Secureum Bootcamp: Epoch 0

Lucas Goiriz Beltrán
&
David Chaparro Misó

October 2021

About the authors

Secureum

Main collaborators

Lucas Goiriz Beltrán

PhD student in Mathematics.

David Chaparro Misó

Software engineer.

Contents

Chapter 1

Ethereum 101: Introduction

1.1 Ethereum: Concept, Infrastructure and Purpose

Ethereum is a **next generation blockchain** that supports **smart contracts** to allow **decentralized applications** to be built on itself. Ethereum was one of the first blockchains to put forth this idea and enter into this concept of a next generation smart contract based decentralized application platform.

One of the fundamental aspects of Ethereum is the fact that it is **Turing complete**: Ethereum supports a **Turing complete programming language**. Turing completeness is a fundamental concept in computer science, which refers to the **expressiveness of a programming language**: what can you do with it, is the logic that you can express with that language arbitrary, is it bounded, is it unbounded...

Many of the high level languages that you may be familiar today (like `C`, `C++`, `Java`, `Python`, `Rust`, `Golang`, ...) are Turing complete.

Therefore, the language supported by Ethereum is expressive enough in arbitrary and unbounded ways. This property is very powerful and it affects both the design and security of Ethereum, smart contracts and the decentralized applications governed by them.

Smart contracts, given that the programming language they are written with is Turing complete, are also Turing complete. This subsequently means that these smart contracts, and the applications they govern, can encode arbitrary rules over arbitrary states in such a way that said states can be read and written using those arbitrary rules. This contributes to what is known as a **state transition**. Think about **finite state automatons from computer science**:

1. You have a `state rule`
2. Said `state rule` is applied to a `state`
3. The `state` is read and modified (which means that it is taken from `state` to `state'`)

The fundamental state transition rule can be done with a Turing complete programming language in arbitrary ways without any constraints on it. These aforementioned rules can be of any kind: rules for ownership, transaction formats, state transition functions...
So any state/any rule allows Ethereum to support any application on it without any artificial constraints coming from the programming language or the platform.

At a high level, Ethereum is an **open source globally decentralized computing infrastructure** that executes smart contracts. By design, everything in the space is open source (the protocol, the specification of the protocol itself and all the code that that actually implements that protocol) so that everything is transparent. This has big implications to security.

Ethereum uses a **blockchain** (the Ethereum blockchain) to **store the various states from the smart contracts**, and given that it's a blockchain, it's **decentralized**: there are many nodes which to agree upon and synchronize the "single view" (global state) that every node agrees on and works with.

So, what is the purpose of Ethereum as a platform? What is the vision that is being worked towards?
Due to the decentralization aspect (there's not one central entity controlling the vision), a lot of these can be thought of as narratives.

Ethereum's initial purpose, put forth in the white paper, was for it **not to be just a currency** and not for it to be just a payment network. This becomes clearer if you're aware of how Bitcoin works: Bitcoin is a predecessor of Ethereum and a large source of inspiration, but Ethereum's vision was to go beyond it being a currency or a payment network.

There is a **native currency** in Ethereum called **Ether** (Ξ). Ether is divisible up to 18 decimals. The smallest unit of ether is known as wei: 10^{18} wei add up to one Ether right. There are other units as well: 1 a Babbage is 10^3 wei; 1

Lovelace is 10^6 wei. These names are in honour of Charles Babbage and Ada Lovelace, which are important people that contributed a lot to computer science.

Ether is used to measure the amount of **resources** that is being used when smart contracts are run. This allows to constrain how long and how many resources the smart contracts use up. It is an important property because it ties with Turing completeness: since smart contracts are Turing complete, the resources and time of execution of a smart contract must be bounded so that it does not take over all the resources of the network, and consecutively collapse it.

While being integral to Ethereum, Ether is not the "*be-all*" or "*end-all*" goal of Ethereum. The idea for Ether was for it to be a utility token: you need the Ether token to utilize the benefits of the Ethereum platform, so if somebody wants to use Ethereum they need to pay using Ether. This is the high level purpose.

You've probably been reading about narratives of Ether being a store of value in a medium of exchange, or a digital gold or a world computer productive asset, things like that. These are all the narratives that are being discussed in the community.

The vision of Ethereum being a world computer is enabled by its rich infrastructure.

1.2 Properties of the Ethereum Infrastructure

High availability and High auditability

High availability refers to the fact that Ethereum is **always up and running** (it's up 24 hours, 7 days a week, 365 days of the year): there's **never a downtime** that is expected because of upgrades or because of any issues (that's the goal) which, again, contrasts with most Web2 services where they might taken down for maintenance, upgrades or any other reasons.

High availability is given by **decentralization**, because of the absence of centralized infrastructure choke points that can go down and bring the whole infrastructure down with them.
High auditability refers to the fact that everything that happens on Ethereum (everything that happens on a blockchain) is auditable (it can be examined, analyzed and reasoned about).

Transparency and neutrality

The fundamental vision of Ethereum is that **applications are permissionless**: if somebody was to build any part of the infrastructure for Ethereum (the

protocol, the Ethereum client, smart contracts...), they can do so **without permission from any centralized entity** within Ethereum. The tools an the infrastructure are **open source**: you can look at it, extend it, deploy and use it without anybody's permission.

This is what lends to **permissionless applications** (decentralized applications), in contrast with how you build, deploy and use nowadays' mobile applications (say on the Apple platform or the Google platform) where you have a centralized entity that you have to register with, get the permission from, test with, follow the regulation set forth by the by that entity (by the Play store or the Apple store) and then deploy it while being subjected to certain "rules" in which you can use it as well.

This is the so-known **contrast** between **Web3** space and the current existing **Web2** space. This is a key aspect of permissionless interaction, development and innovation. All this is **incentivized** because of the built-in economics (crypto economics) which makes people run the Ethereum nodes, deploy and use the applications.

Additionally, as it is built on a blockchain, Ethereum has a high degree of **transparency**: nothing is meant to be proprietary (the source code, the design of the protocols, the transactions that interact with it...) and behind pay walls, or hidden in such a way that you cannot reason about the security or transparency aspects of it.

That's, at least, the high level design goal and all these properties lend themselves to make the platform and everything that's built on it, highly neutral. We'll see more about how decentralization really contributes to neutrality because there's no centralized entity that can change the availability, auditability or transparency aspects of the platform or applications built on top of it.

Censorship Resistance

The aforementioned properties lend themselves to a very high degree of censorship resistance. This is may be something you're familiar with in existing platforms: if an app, a website or anything else does not subscribe itself to the compliance aspects of the platform or any other entity, then it might be taken away from the platform at any time by the entity that is controlling it.

There are many many stories of this being done extremely wrong. There have been accidents where unintentionally some of these apps were taken down because they fit into some larger category type of applications that were being de-platformed. Blockchains in general make this very hard to do at a platform level.

Censorship leads to what is known as "*lowering the counterparty risk*": there is always a risk associated with the party, the platform, the application on top of it or the logic that you're interacting with. Thanks to the transparency, neutrality and censorship resistance, that risk is much much lower.

So none of this is black or white: it's all on a spectrum. We're going towards what is known as "*progressive decentralization*" where some of these properties might not be completely there yet. There might be elements of centralization that over time and by design are removed, so that we reach a point where these applications or platforms are completely decentralized with no single entity or groups of entities that can really manipulate the platform and abuse it. That is where we are headed towards.

1.3 Ethereum Vs. Bitcoin

How does Ethereum compare to bitcoin as a blockchain?

Bitcoin (blockchain) came about in 2008/2009 and it focused by design on the ownership of Bitcoin (cryptocurrency). The consensus of the blockchain (all the operations, states, state transitions...) exclusively focuses on the **ownership of these coins**, and nothing else. So all these state transitions track the transfer of Bitcoin (cryptocurrency) and, in the case of the Bitcoin blockchain, they're referred to as **UTXO**s (Unspent Transaction Outputs).

Compared to that, the **Ethereum** blockchain by design focuses on **general purpose states** (states that do not only focus on the ownership of Ether but anything that can be encoded with the EVM general purpose programming language). So we are looking at a general purpose blockchain that can encode arbitrary states and arbitrary rules for the state transitions that tracks not only the state of Ether cryptocurrency ownership on the platform but the state of the different smart contracts as transactions interact with them.

That is the key difference between Ethereum and Bitcoin: Bitcoin is UTXO based and Ethereum is state based (or account based).

How does the programming language on Ethereum compare to what's available on Bitcoin?

Bitcoin has what is known as `bitcoin script`: it is a **scripting language** (so it's intentionally and by design limited). `bitcoin script` allows a evaluation of spending conditions that evaluate to `true` or `false`, which is what is required for Bitcoin (and what it's supposed to do).

But when you look at **Ethereum**, it supports a virtual machine known as **Ethereum Virtual Machine** (EVM), and by design it is meant to be a **gen-**

eral purpose programming language. Remember it's **Turing complete** (which is a key differentiating feature of Ethereum's expressiveness power when compared to Bitcoin).

1.4 Ethereum Core Components

Network

The underlying network that Ethereum is built on is a peer-to-peer network that's nowadays running on TCP port 30303. The protocol that enables this P2P network is known as ĐΞVp2p.

If you step back and think about the paradigm of the networks that we use today is built around the concept of clients and servers: Laptops, desktops, smartphones, iot gadgets... that we use are all really the clients that are talking to servers sitting in the cloud on AWS or any of the application platforms that you're interacting with.

Compared to that, the key change when it comes to Ethereum is that **the underlying network is all peer-to-peer**: there are no clients and servers, they're all peers that are exchanging messages on a same layer.

On this network we have transactions, which imply the notion of a sender transferring some value (and some data) to a receiving entity.

And on top of that there is this abstraction of a state machine that is driven by the EVM. When it comes to programming that machine, there are high-level languages (HLLs) that programmers and developers work with, being the most common one (the most widely used) `Solidity`, which is converted into EVM instructions (machine language instructions; bytecode).

Data Structures

The Ethereum protocol itself has several common data structures. There is however a very specific data structure known as the **Merkle-Patricia Tree** that is used to optimize the way that Ethereum handles some of the states that are used within the context of the blockchain.

The Merkle Tree is a type of binary tree that is composed of a set of nodes where the leaf node at the bottom of the contains the underlying data, and all the intermediate nodes between the leaf nodes and the root contain the combined hash of their two child nodes. Visually, the data is located at the bottom of the leaf nodes, and all the intermediate nodes (combining their hashes) lead to the root node, which is at the top of the tree and it's usually referred to as "*the root hash*". See **Figure 1.1** shown on the next page.

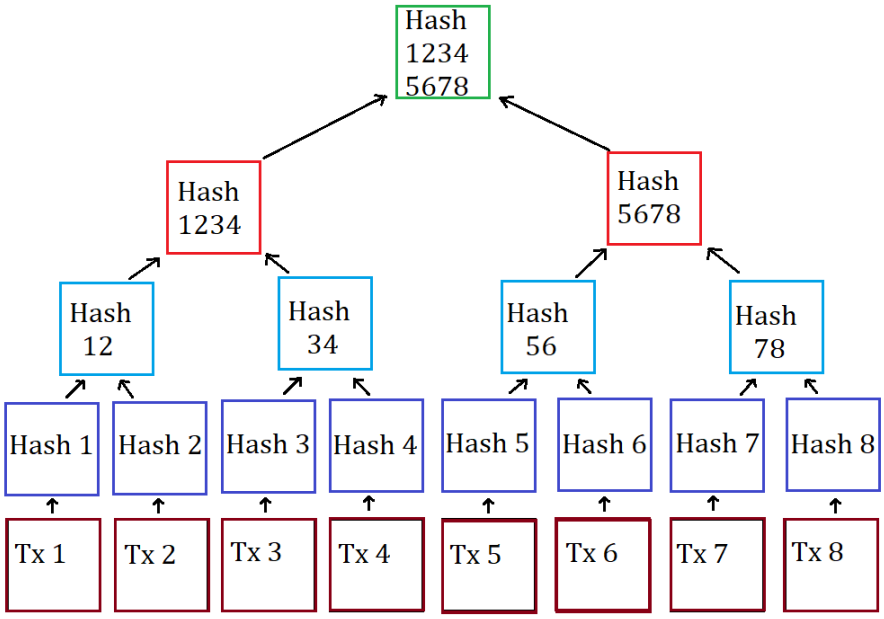

Figure 1.1: Merkle Tree example consisting of 8 underlying transactions.

A Patricia tree (aka prefix free radix tree) is a different data structure where the `(key, value)` pairs that are contained within it have the values at the leaf nodes, and the keys let you traverse a path from the root of the tree to the leaves, so that the nodes that share the same prefix in the key also share the same path down the tree from the root to the leaves. See **Figure 1.2** shown below.

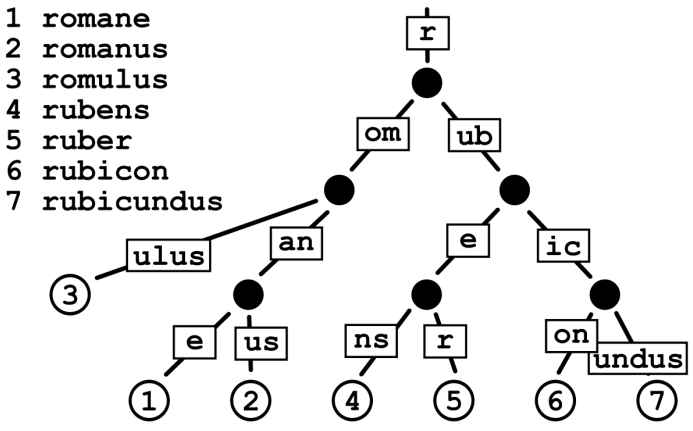

Figure 1.2: Patricia Tree example consisting of 7 words.

Ethereum uses a combination of a Merkle tree and Patricia tree and it's also modified to specifically suit the Ethereum data structures. In this particular case, it uses a Hexane-Merkel-Patricia tree which means that each node has 16 children. There's also a recent proposal to convert this to a binary tree.

When looking at the programming languages supported on Ethereum, let's take for example `Solidity`, there are many common data structures such as arrays, lists, the basic value types, reference types...

Consensus Algorithm

One of the critical core compontents (perhaps the most critical one) is the consensus algorithm.

Why is there a need for a consensus algorithm this?

The Ethereum blockchain, as mentioned earlier, consists in many peer-to-peer nodes: these nodes are all talking to each other and trying to agree upon what is the global state of the blockchain.

Each of the nodes has a local state, which consits on new blocks being formed containing the transactions that are processed by the node. But then, everyone on the Ethereum blockchain should agree upon what is the one canonical blockchain that will be used in the future.

This is agreement is reached through the **consensus algorithm**. Decentralized consensus is critical to any blockchain. In the context of Ethereum it refers to the **Nakamoto Consensus Protocol** that is adapted from Bitcoin. This protocol addresses the problem of which miners' block should be included next to create the canonical blockchain. There are two components to this: **Proof of Work** (PoW) and **The longest chain rule**
PoW is used to determine which entity on the Ethereum blockchain gets to add the next block; **the consensus algorithm** determines which is the longest chain so far, and all the nodes that are chosen to add the next block to the existing canonical blockchain build upon the existing longest chain. This is how a blockchain grows in state.

The consensus algorithm is used as a form of **sybil resistance**. There's this concept of 51% attack, which consists in the scenario of 51% of the participating entities colluding. Then, these malicious entities can change past states (and thus the state transitions) that have been encoded into the blockchain. This breaks the key property of immutability of a blockchain, thus it is very critical.

Ethereum PoW: Present and Future

As mentioned previously, Ethereum Proof of Work is fundamental to how the Ethereum consensus protocol works. Technically, it is referred to as Ethash, and it's formally defined as

$$(m, n) = \text{PoW}\left(H_{\not n}, H_n, d\right)$$
$$m = H_m \wedge n \leq \frac{2^{256}}{H_d}$$

where $H_{\not n}$ is the new block's header but without the nonce (n) and mix-hash (m) components; H_n is the nonce of the header; d is a large data set needed to compute the mixHash and H_d is the new block's difficulty value.

What a miner has to do is to determine a combination of mix-hash m and nonce n for every block to satisfy the constraint on the difficulty for that particular block. This is a trial and error process so the miner has to keep repeating the computation until a combination of m and n that satisfies the difficulty is found. When this is done it means that a sufficient amount of work has been performed by the miner for this particular block, or in other words this indicates that there is proof of work by the miner for this block.

PoW is being replaced by what is known as **proof of stake** (PoS). With Ethereum it's happening with the transition to Ethereum 2.0 or Serenity (also known as **the merge**). The merge is already underway. This change is important from a security perspective: the consensus algorithm is critical to the economic security of a blockchain, as it is what makes the blockchain resistant to attacks from any of the untrusted parties operating the infrastructure or operating on it. For now, this security is provided by PoW.

Ethereum Nodes and Clients

An Ethereum node is a software application that implements the Ethereum protocol specification. It communicates with other Ethereum nodes on the network in a peer-to-peer fashion.

The Ethereum client is a specific implementation of the Ethereum Node. Withing the Ethereum clients, a specification of the protocol itself (the consensus algorithms, data structures,... all these core components) is implemented.
If anyone is running an Ethereum node they're using one of these Ethereum clients that have been built by multiple teams around the world. The most popular ones are

- Geth
- Erigon
- Nethermind

- OpenEthereum

There have been a lot of transitions and changes. Some of the clients have been under development, and some of them are way more popular. Geth is the most popular with more than 80% of the running nodes. But, for the sake of the diversity and decentralization of Ethereum, other clients are being supported and being developed as well. These clients are open source so anyone is free to examine the client code and maybe even contribute to it.

Ethereum transactions are sent to the Ethereum Nodes, and these in turn broadcast them across the peer-to-peer network. This is how Ethereum transactions propagate across the Ethereum network, reach the various nodes, get combined into blocks and result in the blockchain.

Ethereum Miners

Ethereum miners are entities running Ethereum Nodes on the network. They are the ones that receive, validate, execute and combine the transactions into blocks.
They also provide a mathematical proof of their computation (proof of work; PoW). For all this work, if the miner's block gets chosen to be part of the blockchain, then they are rewarded with what is known as a block reward.

This block reward is currently 2 ether and it has decreased over time. This is the crypto economics: incentive for miners to participate and to be honest on the Ethereum network. Along with this reward, they're also rewarded with transaction fees: the ether spent on Gas by all the transactions included in that block.

So block reward and transaction fees are the crypto economic incentives that are paid out to the miner whose block gets accepted into the blockchain.

GHOST protocol

So transactions are sent, and miners validate them. They combine them into blocks and these blocks are propagated over the peer-to-peer network. Multiple miners are doing this process simultaneously. This leads to multiple valid blocks at any level of the blockchain. The canonical blockchain needs to choose one valid block at any level. The choice of that valid block is dictated by the "GHOST protocol" (the Greedy Heaviest Observed Subtree protocol). This protocol allows stale blocks up to seven levels in this calculation. Stale blocks, in Ethereum's nomenclature, are referred to as uncles or armors.

1.5 Gas Metering: Solving the Halting Problem

The Halting Problem

We talked earlier about Turing completeness and how Ethereum supports a Turing complete programming language.

Turing completeness lends itself to a fundamental problem in computer science: the halting problem. Imagine you have a Turing complete programming language, then the halting problem states that: it cannot be predicted if an arbitrary program in that language with an arbitrary input will ever stop execution for that input.

This problem is taken for granted on our personal laptops or on our phones: if there are programs that run into an infinite loop that hangs them, then we just manually kill the execution.

In the context of a blockchain however, we are talking about being able to deploy a smart contract that is not going to run on my Ethereum node exclusively, but it's going to run on all the Ethereum nodes in the blockchain. With that in mind, if any of those smart contracts ran into an infinite loop, then the whole infrastructure would come to a grinding heart, which is obviously undesirable. We would not make further progress and everything would come to a standstill.

The way in which Ethereum deals with this issue is constraining the resources that are given to the smart contract on the Ethereum node by metering them through the concept of Gas.

Gas Metering

The EVM runs smart contracts, which have machine code instructions. Every single one of these instructions has a predetermined execution cost in Gas units. So when a transaction triggers the execution of that smart contract, and the smart contracts starts executing those instructions, then the Gas units for those instructions are "consumed" by the smart contract. This is the concept of Gas metering within a smart contract, where it accounts for every instruction (it could be a computational instruction, a data access,... All those instructions consume Gas units from the transaction).

This implies that a transaction that triggers a smart contract has to include a specific amount of Gas as required by the logic of the smart contract. Depending on what logic needs to be executed by said smart contract, there is a limit to the gas: a certain amount of Gas units are required for a particular flow, so if a transaction is triggering that it must include so many Gas units. In fact, even if the transaction is just triggering a trans-

fer of value, it has to supply the amount of Gas required for what it is triggering.

If the Gas that is required during the execution of the smart contract exceeds what is supplied, or if it exceeds the limit, then the EVM will terminate execution and the transaction will fail. This is fundamental to how Ethereum works, since there is a need to bound the use of resources by anyone interacting with a smart contract that runs on all the blockchain nodes all over the world.

Up until this point, all we have talked about is the units of Gas that have to be provided. However, Gas itself has a price, which is measured in Ether. So this Gas price is not fixed as it depends on the supply and demand of Ether. In this context it depends on the demand for the block space within Ethereum, and if there are many applications (many smart contracts) and/or users competing for that block space, then the Gas price can increase vastly. So just like the automobile analogy (which requires Gas or petrol, and with a fixed amount of petrol one can drive a certain amount of kilometers), the petrol is the equivalent of the Gas units in Ethereum (which allows to run a transaction in a certain number of instructions within the smart contract based on the amount of Gas you supplied). But the Gas price, similar to the price that you pay in Gas stations (petrol bunks) is different and depends on the supply and demand.

This Gas mechanism has recently been updated; take a look at EIP1559 (Ethereum Improvement Proposal 1559).

The take home message for this section is: one needs to obtain gas, which is obtained by purchasing Ether. A transaction requires gas, and if it exceeds the amount of Gas that is supplied, then the transaction fails. But if there is more Gas than what is required, that is supplied as part of the transaction, then the remaining Gas is sent back to the sender who executed the transaction as part of the protocol.

1.6 Web 2.0 Vs. Web 3.0: The Paradigm Shift

In this section, we are going to assume that you are familiar to some extent with Web2 and most of the content will be focused on Web3 and the differences with Web2.

Objectives of Web3

The idea of Web3 is for it to be a permissionless, trust minimized and censorship resistant network for transfer of not only information but also value.

Privacy and anonymity are again very big motivating factors in Web3 and have a huge implication on how we think about security in this space and how we can actually conceive implementing various security measures. However it

isn't a completely fresh start: there are a lot of Web2 security principles, best practices and software engineering best practices that have been researched, experimented, developed and refined over the last 3 or 4 decades that still apply very much to Web3.

In the Web3 world the aspirational goal is that of borderless, permissionless innovation and censorship resistance. This inspires all the applications, smart contracts or any other off-chain components to be open and composable by design
Another Composability means designing components and applications in a modular way, so that other modules (or other applications) can interface with them to increase the utility that is got from either of the two components. This has to be done in a way that is very easy, permissionless and effortless. Again, going back to the Web2 space, a lot of the work that has happened, a lot of the applications that are built by the various enterprises, for various reasons such as protecting their business interests, are designed in such a way that they work well within their ecosystem or their suite of products, and they're not really meant to be very composable or very interactive with other applications or other components from other entities or other vendors (potentially their competitors).

In the Web3 world the aspirational goal is that of borderless, permissionless innovation and censorship resistance. This inspires all the applications, smart contracts or any other off-chain components to be open and composable by design. This means that anybody (any user, any part of the world) can deploy applications and interact with it (be it contracts, be it any other component). This drives innovation in the Web3 space, which is great and has resulted in a very accelerated innovation and compressed time. Again this has implications to to how security is thought of and what is practically possible with security in this space. When you have this unconstrained composability, where any smart contract, any application that is deployed can be acted upon and be combined or connected with any other smart contract on the chain, then it leads to sort of an explosion of the dependencies and configurations that are possible. This makes characterizing Web3 vulnerabilities and exploit scenarios very challenging, because it requires really a very deep knowledge of all these interacting composable components along with their very different constraints that could change because of composability itself, and their configurations could be affected by interactions with other components.
Having said that, there are still many aspects of Web3 security which are really a paradigm shift.

Open-source & Transparency

Due to Web3's ethos (or the design approach) towards everything here being open source and transparent, the way it is thought about the security of the ecosystem has to be changed.

We have known open source in the Web2 world for several decades, so that's not very new. But from a approach perspective, in the Web2 space we do see most of the products, or many of the products, being proprietary from a licensing and from a source code availability perspective. The Web3 ethos stems from the permissionless aspect, the trust minimization aspect and the censorship resistance aspect.

In the context of smart contracts, they're again expected to be open sourced and also expected to be verified contracts. In this case it means that the bytecode (machine instructions) of the contracts deployed on the blockchain are expected to be source code verified using one of the services available (such as Etherscan). This means that the source code of the contract is the same one that was compiled, deployed and that users interact with. That verification is something critical and expected by default in this space.

Remember that everything that happens in the blockchain is transparent; anyone can actually run an Ethereum node so that you don't even have to rely on a block explorer or any service like that (more on that in upcoming secitons). You can look at transactions as they happen in real time on the blockchain, you can also look at transactions that are waiting to get into the blockchain (through mempool explorers). One can even do that by running an Ethereum node themselves. **There is no security by obscurity**.

Code Immutability

In the context of a blockchain being immutable (because of all the blocks being linked with hashes), PoW and what it requires for somebody to go back in time and change one of the blocks or any of its contents, which contributes to the economic security and the immutability aspect of the blockchain.

When it comes to code, the contracts that are deployed on the Ethereum blockchain are designed in a way to be immutable as well. This means that once a contract code is deployed, it is technically considered immutable: it cannot be changed (at least in theory). There are some exceptions, so theoretically it can done, but from a design perspective (from an ethos perspective) this is not desirable. This code immutability affects the way in which security is handled.

As it is already widely known, software will have bugs: one cannot prove the absence of bugs, or the absence of vulnerabilities or errors in a piece of software. Immutability affects this to a great extent: we know that bugs will exist and, if you cannot change the contracts, then how do we fix the code and redeploy the fixed code as we've been used to in the Web2 world (where we keep getting updates to the operating system or to the different apps continuously to fix bugs, optimizations and so on)?

This is something very fundamental to Ethereum or the Web3 space that we

have to keep in mind.

There are practical exceptions: the deployed contract can be modified: the functionality can be modified. This can be done in three ways.

1. The contract can be modified and redeployed at a new address, but then you would have to carry over all the state. And all the users interacting would have to migrate to interacting with the new address.

   This is typically considered impractical but it can theoretically be done.

2. The modified contract (after bugfix or a version 2) can also be redeployed as a new implementation of what is known as a proxy pattern: the proxy contract points to an implementation contract, and that implementation can be modified.

   This is the most commonly used approach to contract upgrading, and again this has huge security applications if it is not done right, if it is not done in a trustworthy manner or if there are certain best practices that are not followed.

3. By using the `CREATE2` opcode, it allows updating a contract in place at the same address using the same unit code that was used to initialize the contract.

We are going to further elaborate on the concepts briefly mentioned here on upcoming sections.

Client-Server Vs. Peer-to-Peer

The pivotal difference between Web2 and Web3 is their underlying paradigm.

Web2 relies on the **client-server paradigm**: we are used to employ cloud services that have servers running, which we interact with using our clients (laptops, phones, smart watches...) that are **centrally managed**, i.e. you have all these corporate entities that determine what the infrastructure is, what the products are and what the next versions of the products will be.

Applications are centrally rolled out and the consumers use them, but they do not have a say in how the infrastructure is managed or how the applications evolve. Sometimes, users don't even have a say as to what kinds of applications can be deployed on the infrastructure: there is a latent **censorship on Web2**.

The former scenario is something that Web3 is trying to get away from: it's trying to go back to the original vision of the web which was for it to be completely peer-to-peer without centralized entities that can dictate what can

be done on the platform, what can be deployed or who can use it.

The idea is employing peer-to-peer communication not only for computing power but also for storage and network, which are the building blocks of Web3. In the case of Ethereum, the peer-to-peer infrastructure that supports these components is known as the **Ethereum triad**

- **Comouting power**: Ethereum itself; Ethereum as a blockchain is used for decentralized compute.
- **Storage**: Swarm.
- **Network**: Whisper (now known as Waku).

Business models

Web2 business models are built around **freemium models** where the basic application is free but if you want to upgrade, you have to pay up. A lot of the business models (Google, Facebook especially) are built around advertising, where the product is free. In some sense, the user becomes the product and the interactions (the data that the user shares with those platforms) is monetized in the form of advertisements that are being delivered to the user. This is something that we've just got used to and we don't even pay attention to anymore.

In Web3, since everything is decentralized, there has to be some incentive for users to deploy the nodes and to contribute to the development of the code, clients, smart contracts and applications. This is driven by what is known as "*incentivized participation*", which goes back to the concept of crypto economics. This has big implications to how security works because there is no real centralized entity that can deal, manage and do instant response.

Programming languages

Programming languages are critical because they are the means of how projects implement their ideas, deploy them and let users interface with them.

Again, programming languages are fundamental to the security of any product, any architecture if you will. In the case of Web2 there have been numerous programming languages, being some of them way more popular than the rest: it started with `C` and `C++` during the Unix days 30 or 40 years ago, and since then we have seen different declarative languages, subject-oriented languages and scripting languages. Some of the most popular ones have been `Javascript` and more recently `Go`, `Rust` and even some unique languages such as `Nim`, that have been used to implement Ethereum 2.0 clients recently.

All those languages are still applicable to the Web3 space, Because Web3 is a combination of smart contracts that run on the blockchain and web interface component (which is how users interact with the contracts on the blockchain). When it comes to the web component, all the Web2 languages are relevant. A lot of them are popular even here in terms of the interface, but also in terms of the tooling that is used for the development of smart contracts themselves.

Smart contracts themselves have a special language. In the case of Ethereum, that's `Solidity`: it is the most widely used language for writing smart contracts on Ethereum. There are others, like `Vyper`, that are gaining traction, but for the majority of the smart contracts that we see deployed, `Solidity` is that language.

The smart contract languages were created specifically for Web3, and specifically for Ethereum in this case. So the security features of those languages have obviously huge implications to the security of the smart contracts themselves and the applications that are built on top of them.

Applications in Web3: ĐApps

Building applications on a decentralized infrastructure (i.e. a blockchain) will cause such applications to be fundamentally different from the mobile applications or the desktop applications that we are used to today (as we can already guess by looking at the programming languages and code immutability in Web3). These applications are referred to popularly as decentralized application or ĐApp.

These applications rely on a concept that is very unique to Web3: **on-chain** and **off-chain**.

- **On-chain** means something that is running or executing on the blockchain or within the blockchain's execution environment.
- **Off-chain** is something that is running outside the blockchain.

In the Web2 space everything is off-chain because, obviously, there is no concept of blockchain. However, in order to function, the Web3 space makes use of an off-chain component combined with an on-chain component.

The off-chain component is all the Web2 or "*the glue*" that binds the web interface to the smart contracts that are running on-chain. This distinction is critical when thinking about security.

To put it simple, a ĐApp combines a web app/web front-end or a mobile app (the off-chain component) that interacts with a smart contract on the blockchain (the peer-to-peer infrastructure which is a combination of compute, storage and network; the on-chain component). In many cases we have one or

more of any of the mentioned.

In the case of Ethereum, what is peer-to-peer is the compute part and the peer-to-peer storage aspects (such as IPFs and some of the other protocols that we won't talk about).

The security of Web3 has to think about the security implications of anything running or interacting off-chain with that running on-chain as well. It's not just smart contract security but also the security of the off-chain web or mobile app components that interface with the on-chain components (the smart contracts).

The main difference however between Web2 and Web3 security is the on-chain component of course. We will need to think about at the pitfalls that are unique to smart contracts and look at the best practices.

Unstoppability & Immutability

Another difference between Web2 and Web3 is that the Web3 applications and infrastructure are unstoppable and immutable.

We talked about ĐApps and how they run on a decentralized infrastructure. The goal is even for the governance of these protocols and infrastructure to be decentralized, this means that there is no single entity that can unilaterally decide to make changes, to start something, to stop the application (be it adapt be the protocol, be the surrounding infrastructure or the governance itself). Specifically, in the case of smart contracts, there shouldn't be a single entity.

This could be the the project team itself that has built out this application or the contracts: they shouldn't be unilaterally allowed to change, stop and withdraw funds. This stems from the trust minimization motivation and the censorship resistance aspect.

They are all very interconnected aspects and as you can imagine they have huge implications to when it comes to upgrading the contracts, fixing the bugs or doing anything with the smart contracts once they're deployed; because they are expected in theory, by motivation and by design (codewise; remember the code immutability we talked about earlier) to be unstoppable and immutable.

From a security perspective, this makes it hard to deploy the software because once you do it, if there are any vulnerabilities found, it's very hard to do an instant response (just what we talked about on code immutability). The latter is something we take for granted in the Web2 world, where we get software updates even without our knowledge.

In the case of Web3, and particularly Ethereum smart contracts, this has huge implications to how we think about design and operationalize security. This is

the ultimate goal, in practice, as we go through different stages of progressive decentralization. These things are evolving, so the best practice right now is to do something known as a "*guarded launch*" (basically initially limiting the functionality of the ĐApp for monitorization purposes). For now the best practice is to have the ability to make changes, to upgrade the contracts, to have an emergency withdrawal function, to remove the tokens in case there is an emergency. It's sort of the stop cap measure as we progress towards complete decentralization. Once we have enough confidence that the contracts are running and there are no bugs, vulnerabilities or exploits, then a lot of these things are disabled in the code: the kill switches or the credibility aspects the governance...

All these things are in a progressive manner made more decentralized over time.

1.7 Decentralization

What does decentralization really mean?

This term is used very casually although it has huge implications to how we think about security, even for smart contracts. There is a definition put forth by Vitalik in his article on decentralization. There are three types of decentralization

- **Architectural decentralization**: it refers to the hardware (the physical computers); who runs them, owns them, who is managing them, who can start them and stop them. This can be done in a decentralized fashion or not.
- **Political decentralization**: it refers to the people behind the hardware or what is commonly referred to as "*wet ware*". Who are the individuals or the organizations who control the hardware or the infrastructure? Is it just one individual or is it a group of individuals? Are the colluding or are they independent and decentralized?
- **Logical decentralization**: it refers to the software: used to build out the applications (the framework itself, the Ethereum code, the protocol itself, the data structures in it, the smart contracts or any other software that runs in that stack). Is that decentralized? Is it a monolithic entity that cannot be split apart and used in a decentralized fashion?

All of these have security implications.

1.8 Cryptography, Digital Signature and Keys

Most of you know tat there are two classes of cryptography: **symmetric cryptography** (there is a single key shared between aprties) and **asymmetric cryptography** (there is a key pair: public key and private key). In the case of

Ethereum, the cryptography that is used is all about digital signatures and not as much about encryption at a protocol level. These digital signatures however depend on the concept of public key and private key.

Private Key

The private key is a secret and the owner has to keep it in a safe place. In the case of Ethereum, it's a 256 bit private key. It's effectively a random number and it's used to derive the public key.

Public Key

The public key, however, is not secret. It is a point on the elliptic curve calculated from the private key using elliptic curve multiplication. The public key is used then to derive the address of an Ethereum account (by hashing the public key by means of the keccak-256 cryptographic hash function and taking the last 20 bytes of the output; it is a very simple calculation) and it is also used by others to engage in cryptographic protocols with the owner of the private key.

It is important to remember that the public key cannot be used to derive the private key. This is should be something obvious to security, because otherwise if the public key could be used to derive the private key, then this key pair system would not deliver any kind of security.

This is the high-level aspect that you need to remember: there's a private key, which is used to obtain the public key, and from the public key we derive the address of the Ethereum account.

keccak-256

We mentioned earlier that the keccak-256 cryptographic hash function is used in the steps of computing the EOA address from the public key.

keccak-256 is actually the cryptographic hash function that is used by Ethereum. It is very closely related to the SHA3 (the secure hash function). The latter was finalized as the standard by MIST (National Institute of Standards and Technology) and in the case of Keccak-256, it was the winning candidate for the SHA3. However, the SHA3 standard was adopted instead.

keccak-256 is critical to a lot of the functioning of the Ethereum protocol and smart contracts as it's a fundamental primitive to how computation in many ways is done on Ethereum.

Digital Signature: ECDSA

The digital signature algorithm used by Ethereum is the same one that is used by Bitcoin. It is known as ECDSA: **Elliptic Curve Digital Signature Algorithm**. Elliptic Curve Cryptography is an approach to public key cryptography based on a particular algebraic structure of elliptic curves over finite fields.
In the case of Ethereum, the particular elliptic curve used is known as Secp-256k1 (this refers to the parameters that are used for the elliptic curve).

Digital signatures are fundamental to how Ethereum works, are powered by public key cryptography (asymmetric cryptography) and habe three main purposes:

1. **Authorization**: inclusion of the signature proves that the owner of the private key who created the signature (and who by implication is the owner of the sending Ethereum account) has authorized the transaction to spend the ether or to execute the contract that it is targeted.
2. **Non-repudiation**: once the signature has been included, it cannot be later denied that authorization was granted for that transaction to execute.
3. **Integrity**: it proves that the transaction data has not been modified or cannot be modified by anyone after the transaction has been signed. This is one of the fundamental security properties.

1.9 Ethereum State & Account Types

Ethereum state is a mapping from the address of the Ethereum accounts to the state contained within it as a data structure. It is implemented as a **Modified Merkle-Patricia Tree**: a combination of a Merkle tree and a Patricia tree with some changes that are specific to Ethereum.

Each of the Ethereum accounts has a unique 20 byte address associated with it, which is used by accounts to "talk to each other". Addresses are critical to how messaging works within the Ethereum protocol and how the accounts engage in transfer of value or information, since accounts need to be able to refer to each other using their addresses.

In addition, accounts have four fields:

- **Nonce**: a counter that's used to make sure that each transaction can only be processed once used to prevent replay attacks.
- **Balance**: a number representing the amount of Ether that the account has at any point in time.

- **Contract code**: smart contract code (absent in Externally Owned Accounts).
- **Contract storage**: associated smart contract storage (absent in Externally Owned Accounts).

Account types

Ethereum has two account types:

- **Externally Owned Account (EOA)**: it is an account that is controlled by a private key. Anyone who has a private key can create a digital signature that can be used to control the Ether that is present in an EOA. These signatures can be used to sign transactions from the EOA, which in turn can trigger messages from the EOA to other accounts. These messages can result in a transfer of value or they can trigger smart contracts.

  An EOA does not have any associated code or storage.

- **Contract account**: it is an account that is controlled by the code that is contained within that account.

  Unlike EOAs, contract accounts have an associated smart contract code and storage. Whenever the contract account receives a message, it triggers the code present and accesses any internal storage associated with it. When the code runs it can send messages to other accounts or even create new contracts.

In this sense, smart contracts can be thought of autonomous agents as they're always present in the execution environment of the Ethereum blockchain. They're always ready to be triggered by a transaction or a message that is sent to them. Through their contract account they have access to the Ether balance and the contract storage. The execution of the code results in manipulation of this balance and the contract storage.

1.10 Transactions: Properties and Components

Transactions are **signed messages** that originate outside of the Ethereum blockchain. They are **triggered by EOA**s (that are managed or controlled by a private key). The trigger happens to be the digital signature, derived from the private key. These transactions are transmitted by the Ethereum network and they trigger state changes on the blockchain. In fact, Ethereum is fundamentally a transaction based state machine, as only transactions are capable of triggering state changes.

Properties

1. Transactions are **atomic**: they run from the beginning to the end completely, it's either all or nothing.

   So the side effects of the transactions are only reflected in the blockchain if they run to completion. If they don't, nothing that they do is reflected on the blockchain and it's as if the transaction never happened. In other words: transactions cannot be divided or interrupted with some of the partial state being reflected on the blockchain and the rest of it not.

   This contrasts withtraditional computing environments where a particular process or a particular control might be interrupted, gets context washed out and then something else (a different process or a thread) executes and then the original context is brought back. None of that happens within the context of Ethereum.

2. Transactions are **serial**: they're executed one after the other, sequentially without any overlapping. There is no parallelism when it comes to the execution of transactions.

3. Transaction **inclusion**. When a user submits a transaction, what is the warranty that it gets included within one of the blocks on the Ethereum blockchain? This property is controlled by entities on the Ethereum blockchain known as miners. They run Ethereum nodes and decide which transactions are included within a block. This depends on multiple factors, the key ones being the congestion on the Ethereum network (or in other words, the other transactions that are competing for the same block space) and the Gas price that the user decides to use for the particular transaction.

4. **Inclusion order**. It refers to the specific order of the transactions included within a block. Again, th is chosen by the miners and, similarly to inclusion, is determined by factors of congestion and Gas price.

   The key takeaway of properties 3 and 4 is that there are entities known as miners on the blockchain who get to decide which transactions get included within a certain block and the specific order of the transactions within that block.

Components

Transactions contain seven components:

1. **Nonce**: the name is an abbreviation for "a number used only once". It's a sequence number that, as part of the protocol, is incremented in

a particular fashion (it changes for every transaction). The application of the nonce is prevention of replay attacks (i.e. replaying the same transaction over and over again).

In the case of an EOA, the nonce value is equal to the number of transactions sent from that account. In the case of a contract, it is equal to the number of other contracts created by this contract account.

2. **gasPrice**: the price for every Gas unit that the sender is willing to pay for a particular transaction. It's measured in wei/gas.

   The gasPrice is not fixed by the Ethereum protocol. The higher the Gas price that the the sender is willing to pay for this transaction, the faster the particular transaction gets included by the miner into a block in the blockchain. This price depends on the demand for the block space at the point in time when the transaction is submitted.

   The reason for this is that there is a limited amount of space in the block, so there's only a limited number of transactions (as determined by the Gas used by each one of them) that can be included within this block.

3. **gasLimit**: the maximum number of Gas units that the sender is willing to pay for a particular transaction.

   This depends on the type of transaction that is being sent. If it is a simple Ether transfer then it costs 21000 Gas units. But if it is a transaction that is targeting a particular contract (or a particular function of the contract) then the required amount of gas is higher.

   If sufficient Gas (in the form of Gas limit) is not set for the transaction (if it's less than what is required to), then it results in what is known as an **out of Gas exception** (OOG Error) and that transaction fails. The way it works is that for any transaction that is being sent by a sender, there is an estimated Gas that needs to be sent as part of the transaction. If that estimated amount of Gas is not sent then it leads to the exception. If there is excess gas, then the remaining Gas is sent back to the sender.

4. **Recipient**: the destination 20 byte Ethereum address for a transaction (i.e. the destination account that this transaction is targeting).

   This could be an EOA address or a contract address, it depends on the target of that particular transaction. It could be any address on the Ethereum blockchain, and the protocol itself does not validate these recipient addresses in the transactions.

So one can send a transaction to any address and that address might not even have a corresponding private key, nor the contract that the sender expects to have. Thus, all such validation should be done at the user interface level. That validation is critical for security reasons (more on that in later chapters). Note that this recipient is really the target address.

There is no "from address" that is a component of the transaction. The reason for this is that the "from address" can be derived from the ECDSA signature components v, r and s: they can be used to derive the public key, which in turn can be used to derive said address.

5. **Value**: the amount of Ether (in wei) that the sender is sending to the recipient address.

   What happens with such funds depends on the recipient: if it happens to be in EOA, then the balance of that account will be increased by this value and the sender's balance correspondingly decreases. If it hapens to be a contract account, then what happens depends on any other data present as part of this transaction (i.e. the contract function being invoken with the transaction data).

   If there is no data being sent as part of this transaction, and the destination happens to be a contract account, then the contracts' `receive` or `fallback` function (if they were defined or if they are present; more on that in the upcoming chapters) are triggered and thus, what happens with the received Ether depends on their implementation.

   If there is no `fallback` function, then the transaction results in an exception and the Ether, that is sent as part of the transaction, remains with the sender account.

6. **Data**: payload of variable length and binary encoded (as per the format required by Ethereum) that is sent as part of this transaction.

   This field is relevant when the recipient is a contract account. As mentioned previously, the data in that case contains the contract function that is being targeted by the transaction plus the specific arguments that are relevant for said function.

7. ***v*, *r* & *s***. The ECDSA signature is 65 bytes in length and has three subcomponents: v, r and s.

   - r and s represent the signature components. They are 32 bytes in length each (adding up to 64 bytes).

- The final subcomponent, v, is the recovery identifier. It's just one byte and its value can be either 27 or 28, or it can be twice the value of chain ID ($2 \times \text{ID}_{\text{chain}}$) plus either 35 or 36.

  The chain ID is the identifier of the blockchain. In the case of the Ethereum mainnet chain, ID = 1.

For a particular transaction, the Ether used to purchase Gas is credited to the beneficiary address that was specified in the block header (more details on this are found in the upcoming sections). Then there's also a concept of a Gas refund: the difference between the Gas limit and the Gas Used is refunded back to the sender of the transaction. This is done at the same Gas price as indicated in that transaction.

1.11 Contract Creation

We mentioned that a transaction can result in contract creation.

The creation transaction is a special one because it's sent to a special destination address called "*the zero address*"(`0x0` address), which is an address that has zero in all its bits. This zero address is treated in a special maner within Ethereum, and it becomes critical to some of the smart contract security properties.

It contains a data payload which represents the byte code of the contract that is being created, and it may also contain an optional Ether amount in the value field, in which case the new contract that is being created, will have a starting balance equal to this Ether value.

1.12 Transactions, Messages and Blockchain

Distinction between Transactions and Messages

So far, we have used both the terms Transaction and Message interchangeably, but in the context of the protocol they are actually very different:

- **Transactions**: originate off chain (by an EOA when an external actor, that is, external to the blockchain, sends a signed data package onto the blockchain) and target an entity on the blockchain.

  This transaction can trigger a message that can do one of two things:

  1. It can trigger a message to another EOA, in which case it leads to a transfer of value (transfer of Ether).

2. It can trigger a message to a contract account, in which case it leads to the recipient contract account running its contract code and doing whatever the code is intended to do.

- **Messages**: the origination and the destination are both onchain. Messages can be triggered in two ways:

  1. externally by a transaction. The destination of that message could be another EOA or another contract account.
  2. internally within the EVM. This happens when a smart contract executes the call family of opcodes and that leads to the recipient contract account running its code, or value transfer to the recipient.

How to build a Blockchain

Blocks are batches of transactions that are grouped together plus the hash of the previous block (a cryptographic hash that is derived from the previous flux data), creating thus a "chain" among the blocks. This is how at high level a blockchain is constructed, and it is also the fundamental reason why a blockchain is considered immutable, which lends itself to the blockchain's integrity.

The reason for that is because if someone were to change any component of a historical block (any of the transaction data: the destination or any other aspect) then that change would affect all of the following blocks: all the hashes that are included in the following blocks would be different from the hash of the modified block, and this is something that anybody running the blockchain or looking at the blockchain would notice. That would break the the immutability of the blockchain.

To preserve the transaction history, blocks are ordered. Therefore, every new block created contains a reference to its parent block and similarly, transactions within the blocks are also strictly ordered. All these critical aspects are the reason why the integrity of a blockchain is maintained and prevents any fraud from happening.

Block Header

So far we have said that the blocks in the Ethereum blockchain contain transactions, but there's more to it: every block contains a block header along with the transactions. The headers of the Ommer's blocks. Each block header itself has several components to it that are critical to how the Ethereum blockchain functions. They contain several things such as:

- **The parent hash**: the hash of the parent block's header. This is what chains the blocks and the Ethereum blockchain together to make it immutable and provides the fantastic integrity property of the blockchain.

- **The Ommer's hash**
- **Beneficiary address**: the address of the Ethereum account to which the block reward for mining this block and all the transaction fees collected from the mining of the transactions included in this block are transferred to. This address is typically controlled by the miner who has mined this block.
- **stateRoot**: one of the three root hashes of the modified Merkle-Patricia tree. These root hashes are 256 bit in length.

  The manner in which the state root is derived is critical to how the Ethereum state is captured within the blockchain: the leaves of the state root are `(key, value)` pairs of all the Ethereum address accounts. The keys are the Ethereum addresses of the accounts and the values represent the Ethereum state within that account.

  Recall that every Ethereum account has four fields: a nonce, a balance, a codeHash and a storageRoot. If that account happens to be an EOA, then the codeHash and storageRoot don't really matter (they don't contain anything in them). But if that account happens to be a contact account, then the codeHash has the Keccak-256 hash of the code that is contained within that contract account, and the storageRoot of that contract account has the rootHash of another Merkle-Patricia tree where the leaves represent the storage that is associated with that contract.
- **transactionsRoot**: one of the three root hashes of the modified Merkle-Patricia tree, where the leaves represent the transactions. Also 256 bit in length.
- **receiptsRoot**: one of the three root hashes of the modified Merkle-Patricia tree, where the leaves represent the transaction receipts. Also 256 bit in length.

  But what is a transaction receipt?

  A transaction receipt can be thought of as the side effects of a particular transaction that are captured on the blockchain. Besides any changes to the account state that transactions might make, there are other side effects that are captured on the blockchain for this particular transaction. It is a tuple that contains four items:

  1. **The cumulative Gas used**: the total Gas used in the block up until right after this particular transaction has happened, so in some sense it captures the ordering of the transactions within the block.

2. A **set of logs**: related to the concept of events in `Solidity` (which we will study in the `Solidity` chapter). These are events that can be generated by any transaction that is captured on the blockchain. They're really critical to how off-chain components, user interfaces and other components monitor what's happening with a smart contract.
3. The **Bloom filters** specifically associated with those logs: they capture the indexed parameters for every event, so that one can query particular parameters of that event in a faster manner.
4. A status quo: what really happened with the transaction.

- **LogsBloom**
- **dificulty**: difficulty of the block in the context of PoW.
- **block number**: the number of blocks that have been mined so far right. This number sort of indicated the position of the block within the blockchain.
- **gasLimit** (called Block Gas Limit under more formal contexts): the Gas limit that's specific to this block. This concept is essential to Ethereum as it dictates **the number of transactions that are added in this block**

  This concept is different from the Gas limit which we talked about earlier that was specific to the transaction. This Block Gas Limit refers to the total Gas that is spent by all the transactions in that block and this effectively caps the number of transactions that can be included within that block.

  So the block size is in fact not fixed in terms of the number of transactions, but it's fixed in terms of the Gas used by all the transactions. The reason for that is thst every transaction can consume a different amount of Gas.

  The Block Gas Limit is set by the Ethereum miners in a very interesting way: by voting on the blockchain. This is currently set to 15 million and it has also changed over time, depending on the miners' voting. It also represents the level of demand there is for the block space on Ethereum.
- **gasUsed**: the total Gas used by all the transactions in this block.
- **extraData**
- **timestamp**: (derived from the unix time) indicates at what point in time was the block was mined.
- **mix-hash**: critical component of the PoW. See **subsection ??**.
- **nonce**: critical component of the PoW. See **subsection ??**.

1.13 EVM (Ethereum Virtual Machine) in Depth

The EVM is the execution component of the Ethereum blockchain: is the runtime environment where all the smart contracts run. Recall that EVM is a quasi turing complete machine: it's turing complete because the underlying programming language supports arbitrary logic unbounded complexity, but it's also bounded by the amount of Gas provided as part of every transaction.

The Ethereum code runs within the EVM and it is written in a low level stack based language referred to as the EVM Machine Code. This code consists of a series of bytes (therefore referred to as a bytecode) where every byte represents a single operation. So the opcodes are very simple and each of them is a single byte.

EVM Arquitecture

Computer architectures are typically classified into either **von Neumann architecture** or **Harvard architecture**. This depends on how code and data are handled within the architecture: Are they stored together? Are they transported over the buses together? How are they cached? And so on...

In the case of the EVM, the code is stored separately in a virtual ROM and there is a special instruction to access the EVM code.

EVM is a very simple stack based architecture: the operands for EVM instructions are placed on the stack and the output of those instructions is also returned on the stack. There's no concept of registers, virtual registers or anything like that.

Every architecture has a concept of a word and in the case of the EVM, the word size is 256 bits. It's believed that this was chosen to facilitate some of the fundamental operations around the 256 hash scheme and the elliptic curve computations.

The architecture is made up of four fundamental components:

1. **The stack**: The EVM has 1024 elements in the stack and each of those elements is 256 bits in length (equal to the word size). EVM instructions are allowed to operate with the top 16 stack elements. Most EVM instructions operate with the stack (because it's a stack based architecture) and there are also stack specific operations.

2. **The volatile memory**: in EVM, data placed in memory is not persistent across transactions on the blockchain. It is also linear (it's a byte array and therefore addressable at byte level) and zero initialized.

There are three specific instructions that operate with memory, such as `MLOAD` which loads a word from memory and puts it onto the stack; `MSTORE` which stores a word in memory from the stack; and `MSTORE8` which stores a single byte in memory from the stack. These instructions (and more) will be reviewed in more detail in the following sections.

3. **The non-volatile storage**. Unlike memory, storage in EVM is non-volatile: data put in storage is persistent across transactions on the blockchain. It is implemented as a `(key, value)` store between 256 bit keys and 256 bit values, and it is also zero initialized.

   To understand how storage fits in within the concept of accounts and the blocks on the blockchain, recall that every account has a storageRoot field. This storageRoot field, implemented as a modified Merkle-Patricia tree, captures all the storage associated with that account. This is relevant for contract accounts that have associated storage. These storageRoots within the account are further captured as part of the stateRoot, which is one of the fields in the block header.

   There are two instructions that operate specifically on storage: `SLOAD` which loads a word from the storage and puts it onto the stack; and `SSTORE` which takes a word from the stack and puts it into storage.

4. **Call data**: it is used specifically for data parameters of transactions and message calls. It is read only (it cannot be written to) and it's also bite addressable.

   There are three specific instructions that operate with call data: `CALLDATASIZE` which gives the size of the supplied call data and puts it onto the stack; `CALLDATALOAD` which loads the call data supplied onto the stack; and `CALLDATACOPY` that copies the supply call data to specific region of memory.

EVM Ordering

Another concept typically associated with architectures is the concept of ordering: **big-endian** ordering versus **little-endian ordering**. In the case of the EVM, it uses the big-endian ordering: the most significant byte of a word is stored at the smallest memory address while the least significant byte is stored at the largest address.

Instruction Set

All the instructions supported by the EVM can be classified into 11 categories. Instructions that are found in categories **a** to **i** operate on the stack.

The format for each of these instructions will be as follows:

```
OPCODE MNEMONIC INPUTS OUTPUTS
```

Let's see an example:
The opcode is the hex representation of the instruction. You will see that the `0x00` opcode is used for the stop instruction. In addition, the word `STOP` is the mnemonic of the instruction. and then the two numbers that you see after the mnemonic refer to the number of stack items placed for this instruction (inputs) and the number of stack items removed (outputs).

So the stop opcode is `0x00`, thus it's the first instruction in the instruction set mapping. The mnemonic is `STOP` (makes sense), 0 items are placed and 0 items are removed from the stack.

In the case of add, you will see that it has 2 items placed onto the stack (the 2 operands) and the computed result (the addition) is placed back onto the stack. That's why you see that there is one item placed onto the stack, which is the result the addition of the two inputs. The same thing holds good for multiplication, and so on...

a. Stop & Arithmetic

```
0x00 STOP 0 0
0x01 ADD 2 1
0x02 MUL 2 1
0x03 SUB 2 1
0x04 DIV 2 1
0x05 SDIV 2 1
0x06 MOD 2 1
0x07 SMOD 2 1
0x08 ADDMOD 3 1
0x06 MOD 2 1
0x07 SMOD 2 1
0x08 ADDMOD 3 1
0x09 MULMOD 3 1
0x0a EXP 2 1
0x0b SIGNEXTEND 2 1
```

Code 1.1: Stop and Arithmetic Instructions.

b. Comparison & Bitwise Logic

```
0x10 LT 2 1
0x12 SLT 2 1
0x20 GT 2 1
0x13 SGT 2 1
0x14 EQ 2 1
0x15 ISZERO 1 1
0x16 AND 2 1
```

```
0x17 OR 2 1
0x18 XOR 2 1
0x19 NOT 1 1
0x1a BYTE 2 1
0x1b SHL 2 1
0x1c SHR 2 1
0x1d SAR 2 1
```

Code 1.2: Comparison and Bitwise Logic instructions.

c. SHA3 Instruction

```
0x20 SHA3 2 1
```

Code 1.3: SHA3 instruction.

This single instruction is critical to Ethereum. It computes the Keccak-256 Hash. The formal notation for how the Keccak-256 hash is calculated is

$$\mu'_s[0] = \text{KEC}\left(\mu_m\left[\mu_s[0]\left(\mu_s[0] + \mu_s[1] - 1\right)\right]\right)$$
$$\mu'_i = \text{M}\left(\mu_i, \mu_s[0], \mu_s[1]\right)$$

This is explained with more detail in the Yellowpaper.

d. Environmental Information Instructions

These set of instructions give information about the environment or the execution context of the smart contract executing them.

```
0x30 ADDRESS 0 1
0x31 BALANCE 1 1
0x32 ORIGIN 0 1
0x33 CALLER 0 1
```

Code 1.4: Address, balance, origin and caller instructions.

The address instruction gives the address of the currently executing account. Balance gives the ether balance of the currently executing account. Origin gives the address of the originator of the transaction that actually led to the execution of the code within the EVM. Caller gives the caller's address in the context of `Solidity`, these would be transaction origin and message sender respectively.

```
0x34 CALLVALUE 0 1
0x35 CALLDATALOAD 1 1
0x36 CALLDATASIZE 0 1
0x37 CALLDATACOPY 3 0
```

Code 1.5: Call value, call data load, call data size and call data copy instructions.

Call value in the context of `Solidity` would be the message value that you would see in the smart contracts.

```
0x38 CODESIZE 0 1
0x39 CODECOPY 3 0
0x3a GASPRIZE 0 1
0x3b EXTCODESIZE 1 1
```

Code 1.6: Code size, code copy and gas price instructions.

Code size this gives the size of the code running in the current environment. Code copy lets you copy the code running in the current environment to memory. Gas price in the context of `Solidity`; you would see this as a `transaction.gasPrice` which gives you the price of the Gas in the current environment.

```
0x3b EXTCODESIZE 1 1
0x3c EXTCODECOPY 4 0
0x3d RETURNDATASIZE 0 1
0x3e RETURNDATACOPY 3 0
0x3f EXTCODEHASH 1 1
```

Code 1.7: Ext code size, ext code copy, return data size, return data copy and ext code hash instructions.

This set of instructions lets you query an external contract account. Ext code size gives you the size of the specified accounts code. Ext code copy copies the specified accounts code to memory. Return data size gives the size of the output data from the previous call in this current environment. Return data copy copies that return data. Ext code hash gives the hash of the external account's code.

e. Block Information Instructions

Similar to environment key information, EVM also has a set of instructions that gives information about transactions block.

```
0x40 BLOCKHASH 1 1
0x41 COINBASE 0 1
0x42 TIMESTAMP 0 1
0x43 NUMBER 0 1
0x44 DIFFICULTY 0 1
0x45 GASLIMIT 0 1
```

Code 1.8: Block hash, coinbase, timestamp, number, difficulty and gas limit instructions.

Block hash gives the hash of one of the specified 256 most recent complete blocks. If the specified block is not one of the most recent 256 ones, then this instruction returns zero, which is something that has a security implication. Coinbase gives the block's beneficiary address (the address to which the block reward and transaction fees are credited to). Timestamp gets the block's timestamp. Number gets the block's number. Block difficulty gets the block's diffiulty and block Gas limit gets the block's gas limit.

f. Stack, Memory, Storage and Flow Instructions

The next category of instructions are related to the stack memory and storage; load and store operations and also those that affect the control flow.

```
0x50 POP 1 0
0x51 MLOAD 1 1
0x52 MSTORE 2 0
0x53 MSTORE8 2 0
0x54 SLOAD 1 1
0x55 SSTORE 2 0
0x56 JUMP 1 0
0x57 JUMPI 2 0
0x58 PC 0 1
```

Code 1.9: Pop, mload, mstore, mstore8, sload and sstore instructions.

Pop pops an element of the stack. Mload and mstore load and store from memory. Again, mstore8 stores a single byte to memory instead of the word. Storage load and storage store load and store words from and to the storage.

The next set of instructions affect the control flow.

```
0x56 JUMP 1 0
0x57 JUMPI 2 0
0x58 PC 0 1
0x59 MSIZE 0 1
0x5a GAS 0 1
0x5b JUMPDEST 0 0
```

Code 1.10: Jump, JumpI, PC, msize, gas and jumpdest instructions.

Jump jumps to the specific location. We also have a conditional jump that conditionally jumps depending on the value specified. Program counter gives you the value of the program counter. Msize gives the size of active memory in bytes as of this instruction. Gas gives the amount of available Gas as of this instruction and this is in the context of the Gas that is supplied with the transaction: how much gets consumed and how much is left. Jumpdests has no effect on the machine state: it does not affect the control flow but it marks a particular destination as being a valid destination for the jump instructions that we talked about.

g. Push Operations

The next set of instructions are specific to the stack. These instructions push operands or place items onto the stack. Depending on the number of items placed, there are 32 such instructions.

```
0x60 PUSH1 0 1
0x61 PUSH2 0 1
 .      .
 .      .
 .      .
0x7f PUSH32 0 1
```

Code 1.11: Push instructions.

Push 1 pushes a single byte onto the stack, push 2 pushes 2 bytes and all the way to push 31. The push 32 instruction pushes a full word (32 bytes or 256 bits) onto the stack.

h. Duplication Operations

The next category of instructions that operate on the stack are the duplication operations, which duplicate items that are already on the stack.

```
0x80 DUP1 1 2
0x81 DUP2 1 2
 .      .
 .      .
 .      .
0x8f DUP16 1 2
```

Code 1.12: Duplicate instructions.

Dup 1 for example duplicates the first stack item, dup 2, dup 3 all the way to dup 16 duplicate those respective stack items.

i. Exchange Operations

The final set of instructions that operate on stack items are the exchange operations. These exchange or swap items that are already on the stack.

```
0x90 SWAP1 2 2
0x91 SWAP2 3 3
 .      .
 .      .
 .      .
0x9f SWAP16 17 17
```

Code 1.13: Exchange instructions.

Swap 1 for example exchanges the first and second stack items, swap 2 exchanges the first and third and so on all the way to swap 16 that exchanges the first and 17th stack items.

j. Logging Operations

These operations append log records from within the execution context of the contract onto the blockchain. We talked about this a bit in the context of the bloom filter in the block header.

```
0xa0 LOG0 2 0
0xa1 LOG1 3 0
0xa2 LOG2 4 0
0xa3 LOG3 5 0
0xa4 LOG4 6 0
```

Code 1.14: Log instructions.

These instructions differ in the number of topics that are specified as being part of the log. So the log itself refers to the event that is fired from within the context of the contract and in the event. The different parameters can be specified as either being indexed or non-indexed. Indexed parameters go into the topics part of the log and the non-indexed parameters go into the data part of the log. This differentiates how fast the the parameters or the records can be queried, searched and looked. These instructions are critical to how the contracts actually communicate some of their state to the off-chain interfaces or the off-chain monitoring tools.

k. System Operations

The next set of instructions include instructions that are critical to how the system functions. They allow one to create new contract accounts, call from one account to another in different ways, revert from the current executing context, invalidate some of the things that have happened and so on.

```
0xf0 CREATE 3 1
0xf1 CALL 7 1
0xf2 CALLCODE 7 1
```

Code 1.15: Create, call and call code instructions.

Create is used to create a new contract account that has associated code and storage with it. Recall that contract accounts can be created from an EOA by sending a special transaction to the zero address (`0x0`) or they can also be created from other contracts when they're executed. The address of the newly created account depends on the sender's address and the nonce of that account. So this makes the newly created contracts address dependent on the previous transactions that have executed from the sender's account. This becomes interesting when we talk about the related instruction called create 2.

Call instruction allows the current executing context to do a message call into another account. So now there is a caller account that is doing a message call into a callee account. This is interesting because it lets contracts call each other in the executing context.

Call code is another related call instruction which lets the caller account call a callee account and lets the callee account execute its code in the context of the state of the caller's account. This distinction is really critical and it has big security implications in some of the future instructions we'll talk about.

```
0xf3 RETURN 2 0
0xf4 DELEGATECALL 6 1
0xf5 CREATE2 4 1
```

Code 1.16: Return, delegate call and create 2 instructions.

Return holds execution and returns the output data. Delegate call is a very interesting instruction part of the call family of instructions which acts

very similar to call code: where there is a caller account that calls into the callee account, where the callee account executes its code in the context of the caller's state. The difference here between call code is that in the case of delegate call, the values of sender and value of the caller is used in the case of callee. In the context of `Solidity`, your message sender and message value of the caller's account is used in the execution context of the callee account.

Create 2 is similar to create and is used to create new contract accounts with associated code and storage. The difference here is that create 2 allows you to create accounts with a predictable address for those accounts, unlike create where the address of the newly created contact account depended on the nonce. So create 2 removes all the transactions that happened from the sender's account so that the address of contracts being generated are very predictable. This has big implications again to security.

```
0xfa STATICCALL 6 1
0xfd REVERT 2 0
0xfe INVALID ∅ ∅
```

Code 1.17: Static call, revert and invalid instructions.

Static call is another instruction in the call family which allows the callee account that is being called into to only read the state of the caller account without letting it modify it. This has security implications as well.

Revert holds execution of the current executing context, it returns the data and it returns the remaining Gas that's left behind after consuming all the Gas that was supplied as part of the triggering transaction so far.

Ihe invalid instruction `0xfe` is again a special instruction in EVM. It consumes all the Gas that's been supplied as part of the triggering transaction and it is used in the context of some of the static analysis tools that we'll touch upon in later chapters.

```
0xff SELFDESTRUCT 1 0
```

Code 1.18: Self destruct instruction

The final instruction `0xff` in the EVM instruction set is a special instruction called self-destruct. As you can imagine, this holds execution but it also destructs the account of the executing context. This account is registered for later deletion. This has huge security implications because the contract account that is executing will not exist after the transaction finishes. This is something we will touch upon and some of the security aspects as when we talk about some of the findings and security pitfalls in later chapters.

Gas Costs

We have talked about Gas in the context of transaction, in the context of the block Gas limit and so on...
But where it really matters is in the context of Turing complexity and quasi-Turing completeness. The boundedness imposed on the EVM programming language is stemming from the Gas costs that are associated with each of the different EVM instructions.

All these instructions have different Gas costs and the reason for that is because each of them has a different requirements when it comes to the computation processing power of the executing Ethereum node, and also the storage requirements, memory accesses and the disk accesses on the real physical hardware that's running the Ethereum node in the context of a miner or anyone else.

When we look at the Gas costs, the simplest instructions like stop, invalid and revert (that only affect the executing context in a very special way, without having a very high demand or no demand on the processing or the storage of that executing physical hardware), the Gas cost is zero.

For most of the arithmetic, logic and stack instructions, the Gas costs vary between 3 to 5 Gas units. Let's contrast this with some of the more demanding instructions like the call family of instructions, the balance, the ext code hash, export, copy... Those kinds of instructions have a much greater processing requirement from the Ethereum node: these now cost 2600 Gas units.

This again contrast with the memory instructions like memory load and store, which within the context of the EVM are very simple instructions that operate on EVM's internal data structures. These memory instructions cost only 3 Gas units. However the storage instructions like sload and sstore; because they deal with persistent state and have to access the disk or the persistent state within the physical machine of that Ethereum node, they cost much more than the memory instructions: sload costs 2100 Gas and sstore costs 20000 Gas units.

To set a slot a storage slot costs 20000 Gas. To change that storage value from zero to a non-zero value (and there are optimizations here) costs only 5000 Gas in some of the other situations. These Gas costs have changed over the duration of the last 5 to 6 years as Ethereum has evolved. These changes happen to prevent some denial of service attacks that have also happened in the past. This can be researched in the documentation by looking at some of the EIPs that have been created specifically to address the Gas cost of these instructions in some of the most recent upgrades (like the Berlin upgrade) to see why these costs Gas costs were changed for some of these more demanding instructions and the rationale behind it.

These become important because not only they address the optimization aspect

when somebody is deploying a contract (Gas usage becomes important because it affects the user experience of the user working or interfacing with these contracts) but from a security perspective (these Gas costs become important from the denial of service context as well)

The final set of instructions where the Gas costs are really high are the create instruction (which is probably the most expensive instruction with a cost of 32000 Gas units; and as you can imagine this is because create results in a new contract account being created, so a lot of the data structures within the EVM context are created, registered, have to be made persistent and so on...) and self-destruct (it costs 5000 Gas units).

1.14 Transaction Reverts and Data

Reverts

A transaction can revert for different exceptional conditions:

- The transaction could run out of Gas depending on how much was supplied as part of it and what that transaction actually needs when it is executing.
- The transaction could also revert because of invalid instructions that are encountered as part of executing the smart contract.

When the transaction gets reverted, all the state changes made in the context of the EVM so far from all the previous instructions in the contract are discarded, and the original state before the transaction started executing is restored. It is as if the transaction never executed from the perspective of the EVM state.

Data

Recall that the data field within a transaction is relevant when the recipient of said transaction is a contract account. In that case, the transaction data contains two components:

- It has to specify the function of that contract that is being invoked and...
- If that function requires any arguments, then it needs to specify those as well

All this is encoded according to the **Application Binary Interface (ABI)**: it's the contract's interface that's specified in a standard way, so that contracts can interact with each other. This is critical for contracts to interact both from outside the blockchain (when a transaction is triggered targeting a destination contract) but also for messages that are sent between two or more contracts

within the EVM context.

These interface functions that are specified as part of the ABI are strongly typed, are known at compilation time and they are static: the types of the function parameters are well known at compile time and they cannot change, because if they did, then what is specified as part of the contract call during execution will not reflect to what the destination contract requires in terms of the function encoding, or in terms of the arguments that are supplied.

So, **how does a callee contract specify the function to be invoked on the destination contract**?

It does that through the **function selector**.ç The way that it is specified is by taking the function signature (of the function that needs to be invoked), running that through a Keccak-256 hash and taking the first four bytes of the output hash: this is the function selector.

How is this function signature calculated from the function declaration?

The function name is taken and appended with the parenthesized list of the parameter types that it accepts. These parameter types are specified one after another, with the comma being the delimiter. Note that there are no spaces used (this is something that is enforced as part of the ABI and it's a standard, because if different contracts use different notations for function signatures, then you can imagine that when a transaction triggers a contract and sends the function selector, the receiving contract will not know which function to execute). So everyone has to know what the format is. This allows the EVM to function in a very deterministic manner.

Besides the function selector we have the function arguments that are also part of the transaction data (like we just discussed). These are encoded as well immediately following the four bytes of function selector: they span from the fifth byte onwards and go on depending on the number of arguments that the particular function needs.

1.15 Block Explorer

If we want to take a look at what has happened in the past in terms of the transactions on Ethereum, the contracts that they interacted with, then the application that allows us to look at all this data is what is known as a block explorer: it lets us **explore the various blocks and their contents on the blockchain**.

It's implemented as an application, a web portal if you will, and it gives us real-time on-chain data about all the transactions, the blocks, the Gas and

everything that we have discussed so far. All this rich information is available in a transparent manner on the blockchain and can be accessed by everyone via this block explorer application.

In the case of Ethereum we have several block explorers. The most popular one is Etherscan. We also have Etherchain, Ethplorer, Blockchair or Blockscout.

1.16 Mainnet

Mainnet refers to the main Ethereum network. There is a distinction because there also exist several testnets. These testnets are test Ethereum networks where protocol and smart contract developers can test their protocol upgrades and smart contracts prior to final deployment in mainnet.

While mainnet uses real Ether, testnets use what is known as "*test ether*", so that you can simulate the Gas, the transfer of value and so on... These test Ether can be obtained from faucets.

Some of the popular Ethereum test nets are Goerli (a proof of authority testnet that allows one to look at a lot of the Ethereum concepts and test them as if they are happening on mainnet). This particular test net works across all the clients. It's called proof of authority because there are a small number of nodes that are allowed to create the blocks and validate them.

Then we have the Kovan testnet, which is again a proof of authority testnet specifically for those running OpenEthereum clients.

We also have the Rinkeby testnet (also proof of authority based) which is specifically for the Geth clients.

Finally we have the Ropsten testnet, which is a proof of work test net (unlike all the other) which means that it's the best representation of mainnet Ethereum (which also uses proof of work). Thus a lot of the main net simulations can be done very closely to how it would behave on the mainnet.

These test nets are also evolving, new ones are being added over time trying to make it as easy as possible for the developers to simulate the real mainnet Ethereum blockchain and all its dependencies. This again becomes very critical to security because **testing is fundamental**: if you do not test, or if the testing environment is not very similar to the production environment, then the assumptions (the dependencies and other aspects that are tested) will be very different from what happens when you deploy the contract, which could end up causing a lot of security issues.

1.17 EIPs & ERCs

EIP stands for **Ethereum Improvement Proposal**: proposals put forward by researchers, developers and/or community members in the Ethereum ecosystem to make changes to different aspects of the Ethereum protocol.

There's a very well defined specific process for EIP from the time somebody proposes one to the way it is discussed, debated, voted upon and finally made it into a standard or a specification.

Depending on the different layers of the Ethereum protocol, the proposal targetting these could be either addressing the core aspects of the protocol, the networking aspects, the interface or some of the token standards.

ERC stands for **Ethereum Request for Comments**. It has (sort of) become the used term for token standards. For example you have probably heard about ERC20 token standard or ERC721 token standard and so on... These are being created as part of the EIP process.

There are also some meta and informational EIPs that don't address the protocol as such, but that address some of the governance aspects of this whole ecosystem, the process and so on... They also address some of the informational aspects of how these standards and specifications are written and distributed within the community.

1.18 Ethereum 2.0

Ethereum 2.0 is a set of interconnected upgrades to the existing Ethereum network that sre being made. The reason why it has been referred to as Eth 2.0 is because it is perhaps the biggest set of upgrades that are being made to the Ethereum protocol since it started. It is not a separate version of the protocol but a continuation of all the research and development activity that has happened on the protocol. The 2.0 signifies the fact that this has some of the biggest upgrades and those upgrades are across three vectors: **scalability**, **security** and **sustainability**.

- **Scalability**: made possible by the concept of sharding.
- **Security**: through the transition from PoW to proof of stake (PoS). This is again a huge change to how the protocol functions and it offers immense economic and security benefits compared to PoW.
- **Sustainability**: with PoW, there's a certain amount of computation that has to be done to pass the difficulty level. This is for sybil resistance part of the consensus protocol. As a result, there is real energy (in terms of running the mining nodes) that is consumed. This goes away to a great

extent when we transition to PoS: it is going to make Ethereum much more sustainable when it comes to environmental impact.

These upgrades already started happening with the deployment of the Beacon chain.

1.19 Legal Aspects in Web3: Pseudonymity & DAOs

When it comes to legal and regulatury aspects of **who is responsible**, **what** are they responsible for **if something goes wrong**, everything changes dramatically in the Web3 space compared to Web2.

One of the things is the **pseudonymity** or **who** is the team behind a particular project. There is an increasing trend towards some of the people involved in the projects being pseudonymous. This could be because of the regulatory uncertainty regarding cryptocurrencies (or crypto space in general), or also be because of the legal implications thereof.

This changes the way we think about reputation and trustworthiness when it comes to applications, projects or products. It also affects the legal or social accountability when it comes to projects: who is responsible, who is accountable if the team is pseudonymous, how do you even know what what they're doing with the project, with the governance and so on... There's this concept of trusting software and not wetware, which is great but there are still social processes where people are involved to a great extent around building the project, rolling it out and the governance of the projects that has a huge implication towards the security posture.

There is also the concept of **DAO**s (**Decentralized Autonomous Organizations**), which stem from the trust minimization aspect and the censorship resistance aspect of Web3. Their objective is to minimize the role and the influence of centralized parties, or a few privileged individuals, in the life cycle of the projects. This means that the project ultimately evolves or aspires to be governed by a DAO, which can be comprised of a **community of token holders for that particular project**. They make voting based decisions on how the project treasury should be spent, what the protocol changes should be and, in some of the cases, all these are decided on-chain and affected on-chain as well.

While this reduces the centralized points of wetware failure, as we call it, it also slows down decision making on a lot of the security critical aspects: imagine if there were vulnerabilities to be found in a deployed contract, and somebody had to create a fix and deploy the fix. If that had to go through a DAO for the decision making, you would have to give a certain amount of time for the token holders to vote for that decision. A centralized party entity in the Web2 space

can make this decision immediately, unilaterally and deploy that fix in a few hours, if not less. In the Web3 (i.e. DAOs), the decision making is decentralized and has that downside.

1.20 Security in Web3

Architectures, Languages & Tools: from Web1 to Web3

Going all the way back to the advent of internet 40 or 50 years ago, the various protocols that were developed as part of the TCP stack, some of the competing ones, the way they were standardized, then the advent of the world wide web that really launched the Web1 to the world, the concept of browsers, the concept of web applications, the client-server paradigm...

Then came the Web2: this is where the enterprises (be it IBM, Microsoft entered the picture), the introduction of Linux to the world, various hardware architectures, various operating systems, the dominance of Microsoft, Apple and more lately Facebook, Amazon and the likes... All these have contributed immensely to the development and the maturity of the Web2 ecosystem over the last 40 years.

This has huge implications to the security in that ecosystem as well, which has been developed in tandem with all those technologies over all those years: the firewalls, anti-viruses, intrusion detection systems, intrusion prevention systems, various kinds of security systems for email, for the world wide web, for your personal laptops...

They've evolved with the technology stack, with the various languages, with the various systems, the new use cases and so on... More lately, if you think about the entire ecosystem of smartphones, the apps around it, the advent of the iPhone, Android... They didn't exist 15 years ago and they entirely changed the way applications were built, deployed, distributed, the containerization withing those mobile devices and the security of those apps...

Now contrast that with Ethereum; with Web3 in general. Ethereum itself is not more than 6 to 8 years old protocol that got inspired from Bitcoin. Bitcoin itself is not more than 12 or 13 years old. This entire ecosystem, and specifically the technical stack of Ethereum (starting from the protocol and going to the EVM) has again taken a lot of inspiration from some simple architectures from the Web2 space, but has some very unique properties: like that of 256 bit words or more uniquely or the associated Gas semantics (which has no parallel in the Web2 world).

The same happens if you look at the languages that are used to write smart contracts, the developer tool chain that is critical to building deploying mon-

itoring applications on Ethereum (Hardhat, Truffle, Brownie, OpenZeppelin libraries...), they are barely 3 to 4 years old. There's an order of magnitude of difference with the Web2 world.

If you look at the security tools like `Slither` from Trail of Bits, `MythX` from ConsenSys Diligence and some of the others from OpenZppelin and other companies in the space; they're fantastic tools but they've been around for not more than 4 or 5 years. The test of time, evolution and adaptation of these tools to differing use cases, protocols and needs is very critical when you start thinking about implications to security, and all these are not happening in a very coordinated manner. They're all happening in different timelines by different teams around the world, often not very coordinated.

The Byzantine Threat Model

This is central to how security is thought about and critical to how security is designed. **Web3** is all about what is known as the **Byzantine Threat Model**, which is based around the byzantine generals problem.

Web2 has very defined concepts of **trusted insiders** and **untrusted outsiders**. Some of this has changed over the years because there is obviously a huge aspect of insider threat that has been recognized in the Web2 system as well. But if you look at the products and their evolution of in the Web2 security space, be it anti-viruses, firewalls or any of the network security (perimeter security devices and applications), there is still an aspect of insiders and outsiders.

This goes away to a great extent (if not completely) with **Web3** because in this case the threat model is really all about byzantine fault tolerance. This means that **anyone** (including the users) could become the **abusers of that system**. This is can be done in a very arbitrary malicious way, which is governed by the crypto economics (or what is known as mechanism design). It has obviously big implications to how security is designed and deployed because you have arbitrarily malicious adversaries that are motivated by mechanism design, and these adversaries could be users, intermediaries or people who are thought of as being critical to the ecosystem. They could include anyone: developers, miners, validators, infrastructure providers and users.

This is the main reason why in Web3, security aspects are challenging and it's the underpinning of web3 being untrusted by default, where the users could become the abusers. Web3 is the ultimate zero trust scenario.

Keys Vs. Passwords

Keys and tokens are very commonly used in terminology as well as the implementations of various protocols in the Web3.

For example we have the private keys that control the EOAs in Ethereum, which is all about the public key cryptography that is used in Web3. More specifically, in Ethereum, cryptographic keys are first class members of the Web3 world, and as much as we unknowingly use cryptography in the Web2 world, Web3 is taking this to everyone because the whole point is for the end users to take control of their assets (their tokens) with keys that are in their control, as there is no centralized entity that is responsible for them. At least aspirationally, the goal is for there to be no centralized intermediaries that can sit between you and your access credentials (your keys) or your assets (your tokens).

Let's contrast keys with passwords (that have become synonymous with the security) or what is wrong with security in the Web2 world. For several decades now, all of us have tens or hundreds of passwords. Most of them very simple and reused, and very few of the users really use password managers. But they rely on passwords being reset or changing them when they are lost by the entity that actually controls access to the website or to any service that is using these passwords.

That ideology of passwords is intentionally by design absent in the Web3 world, at least aspirationally. The goal is that in the future Web3 applications are headed towards this. The pathway to enable this is by the use of keys that are expected to be always under the control of the end user. So, loss of keys (or loss of the seed/secret phrases that generate those keys) is irreversible and there is no recourse or entity that you can go to and have them restored.

This is a significant shift in the security mindset coming from the Web2 world, where passwords again are ubiquitous and we see the problems with passwords being reused despite the use of QFAs, password databases being dumped and the various password replacing technologies such as biometrics still very slowly picking up adoption.

Web3 Tokens Vs. Web2 Financial Data

A similar situation exists with tokens and their data equivalent in the Web2 world: the data that we have on the various services, websites or even the financial assets (the financial data), if something happens to them (i.e. if they're stolen in some fashion; the worst thing that can happen is that the private personal data is maybe sold in the dark web and used to create accounts on your behalf or take loans for some monetary gain, which takes a certain bit of effort on the attacker's side because of the various checks and measures) there are technical and regulatory measures put in place for security.

In Web2 the implications of any data loss is indirect, takes time and effort from the attacker's perspective and in some cases, because of regulations or because of centralized entities, it can also be reversed. With tokens that used

in the Web3 space used by protocols (let's say the example of Ether or any of the cryptocurrencies), if they are taken away from the account that you control with your private key, then there's really no recourse unless these tokens happen to be in a centralized crypto exchange, or in the control of some other centralized parties that take the responsibility for any loss of such tokens.

The end user typically ends up losing those tokens irreversibly. These are again interrelated to the immutability aspects and trust minimization aspects of this whole space, which again contrasts between the fines, regulations and the possible reversals on the Web2 world.

1.21 Web2 Timescales Vs. Web3 Timescales

The timescale of innovation in Web2, although it is seemingly fast (exponential in some ways: smartphones in just 15 years, PCs, Moore's law...), those timescales are really long when you compare that to the compressed timescales of innovation that happen in the Web3 world and specifically Ethereum, which again is driven by a lot of these interrelated concepts we talked about: everything being open source by design, composable, permissionless and borderless. Plus combine that with the mechanism design where a lot of this is incentivized by tokenomics.

As a side effect, unfortunately, security has in some sense taken a back seat: it hasn't been really thought of as much as it should be in the design and development of a lot of these smart contracts (and hence, the applciations they support). This was what contributed to a lot of the vulnerabilities we have seen within smart contracts or Web3 applications which led to exploits causing losses of millions or tens of millions of dollars overnight in a fraction of a second within a few transactions.

And remember this is all irreversible: all these aspects of pseudonymous teams in some cases, the presenting threat model, the use of keys, the use of tokens, the lack of any centralized third party that can reverse the negative side effects of some of these exploits... All these interrelated concepts affect the security aspects of Ethereum and Web3 in general.

1.22 Test-in-Prod. SSLDC Vs. Audits

Test-in-prod is a concept that, although it may have started as a meme, has certainly an element of truth to it in the Web3 space and Ethereum. If you go back to the concepts of compressed time scales, unrestricted composability of contracts and applications in this space, the byzantine threat model and the challenges of replicating the full state of a live blockchain in a test setting; all these are really what make testing in a testing environment very hard.

Again, this contasts with the Web2 world where there are very clear distinctions between a test environment and a production setting for various reasons of owning the complete stack, the maturity of the tools, the lack of sort of unconstrained composability with arbitrary components outside of the stack for that particular product. All those aspects that are very well defined in the Web2 space from a testing perspective, are very challenging to set in the Web3 space. This is further complicated by the maturity of the tools that are still experimental in some sense in the Web3 space, and also the mechanism design aspect of it: the attackers and the users potentially being the abusers.

All these things come together to make testing, which is really fundamental when thinking about security and making something more secure or getting a better level of assurance from the product, very hard to do because the real world failure models cannot be replicated very easily in a test environment.

This implies that it forces "*realistic*" testing to happen only in production. In the case of Web3, in the case of Ethereum, on mainnet. So none of the testnets we talked about can match to some of the assumptions and the constraints that their software contracts will be subjected to. So, the complex technical exploits (i.e. crypto economic exploits), can only be discoverable upon production deployment on the mainnet. This is again a hypothesis, but it is worth thinking about.

An interesting concept to go through, is Web2's concept of **SSDL** which stands for **Secure Software Development Life Cycle**. There are many approaches to this, but in general, any Web2 product software, product hardware or product service has a version of SSDLC which is used during the development life cycle. This version guarantees that some minimum requirements have been met in a combination of testing, internal validation and some sort of external assessment depending on the product. It could be a product audit, a process audit, maybe even penetration testing if it is applicable to that product.

Also depending on the nature of assets that are managed, the risk that is faced, the threat model that is anticipated and even the specific sector or domain that the products are introduced in (such as he financial sector) there exist certifications assuring that the product application or service has to met to be succesfully deployedright. This is prevalent in the Web2 space and has evolved again over the last several decades.

However, when it comes to the **Web3** space, we do not see a mature SSDLC yet. What we see is this concept of **audits**, and unfortunately the life cycle of development has boiled down to building the product (be it a smart contract or a Web3 application), getting an audit done from an external company (a security firm that specializes in, let's say, smart contract security) and then going ahead and launching it.

There is an expectation both from the development team as well as the users (the market in general) to perceive this audit as a silver bullet: something that detects all the security issues in the smart contract, fixes everything and then sort of guarantees that the product is free of bugs and vulnerabilities when it is launched.

Audits are not a "*stamp of security approval*". There are some fundamental aspects that contribute to audits being perceived in this fashion (at least this is a hypothesis). The big one is in general the lack of in-house security expertise: given the rapid innovation time scales in the space, the developers are few and there's a huge demand for developers. There is even a bigger demand for people who not only understand how to develop in Ethereum and the Web3 space, but to understand the security pitfalls, which require a greater level of effort and expertise. This lack of in-house security expertise and the challenge of wanting and having to launch some of these protocols as fast as the team can, forces such teams to seek out external audit firms and get these audits, leading to think of them, market them and brand them as stamps for security approval.

So there's this very unrealistic expectations from audits to be "*catch-all*" for all the security vulnerabilities and bugs that are anticipated in a smart contract or in a Web3 application. For reasons of great demand and very low supply of this expertise, these audits are also very expensive: there are very few audit firms compared to the demand, which leads to a vicious loop where projects want audits but all the audit firms are really booked 6 or 9 months (depending on the market conditions). It is a core problem in the current space and state-of-art.

Chapter 2

Solidity 101: Introduction

As mentioned in the previous chapter, `Solidity` is currently the most popular programming language used to develop smart contract. In fact, it is so commonly used and there are so few alternatives to high-level languages on Ethereum, that it has become a fundamental pillar to smart contracts on Ethereum and therefore their security.

2.1 Solidity: Influence, Features and Layout

`Solidity` is a **high level language** specifically designed for writing smart contracts on Ethereum. It was proposed in 2014 by Gavin Wood and was later developed (and continues to be developed) by the Ethereum Foundation team led by Dr. Christian Reitwiessner, Alex Beregszsaszi and others.

It targets the underlying EVM and is mainly influenced by `C++` (a lot of the syntax and object oriented programming), a bit from `Python` (the use of modifiers, multiple inheritance, C3 linearization and the use of the `super` keyword) and some of the early motivation was also from `Javascript` (things like function level scoping or the use of `var` keyword, although those influences have significantly been reduced since version `0.4.0`).

One of the few alternatives to `Solidity` is `Vyper`: it's a language that is mostly based on `Python` and has just started to catch up with some of the high profile projects on Ethereum. However, to a great extent, due to the maturity of the language and the tool chains built around it, `Solidity` is by far the most widely used, so it becomes critical that in order to evaluate security of smart contracts we understand the syntax semantics, the pitfalls and various other aspects related to it.

`Solidity` is known as a "*curly bracket language*" (it means that curly brackets are used to group together statements within a particular scope), it is also an

object oriented language (so there exitsts the use of inheritance), statically typed (which means that the types of variables defined are static and defined at compile time), there is code modularity in the form of libraries and there are also user defined types.

All these characteristics make `Solidity` a fully featured high level language that allows the definition of complex logic in smart contracts to leverage all the underlying features of the EVM.

So, how does the physical layout of a smart contract written in `Solidity` look like?

This is important to the readability aspect of the file and the maintainability aspect of the smart contract in the context of the the project itself. A `Solidity` source file can contain an arbitrary number of various directives and primitives. These include the `pragma` and the `import` directives, the declarations of structs, enums and contract definitions. Every contract can itself contain structures, enums, state variables, events, errors, modifiers, constructor and various functions that define the various functionalities that are implemented by the smart contract.

This physical layout is something that is specific to the syntax of `Solidity`. When it comes to helping with the readability or the maintainability, it is prime to layout all the components in the order mentioned. This is something that you will commonly see when you evaluate smart contracts in `Solidity`. There might be cases where some of these are out of order from what is considered as best practice, but it's still something to keep in mind.

2.2 SPDX & Pragmas

SPDX

One of the things that you will often see specified at the top of every `Solidity` file is what is known as the SPDX license identifier. **SPDX** stands for **Software Package Data Exchange**. In the case of `Solidity` it's a comment that indicates its license and it is specified as a best practice to be at the top of every file. An example looks like this

```
1 // SPDX-License-Identifier: AGPLv3
```

Code 2.1: Example of SPDX comment.

The specific license obviously depends on what the developer intends for the particular smart contract. This identifier (i.e. the license) is included by the compiler in the byte code metadata that is generated, so it becomes machine readable.

Pragma

The `pragma` keyword in `Solidity` is used to enable certain compiler features or compiler checks. An example looks something like this

```
pragma solidity ^0.8.0;
```

Code 2.2: Example of pragma statement.

At a high level, there are two types of pragmas:

1. The first kind specifies the version. There are, again, two types of versions that can be specified:

   1.1. **The version** `pragma`, which indicates the specific `Solidity` compiler version that the developer expects to be used for that source file, and it looks like

   ```
   pragma solidity x.y.z;
   ```

   Code 2.3: Example of version `pragma` statement.

   where `x`, `y` and `z` are numerals that specify that compiler version. This does not change the version of the compiler used nor enables or disables any features of the compiler. All it does is instructing the compiler at compilation time to check whether its version matches the one specified by the developer. This could be of several formats: it could be a very **simple** format, a **complex** one or even a **floating** one (which has some security implications).

   The latest `Solidity` compiler versions as of now are in the `0.8` range with a different `z` in the `pragma` directive. If you look at `x.y.z`; a different `z` indicates bug fixes and a different `y` indicates breaking changes between the compiler versions. So if we have compiler versions in the `0.5` range, then by looking at the `0.6` range it means that the `0.6.z` range has at least one or more breaking changes compared to the previous versions.

   A floating `pragma` is a `pragma` that has a caret symbol (`^`) prefixed to `x.y.z` in the directive. This indicates that the contract can be compiled with versions starting with `x.y.z` all the way until `x.(y + 1).z`. So, as an example, consider

   ```
   pragma solidity ^0.8.3;
   ```

   Code 2.4: Example of floating `pragma` statement.

   It indicates that the source file can be compiled with any compiler version starting from `0.8.3` going to `0.8.4`, `0.8.5` and whatever else has been released; but not `0.9.z`, so the transition from `0.8` to `0.9` is what is prevented by this floating platform. This allows the

developer to specify a range of compiler versions that can be used with a particular contract, and that has some security implications similar to the floating `pragma`.

A range of compiler versions can be indicated with a complex practice, where you have `>`, `>=`, `<`, `<=` symbols that are used to combine multiple versions of the `Solidity` compiler. This affects the compiler version, which in turn brings in different features that are implemented by said version. Some of those could be security features, others could be security bug fixes or optimizations. All these aspects affect the security posture of the bytecode that is generated from a particular smart contract.

1.2. **The ABI coder** `pragma`. This directive allows a developer to specify the choice between **Version 1** or **Version 2** ABI coder.

The newer Version 2 was considered experimental for a while, but is now activated by default and allows the encoding/decoding of nested arrays and structs.

You might encounter old `Solidity` source code using the old directive, such as shown below

```
pragma experimental ABIEncoderV2;
```

Code 2.5: Example of old ABI coder `pragma` statement.

Version 2 is a strict superset of Version 1: contracts that use Version 2 can interact with other contracts that do not use it without any concern or limitations. This `pragma` also applies to the code defined in the file where it is activated, regardless of where that code ends up eventually; what this means is that a contract whose file is using Version 1 can still contain code that uses Version 2 by inheriting it from another contract. An example of ABI Coder `pragma` statement is

```
pragma abicoder v1; // or v2, which is the default from
    version 0.8.z onwards
```

Code 2.6: Example of ABI coder `pragma` statement.

The ABI coder affects encoding and decoding. The optimizations it does have certain security implications.

2. The second `pragma` directive helps the developer to specify features that are considered experimental as of that point in time.

These features are not enabled by default and have to be explicitly specified as part of this `pragma` directive and within every file where it is required. As of now there is only one experimental feature, which is known as `SMTChecker`.

```
pragma experimental SMTChecker;
```

Code 2.7: Example of experimental `pragma` statement.

SMT stands for **Satisfiability Modulo Theory** which is an approach to formal verification, and in the case of `Solidity` it is used to implement safety checks by what is known as querying an SMT solver.

There are various security checks performed by the SMT checker. The first one is where it uses the `require` and `assert` statements that are included as part of the smart contract. The checker considers all the required statements specified as assumptions by the developer and it tries to prove that the conditions inside the `assert` statements are `true`. If a failure can be established, then the checker provides what is known as a counter example that shows the user how this assertion can fail. There are various other checks that have been added to this empty checker over time. These include the arithmetic overflow, underflow, division by zero, unreachable code and so on.

So SMT checker is a critical security feature that comes packaged as part of `Solidity`. It's implemented in the compiler itself. Formal verification is considered as a fundamental part of programming languages' security, so we can imagine that this particular `pragma` directive affects the security and optimizations of the smart contracts that use them.

What needs to be kept in mind with the `pragma` directives is that they are local to the files where they are specified. So if you have a `Solidity` file that imports other files, the pragmas from the imported files do not automatically carry over to the file that is of concern.

2.3 Imports

These `import` statements are similar to `Javascript`, where the format is

```
import <filename>;
```

Code 2.8: Example of import statement

They help to modularize your code: split what might become a large monolithic code base into multiple components (modules) and import them wherever they are required.

This helps developers to reuse code and, again, not only it affects the readability of code (compare a piece of monolithic code that is hundreds or thousands of lines versus modular code, where they're separated out into independent self-contained modules and used only when required), it also has implications to security and optimization as well.

2.4 Comments & NatSpec

`Solidity` supports single line comments and multiline comments as shown here

```
// This is a single line comment
/* This is a
multiline
comment */
```

Code 2.9: Example of comments

Comments are recommended to be used as inline documentation of what the contracts are supposed to do, what the functions, variables, expressions, various control and data flow expected to do as per the specification and what is really implemented. They can also be used to specify certain assumptions that the developer is making in the implementation and they can also represent some of the invariants that need to be maintained.

Comments become a critical part of documentation that is included or encapsulated within the code itself, affect the readability of the code to a great extent and maintainability. In fact, comments become critical when we start talking about evaluating the security of smart contracts: comments give a lot of vital clues as to what the developer intended to implement or just information related to the various syntax or the semantics itself.

`Solidity` also supports a special type of comment called **NatSpec** which stands for **Ethereum Natural Language Specification Format**. These are specialized comments that are specific to `Solidity` and `Ethereum`. They are written as follows

```
/// This is a single line NatSpec comment
/** This is a
multi line NatSpec comment */
```

Code 2.10: Example of NatSpec comments

and located directly about the function declaration or statements that are relevant to the **NatSpec**. These NatSpec comments come in many different types: there are many different tags such as

- `@title`: describes the contract or the interface.
- `@author`: specifies the developer (i.e. who is authoring the contract).
- `@notice`: explains to an end user what the contract or function does.

- `@dev`: directed towards the developer for any extra implementation related details.

There are also specific tags related to function parameters (`@param`), the return variable (`@return`) and so on... These NatSpec comments are meant to automatically generate `JSON` documentation for both developers as well as users and provide a lot of valuable information that the developer intended for all these various aspects of parameters, returns, contracts and so on... They also form an important piece of the toolset that helps evaluate smart contract security.

2.5 Smart Contracts

Smart contracts (or simply contracts) are fundamental to what Ethereum is all about. Conceptually, contracts are very similar to the concept of classes in object oriented programming and that's because they encapsulate a state in the form of variables, and logic that allow to modify that state in the form of functions. These contracts can inherit from other contracts, they can interact with other contracts and support a very rich environment where one can specify different types of interactions between these components.

Contracts contain different components, including structures, enums, state variables, events, errors, modifiers, constructor and various functions. Some of these concepts should be familiar from other programming languages, but there are also some very Ethereum specific aspects, such as those related to state variables or events, or something that's specific to `Solidity` in the case of modifiers.

Contracts can come in different types: they could be either **vanilla contracts**, **libraries** or even **interfaces**.

2.6 State Variables: Definition, Visibility & Mutability

These are variables that can be accessed by all the contact functions. The data location where these state variables are stored is what is known as the **contract storage**.

Recall EVM has multiple components: the stack, calldata, volatile memory and the non-volatile storage. This non-volatile storage is where the state variables are stored because they need to persist across transactions that affect the contract state.

State Visibility

In `Solidity`, state variables have a concept known as visibility: who can see the state variables and who can access them. Visibility specifiers indicate this

property. There are three specifiers

- `public`: these state variables are part of the contract interface and they can be accessed either internally (from within the contract) or from outside the contract via messages. For such public state variables, an automatic getter function is generated by the compiler, which is used to access their values.
- `internal`: these state variables can be accessed only internally; from within the current contract or contracts deriving from this contract.
- `private`: these state variables can be accessed only from within the contract. They are defined at, and not even from the contracts that are derived from it.

Visibility specifiers are interesting from a security perspective because, although these seem to give an impression that certain state variables are private (in a sort of a privacy centric manner), everything that is within the contract is visible to all the observers external to the blockchain.

The `private` visibility specifier makes these variables private to the contract and prevents only other contracts from reading those private state variables on chain, however all the variables can be looked at can be queried via different interfaces.

State Mutability

State variables also have the concept of mutability. It indicates when can those state variables be modified and what are the rules for those modifications. There are two such specifiers

- `constant`: these state variables are fixed at compile, which means that their value is the same as when they were declared for the life of the contract. There are certain rules for what expressions can be used for defining these constant variables within the contract.
- `immutable`: these on the other hand are fixed at construction time, which means that they can be assigned values within the constructor of the contract or at the point of declaration. They cannot be read during construction time and they can only be assigned once.

The concept of mutability allows the `Solidity` compiler to prevent reserving any storage slot for these variables, making thesm storage and gas efficient: the gas cost of constant and immutable variables are lower.

The reason for this is because the expression that is assigned to it is copied to all the places where it is accessed within the contract and it's also re-evaluated

each time. This aspect allows the `Solidity` compiler to make some local optimizations wherever constant variables are used. And in the case of `immutable` state variables, they're evaluated only once at construction time and then their value is copied to all the places in the code where they are used. For these immutable variables, 32 bytes are reserved even if they require fewer bytes. Due to this, constant variables can sometimes be cheaper than immutable ones. For now the only supported types for these variables are strings and value types.

2.7 Functions

Functions are the executable units of code. In the case of `Solidity`, they are usually defined inside a smart contract, but they can also be defined outside of the contracts in which case they are specified at a file level. Such functions are referred to as "*free functions*".

Functions are what allow modifications to the state that is encapsulated as part of the contract, so they are how logic manifests itself within the smart contracts and the state transitions from one initial state to the modified state, as a result of any of the transactions or messages that interact with the smart contract.

Parameters

Functions typically specify parameters. These are declared just like variables within the function. Parameters are how the caller of the function sends in data into the function for it to work on. Parameters are used and assigned in a very similar manner to local variables within the function, and the nomenclature that the function specifies the parameter and the caller sends in arguments that get assigned to these parameters in the context of the function.

Return Variables

Functions typically also return values. These are returned using the `return` keyword. `Solidity` functions can return single variables or they can return multiple variables. The return variables can also be of 2 types:

- **Named return variables**: they have a specific name or names. They are treated just like local variables within the context of the function.
- **Unnamed return variables**: an explicit return statement needs to be used to return that variable a return value to the context of the function caller.

The caller specifies arguments that get assigned to the respective parameters of the callee function. The caller function works with these parameters (in the context of that function), does something with them along with all the local variables that might be defined within that function (it can also use the state

variables that are declared within that contract) and once it is done with that logic, it can return values back to the caller.

Modifiers

Function modifiers are something unique and specific to `Solidity`. They are declared using the `modifier` keyword and the format is something like this

```
modifier mod() {
    Checks;
    _;
}
```

Code 2.11: Modifier example.

As you can see, they are very similar to a function where, because modifiers have some logic encapsulated within them. The underscore acts as a placeholder for the function that we're attempting to modify; because **modifiers are used along with functions**.

So in this case if there is a function `foo()` on which this modifier is applied, then whenever this function is called, it goes first to the modifier and depending on any of the checks (any of the logic implemented within that modifier), the function's logic gets called at the point where the underscore is placed within that modifier.

So, if there are a bunch of checks in the modifier prior to the underscore, then those checks implement some preconditions that are evaluated before the function's logic is executed. Similarly, if the underscore precedes the checks in the modifier, the function's logic gets executed first and then the modifier executes its checks.

Examples for the usage here could be access control checks that are implemented as preconditions on the function in the modifier, and they could be post-conditions that could be evaluated if the underscore happens to be before the checks in the modifier, and these could implement some sort of accounting checks in the context of the contract.

Function modifiers play a critical role because they're very often used to implement access control checks, things that allow a contract to specify only certain addresses for example, to call the function where the modifier is applied... This is something that becomes critical when you evaluate the security of smart contracts.

Function Visibility

It is similar to the visibility for state variables functions. Functions have the 4 different visibility specifiers

- `public`: these functions are part of the contract interface and they can be called either internally (within the contract) or via messages.
- `external`: these functions are also part of the contract interface, which means they can be called from other contracts and via transactions, but they cannot be called internally.
- `internal`: these functions on the other hand can only be accessed internally (from within the current contract or contracts deriving from it).
- `private`: these functions can be accessed only from within the contract where they are defined and not even from the derived contracts.

Function Mutability

Similar to the state variable mutability, functions also have the concept of mutability. This affects what state can they read or modify. Depending on that there are two function mutability specifiers:

- `view`: these functions are allowed only to read the state but not modifying it. This is enforced at the EVM level using the `STATICCALL` opcode.

  There are various actions that are considered as state modifying that are not allowed for view functions, these include:

  - Writing to state variables (as should be obvious)
  - Emitting events
  - Creating other contracts
  - Using self-destruct
  - Sending ether to other contracts
  - Calling other functions not marked `view` or `pure`
  - Using low level calls
  - Using inline assembly that contain certain opcodes

- `pure`: these on the other hand are allowed to neither read contract state nor modify it.

  The not modification part can be enforced at the EVM, but the reading part cannot because there are no specific opcodes that allow that. There are various actions that are considered as reading from the state

  - Reading from state variables (obviously)
  - Accessing the balance of contracts
  - Accessing members of block

- Transactions or messages
- Calling other functions not marked as `pure`
- Using inline assembly that contain certain opcodes

The read/write mutability aspect of functions again has security implications as you can imagine.

Function Overloading

This is something fundamental to object oriented programming. It means that it supports multiple functions within a contract to have the same name but with different parameters or different parameter types. Overloaded functions are selected by matching the function declarations within the current scope to the arguments supplied in the function call, so depending on the number and the type of arguments the correct function is correctly chosen.

Note that return variables are not considered for the process of resolving overloading, so this notion of overloading is an interesting one that is supported by `Solidity` given that it is an object-oriented programming language.

Free Functions

They are functions that are defined at the file level (i.e. outside the scope of contracts) and thus these are different from the contract functions (defined within the scope of the contract). Free functions always have implicit `internal` visibility and their code is included in all the contracts that call them, similar to internal library functions. These functions are not very commonly seen.

2.8 Events

Events are an abstraction that are built on top of the EVM's logging functionality. Emitting events cause the arguments that are supplied to them to be stored in what is known as the transactions log. This log is a special data structure in the blockchain associated with the address of the specific contract that created the event. This log stays there as long as that block is accessible.

The log and its event data are not accessible from within the contracts, not even from the contract that created them. This is an interesting fact of logs in EVM: they're meant to be accessed off-chain and this is allowed using RPCs (Remote Procedure Calls). So applications, off-chain interfaces or monitoring tools can subscribe and listen to these events through the RPC interface of an Ethereum client.

From a security perspective, these events play a very significant role when it comes to auditing and logging for off-chain tools to know what the state of a

contract is and monitor the state along with all the transitions that happen due to the transactions.

Indexed Parameters in Events

Up to three parameters of every event can be specified as being indexed by using the `indexed` keyword. This causes those parameters to be stored in a special data structure known as topics instead of the data part of the log. Putting parameters into the topics part allows one to search and filter those topics in a very optimal manner. Parameters are commonly part of some of the specifications such as the `ERC20` token standard, and the events in that standard. These index parameters use a little more gas than the non-indexed one but they allow for faster search and query.

Event Emission

Events are triggered by using the `emit` keyword. Every contract would declare a certain set of events as relevant, and within the contract functions, wherever these events need to be created and stored in the log, they would be done so by using the `emit` keyword. An example would look like this

```
1 emit Deposit(msg.sender, _id, msg.value);
```

Code 2.12: Example of Event emission.

for the above example, let's say that we have a deposit event as part of a particular contract, and we have specific parts of functions where we would want to create this event and store them in the log.

So, following the example above, we specify the event plus the arguments that are required according to the parameters of the event. These look in some way very similar to a function call, where the event corresponds to the function and the arguments that are supplied to it correspond to the event parameters.

From a security perspective, it's critical for the contract and for the developers to emit the correct event and to use the correct parameters that are required by that event. This is something that is sometimes missed or not paid attention to because it's harder to be tested perhaps, and not critical to the control flow of the contract. But the only way for off chain entities, any kind of user interfaces or monitoring tools to keep track of the contract state and the transitions is by looking at these event parameters stored in the logs.

2.9 Structs

From a data structure perspective, structs are custom data structures that can group together several variables of the same or different types to create some-

thing very unique to the contract as required by the developer. The various members of the structs are accessed as follows

```
// Create a struct
struct Book {
   string title;
   string author;
   uint book_id;
}
// Fill in some info
Book my_book = Book("El Quixote", "Miguel de Cervantes", 1);
// Access a member
my_book.author
```

Code 2.13: Example of accesing struct members

```
"Miguel de Cervantes"
```

Code 2.14: Output of Code 2.13

Struct is aggregate type that is commonly used within `Solidity` to pack together custom data structures.

2.10 Enums

Enums are another user defined custom type in `Solidity`. They can be used to represent a finite set of constant integer values as represented by the members of the enum. Every enum needs to have a minimum of one member and can have a maximum of 256 members. As they represent the underlying integer values, they can be explicitly converted to and from integers. An example looks as follows

```
enum ActionChoices{GoLeft, GoRight};
ActionChoices choice = ActionChoices.GoRight;
```

Code 2.15: Example of enums

So here `choice` is a variable of `ActionChoices` and it can be assigned the members of `ActionChoices`. Here we are assigning `ActionChoices.GoRight` to `choice` and during the course of the contact function, different members can be assigned to that variable and it can be read from. This is used to improve readability, <so instead of using integer values one can use specific names that correspond to those integer values in the con<text of what makes sense from that co<ntract and its underlying logic.

2.11 Constructor

This concept is specific and unique to `Solidity` because it applies to smart contracts and the way they are created on Ethereum. Recall contracts on ethereum can be created from outside the blockchain via transactions, or from within the `Solidity` contracts themselves. When a contract is created, you can imagine

that one would want to initialize the contract state in some manner. This is made possible by the constructor. So the constructor is really a special function that gets triggered when a contract is created. A constructor is optional and there can be only one constructor for every contract. These special functions are specified by using the `constructor` keyword; some of the syntax and semantics have changed over the course of the solutions but this is how it has been in the most recent versions of `Solidity`

```
contract Base {
   uint data;
   constructor(uint _data) public {
      data = _data;
   }
}
```

Code 2.16: Example of usage of a constructor

So constructors are used to initialize the state of a contract when they are created and deployed on the blockchain. They're triggered when a contract is created and it's run only once. Once the constructor has finished executing, the final code of the contract is stored on the blockchain and this deployed code does not include the constructor code or any of the internal functions that are called from within the constructor.

From a security perspective, constructors are very interesting because they lets one examine what initializations are being done to the contact state because, if not the default values of the specific types of state variables. For example it could be used in the context of the various contract functions, which is an interesting and important aspect when it comes to evaluating the security of smart contracts.

2.12 Receive Function

Another special function in the context of `Solidity` is the `receive()`. This function gets triggered automatically whenever there is an ether transfer made to this contract via send or transfer primitives. It also gets triggered when a transaction targets the contract but with empty `CALLDATA`. Recall that a transaction that targets a contract specifies which function needs to be called in that contract and what arguments need to be used within the data portion of the transaction, but if that data is empty then the receive function is the function that gets automatically triggered in the contract.

There can only be one receive function for every contract and this function cannot have any arguments, it cannot return anything and it must also have external visibility and a payable state mutability. Payable state mutability is something we haven't discussed so far but what it specifies is that the function that has this `payable` specifier can receive ether as part of a transaction and that applies to the receive function as well because it is triggered when ether

transfers happen. The send and transfer primitives are designed in `Solidity` to transfer only 2300 gas. The rationale behind this was to prevent the risk, or mitigate the risk of what are known as "*reentrancy attacks*" which we'll talk more in the security module. This minimal amount of gas does not allow a `receive()` function to do anything much more than some basic logging (using events).

From a security context, `receive()` function becomes interesting to evaluate because it affects the ether balance of a contract and any assumptions in the contract logic that depends on the contract's ether balance.

2.13 Fallback Function

Another special function in `Solidity`. This is very similar to the `receive()`, there are some differences however. The `fallback()` function gets triggered automatically on a call to the contract if none of the functions in the contract match the function signature specified in the transaction. It also gets triggered if there was no data supplied at all in the transaction and there is no `receive()` function.

Similar to the receive function there can be only one `fallback()` function for every contract, however this `fallback()` function can receive and return data if required. The visibility is `external` and if the fallback function is meant to receive ether, then it needs to use the `payable` specified similar to the receive function. The `fallback()` function cannot assume that more than 2300 gas can be supplied to it because this can be triggered via the send or transfer primitives and, similar to the `receive` function, the security implications of the `fallback` function have to consider that the ether balance can be changed via this function, so any assumptions in the contract logic specific to the ether balance need to be examined.

2.14 Statically Typed

`Solidity` is a **statically typed** language, which means that the type of the variables used within the contracts written in `Solidity` need to be specified in the code explicitly at compile time. This applies to the state variables and also the local variables. Statically typed languages perform what is known as compile time type checking according to the language rules. So when variables of different types are assigned to each other at compile time, the language can enforce that the types are used correctly across all these assignments and usages. Many of the programming languages that you may be familiar with such as `C`, `C++`, `Java`, `Rust`, `Go` or `Scala` are statically typed languages. From a security perspective, the type checking is a critical part and helps in improving the security of the contracts.

2.15 Types

`Solidity` has two categories of types

1. `value`: they're always passed by value, which means that whenever they are used as function arguments or in assignments of expressions, they are always copied from one location to the other.

2. `reference`: they can be modified via multiple names all of which point to or reference the same underlying variable (i.e. the same memory address. This is easier to understand when it is thought like the concept of pointers).

From a security perspective you can imagine that this becomes important because it affects which state is being updated and what those transitions are in the states as affected by the transactions.

2.16 Value Type

As discussed value type is one of the two types in `Solidity` where variables of these value types are passed by value (which means they are copied when used as function arguments or in assignments of expressions). There are different value types in `Solidity`: booleans, integers, fixed point numbers, address, contract, fixed size byte arrays, literals, enums and functions themselves.

From a security perspective, value types can be thought of as being somewhat safer because a copy of that variable is made so that the original value of the original state itself is not modified accidentally. But then one should also check that any assumptions around the persistence of the values is being considered properly, so this will become clearer once we talk about the reference types and once we look at some of the security aspects.

2.17 Reference Type

In contrast to value types, reference types are passed by reference: there can be multiple names for the variable, all pointing to the same underlying variable state. There are 3 reference types in `Solidity`: arrays, structs and mappings. From a security perspective, reference types can perhaps be considered a little more riskier than value types because now you have multiple names pointing to the same underlying variable, which could, in some situations, lead to unintentional modification of the underlying state.

2.18 Default Values

Variables that are declared but not initialized have default values. In the case of `Solidity`, the default values of variables are what is known as a zero state

of that particular type. This means is that in the case of a boolean, it has a value of zero as a default which represents a value of false for the boolean. For unsigned integers or integer types, this is 0 (as expected). For statically sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. For dynamically sized arrays, bytes and string the default value is an empty array or string. For enum types, the default value is its first member.

From a security perspective this becomes important because variables that are declared and not initialized end up with these default values. In some cases, such as an address type, the zero address (which is a default value) has a special meaning in Ethereum, and that affects some of the security properties within the contract depending on how those address variables are used.

2.19 Scoping

This is fundamental to every programming language as it affects what is known as variable visibility, or in other words "*where can variables be used in relation to where they're declared.* In the case of `Solidity`, it uses the widely used scoping rules of `C99` standard.

So variables are visible from the point right after the declaration until the end of the smallest curly bracket block that contains that declaration. As an exception to this rule, variables declared in the initialization part of a `for` loop are only visible until the end of the loop. Variables that are parameters, like function parameters, modifier parameters or catch parameters are visible inside the code block that follows the body of the function (or modifier or catch). Other items declared outside of a code block such as functions, contracts, state variables or user defined types are visible even before they are declared. This means that we can see the usage of state variables even before they are declared within the context of a contract. This is what allows functions to be called recursively.

From a security perspective, understanding the scoping rules of `Solidity` becomes important when we are doing data flow analysis. This could be in the context of a manual review, where you're looking at the code yourself or when you're writing tools to do static analysis on `Solidity` smart contracts.

2.20 Boolean

Boolean types are declared using the `bool` keyword. They can have only two possible values: `true` or `false`.
There are five operators that can operate on boolean types: **the logical negation** or **not operator** (`!`), **the equality operator** (`==`), **inequality** (`!=`),

the **and** (`&&`) and **or** (`||`). The latter two operators are also known as logical conjunction and logical disjunction operators.
Operators apply the short circuiting rules. For example, in an expression that uses the logical disjunction or, operator if there are two booleans let's say `x` or `y`, if `x` evaluates to `true`, then the boolean `y` will not be evaluated at all even if it may have side effects. This is because the expression already evaluates to `true` and there's no need for the second boolean to be evaluated at all and similarly this applies to the **and** operator logical conjunction as well. So if there are two booleans that have this operator, let's say `x` and `y`, and if `x` happens to be `false`, then we know that the expression finally will evaluate to `false`, so there is no reason for the compiler to evaluate, because the result is already known to be `false` from a security perspective.
Booleans are used significantly in smart contract functions for various conditionals and evaluations of expressions. **This affects the control flow and specifically when it comes to certain checks access control checks**. *(History).* There have been cases where booleans have been used, and the wrong operator has been used in those checks. So for example using the **not** or logical disjunction instead of logical conjunction. It can have big implications to how that particular expression evaluates and that check, the access control check or whatever that might be, might not be effective at all as intended by the specification. So this is again something to pay attention to when you're looking at booleans and the operators that evaluate that operate on the booleans in smart contracts.

2.21 Integers

Integer types are very common in `Solidity` and any programming language. There are unsigned and signed integers of various sizes. In `Solidity` they use the `uint` or `int` keywords. They come in **sizes from 8 bits all the way to the word size of 256 bits**.

`uint`/`int` keywords sizes: 8 → 256 Bits

So you'll see declarations of unsigned integers or integers signed intgers in the form of `uint8` all the way to 256.

```
uint8, uint16, ... uint256
int8, int16, ... int256
```

Code 2.17: Examples of different **int** and **uint** sizes.

There are various operators for integer types. There are different categories that we saw in the EVM instruction set: **arithmetic operators**, **comparative operators**, **bit operators** and **shift operators**. From a security perspective, given that integer variables are vastly used in `Solidity` contracts, they **affect the data flow of the contract logic** and specifically there is an aspect of integers that becomes **security critical** which is that of **underflow and overflow**.

2.22 Integer Arithmetic

Integer arithmetic is arithmetic that operates on integer operands, signed integer operands or unsigned integers operands `Solidity`. Like in any other language, they are really restricted to a certain range of values, so for example if you have 256, then the range of that variable is from a value of 0 to $2^{256}-1$.

If there is any operation on a variable of, let's say `uint256` type that forces it to go beyond this range, then it leads to what is known as an overflow or an underflow. This causes wrapping. In the case of `uint256` (let's say that the value of one of those `uint256` variables was the maximum value), then if the contract logic incremented it by 1 more, then that integer value would overflow: it would wrap to the other side of the range and would become 0. Similarly an underflow, let's say in the case of the value was 0, if the logic decremented it by one more, then it would again cause wrapping to the other end of the range and the value of that variable would now be $2^{256}-1$. This can have significant unintended side effects when it comes to the integer values used in that logic.

There have been numerous cases of certain integer values being overflowed or underfloored, leading to huge exploits vulnerabilities from a security perspective. This is something that is really really critical when it comes to the security of integers, basically in the smart contract.

To address this specific aspect in versions of `Solidity` below `0.8.0`, **the best practice was to use the safe math libraries from OpenZeppelin** that made operating on integer variables safe with respect to overflows and underflows. `Solidity` itself as a language recognized this aspect of security and introduced in version 0.8.0 default overflow and underflow checks for integers.

In contracts that are written with the compiler version `0.8.0` and above, one can actually switch between the default checked arithmetic (that checks for underflows and overflows and causes exceptions when that happens) versus unchecked arithmetic (where the programmer or the developer asserts that for the expressions used in that unchecked arithmetic there is no way or no cause for concern when it comes to overflows and underflows), so all the default underlying checks in the language in the compiler itself should be disabled.

This is something to be paid attention as it is a critical aspect of smart contact security. When looking at smart contracts, pay attention to the solution compiler version that was used: if it is below `0.8.0`, then there should be the use of safe map from OpenZeppelin, or some of the other equivalents that make sure that the integers don't overflow and underflow and cause security vulnerabilities. If the compiler version is `0.8.0` or beyond, then one should pay attention to any expressions, integer expressions, that are using unchecked blocks to make sure that those don't have any overflows or underflows.

2.23 Fixed Point Arithmetic

Conceptually you would have seen this in other languages, as well for numbers that have an integer part and a fractional part, the location or the position of the decimal point indicates if it is fixed or floating. If that position or location of the decimal point can change for that type then it is referred to as a floating point type.

But if that position is fixed for all variables of that tipe, then it is known as fixed point arithmetic. In the case of `Solidity`, these can be declared but cannot be assigned. There's no real support in `Solidity`. **For any use of fixed point arithmetic**, one has to **depend on some of the libraries such as** `DSMath`, `PRBMath`, `ABDKMath64x64` or others.

2.24 Address

It's a type that is specific to `Solidity` and Ethereum, and it is critical to security: the `address` type. It refers to the underlying Ethereum account address, the EOA or the contract account. This is different from the addresses that you might have encountered in other programming languages such as `C` and `C++`, where they refer to variables' memory address when you're dealing with pointers or references. Here address signifies something very different: an account address.

The `address` types are 20 bytes in size, because remember that is the size of the Ethereum address. They come in two types they can be **plain** `address` types or they can have a `payable` **specifier, and referred to as an** `address payable` **type**, where it indicates that this `address` type can receive Ether. There are different operators that operate on address types, such as shown here

- Operators `==`, `!=`, `<`, `<=`, `>` and `>=`
- Implicit/Explicit Conversions

There are conversions that can be performed on `address` types. Some of them are implicit and others are explicit. For converting `address payable` types to `address` types, implicit conversions can be used because it is safe. Whereas the other way around, where an `address` type is converted to an `address payable` type, that should be an explicit conversion because now this address becomes capable of receiving Ether.

From a security perspective, `address` types play a critical role in contracts. These addresses are used in different types of access control: some may be considered as more privileged than the others in the context of the contract logic. Addresses can also hold Ether balances and token balances, so using

the right addresses in the right places and making sure that the correct access control logic or the balances accounting logic is applied on them, becomes very critical from a security perspective to make sure there are no undefined behavior or unintended side effects leading to security vulnerabilities.

2.25 Address Members

Address types have different members that can give different aspects of the underlying `address` type. There is the `balance`, that gives you (as the name may suggest) the balance of that address in wei. There is the `code` that gives you (surprise...) the code of that address. There is `code hash` that gives you the hash of the code..

There are also the `transfer` and `send` members that **are applicable to the** `address payable` **types**. They make calls to the addresses that are specified by supplying a limited gas stipend of only 2300 gas units. This is not adjustable: it is something hard coded in `Solidity` to address the category of reentrancy attacks on addresses. We'll take a look at the reentrancy aspect later on in the security modules, but it's something to keep in mind for now.

There are also the following members: `call`, `delegate call` and `static call` that can be applied on `address` types. These are used to make low level calls to their specific address that is specified. We talked about some of these calls in the context of the underlying instructions, `call`, `delegatecall`, and `staticcall` instructions, where we talked about how the callee account in the case of `delegatecall` executes with its logic but with the state of the caller account. Similarly in the case of `staticcall` we talked about how the callee contract address can access the state but cannot modify the state.

So as you can imagine, these `address` members play a huge role when it comes to the security aspects, because they deal with the balances, look at the code, the code hash, the reentrancy aspects of `send` and `transfer`, making external calls using `call`, `delegatecall`, and `staticcall`, which are critical when it comes to the trustworthiness of the contracts that are being called at these addresses.

2.26 Transfer

The transfer primitive can be applied on addresses. The `transfer` function is used for transferring Ether to the destination address. This transfer triggers the `receive` function, or the `fallback` function of the target contract. This comes by default supplied with 2300 gas subsidy (remember, it is a fixed amount cannot be changed) and if the `fallback` functions of the target contract uses

more than 2300 gas, then the transaction fails: it reverts and an exception is executed by design.

From a security perspective this primitive affects reentrancy attacks: where the target contract, if it is untrusted, could potentially call back into the caller contract and lead to undesired behavior that could affect token balances or other contract logic in in a very critical way. The 2300 gas assumption is critical when you look at how contracts use transfer and whether that transaction could fail and revert and lead to undefined behavior.

2.27 Send

Similar to `transfer`, there is `send` function in `Solidity` which is somewhat a lower level counterpart for `transfer`. It is used for Ether transfers: it triggers the same receiver `fallback` functions like `transfer`, it also has a 2300 gas subsidy, but **it does not result in a failure if the target contract uses more than 2300 gas unlike transfer**. In the case of `send`, it does not revert but it just sends back a boolean return value that indicates a failure or a false boolean value in the case of failure.

So if the `send` primitive is used to transfer value, then from a security perspective it means that the return value of that `send` primitive must be checked by the caller to make sure that the transfer happened successfully or not, depending on what was returned. Again, the `send` primitive affects reentrancy, which is again why the `send` primitive was introduced as a mitigation in `Solidity`. There is also the 2300 gas subsidy aspect that affects security, and finally the return value check that is critical and and different from its `transfer` counterpart.

2.28 External Calls

There are really primitives that we touched upon earlier: `call`, `delegatecall` and `staticcall`. These are used to interface with contracts that do not adhere to the ABI or where the developer wants more direct control over such calls. They all take single bytes memory parameter, return the success condition as a boolean and return data in a bytes memory. They can also use `abi.*` functions, such as `encode`, `codepath`, encode with selector, encoded signature... to encode structured data as part of the arguments. They can also use gas and value modifiers to specify the amount of gas and Ether for these calls. The latter is applicable for the `call` primitive but not `delegatecall` or `staticcall`.

To summarize: the `delegatecall` is used where the caller contract wants to use the logic specified by the callee contract but with the state and other aspects of the caller contract itself. So while the code of the given address is used, all other aspects such as `storage`, `balance`, `message`, `sender` are taken

from the current caller contract. The purpose of `delegatecall` is to enable use cases such as libraries or proxy upgradability, where the logic code is stored in the callee contract but that operates on the state of the caller contract. `staticcall` is used where we want the called function in the callee contract to look at or to read the state of the caller contract, but not modify it in any way.

The use of external calls have different types of security implications, these are low level calls that should be avoided in most cases unless absolutely required and there are no alternatives available, because these are calling out to external contracts that may be untrusted, in the context of the current applications threat model or trust model. These external contracts could result in undefined behavior, use more gas than expected, cause re-entrances to the caller contract and might also return failures where if the return value is not checked, could result in undefined behavior as well.

2.29 Contract Type

Every contract that's declared is its own type and these contract types **can be explicitly converted to and from address types**. That is what they represent underneath. These contract types do not have any operators supported, and the only members of these types are external functions declared in the contract along with any state variables.

2.30 Bytes Arrays

Bytes array type are used to store arrays of raw bytes. There are two kinds here: if we know what the size of the byte array is going to be in advance, then we can use what are known as fixed size bytes arrays: they come in 32 kinds, `bytes1` for storing 1 byte all the way to `bytes32` for storing 32 bytes, which is the full word size in the context of EVM.

If we do not know the fixed size in advance then we can use the alternative, which is indicated by `byte[]`, but due to padding rules of EVM it wastes 31 bytes of space for every element that is stored in it. So if we have a choice, then it's better to use the bytes type instead of the byte type for these byte arrays. This is something that you will commonly come across in smart contracts for storing raw bytes example in case of hashes.

2.31 Literals

This is something that you would have come across in other programming languages, as well `Solidity` supports five types of literals: **address types**,

rational/integers, **strings**, **unicode** and **hexadecimals**.

The address literals are hexadecimal literals that pass the address checksum test. Remember that Ethereum addresses are 20 bytes in length, so in the case of the hexadecimal address representation, half a byte is represented by a hexadecimal character. This results in the address literal having 40 characters: 2 for every byte. These should pass the checksum test. The checksum is something that has been introduced in EIP55 to make sure that there are no typographical errors when you're using addresses in the context of Ethereum. This is a mixed case addressed exam.

Rational literals and integer literals are also supported. Integer literals have a sequence of numbers in the 0 to 9 range. Decimal fraction literals are formed by using a decimal point, with at least one number in each side. Scientific notation is supported where the base can have fractions and the exponent cannot. Underscores can be used to separate these digits, which is used to help with readability and does not have any semantic significance. String literals are written with either double quotes (`""`) or single quotes (''). They can only contain printable `ASCII` characters and a set of escape characters. Unicode literals they have to be prefixed with the keyword `unicode`. They can contain any `utf-8` sequence. The hexadecimal literals are hexadecimal digits prefixed with the keyword `"hex"` or `'hex'`. The usage of all these literals is in the context of constants.

2.32 Enums

Enums are a way to create user defined types in `Solidity`. They can have members anywhere: from 1 member all the way to a maximum of 256 members, and the default value of an `enum` is that of the first member. This is something that you see sometimes in smart contracts where **`enums` are used to represent the names of the various states** within the context of the contract logic or the transitions in some cases. This is something that helps to **improve readability** instead of using the underlying integers that the `enums` really represent.

2.33 Function Types

Function types are types used to indicate that variables represent actual functions. These variables can be used just like any other variables: they can be assigned from functions because they are of the function type, and they can be sent as arguments to other functions and can also be used to return values from other functions.

They come in two types: `internal` and `external`. Internal functions can only be called inside the current contract. External functions consist of an address of the contract where they're relevant and a function signature along with it. They can be passed and returned from external function calls. The usage of function types is somewhat minimal in most of the common smart contracts.

2.34 Data Location

We talked about value types and reference types. Reference types, which consist of structs, arrays and mappings in `Solidity` allow for a specification of their data location. This is an additional annotation and it indicates where that reference type variable is stored. There are really three locations: **memory**, **storage** and **calle data**. Remember that these are 3 of the 4 locations that the EVM supports besides the stack. These data locations affect the lifetime or the scope and persistence of the variables stored in those locations.

- **Memory** indicates that the lifetime is limited to that external function call.

- **Storage** indicates that the lifetime extends to that whole contract and this is also the location where state variables are stored.

- **Call data** is a non-modifiable and non-persistent area where function arguments are stored. This is required for parameters of external functions but can also be used for other variables. This data location annotation impacts the scope of the variables that use this lotation. From a security perspective this affects the persistence of those variables.

2.35 Data Location & Assignments

The data location annotation we just talked about not only **affects the persistency of those variables**, the scope in which they are relevant, but it also affects what are known as **assignment semantics**. In the context of `Solidity`, what this means is that during an assignment, using such variables is a copy of that variable being created? Or is simply a reference being created to the existing variable? In `Solidity`, storage to memory assignments always create an independent copy. Memory to memory assignments only create references. Similarly storage to storage assignments only create a reference. All other variants, create a copy.

From a security perspective how this impacts the semantics is: if a copy were to be created because of these assignment rules, then any modifications to the copy affect only the copy and not the original variable from where it was copied.

On the other hand, if a reference was created, in the case of memory to memory assignments or storage to storage assignments, then the new variable modifications to that affect the original variable because both of them are just different names pointing to the same underlying variable data (the same memory address on the machine). So this becomes important when you analyze programs and notice what the data locations are for those reference types, because there's a big difference if modifications are being made to the copy versus a reference.

2.36 Arrays

Array types are something that are very common in most programming languages, in the case of `Solidity` they come in two types: they can be `static arrays` **where the size of the array is known at compile time** or **they can have a dynamic size**.

They are formally represented as `T[k]` (a static array of size `k`) or `T[]` (a dynamical array). The elements of these arrays can be of any type that is supported by `Solidity`. The indices that are used with these arrays are 0 based (the first array element is stored at `T[0]` and not `T[1]`.

If these arrays are accessed by the logic past their length, then `Solidity` automatically reverts that access and creates an exception, which causes a failing assertion, In the context of the contract doing such an access. From a security perspective, arrays are very commonly used in smart contracts, **so the things to pay attention to are to check if the correct index is being used especially in the context of indices being zero based and to check if arrays have an off by 1 error, where they're being accessed either beyond or below their supported indices**, in which case such an **access could lead to an exception and the transaction would revert.**
The other aspect to keep in mind with arrays is if the length of the array that is being accessed is really long and if the types are complicated underneath, then the amount of gas that is used for the processing of such arrays could end up in what is known as a denial of service attack(DoS) where those transactions revert because not enough gas can be supplied as part of the transaction so you would end up with no processing really happening because a transaction would revert.

2.37 Array Members

The members that are supported for array types there are four of them: there is the `length`, `push(x)` and `pop`.

- `length` returns the number of elements in the array.
- `push()` appends a 0 initialized element at the end of the array and it returns a reference to that element

- `push(x)` appends the specified element `x` to the end of the array and it returns nothing.
- `pop` on the other hand removes an element from the end of the array and implicitly calls delete on that remote element.

2.38 bytes & string

Bytes are used to stir arbitrary byte data of arbitrary length. Remember that **if we know beforehand the size of the byte array**, then we can **use the fixed size byte arrays** to store those number of bytes. But if you do not know what the size is beforehand, then we can use the `bytes` type, and even there we have a choice of bytes or the byte array we talked about earlier. Remember that the byte array uses 31 bytes of padding for every element stored and leads to waste of that space so **it's preferable to use bytes over the byte array**.

String type is equivalent to the byte style except that it does not allow accessing the length of the string and the index of a particular byte in that string, so it does not have those members. `Solidity` **does not yet have inbuilt string manipulation functions but there are third party string libraries** that one can use.

2.39 Memory Arrays

Memory arrays are arrays that are created in memory, they can have dynamic length and can be created using the `new` operator. But as opposed to storage arrays, **it's not possible to resize them**. So the `push()` **member functions are not available** for such memory arrays. So the options are for the developer to either **calculate the required size in advance** and use that appropriately during the creation of these arrays, **or create a new memory array and copy every element of the older memory array into the new one** an example is shown here.

```
1 uint[] memory a = new uint[](7);
```

Code 2.18: Example of memory array definition.

2.40 Array Literals

They are another type that is supported by `Solidity`. They are a comma separated list of one or more expressions, enclosed in square brackets (which is how arrays are represented in `Solidity`). These are always statically sized memory arrays, whose length is the number of expressions used within them. The base type of the array is the type of the first expression of that list, such that all other expressions can be converted to the first expression. If that is

not possible then it is a type error indicated by `Solidity`. Fixed size memory arrays cannot be assigned to dynamically sized memory arrays within `Solidity`, so these are some aspects to be kept in mind when evaluating contracts that have array literals.

2.41 Array Gas Costs

Arrays have `push` and `pop` operations. Increasing the length of a storage array by calling `push`, has constant Gas cost because storage is zero initialized. Whereas if you use `pop` on such arrays to decrease their length, the Gas cost associated with that operation depends on the size of the element being removed. If the element being removed happens to be an entire array, then it can be very costly because it includes explicitly clearing the removed elements, which is similar to calling `delete` on each one of them.

2.42 Array Slices

`Solidity` supports the notion array slices. Array slices are views that are supported on contiguous array portions of existing arrays. They are not a separate type in `Solidity`, but they can be used in intermediate expressions to extract useful portions of existing arrays as required by the logic within the smart contracts. These are written as

```
X[start:end]
/** This expression takes the array from element X[start]
up to element X[end-1]
*/
```

Code 2.19: Example of array slicing.

From an error checking perspective if `start > end` or if `end > n` (where `n` is the size of the array) then an exception is thrown. Both these `start` and `end` values are optional, where `start` defaults to `0` and `end` defaults to the length of the array `n`. Array slicers do not have any members that are supported, and for now `Solidity` only supports array slices for call data arrays.

2.43 Struct Types

Struct types and Solidity are another user defined type: they let the developer combine different variables of value or reference types into one unit and treat them together in the contract logic. So these are used extensively within smart contracts, they're very commonly encountered.

Some of the properties of `struct` types are that they can be used inside mappings, arrays and they themselves can contain mappings and arrays. All these different complex reference types can be used in a very interrelated manner and

allows for a versatile usage of these data structures to support different kinds of encapsulation logic when it comes to the different data types within a smart contract. There's one exception: `struct` types cannot contain members of the same `struct` type.

2.44 Mapping Types

It is an interesting reference type somewhat unique to `Solidity`. Mapping types define `(key, value)` pairs, they're declared using the following syntax

```
mapping(_key => _value) _Var
```

Code 2.20: Example of mapping definition

The `key` type in a `mapping` can be really any built-in value type: byte, string or any contract or enum type even. Other user defined or complex types, such as `mapping structs` or `array` types are not allowed to be used as the key type, so there are some restrictions here. On the other hand, the `value` type of that `(key, value)` pair, can be any type including mappings, arrays and structs. There are some interesting aspects of how mappings are created and maintained by `Solidity`: the `key` data is not stored in the `mapping`, it is only used to look up the value by taking a Keccak-256 hash of that `key` data. They also do not have a concept of length nor a concept of a `key` or `value` being set in the mapping. They can only have a storage data location, so they are only allowed for state variables. They cannot be used as parameters or return values of contact functions that are publicly visible.

These restrictions are also true for arrays and structs that contain mappings, not just mappings themselves. Also one cannot iterate over the mappings, you cannot enumerate their keys and get the resulting values. This is not supported by default but it is possible where required by implementing another data structure on top of mappings and iterating over them. So very versatile type in `Solidity` again, very commonly encountered in smart contracts to store associations between different data structures that are used in that contract logic.

2.45 Shorthand Operators

These are concise notations of slightly longer expressions as shown here

```
Long expression } Shorthand notation
----------------+--------------------
a = a + e       } a += e
a = a - e       } a -= e
a = a*e         } a *= e
a = a/e         } a /= e
a = a%e         } a %= e
a = a}e         } a }= e
a = a&e         } a &= e
```

```
10 a = a^e          } a ^= e
```

Code 2.21: Example of shorthand operators

Basically it consists on simplifying the expression of increments and decrements, where the result of the expression is really the value of a after the increment or decrement has been performed.

2.46 Delete

The `delete` keyword that can be used within smart contracts to reclaim the underlying storage of a variable when it is no longer required in in that context of the contract. Applying this keyword on a variable `a`, of a particular type, assigns the initial value for that type to `a`. So if it is applied on integers, then the value of that variable is set to 0, for arrays it assigns a length of 0. For dynamic arrays and for static arrays the length remains the same but all the elements are set to their initial value.

`deleta A[x]` where `A` is an array and `x` specifies a particular index, deletes the item at that index of that array and leaves all the other elements and even the length of that array intact. For structs, `delete` assigns a `struct` with all the members reset to their initial values. Delete has no effect on mappings, this is an exception that has to be paid attention to. So if you `delete` a struct which in turn has a mapping as one of its fields, then `delete` will reset all the members of that struct that are not mappings and will also recurse into each of those members unless they are mappings. But if you want to `delete` a particular `key` of that `mapping` then that is possible.

2.47 Implicit Conversions

Every programming language that supports different types supports the concept of conversions, where variables of different types can be converted between each other.

There are two types of these conversions, some of them can happen **implicitly** where that conversion is applied by the compiler itself. These typically happen where that conversion makes sense semantically and there is no information that is lost, so this is a very safe conversion applied by the compiler. Such conversions happen during assignments of variables when variables are passed as arguments to functions, and the parameter types of those functions are of a different type than the arguments applied (and in other contexts as well).

Examples of implicit conversions in the case of `Solidity` are converting a `uint8` to `uint16` or `uint128` to `uint256` and so on, where the resulting type is bigger in the sense of the storage supported than the type that is being converted

from. So `uint16` has 16 bits that can safely store `uint8`. However exceptions to implicit conversions are converting from signed integers to unsigned integers, and that doesn't make semantic sense because unsigned integers cannot hold or represent negative values.

2.48 Explicit Conversions

The flip side of implicit conversion are **explicit** conversions, where the type conversions are explicitly applied by the developers themselves and not by the compiler. The reason for that is the compiler cannot deduce or prove the type safety of such conversions and they may result in an unexpected behavior.

There are various rules to such explicit conversions: in the case of integers when they are converted to a smaller type, the higher order bits are cut off when they are converted to a larger type they are padded on the left with the higher order end. So these apply for example when a `uint8` is converted to `uint16` the padding happens on to the left and when a `uint16` is converted to `uint8`, the higher order bits are cut off. Similarly for fixed size bytes, the `bytes` arrays or `bytes1` all the way to `bytes32`, converting to a smaller type cuts off bytes to the right and converting to a larger type will pad bytes to the right.

So these rules are something that the developer has to pay attention to when forcing explicit conversions and if not done right, they could really result in undefined unexpected behavior, because the values underlying variables are chopped off in an unexpected fashion.

2.49 Conversions Literals

There are various rules that apply to these conversions. Decimals and hexadecimal number literals can be converted implicitly to any integer type that's large enough to represent it without getting it truncated. However decimal number literals cannot be implicitly converted to fixed size byte arrays.

Hexadecimal number literals can be converted to fixed size byte arrays but only if the number of hex digits fits the size of the bytes type exactly, although there are some exceptions to this. As well string literals and hex string literals can be implicitly converted to fixed size bite arrays, but only if the number of characters matches the size of the `bytes` type. So these are various `Solidity` rules that need to be considered while converting literals and again it's something that you might encounter while analyzing smart contracts.

2.50 Ether Units

Ether is 18 decimals, the smallest unit is a wei. There are various names given for different numbers of weis: 1 gwei = 10^9 wei, 1 ether = 10^{18}.

In the case of the `Solidity` types, a literal number can be given a suffix of a wei, or a gwei (gigawei) or an ether. These are used to specify sub denominations of ether, as we see here, which are used when contracts want to manipulate different denominations of ether in the context of the logic.

2.51 Time Units

As you can imagine contracts might want to work with different notions of time for various types of logic that they want to encode. `Solidity` supports different suffixes that represent time, and these can be applied to literal numbers and these suffixes are: seconds, minutes, hours, days and weeks.

The base unit for time is seconds, so literally when 1 is used it is the same as representing 1 seconds. The suffixes cannot be directly applied onto variables, so if you want to apply time units to certain variables, then one needs to multiply that variable with that time unit. So as an example shown, if we have a `daysafter` variable and we wanted to represent the number of days then we have to proceed like follows: `daysafter * 1 days`. That is really how Solidity allows one to use these units with variables.

2.52 Block & Transaction Properties

`Solidity` allows accessing various block and transaction properties within smart contracts. These allow developers to perform interesting logic that are dependent on different aspects of the current block or the transaction. So in the case of the block we have the **block hash**, that gives the hash of the specified block, but only works for the most recent 256 ones, otherwise it returns zero. There is **block chain id** which gives the current id of the chain that this is executing on. There is the **block number** which is the sequence number of the block within the blockchain. There is the **block timestamp**, which is the number of seconds since the unix epoch.

There is also the **coinbase address**, which is controlled by the miner and it's the beneficiary address where the block rewards and transaction fees go to. There is a **block difficulty** related to the proof of work. There's a **Gas limit** related to the block.

Additionally, yhere are also fields related to the message, the **message value**, representing the amount of ether that was sent as part of this transaction, **message data**, which gives access to the complete call data sent in this

transaction, **message sender**, which is the center of the current call or the message, and the **message signature**, which is the function identifier or the first four bytes of the call data representing the function selector that we talked about earlier.

Then there are transaction components of **Gas price** used in this transaction, the amount of **Gas left** in this transaction after all the competition so far, and also the **transaction origin**, which is the sender of the transaction, representing the EOA.

2.53 Message Values

Message value represents the amount of ether sent in way as part of the transaction. Message sender is the sender's address. The thing to be kept in mind is that every external call made, changes the sender. Every external call made can also change the message value.

So if we have three contracts `a`, `b` and `c` where `a` calls `b` and `b` calls `c`, in the context of the contract `b` the message sender is `a`, but in the context of the contract `c` the message sender is contract `b` and not `a`.

These aspects should be kept in mind when analyzing the security of smart contract because the developers could have made incorrect assumptions about some of these that could result in security issues.

2.54 Randomness Source

The block timestamp and block hash that we just discussed are not good sources of randomness, that's because both these values can be influenced by the miners mining the blocks to some degree. The only aspects of timestamps that are guaranteed, is that the **current blocks timestamp must be strictly larger than the timestamp of the last block** and the other guarantee is that **it will be somewhere between the time stamps of two consecutive blocks in the canonical blockchain**. Therefore smart contract developers should not rely on either the block timestamp or the block hash as a source of good randomness.

2.55 Blockhash

`Solidity` supports a block hash primitive, where the block number can be specified to obtain the corresponding hash. This is possible only for the most recent 256 blocks excluding the current block, for all other historical blocks the value returned is 0. So as you can imagine this becomes important when you are considering the security aspect of code using this primitive, because **the**

block number should really be the most recent 256 blocks otherwise a value of zero is obtained.

2.56 ABI Encoding/Decoding

`Solidity` supports multiple functions in these categories. The obvious ones are the `encode` and `decode` functions that take arguments and encode them or decode them, specifically with respect to the ABI. There are also functions that encode with the function selector or with the signature, and finally there is an encode packed function that takes the arguments and performs the encoding in a packed fashion, so there's no padding applied between the arguments applied.

For this reason the packed encoding can be ambiguous. This is something that affects security when you're considering these functions, specifically this encode packed function.

2.57 Error Handling

Error handling is one of the most important fundamental and critical aspects of programming languages' security. The reason is that errors during program execution are what result in security vulnerabilities. These could be errors resulting from user inputs when they interact with the smart contract and the inputs are not as expected by the developer during the coding of the contract. They could also be due to assumptions made within the smart contract that are not really valid for the various control and data flows that happen during program execution. They could also be related to the programming variants that are expected from a specification perspective and these invariants might not hold good during certain control and data flows.

`Solidity` supports multiple primitives for error handling, being the first set of primitives are functions that let the developer assert or require certain conditions to be held. The `assert(x)` primitive for example, specifies a condition `x` as its argument, and if that condition is not met (if it evaluates to `false` during run time), then a panic error is raised and this panic error reverts all the state changes made to the contract logic so far in the context of the transaction that triggered it. The `assert` primitive is meant to be used for internal errors for program invariants that are meant to be held at any part of the program during execution.

This contrasts with the `require` primitive, which is another error handling primitive supported by `Solidity`. Similarly it also specifies a condition that gets evaluated at runtime, and if that condition evaluates to `false`, then it again raises a `revert`, that reverses all the state changes made to the contract so far. The `required` primitive is meant to be used for errors in inputs from

users or from external components that the contract interacts with. The `require` primitive takes an optional string as an argument: this is a message that gets printed if the required condition is not met.

Finally there is a `revert` primitive that unconditionally aborts execution when triggered, reverting all the state changes similar to when the conditions are not met for a certain `require` primitive. This again takes an optional string that gets printed when that happens. The thing to be kept in mind is the difference between `assert` and `require`. There were some historical differences as well in the use of a particular opcode: a different opcode for `assert` versus `require` that affected the Gas consumption, but some of these have been changed in the recent celebrity versions.

2.58 Math/Crypto Functions

Solidity supports the addition and multiplication operations with modulus: `addmod()` and `mulmod()`.

It obviously supports the Keccak-256 hashing function that is fundamental to Ethereum and used extensively within Ethereum and smart contracts themselves.

It also supports the standardized SHA-256 algorithm (related to Keccak-256), but the standardized version, it further supports one of the older hashing function, the ripe message digest `ripemd160(bytes memory)` for historical reasons.

finally it supports what is known as the `ecrecover` primitive. This is the elliptic curve recover function that takes in the hash of a message as an argument along with the signature components, the ECDSA signature components of v, r and s. `ecrecover` takes in these arguments and returns the address (or recovers the address) associated with the public key from the elliptic curve signature that is specified in the parameters. This is used in various smart contracts and it is used for different types of logic within them.

2.59 ecrecover Malleability

`ecrecover` is susceptible to malleability, or in other words non-uniqueness. In the context of signatures this means that a valid signature, can be converted into a second valid signature without requiring knowledge of the private key to generate those signatures. This depending on how signatures are used within the contract logic or the reason why the signatures are used within that contract logic can result in replay attacks, where the second valid signature can be used by the user or even by the attacker to bypass the contract logic that is

using these signatures.

The reason for this malleability is the math behind how elliptic curve cryptography works, so the signature components of v, r and s. The s value can either be in the lower order range or in the higher order range, and `ecrecover` does not prevent the s value from being in one of these two ranges. This is what allows the malleability. If the smart contract logic using `ecrecover` requires the signatures to be unique, then currently the best practice is to use the ECDSA wrapper from OpenZeppelin, that enforces the s value to be in the lower range (it forces there to be a single valid signature for these signature components).

2.60 Contract Related

Solidity supports some contract related primitives that need to be understood, the keyword `this` refers to the current contract and **it can be converted to an `address` type explicitly**. More importantly, there is a primitive known as `selfdestruct` that `Solidity` supports, recall that there is a self-destruct instruction in the EVM. This primitive is really a high level wrapper on top of that instruction, it takes in a single argument: an `address` type specifying the recipient, and what this recipient signifies is that when the contract is destroyed by this primitive, all the funds (the ether balance in that destroyed contract) are sent to this recipient address when the execution ends. There are some specifics of `selfdestruct` that need to be kept in mind from a security perspective.

2.61 selfdestruct

The recipient address that is specified in this primitive does not execute the receive function when it is triggered. Recall that contracts can specify a receive function that gets triggered on ether transfers or under other conditions. In the context of `selfdestruct` primitive, the recipient address happens to be a contract and it specifies a receive function that does not get triggered when `selfdestruct` happens. This is critical because any logic that might be within that receive function might have been anticipated by the developer to be triggered anytime ether is received by the contract, but `selfdestruct` is an exception to that logic.

Also the contract gets destroyed by `selfdestruct` only at the end of the transaction that has triggered this flow. What this means is that if there is any other logic after `selfdestruct` that may revert for various reasons, then that revert undoes the destruction of the contract itself. So just because we see a `selfdestruct` in the control flow does not mean that the contract gets destroyed, because logic after that might revert and really not result in the destruction of this contract.

2.62 Contract Type

`Solidity` supports some primitives specific to the contract type. `type(x)` is the syntax for doing so, where `x` is a contract type. The primitives supported are `type(C).name` that returns a name of the contract, `type(C).creationCode` and `type(C).runtimeCode` primitives return the creation and runtime byte codes of that contract. These are interesting details that are best examined by writing a simple contract and looking at what these primitives return.

There's also the interface id primitive `type(I).interfaceId` that returns the identifier for the interface specified. We'll take a look at the differences between interfaces and contracts later on, but these are primitives that are supported by `Solidity` specific to the contract or interface type.

2.63 Integer Type

There are similar parameters supported for integer types. This again represented by `type(x)` where `x` happens to be an integer. `type(t).min` returns the smallest value representable by the type `t`. Similarly `type(t).max` primitive returns the largest value that is representable by the type `t`. If we say `type(uint8).max` then it returns the maximum value representable by the unsigned integer of size 8 bits, and in this case it happens to be 255 which is $2^8 - 1$..

2.64 Control Structures

These are really fundamental to any programming language because there is a control flow to the sequence of instructions specified in the high-level language that get translated into machine code by the compiler.

In the case of `Solidity`, the control structures supported are `if`, `else`, `while`, `do`, `for`, `break`, `continue` and `return`. These are very similar to the ones found in any of the programming languages. Although are some differences in `Solidity`. Paranthesis for example cannot be omitted for conditionals that some of the other languages support, however curly braces can be omitted around single statement bodies for such conditionals.

Also note that there is no type conversion from a non-boolean to a boolean type. As an example `if(1)` is not allowed in `Solidity` because 1 is not convertible to the boolean `true`, which is supported by some of the other languages. So control structures play a critical role in the security analysis of smart contracts whether you're doing a manual review or whether you're writing a tool to pass the `Solidity` smart contract. These are some things to be understood really well because that's how control flows, and any analysis depends critically on making sure that the control flow is accurately followed and representative of what really happens at runtime.

2.65 Exceptions

`Solidity` supports the concept of exceptions to a great extent. We touched upon this under error handling. What this really means, is that in Solidity exceptions are used to handle errors, and these exceptions are state reverting. Which means that an exception undoes or reverses all the changes made to the state of the smart contract in the context of the current transaction: the calls and all the subcalls that may be several levels deep. All the changes that they have affected to the contact state are reversed, and this also flags an error to the caller so that they can take appropriate action.

So when exceptions happen within subcalls in that call hierarchy, during runtime they bubble up, and What this means is that exceptions are rethrown at the higher level calls automatically. There are some exceptions to this rule. There are some differences here in the context of the `send` primitive, and the low level function calls: `call`, `delegatecall` and `staticcall` which we talked about earlier. These primitives (`send`, `call`, `delegatecall` and `staticcall`) return a boolean `true` or `false` as their first return value instead of an exception bubbling up.

This is an important distinction to be kept in mind when analyzing smart contracts because the exception behavior is different for these primitives, compared to the standard message calls. Exceptions that happen in external calls made during the contact execution can be caught with the `try catch` statement.

These exceptions can contain data that is passed back to the caller and this data consists of a function selector indicating which function the exception happened in, and also some other ABI encoded data that gives more information about the exception.

`Solidity` supports two error signatures, the error and panic. Error takes a string parameter whereas panic takes an unsigned parameter. Error is meant to be used for "*regular error conditions*", such as input validation and so on. The panic is used for errors that should not be present in bug free code. So this is the `assert` primitive situation where program invariants are being violated and panic is meant to be used.

2.66 Low-level Calls

Let's now revisit the concept of low level calls and talk about very specific peculiarity or counterintuitive aspect of these calls. As we first talked about these calls under the EVM instruction set, there is `call`, `staticcall` and `delegatecall`, each of them corresponding to the specific instruction that implements these calls.

The counter-intuitive aspect is that **if these calls are made to contract accounts that do not exist for some reason they still return `true` based on the design of the EVM**.

This can have some serious side effects if the contract logic really assumes that the external call was successful and executed the logic that it expected it to execute because it got a value of `true` from these primitives. The mitigation for this aspect of low level calls is to check for contract existence before these calls are made, and have the logic handle it appropriately if they do not exist.

This has resulted in some serious security vulnerabilities being reported in various high-profile smart contract projects, so something to be kept in mind when analyzing the security of smart contracts that use these low level calls.

2.67 Assert

As we know, `assert` is really meant to be used for the *program invariants* that should never be violated within the smart contract if it does not have bugs as intended by the developer.

These asserts result in the panic errors that take the `uint256` type, to reiterate they should be used for internal errors for checking invariants, normal code bug free code should never cause panic. Comparing it to the `require` primitive, `assert` should only be used again for program nvariants, and not for checking things like invalid external inputs or invalid interactions with external dependencies.

2.68 Panic

The Panic exception is generated in various situations in `Solidity`, and the error code supplied with the error data indicates the kind of panic.

There are many of these error codes. Some of them are

- `0x01`: indicates that `assert` has an argument that evaluated to `false`.
- `0x11`: an overflow or underflow happened in arithmetic.
- `0x12`: division by zero or modulus by zero occured.
- `0x31`: `pop()` of an empty array occurred.
- `0x32`: out of bounds access for an array.

There are numerous error codes for panic.

2.69 Require

Under the `require` primitive, `Solidity` either creates an error of type error strain or an error without any error data.

This should be used to detect invalid conditions during runtime that cannot be prevented at compile time, including input validation (which is a fundamental pillar of programming languages' security).

Inputs at runtime that are obtained from users interacting the contracts should be validated to make sure they are within the thresholds of what is acceptable with the smart contract logic, so some sanity checks on those values are necessary.

Require is also used for checking the return values from calls that are made to external contracts. So any type of external interaction, be it inputs from users or return values from external contact calls, are what require is meant to be used with. Require takes in an optional message string that is output when the condition fails.

2.70 Error

Error string exception, as discussed earlier, are generated when `require` executes and its argument evaluates to `false`.

The error string is also generated in other situations such as an external call made to a contact that contains no code, or if the contract receives ether via public function without the payable modifier. Or if the contract receives ether via a public getter function.

2.71 Revert

There are two ways to explicitly trigger a `revert` in `Solidity`: using the `revert CustomError(arg1,...)` primitive or the `revert([String])` function, where the `String` parameter is optional.

In both these cases, the execution is aborted and all the state changes made. as part of the transaction, are reversed. This distinction between `CustomError` and the string, is interesting from an optimization and use case perspective.

2.72 try/catch

These primitives supported by `Solidity` are fundamental error handling. The syntax is

```
try Expr [returns()] {...}
catch <Block> {...}
```

Code 2.22: try/catch example

So, we have the `try` and `catch` keywords coupled with an expression that contains an external function call or creation of a contract. These are coupled with code blocks corresponding to the success blocks or the catch blocks. These are code segments within the curly braces shown here. Which block gets executed depends on whether there was a failure or not in that external call within that expression.

If there were no errors then the success block gets executed (the block that immediately follows the `try` expression in the syntax shown here). But if there was an error in that external call then the catch block, or one of the catch blocks, gets executed. Which catch block gets executed depends on the error type, and there are multiple of them.

2.73 catch Blocks

`Solidity` supports different kinds of `catch` blocks depending on the error type. There is a `catch` block that supports an error string `catch Error(<string reason>)`. This is executed if the error was caused by `revert` with a reason string, or `require` where the condition evaluated to `false`.

Then there is a `catch` kind that supports panic error code `catch Panic(uint <error code>)`. If this error was caused by a panic failing `assert`, (remember: division by zero, outer bound array accesses, arithmetic underflow/overflow) this is the `catch` block that will be run.

In addition there is a `catch` that specifies the low level data `catch (bytes <LowLevelData>)`. This one gets executed if the error signature does not match any other clause shown above. Or if there was an error while decoding the error message itself, or if no error data was provided with that exception. This variable that is declared the low-level data gives us access to the error data in that case.

Finally if the developer is not really interested in the type of error data, one can simply use `catch` as is. These give various options to deal with different types of exceptions, that might come from the external call that is used within the `try catch` permit.

2.74 try/catch State Change

As we just discussed, there exists the concept of the success block, that gets executed when there are no exceptions in that external call. There are also

error blocks that correspond to the different `catch` blocks which get executed when there are exceptions encountered in that external call.

If execution reaches the success block, it means that there were no exceptions in that external call, and all the state changes that are done in the context of the external call are committed to the state of the contract.

But if execution reaches one of the `catch` or error blocks, then it means that the state changes in that external call context have been reverted, because of the exception. There could also be a context where the `try catch` statement itself reverts for reasons of decoding or low level failures.

2.75 External Call Failure

These failures of the external call made in the context of the catch primitive, could happen for a variety of reasons and one cannot always assume that the error message is coming directly from the contract that was called in that external call, because the error could happen deeper down in the call chain resulting from that call. It was forwarded to the point where it was received.

This could also be due to an out of Gas or OOG situation, in which case the caller still has a bit of Gas to deal with that exception because not all of it is forwarded to the callee.

2.76 Programming Style

So far so all the aspects of the various basics of `Solidity`: syntax, semantics,... are rules that are enforced in `Solidity` grammar.

Programming style on the other hand is coding convention, and these are different across different developers. Different styles are adopted based on what the developer is comfortable with based on what they believe is an optimal way of programming, **but fundamentally the programming style is about consistency**.
The reason is that programming style affects the readability and maintainability of the code. So when anyone other than the developer looks at the code to evaluate the security, to audit it or to make fixes or extend those smart contracts, the consistency of the programming style becomes important. If different styles are used within the same function, within the same module or within the same project, because of the same developer not being consistent or because multiple developers are involved in that project having different styles, then this significantly affects the readability and maintainability of the code, impacting both significantly on the security life cycle of the code. There

are two main categories of programming style, that of code **layout** and that of **naming**.

2.77 Code Layout

Code layout refers to the physical layout of the various programming elements within a source code file. There are many programming style aspects related to code layout: those related to indentation, where the best practice and `Solidity` is to recommend 4 spaces per indentation level and to prefer spaces over tabs and to definitely not mix them.

There are also style guidelines with respect to blank lines used to surround declarations, with the max line length that's recommended to be 79 or 99 characters for best readability. There are also recommendations for wrapping lines, for the encoding used in the source files (`ASCII` or `UTF-8`), and for keeping the import statements at the top of the file and not anywhere else.

Finally, the ordering of the functions within the contract, where the recommendation is to have the constructor as the first in that order followed by functions of different visibilities. Grouping all the external functions first, followed by public functions and then internal and then private functions.

2.78 Code Layout (More)

There are more code layout programming style guidelines specific to the use of white space within expressions, curly brackets and spaces and control structures function, declarations, mappings and variable declarations.

Strings are recommended to be used with double quotes instead of single quotes operators, spaces have some guidelines as well. Finally the ordering of different program elements within a `Solidity` file also have a guideline, which is to have the `pragma` declaratives all the way at the top, followed by the `import` directives and then the contract library or interface definition itself.

Within each of the contracts libraries and interfaces, the guideline suggests using the types first, followed by declarations of state variables, then events and finally all the various functions.

2.79 Naming Convention

The next aspect of programming style is naming convention. This refers to the names that are given to various variables, events, contracts, libraries and all the different program elements used within smart contracts.

There are different types of names: lower case names, lower case with underscores, all upper case, upper case with underscores, capitalized words, mixed case, capitalized words with underscores and so on. All these different types are recommended to be used for different program elements, and as a general rule the guideline is to avoid letters that can be confused with the different numerals, like the lowercase letter`"l"`, or the uppercase letter `"O"` and uppercase letter `"I"` that could be confused with 0 and 1 numerals.

Contract and libraries should be named with cap word style, they should also match their file names and if a contact file includes multiple contracts and libraries then the file name should match the core contract (or what is considered as a core contract for that file by the developer). Structs should be named using the cab word style. Event should be named using cap word style again. Functions should be named using mixed case.

This is something that you'll encounter often within contracts developers are sometimes consistent with these naming, sometimes they're mixed up because of multiple developers or the style just not being consistent or being confused with that of some external libraries.

Again all these aspects affect the readability and maintainability, they do not have any impact on the syntax or the semantics of the contract itself. The byte code is just the same but it has an effect to a certain extent, on the security audit aspect when you look at this code, and when the naming convention is different or not consistent then it could lead to some assumptions being made on these variables being the same ones or different ones.

2.80 Naming (More)

Some more naming conventions are: function arguments should be in mixed case, local state variables again in mixed case, constants however should be with all capital letters and underscores separating the multiple words if they are present.

Modifiers in mixed case, enums in cap word style and finally one should avoid naming collisions where the desired names, variables or functions collides with that of a built-in within `Solidity` or any other reserve name. So those should be resolved using single trailing underscores in those names.
Programming style is subjective in nature: different developers different teams might have different philosophies as to what works best for them. The key is consistency within the function modules, contracts or projects, as it affects readability and maintainability, which are critical for security, specifically with respect to smart contract audits.

Chapter 3

Solidity 201

3.1 Inheritance

Remember that `Solidity` is an object-oriented programming language, so it supports various aspects of inheritance. It supports multiple inheritance and polymorphism. If you have studied other object-oriented programming languages, a lot of these concepts must be familiar to you, and are very similar in `Solidity`.

Polymorphism means that a function call executes the function of the specified name and parameter types in the most derived contract in the inheritance hierarchy. When a contract inherits from multiple other contracts, only a single contract is created on the blockchain with the code from all the base contracts compiled into the created contract.

Function overriding means that functions in the base classes can be overridden by those in the derived classes which can change their behavior. If they are marked as virtual using the `virtual` keyword the overriding function must then use the `override` keyword to specify that it's overriding the virtual function in the base classes.

Languages that allow multiple inheritance have to solve some problems. One of them is known as the **diamond problem**: this is solved in `Solidity` in a very similar way to how it is solved in `Python`, using what is known as `C3` linearization, that forces a specific order in the directed acyclic graph constructed from the base classes. At a high level, when a function is called that is defined multiple times in different contracts (in the base and derived classes) the given bases are searched in a specific order from right to left, in a depth first manner and stopping at the first match that is found. The difference between how `Solidity` implements this versus `Python` is `Solidity` searches these classes from right to left in the specified order as opposed to left to right in `Python`.

3.2 Contract types

Besides the typical contracts supported by `Solidity` it also supports three other contract types. Those are abstract contracts, interfaces and libraries.

- Abstract contracts are where at least one of the functions in the contract is not implemented. These are specified using the `abstract` keyword.
- Interfaces on the other hand can't have any of the functions implemented within them, they can't inherit from other contracts, all the declared functions must be external, they can't declare a constructor and they can't have any state variables. These are specified using the `interface` keyword.
- Libraries are meant to be deployed only once at a specific address. The callers call the libraries using the `DELEGATECALL` opcode. This means that if library functions are called, their code is executed in the context of the calling contract. Libraries are specified using the `library` keyword

3.3 Using for

`Solidity` supports `using for` directive, which is used for attaching library functions to specific types in the context of a contract. So for the directive `using A for B`, `A` specifies the library and `B` specifies a particular type. This means that the library functions in a will receive objects of type `B` when they are called on such types, and they will receive the object of that type as their first parameter. This directive is applicable only within the current contract, including within all its functions. It has no effect outside of the contract in which it is used, so for example, if this directive is used as shown here saying `using safeMath for uint256;`, it means that variables of type `uint256` within that contract where this directive is used can be attached functions from the `SafeMath` library.

3.4 Base Class Functions

When considering the inheritance hierarchy: we have base classes and then the derived classes. If in the derived classes one would like to call functions further up in the inheritance hierarchy (e.g. the base classes), this is possible. If we specifically know the contract that has the function that we would like to call, then we could specify that as shown here

```
1 Contract.function();
```

Code 3.1: Explicit function call from base class example

If we wanted to call the function exactly one level higher up in the flattened inheritance hierarchy, this can be done by using the `super` keyword as shown here

```
super.function();
```

Code 3.2: "One level up" function call from base class example

3.5 Shadowing

It was supported in `Solidity` for the state variables until version `0.6.0`. This effectively allowed state variables of the same name to be used in the derived classes as they were declared in the base classes. These shadowed variables could effectively be used for purposes other than those declared in the base classes. This used to be allowed until version `0.6.0` and it was removed from that version on because it caused quite a bit of confusion and potentially could lead to serious errors from a security perspective. As of the latest versions, state variable shadowing is not allowed in `Solidity`. This means that state variables in the derived classes can only be declared if there is no visible state variable with the same name in any of its base classes.

3.6 Overriding changes

Remember function overriding means that functions in the derived classes can override the virtual functions in their base classes to redefine the logic within them. These overriding functions may also change the visibility of the overridden function, but this can only be done from changing them from external to public. The mutability of these functions may also be changed, but only to a more stricter one following this order: non-`payable` mutability can be changed to either `view` or `pure`. `view` mutability may be changed to `pure`. `payable` mutability is an exception: it can't be changed to any other mutability.

3.7 Virtual Functions

Virtual functions are functions without implementation. These have to be marked as `virtual` outside of interfaces. In interfaces all functions are automatically considered `virtual`, so they don't need to use the `virtual` keyword. However in `abstract` contracts for example, if a function has to be considered as `virtual`, that is without specifying an implementation, then it should specifically use the `virtual` keyword to indicate as such. Functions with `private` visibility can't be made `virtual`.

3.8 State Variables

Remember that state variables in `Solidity` can have different visibilities. One of them is `public`. `public` state variables have automatic getter functions generated by the `Solidity` compiler. These getters are just functions that are

generated to allow accessing the value of the `public` state variable, so they return the value of those state variables. Such `public` state variables can override external functions in their base classes that have the same name as the `public` state variables, parameter and return types of those `external` functions match the getter function of these variables. so while public state variables in `Solidity` can override `external` functions according to thpse, they themselves can't be overridden.

3.9 Function Modifiers

Function modifiers can also override each other. This is very similar to how function overriding works except that there is no concept of overloading for modifiers. The `virtual` keyword again must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier. Again very similar to the concept of `virtual` and `override` functions.

3.10 Base Constructor

When you have classes deriving from other base classes, then the base and the derived classes could have constructors. The constructors of all the base contracts will be called following the linearization rules (which we touched upon earlier in the context of `Solidity`). If the base constructors have arguments, then the derived contracts need to specify those arguments. This can be done either in the inheritance list of the derived contract or it can be explicitly done, so within the derived constructor itself.

3.11 Name Collision

Name collision is always an error in `Solidity`. It is an error when any one of the following pairs in a contract have the same name due to inheritance. A function and a modifier can't have the same names in the base and derived classes. A function and an event can't have the same name either. Finally, an event and a modifier also can't have a same name, if this happens, then this is a compile time error.

3.12 Library Restrictions

`Solidity` supports different types of contracts, there are the typical vanilla contracts that one always encounters. But there are also abstract contracts, interfaces and libraries. Libraries in particular have several restrictions compared to typical contracts: they can't have state variables, they can't inherit from other classes or be inherited themselves, they can't receive Ether, they can't also be destroyed, they have access to state variables of the calling contract

only, if they are explicitly supplied. Library functions can only be called directly without the use of `delegatecall`, if they do not modify the state, that's, if they are `view` or `pure` functions. This is because libraries are assumed to be stateless by default.

3.13 EVM Storage

Let's how some of the `Solidity` concepts map to the EVM storage. Remember: it is a `(key, value)` store that maps 256 bit words to 256 bit words, so the `key` and `value` are both considered to be the word size supported by the EVM. The instructions used to access the storage are `SLOAD` to load from storage and `SSTORE` to write to storage from the stack. Remember that all locations in the storage are initialized to zero.

3.14 Storage Layout

Let's now understand how state variables declared within a smart contract map to the underlying EVM storage. State variables are stored in the different storage slots. Each slot in the EVM storage corresponds to a word size of 256 bits. The various state variables declared within the smart contracts are mapped to these storage slots in the EVM, and if there are multiple state variables that can fit within the same storage slot depending on their types, then they are done so to maintain a compact representation of the state variables within that storage slot. The mapping is done in the same order as the declaration of the state variables, so state variables that are declared within a contract are stored contiguously in their declaration order in the different storage slots of the EVM, which means that the first state variable is stored in slot 0 the second one in slot 1 or maybe the same slot 0... If the first variable was of a size smaller than 256 bits, the second one could fit as well within that slot, so except for dynamic arrays and mappings, all the other types of state variables are stored contiguously item after item starting with the first state variable.

3.15 Storage Packing

This means that state variables which are declared next to each other are contiguously stored within the same storage slot, if the size of their types allow to do. Remember that `Solidity` supports different types and each type has a default size and bytes, so it all depends on the types of the state variables declared and their underlying sizes. If there are multiple contiguous state variables that need less than 32 bytes, then those are packed into the single storage slot where possible. There are some rules that are followed: the first item in a storage slot is stored lower-order aligned value types that only use as many bytes that are necessary to store them, and when a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot. This concept of storage

packing becomes important when we are looking at a smart contract code and trying to determine which storage slot a particular state variable fits in, which depends on the other state variables that are declared around it.

3.16 Structs & Arrays

State variables of type structs and arrays have specific rules with respect to the storage slot allocation. Such state variables always start a new storage slot as opposed to being packed into existing ones. The state variables following them also start a new storage slot. The elements of the structs and arrays themselves are stored contiguously right after each other as if they were individual values. and depending on their types the rules we just discussed in the previous section apply to these as well.

3.17 Inheritance

How does inheritance affect the storage slot allocation? For contracts that use inheritance, the ordering of state variables is determined by the `C3` linearization rule of the contract orders, starting from the most base-word to the most derived contract. If allowed by any of the rules discussed, state variables from the different contracts (the different base and derived contracts) are allowed to share the same storage slot with respect to the storage packing concept we talked about.

3.18 Layout & Types

Storage packing allows us to optimize the storage slot layout depending on the types of the state variables. So state variables can be made to have a reduced size type depending on the values that they're supposed to hold, then storage packing allows such state variables to share a storage slot. This allows the service compiler to combine multiple reads or writes into a single operation when it generates the corresponding byte code.

However, if those state variables sharing the same slot are not read or written at the same time, depending on the contract logic, this can have an opposite effect, which results in more Gas being used than expected. This is because when one such value of a state variable that shares that slot with other state variables is being read or written, then the entire slot is read or written because that is the size that the EVM and `Solidity` work with. Now the specific state variable within that slot has to be separated out for reading or writing, this is done by masking out all the other state variables that share that slot.

This masking results in additional instructions being generated which lead to additional Gas being used in this case, so depending on the specific sizes of

the types and on the pattern of reading or writing, the types of state variables that are adjacent to each other in the declarations should be bid for efficient optimization from storage packing.

3.19 Layout & Ordering

To summarize: the ordering of the state variable declarations within a smart contract impact the layout of their storage slots and affects if multiple state variables declared contiguously can be packed within the same storage slot or if they need separate storage slots. This packing has a huge impact on the Gas Cost because the instructions that read and write state variables (if you remember are `SLOAD`s and `SSTORE`s) are the most expensive instructions from a Gas Cost perspective supported by EVM.

`SSLOAD`s costs as much as 2100 Gas or 100 Gas depending on how many times the state variables has been read. So far in the context of this transaction. `SSTORE`s cost as much as 20000 Gas in the most recent EVM versions. As an example, if we have three state variables of types `uint128`, `uint128` and `uint256` that are declared within the same smart contract contiguously, then these variables would use 2 storage slots because the first 2 storage variables can share the same storage slot. The 2 variables of size 128 bits will fit into the same storage slot: slot 0 in this case (which is 256 bits in size). The third state variable of type `uint256` would go into the second storage slot (or slot 1). But if the declaration order is slightly changed (so for example putting the 256 bit state variable in between the 128 bit variables), then the new order would require 3 storage slots instead of 2: the first 128 bit one would go into slot 0, the second one would not fit within slot 0, so it would go to slot 1 and consume the whole slot 1. The third state variable would then take up a slot. This gives you an idea of how the state variable declaration order impacts a number of storage slots, which has a big impact on the Gas Cost used by that contract.

3.20 Mappings & Dyn Arrays

Storage slot allocation for mappings and dynamically sized arrays is a bit more complex than their value type counterparts. These types are unpredictable in their dynamic size by definition and because of that reason the storage slots allocated for them can't be reserved in between the slots for the state variables that surround them in the declaration order within their contract.

Therefore these are always considered to occupy a single slot, that's 32 bytes in size with regard to the rules discussed so far and the elements that they contain within, that that can change dynamically over the duration of the contract, are stored in a totally different location: the starting storage slot for those elements is computed using `keccak-256` hash.

3.21 Dyn Arrays

Let's say we have a state variable of type dynamic array and based on the declaration order within that smart contract, let's say that it is assigned slot number `p`. This slot only stores the number of array elements within that state variable and is updated during the lifetime of the contract when this changes. The actual elements of the dynamic array itself are stored separately in different storage slots. The starting slot for those elements is determined by taking the `keccak-256` hash of the slot number b. The elements themselves starting from that storage slot that we just calculated are stored contiguously and can also share those storage slots if possible, depending on their types, on the size of those types and, if we have dynamic arrays that in turn have dynamic arrays within them, then the same set of rules apply recursively to determine their corresponding storage slots.

3.22 Mappings

Again, depending on the declaration order, if there is a state variable of mapping type and it gets assigned a slot number `p`, then that particular slot stores nothing: it's an empty slot just assigned to that mapping. Compare this to the dynamic array that we just discussed where this slot stores the number of those array elements. In the case of mappings, there is no such concept, so this slot `p` stores nothing: it's just a blank slot that is allocated for this mapping. The slots corresponding to the values for keys of this mapping are calculated as follows: for every key `k`, the slot that is allocated is determined by taking the `keccak-256` hash of `h(k).p`. The `.` is a concatenation of the two values of `h(k)` and `p`. We know that `p` is the slot number, which we mentioned earlier on. `h` is a function that is specific to the type of the key that we're talking about and if this is a value type then, there is a padding that is done to make it up to 32 bytes. If it is a string or byte arrays, then `h()` computes the `keccak-256` hash of the unpadded data. The type specific rules that determine what `h()` is and those details are specified better at the reference provided.

3.23 bytes & string

For this case, there is an interesting optimization. The storage layout for these is very similar to arrays, so the actual storage slot depending on the declaration order, stores the length of these types, the elements themselves of the variable are stored separately in a storage slot that is determined by taking the `keccak-256` hash of the storage slot assigned to store the length. However, if the values of these variables are short, then instead of storing these elements separately they are stored along with the length within the same storage slot. The way this is done is: if the data is at most 31 bytes, then the first byte in the lowest order stores the value `length*2` and all the other bytes (the higher order bytes) store the elements that fit within the remaining 31 bytes. If the

length of the data is more than 31: so if it is 32 bytes or more, then the lowest order byte stores the value l`length*2 + 1`, the elements themselves don't fit within this storage slot that stores the length, so they are stored separately using the `keccak-256` hash of this slot's position.

The distribution of whether the data values values are stored within the same storage slot as the length or if they're stored separately, is made by looking at the lowest order bit. If that is set (`1`) it means that they are stored separately and, if they're stored within the same slot as a length, then this bit will not be set (`0`). This is because of the length being stored as `length*2 + 1` or just `length*2`.

3.24 Memory

Remember: EVM is a stack based architecture, it has call data, the volatile memory, the non-volatile storage. EVM memory has a linear layout, which means that all the memory locations are stored linearly next to each other, and memory locations can be addressed at a byte level. The EVM instructions that are used to access memory are `MLOAD`/`MSTORE` that operate on the word size (256 bits) and, if one wants to store a single byte from the stack to memory, one can use the `MSTORE8`. All locations in memory are zero initialized.

3.25 Memory Layout

`Solidity` places new memory objects at the free memory pointer and all this memory that is allocated is never freed or deallocated. This free memory pointer in `Solidity` initially points to a hex value of 80. All these concepts related to memory layout matter from a security perspective only: if the developer is manipulating this memory directly in the assembly language support provided by `Solidity` because, if one is using `Solidity` as a high-level language without using assembly, then all this is automatically handled by the `Solidity` compiler itself.

3.26 Reserved Memory

The initial value of the free memory pointer and `Solidity` is hex 80 because the first four 32 byte slots are reserved by `Solidity`, these add up to 128 bytes and therefore the `x80` value. The first two of those 32 byte slots are used by `Solidity` as a scratch space for the hashing methods. The third slot again 32 bytes is used for the free memory pointer, so this points to the next byte of memory within `Solidity` that is considered as "*free*" or in effect this also indicates the amount of allocated memory currently within `Solidity`. The fourth such slot is referred to as a zero slot and is used by `Solidity` as an initial value for dynamic memory arrays. We'll talk about that shortly.

3.27 Memory Arrays

For memory, arrays every element in it within `Solidity` occupies 32 bytes. This is something we mentioned in the context of the byte array and how every element occupying 32 bytes wastes a lot of space. Despite the type of the memory array, this is not true for bytes and string types. And for multi-dimensional memory arrays, those are pointers to memory arrays. For dynamic arrays, very similar to the storage even within memory, these are stored by maintaining the length of the dynamic array in the first slot of that array in memory followed by the array elements themselves.

3.28 Free Memory Pointer

This pointer is maintained at position hex 40, because the first two 32 byte slots are reserved for the hash function scratch space as we just talked about. The initial value in the free memory pointer is hex 80 just beyond the reserved slots, this pointer effectively points to memory that is allocatable in the context of `Solidity` at any point in time and whenever memory is allocated by the compiler, it updates the free memory pointer. These concepts should be familiar, if you have looked at memory allocation of any other programming languages, these just happen to be the specific ways in which `Solidity` handles memory allocation using the familiar concept of the free memory pointer.

3.29 Zeroed Memory

With respect to zeroed memory or memory containing zero bytes, there are no guarantees made by the `Solidity` compiler that the memory being allocated has not been used before, so one can't assume that the memory contents contain zero bytes. The reason for this is that there is no built-in mechanism to automatically release or free allocated memory in `Solidity`. As you can imagine, this has a security impact because if one is using memory allocated objects, those are not guaranteed to be zeroed memory. Then the default values may not be zeros. These again are relevant only if memory is being manipulated directly in assembly within `Solidity`. This should not be much of a concern if one is not using assembly.

3.30 Reserved Keywords

These are keywords in `Solidity` that are reserved for future use, so they are not currently used by any of the syntax that is supported. These may be used for any anticipated new syntactic features within `Solidity`. There are many such reserved keywords, some of them are: `after`, `alias`, `apply`, `auto`, `case`, `null`, etc... And you can imagine why these could potentially be reserved: because they all have a specific significance in the context of programming languages or

in the context of object-oriented programming languages. `Solidity` anticipates that it may support features that may end up using these reserved keywords. An example of a keyword that was reserved earlier is `unchecked`, which is now used as of version `0.8.0` for declaring any block within `Solidity` as being unchecked for arithmetic overflow and underflow checks. So we can assume that some of these reserved keywords might be supported in future `Solidity` versions for different features.

3.31 Inline Assembly

Inline assembly is a way to access the EVM features directly at a low level, and from a security perspective this is important because it bypasses some of the safety features provided by `Solidity` at a high level language. Type safety is one such aspect, so if a developer is manipulating in inline assembly, then the corresponding code does not enjoy the type safety benefits provided by the `Solidity` compiler. The language used by solution for inline assembly is called `YUL`. This is somewhat of a recent feature.

There have been a lot of developments in the inline assembly support by `Solidity` in the most recent versions. This sees constant updates. As you look at the most recent versions of `Solidity`, an inline assembly block is marked using the keyword `assembly{...}`. The inline assembly code is placed within the curly braces, annd is specified in the `YUL`.

3.32 Assembly Access

`YUL` support assembly access to various features such as the external variables, functions and libraries. Local variables of value type are directly usable in inline assembly, and local variables that refer to memory or call data evaluate to the variable address and not the value itself, effectively serving as a reference. For local storage variables or state variables that are also allocated in the storage, a single `YUL` identifier is not sufficient, because remember that storage has a concept of packing, where multiple variables can share the same storage slot and therefore their address is in two parts: it refers to the slot and the offset within the slot. Assignments are possible to assembly language variables which allow rich manipulation of these variables within inline assembly. One should take care when manipulating in this aassembly language: one should remember that variables that point to memory or storage changed the pointer itself and not the data and, there are many other rules and restrictions as you can imagine when it comes to manipulating all these aspects within assembly as supported by `YUL`.

3.33 YUL Syntax

YUL supports literals and calls. there are variable declarations that are possible in the form of `let x : 7` which declares a new variable `x` and assigns an initial value of 7 to it. There are scoping blocks that are supported by YUL, so that multiple blocks can be considered within the assembly blocks. There is rich control flow that is supported using, `if`, `switch` and `for`. There are also function definitions that are supported by YUL, so that within inline assembly you can have multiple functions that help you modularize code.

Take a look at the developments in the YUL language as supported by Solidity, this is happening at a great speed: there are a lot of features being added to provide a lot of richness and expressiveness by the YUL language for developers who want to code directly in assembly, but like mentioned before from a security perspective this becomes even more critical than programming in Solidity itself because inline assembly is typically considered as very close to the underlying virtual machine. So in this case, very close to the EVM and, if the internals of the EVM layout and all the nuances with respect to that are not paid attention to, then coding directly in YUL in Solidity's assembly language might result in some serious bugs where the manipulations are not done correctly and corruption happens or maybe even vulnerabilities.

3.34 solc 0.6.0 Breaking

Remember that breaking versions are versions that are not backwards compatible, or in other words they've introduced significant changes to the syntax, to the underlying semantics, that are not compatible with the previous changes. These breaking versions increment the number that you see in the middle of the version, so for a Solidity version `x.y.z`, the next breaking version would be `x.(y + 1).z`. In Solidity 0.6.0 a breaking semantic feature that was introduced changed the behavior of the existing code without changing the syntax itself. It was specifically related to the exponentiation. The type of the result until this version was the smallest type that could hold both the type of the base and the type of the exponent. With this change the resulting type was always the type of the base.

3.35 solc 0.6.0 Explicitness

Solidity 0.6.0 also introduced 6 sets of explicitness requirements. Explicitness, as you can imagine, is good for security because it reduces ambiguity and any vulnerabilities that result because of that ambiguity. In this case, keywords `virtual` and `override` were introduced for functions in base and derived classes. Functions and base classes can now only be overridden when they are marked with the `virtual` keyword, and their corresponding overriding functions need to use the `override` keyword array length is read-only: it's no longer possible

with version 0.6.0 to resize storage arrays by assigning a new value to their length. An abstract keyword was introduced for what became abstract contracts or contracts where at least one function is not defined. Libraries have to implement all their functions, not only the internal ones, as of this version and there are various restrictions (explicitness restrictions) brought forward for the assembly variables. The other big change was state variable shadowing being removed. State variable shadowing is when you have state variables in the base contracts that are shadowed in the derived contracts, or the derived contracts have state variables with the same name as the base contract. This can be confusing results in ambiguity and have impacted the security of smart contracts, so with the 0.6.0 this aspect of shadowing was not allowed henceforth.

3.36 solc 0.6.0 Changes

There were many other syntactic and semantic changes brought forward by Solidity 0.6.0. external function type conversions to address types are not allowed. Instead they have an address member that allows similar functionality. Dynamic storage arrays, for their push(x), that until then returned the length of the array, now returns nothing. Also, until 0.6.0 there was a concept of unnamed functions. This version split the functionality implemented by such a function into a fallback function and a separate receive function. These are two types of functions that we have discussed in the Solidity module. There are differences between these two functions and specific use cases where one of them is applicable versus the other. The receive function for example is called whenever the call data is applied is empty, this is implicitly payable whereas the fallback function is called when no other function matches. The behavior of this also depends on if there is a receive function in that contract or not, and a fallback function can be made payable or not. If it is not payable, then transactions not matching any other function that send value will revert. There are all these aspects that until 0.6.0 were combined under the concept of the unnamed function.

3.37 solc 0.6.0 New Features

0.6.0 also introduced several new features: the try/catch blocks for exception handling. This, if you remember, is used to react and handle failed external call. struct and enum types can be declared at a file level with this version. Until then, it was only at a contract level. Array slices can be used for all data arrays. NatSpec, as of this version supports multiple return parameters for developer documentation; it enforces the same naming checks as the param tag. The inline assembly language YUL introduced a new statement called leave to help exit the current function. Furthermore, conversions from address type to address payable type are now possible via the payable(x) primitive, where x is of type address.

3.38 solc 0.7.0 Breaking

The next breaking release was `Solidity 0.7.0`. With this version, exponentiation and shift of literals by non-literals will always use `uint256` or `int256` to perform the operation. Until this version the operation was performed using the type of the shift amount or the type of the exponent, which can be misleading, so this became very explicit. This again is a breaking semantic change because the behavior of exponentiation and shifts changed underneath without any changes to the syntactic aspect.

3.39 solc 0.7.0 Changes

This version also introduced several syntactic changes that could cause existing contracts to not compile anymore and therefore considered a breaking change. Examples of such changes were the syntax of specifying the Gas and Ether values applied during external calls. The `now` keyword for time management within contracts was deprecated in favor of `block.timestamp` because `now` gave the perception that time could change within the context of a transaction whereas it is a property of the block, correctly indicated by `block.timestamp`. The `NatSpec` aspect for variables was also changed to allow that for only `public` state variables and not for `local` or `internal` variables. `gwei` was declared as a keyword and therefore can't be used for identifiers. string literals can contain only printable `ASCII` characters. As of this version `unicode` string literals were also supported with the use of the `unicode` prefix. The state mutability of functions during inheritance was also allowed to be restricted with this version, so functions with the default state mutability can be overridden by `pure` and `view` functions while the `view` functions can be overridden by `pure` functions. There are also multiple changes introduced to the assembly support within `Solidity`.

3.40 solc 0.7.0 Removed

This version also removed some features that were considered as unused or unsafe and therefore beneficial for security. Struct or arrays that contain mappings were allowed to be used only in storage and not in memory, the reason for this was that mapping members within such structural arrays in memory were silently skipped. This as you can imagine would be error prone.

The visibility of constructors, either `public` or `external` is not needed anymore. The `virtual` keyword is disallowed for library functions, because libraries can never be inherited from and therefore the library functions should not need to be `virtual`. Multiple events with the same name and parameter types in an inheritance hierarchy are disallowed, again to reduce confusion. The directive `using A for B` with respect to library functions and types affects only the contract it is specified in as of this version. Previously this was

inherited, now it has to be repeated in all the derived contracts that require this feature.

Shifts by sign types are disallowed as of this version. Until now shift by negative amounts were allowed, but they caused a `revert` runtime. The Ether denominations of `fini` and `zabo` were considered as rarely used and therefore were removed as of this version. The keyword `var` was also removed because this would until now pass, but result in a type error as of this version.

3.41 solc 0.8.0 Breaking

`Solidity 0.8.0` is the latest of the breaking versions of `Solidity`. This version introduced several breaking changes. The biggest perhaps is the introduction of default checked arithmetic. This is the overflow and underflow arithmetic checks that are so commonly used in `Solidity` contracts to prevent the wrapping behavior that results in overflows and has resulted in several security vulnerabilities. Until this version, the best practice was to rely on the `OpenZeppelin SafeMath` libraries or their equivalents to make sure that there are runtime checks for overflows and underflows. These never result in vulnerabilities. This is so commonly used that `Solidity 0.8.0` introduced the concept of checked arithmetic by default, so all the arithmetic that happens with increment, decrements, multiplication and division is all checked by default. This might come at a slight increase of Gas Cost, but it also increases the default security level significantly and it also improves the readability of code because now one doesn't have to use or see the use of calls to the `SafeMath` libraries in the form of `.add()`, `.sub()` and, so on... And as an escape hatch where the developer knows for sure that certain arithmetic is safe from such underflows and overflows, `Solidity` provides the `unchecked` primitive that is allowed to be used on blocks of arithmetic expressions where this default underflow and overflow checks are not done by the `Solidity` compiler.

ABI coder version `v2` is activated by default. As of this version, it doesn't have to be explicitly specified but if the developer wants to fall back on the older `v1` version that has to be specified. Exponentiation is right associative as opposed to being left associative that was the case. This is the common way to parse exponentiation operator in other languages, so this was fixed. As of this version the use of the `revert` opcode versus the use of the `invalid` opcode for failing asserts and internal checks was removed. As of this version both these now use the `revert` opcode and static analysis tools are allowed to distinguish these two differing situations by noticing the use of the panic error in the case of failing asserts and internal checks. When storage bite arrays are accessed where the length is encoded incorrectly a panic is raised that's another change introduced. Finally, the `byte` type which used to be an alias of `bytes1` has been removed as of this version.

3.42 solc 0.8.0 Restrictions

`Solidity 0.8.0` also introduced several restrictions. Explicit conversions of multiple types are disallowed. Remember that explicit conversions are where the user forces conversions between certain types without the compiler necessarily thinking those are safe from a type safety perspective, so these explicit conversions being disallowed may be considered as a good thing from a security perspective. Address literals now have the type `address` instead of `address payable`, these have to be explicitly converted to `address payable`.

If required, the function call options for specifying the Gas and Ether value passed can only be provided once and not multiple times. The global functions `log0`, `log1` all the way to `log4` that may be used for specifying events or logs have been removed because they were low level functions that were considered as largely unused by `Solidity` contracts and, if a developer wants to use them, they need to resort to inline assembly. `enum` definitions now can't contain more than 256 members. This makes it safer because the underlying type is always `uint8`, so 8 bits that allows only 256 members to be represented by that type. Declarations with name `this`, `super` and `_` are disallowed. Transaction origin and message sender global variables now have the type `address` instead of `address payable`, and again require an explicit conversion where `address payable` is needed. The mutability of `chainID` is now considered `view` instead of `pure`, so all these different types of restrictions were introduced in this version that have an impact on security.

3.43 Zero-address Check

Also known as "*the first category of security checks*".

Remember that Ethereum addresses are 20 bytes in length and, if those 20 bytes all happen to be zeros which is referred to as a zero address that is treated specially in `Solidity` contracts and also in the context of the EVM, because the private key for a zero address is unknown so, if Ether or tokens are transferred to the zero address, then it is effectively the same as burning them or not being able to retrieve them in the future. Similarly, setting access control roles within the context of smart contracts to the zero address will also not work because transactions can't be signed with the private key of the zero address, because nobody knows the private key. Therefore zero addresses should be treated with a lot of extra care within smart contracts, and from a security perspective zero address checks should be implemented for all address parameters specifically for those that are user supplied in external or public functions.

3.44 tx.origin Check

Also known as "*the second category of security checks*".

Again, remember that Ethereum has two types of accounts: EOA and contract accounts. Transactions in Ethereum can only originate from EOA, so `tx.origin` is representative of the EOA address where the transaction originated from, so in situations where smart contracts would like to determine if the message sender was a contract or whether it was an EOA, then checking if the message sender is equal to `tx.origin` is an effective way to do it. There are some nuances in the usage of this, but at a high level this is a check that you may encounter in smart contracts and has security implications.

3.45 Arithmetic Check

Also known as "*the second category of security checks*".

We have talked about the concept of overflows and underflows. Just to refresh: where arithmetic is used with integers in `Solidity`, if the value of that integer variable exceeds the maximum that can be represented by that integer or goes below the lowest value that can be represented by that integer type, then it results in what is known as wrapping, where the value overflows from the maximum integer value of the type and becomes zero or underflows below the lowest value representable (which is typically zero) and becomes equal to the maximum value representable.

This can have big security implications because the values of those variables (maybe it's representing the balance of that account or something else within the context of the application logic) wraps around and becomes either zero or the maximum value, which can totally change the application logic that is working with them. So such overflows are underflows of balances or other accounting aspects related to such variables can and have resulted in critical vulnerabilities.

These checks until `Solidity 0.8.0` had to be implemented by the developers themselves, either within their own smart contracts or by using `OpenZeppelin`'s `SafeMath` library, which provided various arithmetic library functions for addition, subtraction, multiplication, division and so on... that implemented these checks in the library functions. `Solidity` recognize this aspect of arithmetic checks and how they are used all over in most of the smart contracts because nearly every contract deals with such integers and therefore these checks, as of version `0.8.0`, are checked by default for integer types. Furthermore, they can be overridden by the developer where that those checks are not necessary.

To sum it up: arithmetic checks are one of the most critical checks that one would encounter in `Solidity` contracts, and until version `0.8.0` you would see

a an extensive use of `SafeMath` library from `OpenZeppelin` for doing so. From `0.8 0` onwards, these are implemented by default.

3.46 OZ libraries

(Note that OZ stands for OpenZeppelin).

Most libariesthat you'll encounter inside smart contracts are written and maintained by `OpenZeppelin`, which is one of the leaders in the space in not only developing these libraries, but in security of Ethereum smart contracts. They provide multiple services and multiple tools in this context, so these `OpenZeppelin` libraries are widely used and have been time tested for several years now. Furthremore, they've also been optimized over time with respect to the Gas consumed by them and also with respect to the various `Solidity` versions that have been released over time.

One of the most common `OpenZeppelin` libraries is the `SafeMath` library that we discussed in the context of arithmetic checks. There numerous other `OpenZeppelin` libraries related to the implementation of token standards, various security functionalities, proxy contracts and utilities. You'll encounter one or more of these `OpenZeppelin` libraries when you're developing smart contracts as a developer or when you're auditing smart contracts for the security.

3.47 OZ ERC20

Let's start with the `OpenZeppelin` library that implements ERC20 token standard. This is perhaps the most popular, widely used and commonly seen token standard that you would encounter as a developer or as a smart contact security auditor. This library implements all the required functions specified by the token standard. It implements the name, symbol decimals, the total supply (that returns the amount of tokens in existence so far), the `balanceOf` function (that returns the amount of tokens owned by specific accounts), the `transfer` (and `transferFrom`, that help moving tokens from one address to another), the notion of allowance (which specifies a spender in addition to the owner of the tokens where the owner grants a certain allowance to the spender after which the spender can spend those tokens and send them to different other addresses) and in the context of allowance, there is the notion of increasing or decreasing allowance (that the owner implements for a specific spender). There are various extensions and presets and utilities related to these standards which we'll talk about next.

3.48 OZ ERC20 Util SafeERC20

One such utility related to the `ERC20` token standard is what is referred to as `safeERC20`. The `transfer`, `transferFrom`, `approve`, `increase` and `decrease allowance` functions of `ERC20` tokens are expected by the specification to return a `bool` value. Contracts implementing the standard which might choose not to return a `booln` deviate from the specification effectively. They may revert for these tokens on these functions under certain conditions or they may return no value. These differing return values, or exception handling in the case of `ERC20` tokens, have resulted in security vulnerabilities, therefore this `safeERC20` utility implements wrappers for these functions. It implements the safe version, so `safetransfer`, `safetransferFrom`, `safeapprove`, `safeincrease` and `safedecrease` relevance that always revert to failure after checking the different conditions for these functions. You may notice this utility being used with the contracts with the `using` directive of `Solidity` as `usign safeERC20 for IERC20`.

3.49 OZ ERC20 Util TokenTimelock

The next utility is what is known as token timelock. This implements a token holder contract where tokens are held by the contract and there is a specific address that is defined as the beneficiary address for all the tokens held by this contract, that are only released to that beneficiary address after a particular time has expired. The application for this as you can imagine are for things like token investing, where a certain number of tokens are allocated to the various team members: to the advisors and so on... that need to be claimable by them only after a certain point in time. This library implements the notion of a token, the beneficiary address and specifically a release function that, when triggered, checks if the block timestamp is greater than the release time that was declared earlier and if so, transfers the amount of tokens held by the contract to the beneficiary address.

3.50 OZ ERC721

The next one is the OpenZeppelin library that implements `ERC721` token standard. This is the token standard that is commonly referred to as NFTs or non-fungible tokens. It is perhaps the other widely used popular token standard besides `ERC20` that we just talked about. Unlike `ERC20` tokens, `ERC721` tokens are considered as non-fungible because every token is distinguishable from the other every token has a token id, nlike ERC20 tokens that are indistinguishable from each other. So this library implements all the required functions as per the specification: there's a `balanceOf` function (that returns a number of tokens in the specified owner address), there's a `orderOf` function (that returns the address that owns the specified token id), there are the `transferFrom` and `safetransferFrom` functions (that allow transferring tokens from one address

to another address; the `safetransferFrom` function makes certain checks before doing the transfer).

There are multiple checks implemented with respect to the zero address, the ownership of the tokens and specifically to check if the recipient is a contract account, and if so, if that contract recipient is aware of the `ERC721` protocol itself. This is done to prevent these tokens from getting locked in that address forever. Approvals with `ERC721` work differently from `ERC20`: unlike ERC20 (that has a notion of spender for the tokens), `ERC721` introduces the concept of an operator, which is somewhat similar. The approve function in this case specifies the address of the operator, the specific token id and it gives permission to the operator to transfer this particular token to another account. This approval is automatically cleared when the token is transferred and only a single account can be approved at any time, which means that approving the zero address clears the previous approvals. There are other functions associated with the `ERC721` as part of this library and there are also various extensions presets and utilities similar to the `ERC20` contract.

3.51 OZ ERC777

The next library is one that implements the `ERC777` token standard. This is a token standard similar to `ERC20`. It's backwards compatible with `ERC20`, so it implements a standard for fungible tokens and it's considered as implementing several improvements over `ERC20`. one of the key features is the notion of hooks, which are functions within the contract that are called automatically when tokens are being sent from it, or when tokens are being received. This allows the contract to control and reject which tokens are being sent and which tokens are being received. These features allow us to implement several improvements over `ERC20` such as avoiding the need for a separate `approve` and `transferFrom` transactions, which is considered as a significant user experience challenge for `ERC20` contracts. `ERC777` also allows one to prevent tokens from getting stuck in the contracts using the hooks feature. This also implements the decimals as being a fixed value of 18, so there's no need for the contract to set or change it. It introduces a notion of operators that are special accounts that can transfer tokens on behalf of others and it also implements a `send` function where, if the recipient contract is not aware of `ERC777` by not having registered itself as being aware, then transfers to that contract are disabled to prevent tokens from getting stuck in that contract.

3.52 OZ ERC1155

`ERC1155` is another token standard that allows a contract to manage tokens in a fungibility agnostic and Gas efficient manner, so a single contract that implements a standard can manage multiple tokens, some of which can be fungible

tokens like `ERC20` or NFTs. All these are managed within a single contract: this means that a single transaction can manipulate multiple tokens within that transaction. This makes it very convenient from a user experience perspective. It also makes this standard very Gas efficient. This standard specifically provides two functions: `balanceOfBatch()` and `safeBatchtransfersFrom()` that allow querying balances of multiple tokens and transferring multiple tokens in the same transaction. This makes the management of these tokens within the contract very simple and Gas efficient.

3.53 OZ Ownable

The oracle library of `OpenZeppelin` allows a smart contract to implement basic access control by introducing the notion of the owner for a particular contract. The default owner is the address that deployed the contract, this allows the smart contract to implement access control on special or critical functions that modify critical parameters within that contract to only be accessible by this owner address. This is made possible by the modifier `onlyOwner` within this library. This library also supports the transferring of ownership where a new owner can be specified to be switched over from the existing owner. There's also the `renounceOwnership` where the ownership is set to the zero address, which essentially makes all the only owner functions uncallable thereafter.

3.54 OZ AccessControl

`OpenZeppelin` provides a second library to implement a more flexible access control known as role based access control or RBAC for short. This allows a contract to define different roles that are mapped to different sets of permissions, and by using the `onlyRole` modifier, access to different functions can be restricted to specific roles. Every role also has an associated `admin` with it that can grant and revoke those roles. So unlike `ownable` which implements a very basic access control using the notion of an owner address and all other addresses, this library allows for a more flexible role-based access control.

3.55 OZ Pausable

The pausable library from `OpenZeppelin` is interesting from a security perspective because it allows teams to execute what is known as a "*guarded launch*". What this means is that when the team is launching a new project with smart contracts, it's good for the team to anticipate potential emergencies that could arise and using this functionality of the pausable library, they can pause the smart contracts to deal with the emergency, remediate any risks and then unpause the contract to continue normal operations. This is made possible using the `pause` and `unpause` functions that can be triggered by authorized

accounts.

These functions allow the authorized accounts to pause the contract and unpause it at the desired times. The way this works is by using the `whenPaused` and `whenNotPaused` modifiers on different functions. So in all functions that should be callable during the normal operations of the contract, the `whenNotPaused` modifier should be used and for those functions that should still be callable during emergencies, the `whenPaused` modifier should be used. Effectively, this library allows project teams to implement a circuit breaker mechanism to deal with any vulnerabilities discovered in the contract or to also deal with exploits that are happening with the contracts when they can use the `pause` functionality, pause the contracts and all the user interactions with the contract, mitigate the risk from that emergency, if possible, then resume normal operations by unpausing the contract.

3.56 OZ ReentrancyGuard

The other OpenZeppelin library that is very critical to security is the reentrancy guard library. This is used to mitigate the risk from re-entrancy vulnerabilities that are somewhat unique to smart contracts and very dangerous. This is the vulnerability category that was exploited during the DAO hack, which has historical significance to Ethereum.

Reentrancy vulnerability is: if our smart contract is making an external call to any function of an external contract where that external contract is potentially untrusted (it is not one of our own contracts and it's been deployed by some other project team), then in such cases those external contracts can make a nested call to our contract. So they can re-enter our contract function (the function that made that external call or any other function) and in cases where certain contract state has not been updated within our contract, that aspect can be exploited by this nested call to do things such as transferring tokens multiple times or triggering logic multiple times where in fact it should have been able to do that only one time.

This at a high level is known as a reentrancy attack because because of the concept of re-entering or nesting that happens, that can be exploited in many different ways. This particular library introduces a modifier called `nonReentrant` and when this modifier is applied to different functions in our contract, those functions can't be re-entered after making an external call. This aspect can be used to mitigate reentrancy risk and is one of the standard security best practices that is recommended. Note that all these security features implemented in these different libraries where specific modifiers need to be used for implementing those checks, all these aspects are applicable only on functions that use those modifiers. So just by using those libraries in those contracts we do not get the security benefits. Those benefits are realized only on functions where

this modifier is used in the expected manner.

3.57 OZ PullPayment

`OpenZeppelin` implements a pull payment library that is relevant in the context of payments. Payments between two contracts can be done either by the paying contract (by pushing the payment to the receiver account) or the receiving contract (by doing a pull of the payment from the paying contract). This is interesting in the context of avoiding re-entrancy attacks, so in the case of the pull payment library, the paying contract makes no calls on any of the functions of the receiver contract because the receiver contract may be potentially malicious, and it's better for that receiving contract or account to withdraw the payment itself by using the notion of pull. This prevents reentrancy by favoring the pull payment as opposed to the push payment and therefore is a standard security best practice that is recommended.

3.58 OZ Address

The `OpenZeppelin` address library implements a set of functions related to the address type. The first one is the `isContract` function that we often encounter within different smart contracts. It takes an address and a contract as parameters and returns a `booln`. This function returns `true` if the account address is a contract. However, if it returns `false`, then it is not safe to assume that the specified address is an EOA. The reason for that is because `isContract` will return `false` in 4 different situations: one, if it is an EOA. Two, if it is a contract account that is in construction (so within the constructor of that contract account). Three, if it is an address where a contract will be created and four, if the address specified had a contract in it, but was later destroyed.

So for all these 4 cases, this function will return `false` and an EOA is only one of the four reasons, so this is something where contracts using this function typically make incorrect assumptions about what this function does and something that has to be paid attention from a security perspective.
The second function is `sendValue`. Remember that `Solidity` has a `transfer` primitive that sends wei to a recipient contract, but limits the Gas supplied to 2300 Gas units. This has the drawback that if the Gas Cost of certain opcodes changes (for example, increases over time) then the 2300 subsidy is not going to be sufficient for some of the logic that would be implemented within the `fallback` function of that contract. So the `sendValue` function removes this 2300 limitation and forwards all the available Gas to the callee contract. This library further implements wrappers around the low-level call primitives supported by `Solidity`, so for `call`, `staticcall` and `delegatecall` primitives, there are equivalent wrappers that are considered as safer alternatives to

using these low primitives directly (`functionCall`, `functionCallWithValue`, `functionStaticCall`, `functionDelegateCall`).

3.59 OZ Arrays

The `OpenZeppelin` arrays library implements array related functions. There is a `findUpperBound()` function that takes in a `uint256` array along with the `uint256` element. The array is expected to be sorted in ascending order with no repeat elements in it, and it returns the first index in that array that contains a value greater or equal to the specified element. If there is no such index which means that all the values in the array are strictly less than the element, then in those cases the length of the array itself is returned.

3.60 OZ Context

The context library provides current execution context, specific to the `msg.sender` and `msg.data` primitives. Remember that these parameters are provided by `Solidity` in situations where our smart contract is working with what are known as meta transactions, where the account sending the transaction and paying for the Gas costs may not be the actual user as far as our applications context is concerned. In such situations, which happen where there are relayers between the user and our smart contract, the functions implemented by this library help us distinguish between the users context and the relayers context.

3.61 OZ Counters

There's a simple counters library that allows a contract to declare new counters, increment and decrement them. This is useful for doing things like tracking the number of mapping elements for `ERC721` token IDs or for request IDs depending on the application context. There are different functions that let the contract get the current value of a counter, reset it to zero, increment and decrement the counter by one.

3.62 OZ Create2

`OpenZeppelin` has a create2 library that provides library functions to use the `create2` EVM opcode functionality in an easier and safer manner. Remember that EVM has two instructions: `create` and `create2` that allow contracts to programmatically create other contracts. This is in contrast to creating contracts by sending a transaction to the zero address so, if we think of this as a deployer contract that is creating a newly deployed contract, then the create opcode uses the address of the deployer contract along with the state

of the deployed contract in the form of the nonce of that contract account to determine the address of the newly deployed contract.

Contrast to this the `create2` opcode. It does not use the state of the deployer contact at all. Instead it only uses the byte code of the newly deployed contract along with a value provided by the deployer contract (known as the salt), to determine the address of the newly deployed contract. Because of this change, the address of the newly deployed contract becomes deterministic. In this case the deploy library function uses 3 parameters: the amount, salt and byte code to create and deploy a newly deployed contract. The amount is the amount of the Ether balance the newly deployed contract will start off with, if one only wants to determine the address of the new contract without actually deploying it, there is a library function called the `computeAddress` that helps one to do that and, if one wants to compute the address of this contract, if it is going to be deployed from a different deployer address, then there's a different library function `computeAddress` that takes an additional parameter which is the address of the deployer.

3.63 OZ Multicall

`OpenZeppelin` provides a multi-call library that allows a smart contract to batch multiple calls together in a single external call to this contract. This function is `multicall`. It takes in a single data parameter and it returns a bytes array of all the return parameters from those multiple points. It helps the contract to receive and execute multiple function calls in a batch. The benefit of this is that it is less overhead and makes it more Gas efficient because all these multiple calls are now packaged in a single call within the same transaction of the same block.

3.64 OZ String

`OpenZeppelin` provides a string library that allows one to perform some basic string operations, there is a `toString` function that converts a `uint256` to its `ASCII` string decimal representation. There is a `toHexString` that converts it to an `ASCII` string hexadecimal representation and finally, there is a `toHex String` that takes in a length parameter that converts a `uint256` to a hexadecimal representation with a fixed length.

3.65 OZ ECDSA

`OpenZeppelin` provides an ECDSA library. Remember that ECDSA signatures are used very commonly in Ethereum smart contracts. The signature itself has three components v, r and s which are `bytes1`, `byte32` and `bytes32` in length respectively, making the signature 65 bytes. The EVM has an `ecrecover`

opcode and `Solidity` has a similar primitive that supports this opcode. But that opcode allows for what are known as malleable (or non-unique signatures if you remember).

This library prevents that by providing a library function recovered that is not susceptible to this malleability. The way that it's made possible is that this function requires the s value that signature to be in the lower half order, the v value to be either 27 or 28, so this becomes important depending on how the smart contract is using the signatures and, if malleability is a concern or a risk, for that use case the `ecrecover` function takes in the hash of the message (the signature component of that message) and returns a signer address. To sum it up, the EVM `ecrecover` is malleable which may be a concern depending on how the signature is being used in the smart contract logic. This library provides a non-malleable way of using `ecrecover`.

3.66 OZ MerkleProof

The MerkleProof library provides functionality to help with the verification of Merkle tree proofs. Remember that Merkle trees are data structures where the leaves contain the data and all the other nodes in the tree contain a combination of the hashes of their two child nodes. This library provides a verify function that takes in three parameters: the leaf, the root, the proof, and returns a `bool` value which is `true` if the leaf parameter can be proved to be a part of the Merkle tree defined by the root parameter. In order to do that, a proof must be provided to this function that contains all the sibling hashes on the branch from the leave to the root of the tree. This is an interesting library that is used often where Mertkle tree proofs are required within smart contracts.

3.67 OZ SignatureChecker

The SignatureChecker library provides functionality that allows smart contracts to work with both ECDSA signatures and `ERC1271` signatures. We've talked about ECDSA signatures that are signatures that can be created with the use of a private key which is possible only with EOAs. The reason for this is that contracts can't possess a private key because all contract state is public. `ERC1271` allows the concept of contract signatures in in a manner that is different from ECDSA signatures. This library becomes interesting for applications such as smart contact wallets that need to work with the contract signatures and ECDSA signatures.

3.68 OZ EIP-712

There is an `EIP712` library that provides support for the hashing and signing of typed structured data as opposed to binary blobs. This supports the notion

of an `EIP-712` domain separator. The source code of this library this is again often used in smart contracts and from a security perspective, what becomes interesting here is whether this signature includes the chainID of the chain where the smart contract is deployed and being executed and whether this also includes the address of the smart contract itself. Not using these two values within the signature can allow replay attacks, if the contact is redeployed to some other address on the same chain or to a different chain.

3.69 OZ Escrow

The Escrow library allows a smart contract to hold funds for a designated payee until they withdraw them. The contract that uses this as the payment method is its owner and it provides three functions to allow this functionality: there is the `depositsOf` function that returns the the amount of the funds designated for the payee, there are the `deposit` and the `withdraw` functions themselves that are only callable by the owner.

3.70 OZ ConditionalEscrow

The ConditionalEscrow library is derived from the Escrow library and as the name says it only allows withdrawal if a particular condition is met. The withdrawal allowed function checks for this condition and returns `true` or `false`, if it is met or not. The withdrawal function itself is of public visibility and does not have the `only owner` modifier here, but it checks the withdrawal allowed condition and if that is met it calls the base contract's withdraw function that has the `only owner` modifier.

3.71 OZ RefundEscrow

The refunds Escrow library is further built on top of the ConditionalEscrow library that we just discussed, this allows holding funds for a beneficiary that are deposited from multiple parties multiple depositors, this contract has three states in which it can be it has the active state refunding state, the closed state active state is when deposits are allowed to be made by the multiple depositors refunding is where refunds are sent back to the depositors and finally, there is a closed state in which the beneficiary can make the withdrawals.

3.72 OZ ERC-165

The `ERC165` library allows one to determine if a particular contract supports a particular function interface. This runtime detection is implemented using a lookup table. It provides two functions: the first one is `_registerInterface` and is used for registering function interfaces. The second one,

supportsInterface, is to determine if a particular interface is supported which returns a bool either true or false.

3.73 OZ Math

OpenZeppelin provides a math library that has some basic standard math utilities that are missing in the Solidity language itself. There's a max function that returns the maximum of two uint256 values. There's a minimum that provides the minimum of those two values. Then the average function that returns the average of those two numbers, which is rounded towards zero.

3.74 OZ SafeMath

Then there is the SafeMath library which we have talked about earlier. It provides the basic math functions that are safe from overflow and underflow conditions because of wrapping. This has support for add, sub, mul, div and mod functions. The typical usage is done via the using directive where you would see something like using SafeMath for uint256 where the SafeMath library functions are applied to all variables of type uint256 in that contract. There are the try variants of these functions where instead of reverting, if the overflow and underflows happen a flag is returned. This is useful for exception handling, so this SafeMath library is almost absolutely required for smart contracts that deal with integers and use a Solidity compiler version below 0.8.0 (because remember that Solidity 0.8.0 introduced default overflow and underflow checked arithmetic).

3.75 OZ SignedSafeMath

SignedSafeMath library provides the same mathematical functions as SafeMath, but for signed integers. The only operation that is missing is the modulus operation which does not make sense for signed integers. The motivation for this is the same as SafeMath.

3.76 OZ SafeCast

Remember that Solidity allows both implicit casting of types and explicit casting between types. Explicit casting is where the developers can force the compiler to cast one type into another type where the compiler may not be able to determine that it is safe to do. So in cases where the developers want to do what is known as downcasting, the OpenZeppelin SafeCast library provides various functions to do so in a safe manner.

Downcasting is when the developer wants to cast a source type into a target type where the target type has fewer storage bits to represent it than the source type. In such cases, because the target type has fewer storage bits, it may not always be safe to do so. If the variable of that type actually requires the storage bits being reduced from the source type to destination type. SafeCast library provides functions that allow the developer to determine if that downcasting is safe and if not, it raises an exception by reverting the transaction. There are various functions to safely downcast from unsigned integers of 256 bits to 224, 128, all the way to 8 bits. Similarly, there are functions for signed integers to do so as well, so these functions become very useful for developers when they're doing downcasting to prevent overflows because of doing so.

3.77 OZ EnumerableMap

Remember that the mapping types and `Solidity` can't be enumerated for all the keys and values that they contain. The EnumerableMap library of `OpenZeppelin` allows a developer to create and use EnumerableMaps. Adding and removing entries from this mapping type can be done in constant time. Checking for existence of entries can also be done in constant time. Enumerating the maps can be done in $\mathcal{O}(n)$, n is the size of the mapping. As of the latest version, the only supported mapping type is the one where keys are of `uint256` and the values are of `address` type.

3.78 OZ EnumerableSe

The EnumerableSet library allows the developers to use enumerated sets. There are various functions that are provided to manage the sets, adding and removing entries to the set and checking entries for existence. Again can be done in $\mathcal{O}(1)$ time (that's constant time). Enumerating them can be done in $\mathcal{O}(n)$ time. As of the latest version, the only supported set types are those that contain `bytes`, `address` or `uint256`.

3.79 OZ BitMaps

Bitmaps are commonly encountered data structures in computer science, where every bit of the underlying type can be thought of as representing a different variable. the BitMaps library maps a `uint256` type to `bool` types, where this bitmap can be used to represent 256 different `bool` values within that single `uin256` type. This library allows developers to do that in a very compact and efficient manner. The library provides 4 different functions to operate on these BitMaps: the `get` function (returns the `bool` value at a particular index of the bitmap), the `setTo` function (allows us to set the value at a particular index of the bitmap to the specified value), the `set` function (sets the value of the

bitmap at that index to 1) and `unset` sets the value of the index at that bitmap to 0.

3.80 OZ PaymentSplitter

The PaymentSplitter library provides functions that allow us to split Ether payments among a group of accounts. The sender, who sends Ether to this contract that uses this library does not know about the splitting aspect, so it is sender agnostic. The splitting can be done in equal proportions or in an arbitrary manner.

This is done by assigning a particular number of shares to every account. That account can later claim an amount of Ether that is proportional to the percentage of the total shares that they were assigned. This follows the PullPayment model that we have discussed earlier, which is much safer from a security perspective than a PushPayment model.

3.81 OZ TimelockController

The TimelockController library provides library functions for enforcing `Timelocks`. `Timelocks` are nothing, but time delayed operations. If there are operations that need to be executed only after a certain window of time delay has passed or occurred, that is referred to as `Timelock`. This library provides various functions to enforce `Timelock` on `onlyOwner` operations. `OnlyOwner` here refers to the modifier for access control which when applied to functions allows only the Owner of that smart contract to execute that function. This becomes critical from a security perspective because `onlyOwner` operations are used in smart contracts to make changes to critical parameters of that protocol or project.

They're also used on functions that enforce or change access control for that smart contract, so in all these scenarios, if we want to give the users who interact with the smart contract an opportunity to notice these operations that are making these critical changes, then decide if they would like to continue engaging with the smart contract or if they would like to exit from engaging with the smart contract by removing the funds from the smart contract or some other logic, then `Timelock` becomes useful for providing a mechanism to do so.

This library provides various functions that help us schedule, delay, execute, cancel such operations or do, so in batches all in a `Timelock` specific manner. There are also functions that let us query, if an operation is pending, if it is ready, if it is already done in the context of the `Timelock` and one can also update the delay that is specific to the `Timelock` operation.

3.82 OZ ERC2771Context

Remember that we talked about the context library earlier, that provides support for what are known as Meta-transactions. This `ERC2771` context library provides a variant of that library, that's specific to `ERC2771`. At a high level, there is a **transaction signer** who originates transactions, by signing it from an EOA, and sends this signed transactions to a relayer off-chain, this relayer is responsible for paying the Gas. `ERC2771` specifies a secure protocol for a particular contract to accept such Meta-transactions. This protocol is concerned about the Gas layer from forging, modifying or duplicating the requests that are sent by the transaction signer.

It specifies four different entities there's a **transaction signer**, who signs and sends a transaction off-chain to the Gas layer, **the Gas layer** receives these transactions and is expected to pay for the Gas, then forward it to a **trusted forwarder** contract on-chain, which is further responsible for verifying the assigned transaction to look at the nonce, the signature and make sure they are correct.

Then finally, forward that verify transaction to the **contract** that is the ultimate destination for the transaction. So this protocol is defined by this `ERC`, the library provides various functions to help with it.

3.83 OZ MinimalForwarder

The MinimalForwarder library provides support for implementing the trusted forwarder that we discussed in the context of the `ERC2771` Meta-transactions. It implements a very simple MinimalForwarder that verifies the nonce and signature of the forwarded transaction before calling the destination contract and it does.

So with two functions, the `verify` function for verification of nonce and signature; and the `execute` function for executing the specific function on the destination contract.

3.84 OZ Proxy

`OpenZeppelin` provides support for different libraries that help with proxies. At a high level the Proxy setup requires two contracts: the Proxy contract, and what is known as the implementation contract.

The Proxy contract receives the calls from the user, and forwards it to the implementation contract, this forwarding is done via `delegateCall`. In this setup the Proxy contract is typically the one that holds the contract state, the implementation contract is the one that implements the logic. So when the

forwarding is done via `delegateCall`, the implementation logic executes that logic on the state held in the Proxy contract.

As you can imagine this has to be done in a very careful manner because it can lead to a variety of security issues, there are many many articles that have been written on this topic by `OpenZeppelin` and also by Trail of Bits and other security firms.

So coming back to this particular library, it provides a `fallback` function, that forwards the call to an implementation. It also provides a `delegate` function, that allows one to specify, the delegation to a specific implementation contract. This also allows us to specify a hook, via the `beforeFallback` function, that gets called before falling back to the implementation.

3.85 OZ ERC1967Proxy

The `ERC1967` Proxy library helps us implement what are known as upgradable proxies. These are upgradable because the implementation contract that sits behind the Proxy can be changed to point to a different implementation contract.

Remember the Proxy setup where the application state is held in the Proxy contract, the logic may be implemented in the implementation contract. So, if you want the logic to change for whatever reason maybe to fix a bug, in the current implementation or to enhance and add more logic, upgradeable proxies are one way to do so.

In this case, the address of the implementation contract that can be changed is stored in the storage of the Proxy contract. This specific storage location is specified by the EIP, so that it does not conflict with the layout of the implementation contact that sits behind the Proxy.

The address of the logic or the implementation contract can be specified as part of the constructor, the address of the new implementation can be provided while upgrading using the upgrade function. So upgradeable proxies are something that we encounter commonly in smart contracts, this again has to be done in a very careful manner because it can lead to security issues such as the storage conflict that is specified here.

3.86 OZ TransparentUpgradeableProxy

Another Proxy related library is the TransparentUpgradeableProxy. This helps one implement a Proxy that is upgradable only by an admin. It specifically

helps us mitigate the risk due to attacks from **Selector Clash**.

What this means is that, if a function is present both in the Proxy and the implementation such that their selectors, their **function selectors clash** or they evaluate to the same value, then that could lead to problems, because if there is a function call to that function, then it will not be clear if the function should be executed in the context of the Proxy contract or, if it should be forwarded to the implementation contract.

So this library specifies that all function calls coming from the Non-Admin users will be forwarded to the implementation contract even, if those calls match the function selected of the Proxy contract. Similarly, the function calls made by the admin users are restricted to the Proxy contract, they are not forwarded to the implementation contract.

This allows for clean separation where the admin functions are restricted to the Proxy contract and Non-Admin functions are forwarded to the implementation contract. So the admin can do things such as upgrade the implementation contract or create the admin address itself.

3.87 OZ ProxyAdmin

The ProxyAdmin library is meant to be used as the admin of the TransparentUpgradeableProxy that we just discussed. It provides support for various functions that are required by the admin, these include:

- The `getProxyImplementation()` which returns the implementation contract address.
- The `getProxyAdmin()` which returns the admin address.
- `changeProxyAdmin()`, that changes the ProxyAdmin.
- `(upgrade(proxy, implementation)`, that upgrades the implementation contract pointed to by the Proxy.
- The `upgradeAndCall(proxy, implementation, data)` function that both upgrades implementation, then makes a call to that new implementation.

3.88 OZ BeaconProxy

The BeaconProxy library allows one to implement a Proxy where the implementation address is obtained from a different contract known as a beacon contract. That beacon contract itself is upgraded:

`Implementation Addr -> UpgradeableBeacon`

The address of the beacon contract is stored in the Proxy storage at a slot specified here, again that is specified by `EIP1967`:
`Beacon Address -> Slot uint256(keccak256('eip1967.proxy.beacon')) -1`.

The constructor can be used to initialize where the beacon contact is located. There are functions that allow us to get the address of the beacon the address of the implementation: `Constructor -> Beacon Init`, `_beacon() -> Beacon Addr`.

Finally, to set the beacon contract to a different address than what was initialized: `_implementation()`, `_setBeacon(beacon, data)`.

3.89 OZ UpgradeableBeacon

The UpgradeableBeacon library provides support for implementing the beacon contract in the context of the BeaconProxy that we just discussed. The Owner of this contract can change the implementation contract that this BeaconProxy points to. The initial implementation contract is specified in the constructor, the Owner is the one who deployed the contract.

There are functions that allow one to determine what that implementation contract is and also to upgrade it to a new implementation: `_implementation()`, `upgradeTo(newImlementation)`.

3.90 OZ Clones

The OZ Clones library helps one implement what are known as minimal Proxy contracts as specified by `EIP1167`. In this case all the implementation contracts are clones of specific byte code, where all the calls are delegated to a known fixed address. The deployment can be done in a traditional way using create or it can be done in a deterministic way using `CREATE2`.
Corresponding to these two deployment options, there are two functions:

- There's the `clone(implementation)` function that clones that implementation and returns the address of the instance deployed using create
- There is the equivalent version for `CREATE2` the `cloneDeterministic(implementation, salt)` that takes in the implementation, the sort and returns the instance of the clone that was created.

3.91 OZ Initializable

The OZ Initializable library provides critical functionality that is required for applications that work with Proxy contracts.

Remember that in the Proxy setup we have a Proxy contract that forwards all the calls to an implementation contract. The Proxy contract maintains the data or the application state and delegates the calls to the implementation contract, which implements the logic that works on the application state maintained by the Proxy contract. So in this setup ,if there are functions in the implementation contract that need to work with certain initialized values, then all such initialization should not be done in the constructor of the implementation contract, because the constructor would modify the state of the implementation contract which is never used in this setup.

So all this initialization is expected to be moved to a different function, which is typically called the initialize function that has an external visibility, this initialized function is expected to be called by the Proxy contract. This aspect of not using constructors for initialization, but using a separate initialize function applies not only to the implementation contract, but to all the base contracts that it derives. This initialization should be performed only once and should be performed immediately after the implementation contract is deployed, either from a deploy script or from a factory contract.

The OZ Initializable library provides an initializer modifier, which when applied to this initialize function allows that to be called only once. So these concepts of the Proxy setup, the fact that the implementation contract should not be using a constructor, but instead an OZ Initializable library function that needs to be called immediately after deployment, more importantly needs to be called only once.

These are very critical from a security perspective there have been multiple vulnerabilities reported because of this not being followed multiple exploits and something that therefore needs to be paid very careful attention to.

3.92 Dappsys DSProxy

We now move on to a different set of libraries provided by the DAppSys teams at DappHub. These are used commonly in smart contracts as an alternative to the OpenZeppelin libraries that we have discussed.

The first one is the DAppSys DSProxy this implements a simple Proxy that is deployed as a stand-alone contract and can be used by the Owner to execute the code the logic that is implemented in the preventation contract. The user would pass in the contract byte code along with the function call data, the call

data remember that it specifies the function selector of the function to be called along with the arguments for that function. This library provides a way for the user to both create the implementation contract using the bytecode provided, then delegating the call, to that contract, the specific function, the arguments as specified in the call data. There are associated libraries related to DSProxy that help implement a factory contract as well as some caching mechanism.

3.93 Dappsys DSMath

Dappsys provides a DSMath library that provides math parameters for arithmetic functions. The first set of primitives are arithmetic functions that can be safely used without the risk of underflow and overflow, these are equivalent of the SafeMath library from OpenZeppelin this has the `add`, `sub`, `mul` functions. There is no `div` function because the `Solidity` compiler has built in divide by zero checking. DSMath also provides support for Fixed-point map.

It introduces two new types: the `Wad` type, the `Ray` type. The `Wad` type is for decimal numbers with 18 digits of precision while the `Ray` type is for decimal numbers with 27 digits of precision. There are different functions that help one operate on the `Wad` and `Ray` types.

3.94 Dappsys DSAuth

The DSAuth library provides support for developers to implement an authorization pattern that is completely separate from the application logic. It does so by providing an auth modifier that can be applied to different functions and internally this modifier calls the `isAuthorized()` function that checks, if the message sender is either the Owner of this contract or the contract itself. this is the default functionality. This can also be specified to check, if the message center has been granted permission by a specified authority. We'll talk about this aspect of authority in the next slide.

3.95 Dappsys DSGuard

The DSGuard library helps implement an access control list or ACL. This is a combination of a source address destination address and a function signature.

This library can be used as the authority that we just discussed in the context of the DSAuth library. This implements a function `canCall()`that looks up the access control list and determines, if the source address can call the function specified by the function signature at the destination address. So it's a combination of the source, destination and the signature that determines the value of the `bool` that's either `true` or `false`: `[src][dst][sig] => boolean`.

When used as an authority by DSAuth, the source refers to the message sender, the destination is the contract that includes this library, the signature refers to the function signature.

3.96 Dappsys DSRoles

The DSRoles library provides support for implementing role-based access control this is something we discussed in the context of the OpenZeppeline AccessControl library as well. In this case it implements different access control lists, that specify roles and associated capabilities. It provides a `canCall()` function that determines, if a user is allowed to call a function at a particular address by looking up the roles and capabilities defined in the access control list.

RBAC is implemented via mechanisms, there is a concept of root users, who are users. Allowed to call any function regardless of what roles and capabilities are defined for that function. There's a concept of public capabilities that are global capabilities that apply to all users. Finally, there are role specific capabilities that are applied when the user is not the root user and the capability is not a public capability.

3.97 WETH

Let's now talk about WETH. Protocols often work with one or many `ERC20` tokens either their own or of other protocols. They also work with the Ether that is sent to their smart contracts via message value. Instead of having two separate sets of logic and two separate sets of control flow within their contracts, one to deal with Ether, the other to deal with `ERC20` tokens, it would be very convenient, if we could have a single logic single, control flow to deal with both Ether and `ERC20` tokens.

The WETH concept provides this capability. It allows smart contracts to convert Ether that's been sent to their contracts to its `ERC20` equivalent which is known as WETH. This conversion is a process called wrapping, the other direction of converting the `ERC20` equivalent of WETH back to Ether is called unwrapping.
This is made possible by sending the Ether to a WETH contract which converts it into its `ERC20`equivalent at a one is to one ratio. There are multiple versions of WETH contracts the most popular right now, is the `WETH9` [14] contract which holds anywhere between 6.5 to 7 million Ether as of this point. There are also some improvements being done, there is `WETH10` that is more Gas efficient than the version 9, this version also supports flash loans as per the `EIP3156` standard. So this WETH concept is something that we often come across in smart contact applications.

3.98 Uniswap V2

Uniswap is an automated market making protocol on Ethereum. That's powered by what is known as a constant product formula:

$$xy = k$$

where `x` and `y` are token balances of two different tokens and `k` is their constant product.

Uniswap allows liquidity providers to create pools of token pairs, and whenever anyone provides liquidity to either of the two tokens of the token pair, new tokens known as LP tokens liquidity provided tokens are minted and sent back to the liquidity provider. This represents their share of the liquidity in the tokens.

Uniswap is the most popular protocol on Ethereum currently for swapping between tokens belonging to a token pair, and a big part of that is because of the simplicity of the constant product formula as determined by the curve $xy = k$.

Uniswap also provides support for on-chain Oracles. A price Oracle is a tool that allows smart contracts to determine the price information about a given asset on the blockchain. In the case of Uniswap V2 every token pair, measures the price of further tokens at the beginning of each block.
So in effect this is measuring the price at the end of the previous block that is maintained within a cumulative price variable that's weighted by the amount of time this price has existed for the token pair. This particular variable can be used by different contracts on the Ethereum blockchain to track what is known as "*time weighted average prices*" or TWAPs across any particular time interval.

3.99 Uniswap V3

Uniswap recently introduced their version 3 of the protocol, which is considered as a big improvement over their version 2. This improvement is specifically around the concept of concentrated liquidity. What this means is it allows liquidity providers to provide liquidity for the token pair, across custom price ranges instead of across the entire constant product curve.

This brings about a big improvement to their **capital efficiency**. This version of the protocol also introduces flexible fees across different values as shown here.

Finally, for Oracle support version v3 introduces advanced TWAP support where the cumulative sum instead of being maintained and trapped in one variable is now done, so in an array. This allows smart contracts to query the TWAP on demand for any period within the last 9 days.

3.100 Chainlink

Chainlink is perhaps the most widely used Oracle and source of price feeds for smart contracts on Ethereum. Price data and even other kinds of data are taken from multiple off-chain data providers and they are put on-chain to create these feeds by the decentralized Oracles on the chainlink network.

Chainlink has mechanisms for aggregating this data across the various data providers and itself provides an extensive set of APIs for working with these Oracles and price feeds.

Chapter 4

Security Pitfalls & Best Practices 101

4.1 Solidity versions

The `Solidity` language has evolved considerably in the last several years. There have been many features added, some of them removed. Security has been improved in several cases, optimizations have been made.

As a result, there are many versions of `Solidity` that are available for projects and developers to choose from. At least one version is released every few months that make some optimizations and fixes some bugs a couple of breaking changes are introduced every year or so. As a result, the question is always about which version of the `Solidity` compiler to use for a particular project, so that the best combination of features and security aspects are considered.

The older compiler versions are time tested, but they have bugs. The newer versions have the bug fixes which is good, but they may also have new bugs which have been undetected so far.

The older versions have lesser features compared to the newer versions, that have more features. Some of these are language level features that are visible syntactically. Some of them are semantic changes. Some of them are security features. Some of them are optimizations that are not very visible.

As a result, the choice of an optimal version of the compiler for a particular project is always a tricky thing. This has to take account not just the functionality, but also the security aspect. As a result, there is a trade-off to be made, there are risks as well as rewards. As of this point many of the projects are transitioning to the `Solidity` version 0.8.0 and beyond, because among other things, this version has introduced default arithmetic checks for underflow and

overflows. So these aspects of security and functionality, the range of choices available across the various `Solidity` compiler versions, have to be kept in mind when determining which version to use for a particular project.

4.2 Unlocked Pragma

We remember that `Solidity` supports the concept of `pragma` directives and one of them is related to the `Solidity` compiler version, that can be used with this smart contract.

There are many aspects related to that fragment directive, but the one that is relevant from a security perspective, is the concept of that `pragma` being unlocked or floating and what this means is that in the `pragma` directive that specifies the compiler version, if the `^` symbol is used, then it is referred to as being unlocked. What this means, is that the use of this caret symbol, specifies that any compiler version starting from the one specified in that `pragma` directive all the way to the end of that breaking version can be used to compile this smart contract. As an example, if the `pragma` directive is `^0.8.0` it means that any compiler version from `0.8.0` all the way to the last version in the `0.8.z` range can be used according to this `pragma` for compiling this smart contract.

This becomes interesting from a security perspective. The use of such an unlocked or floating `pragma` allows one `Solidity` compiler version to be used for testing, but potentially, a different one that is used for compiling the contracts while being deployed. This aspect of using a different version for testing and deployment is risky from a security perspective. That's because one could test with a totally different set of compiler features and security checks, the newer version or a different version that is used for deployment may support a different set of features and a different set of security checks, so this mismatch between testing and deployment is allowed by the use of this unlocked `pragma` and hence is not recommended to be used.

So what is recommended is to `lock` the `pragma` by not using the caret symbol in that `pragma` directive, this will enforce that the same compiler version is used for testing as well as for deployment.

4.3 Multiple Pragma

Another security aspect related to the use of the solution compiler `pragma` in contracts is the use of different pragmas across different contracts.

Remember that the `pragma` applies only to the contract where it is used so. If there are different multiple contracts that are used within a single project, then

each one of them could have a different `pragma` specifying a different compiler version.

The reason why this is not recommended is because these different compiler versions like we just discussed can have different bugs, different bug fixes, different features and even different security checks across the versions. This will result in different components of the application having different security properties which is not desirable.

So from a security perspective, what is recommended is to use the same `pragma` across all the different contracts that form that smart contract application. This will result in all of them having the same set of bugs, features and security checks which can be accounted for while one is testing that smart contact application.

4.4 Access Control

Access control is perhaps the most significant and fundamental aspect of security. When it comes to smart contracts, what it means is access to functions.

Remember that functions can have different visibility. `public` and `external` functions are those that can be called by any user interacting with the smart contract.

So from an access control perspective, we need to make sure that the right set of addresses can call these functions. We need to ask know if it might be okay for anyone to access these functions, any address to access this function, or it might be required only for the Owner to access this or there could be an extensive role based access control that is desirable as well.

This means that when we are reviewing smart contracts for security, we need to make sure that the right access control is enforced by the use of the correct modifiers. That make sure that the correct checks are enforced on the different sets of addresses used with the smart contract. Any of these missing checks either missing modifiers or the use of incorrect addresses or even the access control specification might allow attackers to control critical logic that is executed within some of these critical functions.

4.5 Withdraw Funds

Smart contracts typically manage a significant amount of funds related to the amount of Ether or the `ERC20` tokens that they hold and manage it in different ways for different users. So they have different functionality for users to deposit

these funds and similarly they have different mechanisms for users to withdraw their funds.

These withdrawal functions need to be protected, from an access control perspective. What this means is that, if these withdrawal functions are unprotected, that's if they are public and external and they do not have the right access control enforced on the different addresses via checks implemented within the modifiers applied on these functions, then it may let attackers call these unprotected withdrawal functions and withdraw Ether or `ERC20` tokens that belong to other users. This unauthorized withdrawal leads to loss of funds for the users and loss of funds for the protocol itself.

So in this context of withdrawal of funds access control again becomes important, the security checks have to make sure that the right access control is applied with respect to the different addresses or different modifiers on these withdrawal.

4.6 selfdestruct

The use of the self-destruct primitive is critical and dangerous from a security perspective. Remember that self-destruct is an EVM instruction that is further supported by a `Solidity` primitive, which when used within the smart contract, destroys or kills that contract and transfers all its Ether balance to the specified recipient address.

So from a security perspective, any smart contract that uses self-destruct within a particular function, needs to protect access to that function because, if not, an user can mistakenly call that function or an attacker can intentionally call that function to kill that contract and remove its existence thereafter.

This means that from a security perspective, unauthorized calls to functions within smart contracts that may use the self-destruct primitive should be prevented, so that the contract does not get killed intentionally or mistakenly.

Access control to such functions again becomes critical to make sure that only authorized users may call such functions. At a high level, even the use of self-destruct is considered as being very risky and dangerous from a security perspective.

4.7 Modifiers Side-effects

Modifiers in `Solidity` smart contracts are typically used to implement different kinds of security checks access control checks, or accounting checks on fund balances and so on. Such modifiers should not have any side-effects, they should not be making any state changes to the contract or external calls to

other contracts.

The reason for that is any such side-effects made by the modifiers, may go unnoticed both by the developers as well as the smart contract security auditors evaluating the security of these contracts. They go unnoticed not only because developers and auditors assume that modifiers don't make side-effects, but also because the modified code is typically declared in a different location from the function implementation itself. Remember that the best practice is for the modifiers to be declared in the beginning of the contract and function implementations in the later part of the contract.

So as a security check, one should make sure that modifiers declared in contract should not have any side-effects and they should be only enforcing checks on different aspects of the contract.

4.8 Incorrect Modifier

Incorrect modifiers are a security risk. Modifiers should not only implement the correct access control or accounting checks as relevant to the smart contract logic, but they should also `execute` underscore or `revert` along all the control flow paths within that modifier. Remember that in the context of `Solidity` underscore inlines the function code on which the modifier is applied.

So, if this does not happen along any particular control flow path within the modifier, then the default value for that function is return.

This may be unexpected from the context of the caller who called this function on which this modifier is applied, so the security check is to make sure that all the control flow paths within the modifier either `execute` undiscovered or revert.

4.9 Constructor Names

Constructor names in `Solidity` have had security implications historically. If you go back all the way to `Solidity` compiler version 0.4.22 versions until that version, required the use of the contract name as the name of the constructor. And between that version and 0.5.0 one could either use the contract name as a constructor or use the constructor keyword itself. It was only after 0.5.0 that `Solidity` forced the use of the constructor keyword for constructors.

So this flexibility, the use of the contract name as the constructor name, has historically caused bugs, where the contact name was misspelled which led to that function not being the constructor, but a regular function. Also the flexibility between allowing both the old style, the new style constructor names caused

security issues, because there was a precedence that was followed, if both of them existed. So this constructor naming confusion has been a historical source of bugs and `Solidity` smart contracts, this is not a concern anymore.

4.10 Void Constructor

There's a security concern related to `void` constructors. What this means is that if a contract derives from other contracts, then it makes calls to the constructors of base contracts assuming they're implemented, but if in fact they are not, then this assumption leads to security implications.

So the best practice for derived contracts is to check if the base constructor is actually implemented and remove the call to that constructor, if it is not implemented at all.

4.11 Constructor callValue

This security pitfall is related to constructors, the checks for any value sent in contact creation transactions triggering those constructors.

Typically, if a constructor is not explicitly payable and there is value Ether that is sent in a contract create transaction, that triggers such a constructor, then the constructor reverts that transaction.

However because of a compiler bug, if contract did not have an explicit constructor, but the contract had a base contract that did define constructor, then in those cases, it was possible to send Ether value in a contract creation transaction, that would not cause that reward to happen. This compiler bug was present all the way from version 0.4.5 to version 0.6.8.

4.12 delegateCall

The security pitfall is related to the use of `delegateCall` in contracts where the `delegateCall` may be made to an address that is user controlled. Remember that in the case of `delegateCall`s, the calling contract makes a `delegateCall` to a called contract, where the called contract executes its logic on the state of the calling contract.

So, if the address of the called contract is user controlled, then the user may accidentally or maliciously make this `delegateCall` to a malicious contract, that can make unauthorized modifications to the state of the calling contract. Therefore, `delegateCall`s should be used with extreme care in contracts. All precautions should be used to ensure that the destination addresses for such `delegateCall`s are trusted.

4.13 Reentrancy

The reentrancy security pitfall is perhaps unique to smart contracts where external calls made to contracts can result in what can be thought of as callbacks to the called contract itself.

So for example, if there is a contract `C1`, that makes a call to an external contract `C2`, where `C2` could potentially be untrusted could be malicious because it is not developed by the same team or within the same project as `C1`, then that external contract `C2` could call back into `C1` to the same function that called it or to any other function of `C1` that allows such a call.

This could be exploited to do malicious things, such as multiple withdrawals or something less harmful, such as out of order emission of events. There have been multiple exploits that have taken advantage of this class of reentrancy attacks, some of them are historical in nature such as the DAO hack on Ethereum.

So this class of security vulnerabilities, that is specific to smart contracts needs to be paid attention to, the best practice to prevent such reentrancy vulnerabilities from being exploited is to follow what is known as the **Checks Effects Interactions** pattern or the CEI pattern for short, where the interactions with external potentially untrusted contracts is only made after performing all the checks and all the effects where effects are nothing, but changes to the state of the calling contract, so that any anticipated side-effects of interactions with the external contracts are already reflected in the state of the calling contract.

So this CEI pattern is recommended as a best practice to be followed in all functions that are making external contract calls specifically to contact calls that could be malicious because they're untrusted. The other best practice is to use what are known as reentrancy guards, we talked about this in the context of the reentrancy guard library from OpenZeppelin where a modifier, a non reentrant modifier, is provided. This modifier when applied to specific functions prevents them from being called within a callback, so it avoids any reentrances to that function itself.

4.14 ERC777

This security pitfall is related to the use of `ERC777` standard, the potential for re-entrancy vulnerabilities due to the callbacks it supports.

Remember that `ERC777` standard is considered as an extension to the `ERC20` standard it's considered as making improvements to it. One improvement is the notion of hooks that it supports during token transfers, if such an `ERC777` token contract is potentially malicious, then it could use these hooks to cause reentrancy into the calling contract. So for example, if there's a contract `C1`

that calls an `ERC777` token contract that is malicious, then that contract could use the hook functionality to cause a reentrancy into the calling contract `C1` and take advantage of it as we just mentioned.

The best practice again is to follow the Checks Effects Interaction (CEI) pattern in the calling contract and also to consider the use of reentrancy guards.

4.15 transfer() & send()

This security pitfall is related to the use of the `transfer` and `send` primitives in `Solidity`.

These primitives were introduced as reentrancy mitigations, because they only forward 2300 Gas to the called contract, which is typically sufficient only for basic processing such as emitting a few logs or something even simpler, this Gas is not enough to make a real currency call back to the calling contract which requires more than 2300 gas.

So this has been recommended for a long time as a security best practice for preventing reentrancy attacks however over time some of the opcodes have been reprised when it comes to their Gas usage, so their Gas Cost has increased in some of the recent hard forks on Ethereum and because of that the use of these primitives that enforce the Gas subsidiary 2300 Gas could break the contract because it might not allow the called contract to even do the basic processing that we just talked about.

So the latest security best practice recommendation is to not rely on transfer and send as reentrancy mitigations, but instead to use the low-level call directly that does not have those hard-coded Gas Limits and couple that with a CEI pattern or reentrancy guard or both for re-entrance mitigation.

4.16 Private Data

This security pitfall is related to the notion of what is private data on a blockchain or the privacy of on-chain data.

Remember that state variables and `Solidity` have a function visibility specifier. By making this specified private, such private state variables can't be read only by other smart contracts on the blockchain, this does not mean that they can't be read at all. We don't have to believe that they are considered private in a confidentiality perspective because the state of such variables and contracts and transactions in general on the blockchain can be read by anyone on the chain itself or via off-chain interfaces by querying the mempools for transactions or by querying the contract state itself to look at what values such

private variables contain.

This effectively means that, there is no notion of data being private on the blockchain and any such data for confidentiality reasons that needs to be private should be encrypted and stored off-chain.

4.17 PRNG

This security pitfall is related to pseudo-random number generation on the blockchain within smart contracts applications that `require` such random numbers.

Remember that these values could be influenced to a certain extent by miners who are mining the blocks that contain these values. So if the stakes in those applications using these as sources of randomness is high, then such actors could use their influence to a certain extent to gain advantage.

So this is a risk from randomness that needs to be paid attention to something to be aware of and, if the stakes are high for the applications where you desire a much better source of randomness then, there are some alternatives such as the verifiable random function.

4.18 Time

Similar to randomness, the notion of getting the time on-chain is also tricky. Often smart contracts resort to using `block.timestamp` or `block.number` as sources for inferring the time within the application's logic.

Again, what needs to be paid attention to is that this notion of time can be influenced to a certain extent by the miners. There are issues with synchronization across the different blockchain nodes and there are also aspects of the block times that change by a certain degree over time.

This is again a risk that needs to be paid attention to and, there are some alternatives to this using the concept of Oracles.

4.19 Overflow/Underflow

This security pitfall is related to the notion of overflows and underflows in `Solidity` smart contracts. This is applicable to any integer arithmetic that is used within the contracts which is very often encountered.

When such arithmetic is used in a way where the increments or decrements to those integer variables are done without checking for the bounds, then they

could result in wrapped values where the value exceeds the maximum storage for that integer type and hence overflows or wraps to the lower end of that type or, If it's being decremented it could be decremented below `zero` in which case it results in wrapping to the maximum value of that integer type.

If those extremely high or extremely low data values resulting because of wrapping are invalid in the applications logic, then it is okay. But if it is not, if it's valid in the applications logic, then this could result in unexpected behavior in the best case or in the worst case it could result in some very serious vulnerabilities that can be exploited. We have seen multiple vulnerabilities and exploits led to overflow and underflow historically.

So the recommended best practice is to use the SafeMath libraries from OpenZeppelin that enforce the overflow and underflow checks during integer arithmetic or to use the latest `Solidity` versions greater than or equal to 0.8.0 that introduce check arithmetic by default.

4.20 Divide before Multiply

Another security pitfall or best practice related to integer arithmetic is the use of divide before multiply. `Solidity` integer division might truncate the value of results therefore, if division is done before multiplication, then this may result in the loss of precision of the values being computed.

So the recommended best practice is to always do the multiplication operations first followed by any division that is required.

4.21 TOD

This security pitfall is related to **transaction order dependence** or TOD for short. Remember that in Ethereum transactions submitted by users sit in a data structure known as the `mempool` and get picked by the different miners for inclusion within blocks.

The specific transactions that are picked, the specific order of those transactions included within the blocks depends on multiple factors and specifically the Gas Price of those transactions itself.

So from an attacker's perspective one can monitor the `mempool` for interesting transactions that may be exploited by submitting transactions with a Gas Price appropriately chosen, so that the attackers transaction either executes right before or right after the interesting transaction. This is typically known as Front-running and Back-running and may lead to what are known as sandwich

attacks.

All these aspects are related to assumptions being made on the transaction being included in a specific order by the minor within a block.

So from a security perspective logic within smart contracts should be evaluated to check, if transactions triggering that logic can be front run or background to exploit any aspect of it. A classic example of transaction order dependence is the `approve()` functionality in the popular `ERC20` token standard.

4.22 ERC20 approve()

Remember that the `ERC20` token standard has the notion of an owner of a certain balance of those tokens and there's also the notion of a spender which is a different address that the owner of tokens can `approve` for a certain allowance amount which the spender is, then allowed to transfer.

This approval mechanism is susceptible to a Race-condition. So let's take an example to see how that works let's say that I am the owner of a certain number of tokens of an `ERC20` contract and I want to `approve` a particular spender with 100 tokens of allowance, so I go ahead and do that with an `approve()` 100 transaction and later I change my mind and I want to reduce the allowance of the spender from 100 to 50.

So I submit a second approved 50 transaction and, if that spender happens to be malicious or untrustworthy and monitors the `mempool` for this approval transactions they would see that I'm reducing their approval to 50 by noticing the `approve()` 50 transaction.

In that case they can front run the reduction of approval transaction with a transaction that they send that spends the earlier approved hundred tokens. So that goes through first because of Front-running and when my `approve()` 50 transaction goes through that, would give the spender an allowance of 50.

Now the spender would further go ahead and spend those 50 tokens as well, so effectively instead of allowing the spender to spend only 50 tokens I have let them spend 150 tokens of mine, this is made possible because of transaction order dependence or Front-running.

The mitigation to this the best practice recommended is to not use the `ERC20` `approve()` that is susceptible to this Race-condition, but to instead use the `increaseAllowance()`, the `decreaseAllowance()` functions that are supported by such contracts.

4.23 ercrecover

This security pitfall is related to the use of the `ercrecover` primitive in EVM and supported by `Solidity`.

The specific pitfall is that it is susceptible to what is known as signature malleability or non-unique signatures. Remember that elliptic curve signatures in Ethereum have three components `v`, `r` and `s`. The `ercrecover` function takes in a message hash the signature associated with that message hash and returns the Ethereum address that corresponds to the private key that was used to create that signature.

In the context of this pitfall, if an attacker has access to one of these signatures, then they can create a second valid signature without having access to the private key to generate that signature.

This is because of the specific range that the `s` value or the `s` component of that signature can be in it can be in an upper range or a lower range and both ranges are allowed by this primitive which results in the malleability.

This depending on the logic of the smart contract, the context in which it is using these signatures can result in replay attacks, so the mitigation is to check that the `s` component is only in the lower range and not in the higher range, this mitigation is enforced in OpenZeppelin's ECDSA library which is the recommended best practice.

4.24 transfer()

This security pitfall is related to the transfer function of `ERC20` tokens. The `ERC20` specification says that a transfer function should return a `boolean` value, however a token contract might not adhere to the specification completely and may not return a `boolean` value may not return any value.

This was okay until the service compiler version `0.4.22`, but any contract compiled with a more recent `Solidity` compiler version will `revert` in such scenarios. So the recommended best practice for dealing with this scenario is to use the OpenZeppelin's `SafeERC20` wrappers for such interactions.

4.25 ownerOf()

This pitfall is similar to the previous one and applies to the `ownerOf()` function of the `ERC721` token standard.

The specification says that this function should return an address value however contracts that did not adhere to this specific aspect would return a `boolean`

value.

It used to be okay until the `Solidity` version `0.4.22`, but with any newer compiler version returning a `boolean` value would cause a revert. So the best practice again is to use the `ERC721` contract from `OpenZeppelin`.

4.26 Contract Balance

This security pitfall is related to the Ether balance of a smart contract and how that can change unexpectedly outside the assumptions made by the developer.

Remember that smart contracts can be created to begin with a specific Ether balance. Also there are also functions within the smart contract that can be specified as being payable, which means that they can receive Ether via message value.

These two ways can be anticipated by the developer to change the Ether balance of the contract. But there are also other ways in which the Ether balance of the contract can change.

One such way is the use of `coinbase` transactions. These are the beneficiary addresses used in the block headers where the miner typically specifies the address to which the block rewards and all the transaction Gas fees should go to. That `coinbase` address could point to a specific smart contract where all the rewards, the Gas fees go to, if that block is successfully included in the blockchain.

The other unexpected way could be via the `selfdestruct` primitive where, if a particular smart contract is specified as the recipient address of `selfdestruct()`, then upon that executing the balance of the contract being destructed would be transferred to the specified recipient contract.

So these two ways the `coinbase` and `selfdestruct` although very unusual and unexpected could in theory change the Ether balance of any smart contract, this could be well outside the assumptions made by the developer or the team behind the smart contract.

So what this means is that, if the application logic implemented by a smart contract makes assumptions on the balance of Ether in this contract and how that can change, then those assumptions could become invalid because of these extreme situations in which it can be changed. So this is something to be paid attention to while analyzing the security for contract from the perspective of the Ether balance that it holds.

4.27 fallback vs receive

This security consideration is related to the use of `fallback` and `receive` functions within a smart contract.

Remember from our discussion in the `Solidity` modules, there are differences between these two functions, there are some similarities. These are related to the visibility, the mutability and the way that Ether transfers are handled by these two different functions.

So from a security perspective, if these functions are used in a contract, then one should check that the assumptions are valid and if not, what are the implications thereof.

4.28 Strict Equalities

From a security perspective strict equalities are considered as dangerous in specific contexts of the smart content applications.

Strict equality is referred to the equal to operator or the not equal to operator as compared to the less stricter less than or greater than or equal to operators.

When these strict equalities are applied to Ether or token values, then such checks could fail because the transferred Ether or tokens could be slightly less or greater than what the strict equalities expect or the balances computed could be different because of the different number of decimals expected or the precision of the operations being slightly different from the assumptions being made. Hence the use of strict equalities with such operands and operations is considered dangerous because they could lead to failed checks.

So the security best practice is to default to less stricter equalities and make sure that those constraints are satisfied as per the assumptions.

4.29 Locked Ether

Locked Ether refers to the situation where the contract has an Ether balance that gets locked because Ether can be sent to that contract via payable functions, but there's no way for users to withdraw that Ether from that contract.

This is in a very simple scenario possible when smart contact has functionality to allow users to deposit Ether. But there is no functionality to withdraw the Ether from that smart contract.

The obvious solutions are to remove the payable attributes for functions to prevent Ether from being deposited via those functions or to add withdrawal capabilities to the smart contract. The simple situations for this particular pitfall can be easily recognized and fixed. But also, there could be complex scenarios where the contract can be taken to a particular state either accidentally or maliciously where the Ether or the token balance of the contract gets locked and can't be withdrawn.

4.30 tx.origin

The use of `tx.origin` is considered dangerous in certain situations within smart contracts. Remember that in the context of Ethereum, `tx.origin` gives the address of the externally owned account that originated the transaction.

If the `tx.origin` address is used for authorization, then it can be abused by attackers for launching replay attacks by coming in between the user and the smart contract of concern.

This is sometimes known as man in the middle replay attack or MITM as an abbreviation because attacker comes in between the user and the contract, captures the transaction and later replace it. Because the smart contract uses `tx.origin`, it fails to recognize that this transaction actually was originated from the attacker in the middle.

So in this case the recommended best practice for smart contracts using authorization is to use `message.sender` instead of `tx.origin`, because `message.sender` would give the address of the most recent or the closest entity. So in this case, if there is a man in the middle attacker, then `message.sender` would give the address of the attacker and not that of the authorized user pointed to by `tx.origin`.

4.31 Contract check

There may be situations where a particular smart contract may want to know, if the transaction or the call made to it is coming from a contract account or an externally owned account.

There are two popular ways for determining that. The first one is to check, if the code size of the account of the originating transaction is greater than zero and if this is not zero, it means that that account has code and therefore is a contract account, the second technique is to check, if the `message.sender` is the same as the `tx.origin` and, if it is, then it means that the `message.sender` is an externally owned account. Remember that `tx.origin` can only be an externally owned account in Ethereum as of now.

So these two techniques have pros and cons and depending on the specific application it may make more sense to use one over the other.

There are risks associated and implications thereof of either of these two approaches particularly with the code size approach the risk is that, if this check is made while a contract is still being constructed within the constructor the code size will still be zero for that account so, if we determine based on that aspect that this is an externally owned account, then it would be a wrong assumption.

4.32 delete Mapping

The next security pitfall is related to the concept of the `delete` primitive and `Solidity` and how it applies to mappings. If there is a struct data structure in a smart contract that contains a mapping as one of its fields, then deleting that structure would `delete` all the fields of the struct, but the mapping field itself would remain intact, so this is one of the `Solidity`'s behaviors that needs to be kept in mind.

That can have unintended consequences, if the developer assumes that the mapping field within the struct also got deleted and re-initialized to its default values.

The best practice is to use an alternative approach such as considering the data structure that is meant to be deleted as being logged to prevent future logic from using the data structure or the mapping fields within that data structure.

4.33 Tautology Contradiction

An interesting security consideration is that of tautologies and contradictions. A tautology is something that is always true whereas a contradiction is something that is always false.

Within smart contracts this can be found in certain primitives used, such as an unsigned integer variable `x` and then there is a predicate that checks, if `x` is greater than or equal to 0. This predicate because of `x` being an unsigned integer is a tautology it's always going to be true because `x` can't take a negative value.

The presence of such tautologies or contradictions in smart contracts indicates either flawed logic or mistaken assumptions made by the developer or these may just be redundant checks.

In either scenario these may be interesting from a security perspective, so it is something to be paid attention to and flagged as potential concerns.

4.34 Boolean Constant

The use of `boolean` constants `true` or `false`, directly in conditionals is unnecessary.

The reason for this is that if there's a conditional whose predicate is true, then that can be removed because that code block would get executed nevertheless and similarly, if the predicate is the `boolean` constant `false`, then that could be removed as well and along with the code in that associated block because that code would never `execute` because the conditional is always going to be `false`.

So these usages of `boolean` constants specifically within conditionals is indicative of flawed logic or assumptions or they could just be used in a redundant manner. The recommendation upon identifying such usage, it is removing those constants and any code blocks associated with them, so that it becomes simpler to read and to maintain.

4.35 Boolean Equality

An aspect related to `boolean` constants is that of `boolean` equality, this is where the `boolean` constants true or false are used within conditionals for an equality check, so the `x` variable is checked against the `true` constant.

This usage is redundant because the variable `x` can be used directly within the conditionals predicate without actually comparing it to true and both of them are equivalent

So the use of the `boolean` constant `true` within the predicate is actually unnecessary, so while this may not be a big security consideration and perhaps indicative of the developer not fully understanding how `Solidity booleans` work.

It is interesting from an optimization perspective and certainly improves the readability aspect of the code.

4.36 State Modification

Contract state modifications made in functions whose mutability is declared as `view` or `pure` will `revert` in contracts compiled with `solc` version greater than

or equal to `0.5.0`.

This is because this compiler version started using the `staticCall` opcode for such functions, this instruction leads to a revert, if that particular function modifies the contract state.

So when analyzing the security aspects of contracts it's good to pay attention to the mutability of the functions to see, if they are viewer `pure`, but they actually modify the contract state in which case they would lead to reverts at runtime.

4.37 Return Values of low-level calls

Checking the `return` values of functions at `call` sites is a classic software engineering best practice that's been recommended over several decades and in the case of `Solidity` this specifically applies to `return` values of function `calls` made using the low level `call` primitives.

These are the `call`, `delegateCall` and `send` parameters that do not `revert` under exceptional behavior, but instead return `success` or `failure` as a `return` value.

So because of this particular characteristic it becomes critical for the `call` sites in contracts that use these primitives to check the `return` values and act accordingly, if not it could lead to unexpected failure.

4.38 Account Existence

Checking for the existence of a smart contract account at a particular address before making a call is a security concern.

The reason for that is because when such calls are made using low level call primitives `call`, `delegateCall` or `staticCall` these functions return true even, if the account does not exist at that address.

So if the contract making such a call looked at the `return` value, saw it was true and assumed that the target contract existed at the address that it called and also assumed that the contract executed successfully, then that would be a faulty assumption.

This as you can imagine could have some serious implications to security, so the best practice here is before making low-level calls to external contract addresses one should check that those accounts do indeed exist at those addresses.

4.39 Shadowing

Shadowing of built-in civility variables was a concern in some of the older `Solidity` versions built-in variables such as now `assert` and some others could be shadowed by other variables functions or modifiers in the contract to override their behavior.

This as you can imagine is dangerous and could lead to many unexpected behavior and therefore this Shadowing was disallowed in the later `Solidity` version.

4.40 Shadowing pt.2

Similar to the Shadowing of built-in variables the older versions of `Solidity` also allowed state variable Shadowing.

This meant that the right contracts could have state variables that had the same name as some of their base contracts. You can imagine that the base variables and shadowed variables with the same names could be confusing even for the developer and they could end up using or modifying the wrong variable from the base contracts.

This dangerous and unexpected consequences was recognized, so `Solidity` compiler `0.6.0` disallowed Shadowing of state variables.

4.41 Pre-declaration

Earlier versions of `Solidity` allowed the use of local variables even before they were declared.

These variables could be declared later or they could have been declared in another scope. This led to undefined behavior as you may expect.

`Solidity` version `0.5.0` and beyond change this, to implement the popular C99-style scoping rules where variables can only be used after they have been declared and only in the same or nested scopes.

4.42 Costly Operations

Certain operations in `Solidity` are considered costly or expensive in terms of the amount of Gas units they use.

If such operations are used inside loops they end up consuming a lot of Gas which could result in unexpected behavior.

The best example of a costly operation in `Solidity` is that of state variable updates remember that insularity state variables are stored in the storage area of the EVM updates to such state variables use the SSTORE instructions of the EVM which are one of the most expensive EVM instructions.

As of the latest upgrade from Berlin, `SSTORE` costs 20000 Gas units, if they are a cold store where the state variable is being updated for the first time in the context of this transaction. Or they cost 5000 Gas units if it is a warm store, in which case this variable has already been updated in the context of this transaction.

So either 5000 or 20000 Gas units are consumed every time a state variable is updated, so as you can imagine, if such updates are done inside loops, then they could end up consuming a lot of Gas and result in an Out-of-Gas error, if the amount of Gas supplied in this transaction is less than what is required.

The solution here is to use local variables instead of state variables as much as possible, the reason is because local variables, if you remember are allocated in memory and memory updates using MSTORE only cost 3 Gas units compared to the 5000 or 20 000 that storage updates cost.

So this notion of costly operations being used inside the loops leading to Out-of-Gas errors and in the worst case leading to a denial of service (DoS) can be mitigated by caching and using local variables as much as possible instead of storage variables.

4.43 Costly Calls

Similar to state variable updates, external calls inside loops should also be used very carefully. The reason is external calls cost 2600 Gas as of the latest upgrade this is more of a concern if the index of the loop is controlled by the user because in that case the number of iterations of the loop is also user controlled.

Because that could result in a denial of service, if one of the calls inside the loops rewards or, if the execution runs Out-of-Gas because the Gas applied in the transaction wasn't enough.

So the mitigation here is to avoid or reduce a number of external calls made inside loops and also check that the loop index can't be user controlled or that it is bounded to a small number of iterations, this again is in the context of preventing opportunities for denial of service.

4.44 Block Gas Limit

Costly operations such as state variable updates and external calls especially made inside loops are also relevant in the context of the block Gas Limit.

Remember that Ethereum blocks have a notion of a block Gas Limit which limits the total amount of Gas units consumed by all the transactions included in the block to a maximum upper bound. This upper bound until recently was 15 million Gas units. This has changed significantly in how it works because of EIP 1559, but the notion of a block Gas Limit still remains.

The reason why this is relevant is because, if expensive operations are used inside loops where the loop index may be user controlled. Then such expensive operations may result in an Out-of-Gas error, this Out-of-Gas could not only come from the amount of Gas units supplied in the transaction that resulted in all this execution, but it could also arise because of the Gas consumed by this transaction exceeding the Gas Limit for this block.

So the mitigation here is again to evaluate the loops and make sure that a lot of these expensive operations are not used inside the loops and also to check if the loop index is user controlled, and if it can be bounded to a small finite number, so that opportunities for denial of service are prevented.

4.45 Events

Events should be emitted within smart contracts for all critical operations. Emission of events that are missing for such critical operations is a security concern.

The reason for this is because it affects off-chain monitoring remember that events emitted from smart contracts end up storing the parameters of such events in the log part of the blockchain.

These logs either the topics part or the data part can be queried by off-chain monitoring tools or off-chain interfaces to understand what is happening in the smart contracts. This is an easier way to understand the state of the smart contracts without having to query the contracts themselves.

These events become very important from a transparency and user experience perspective. So the best practice is to recommend the addition of events in all places within the smart contracts where critical operations are happening, these could be updates to critical parameters from the smart contract applications perspective this could be operations that are being done only by the Owner or privileged roles within the smart contract. So in all such cases events should be emitted to allow transparency and a better user experience.

4.46 Event Parameters

Having talked about events, let's now focus on the event parameters. Event parameters not being indexed may be a concern in certain situations.

Remember that event parameters may be considered as either indexed or not depending on the use of the `indexed` keyword. This results in those parameters being stored in the topics part of the log or the data part of the log, and being stored in the topics part of the log allows for those parameters to be accessed or queried faster. Because of the use of the bloom filter, if they're stored in the data part, then it results in a much slower access.

There are certain parameters for certain events that are required to be indexed as per specifications `ERC20` token standard for example, has transfer and approval events that require some of their parameters to be indexed.

Not doing it will result in the off-chain tools that are looking for such index events to be confused or thrown off track.

So the best practice here is to add the ind`indexed` keyword to critical parameters in an event. Especially if the specification requires them to be in text, this comes at cost of some additional Gas usage, but allows for faster query.

4.47 Event Signatures

The concern here was that of incorrect event signature in libraries. The reason for this happening was because, if events used in libraries had parameters of contract types, then because of a compiler bug, the actual contract name was used to generate the signature hash instead of using their address type.

This resulted in a wrong hash for such events being used in the logs. The mitigation here was to fix the compiler bug which happened in version 0.5.8 where the address type was used instead of using the contract name incorrectly.

4.48 Unary Expression

Unary expressions are where an operator is used on a single operand as opposed to two operands, in which case it would be a binary expression and such `unary` expressions are susceptible to typographical errors by developers.

For example let's take a look at the scenario where there's a variable `x`. The developer wants to increment it by one, so the way to do that is to say `x += 1` which effectively is `x = x + 1`. But if the developer interchanges the order of `+` and `=` and instead uses `x = +1`, then this would result in re-initializing the value of `x` to `+1`. The reason for this is that `+ 1` is a `unary` expression whose

value is `1` and `x` would get initialized to that value.

As you can imagine such typographical errors are likely to be made by developers it's very easy to make these switching the order and, if they are considered as valid by the compiler, then it's very hard to notice such errors both by the developer as well as by the security auditor.

So in order to prevent some of the most common usages that result in such typographical errors the `unary +` was `deprecated` as of compiler `Solidity` version `0.5.0`.

4.49 Zero Addresses

Zero address in Ethereum and `Solidity` has a special consideration. Remember that Ethereum addresses are 20 bytes in length and, if all these bytes are zeros, then it's referred to as a Zero-address.

These Zero-addresses become significant, because state variables or local variables of address types have a default value of zero in `Solidity`. These Zero-addresses are also used as burn addresses because the private key corresponding to this Zero-address is not known, so any Ether or tokens that are sent to the Zero-address gets burnt or inaccessible forever.

If addresses used for access control within the smart contracts end up being Zero-addresses, such functions can't be invoked again because of the lack of knowledge of the private key. So such functions can't be called which might in the worst case end up in such contract getting locked.

So the best practice is to perform input validation on all address parameters that are of address type to check that they are not Zero-addresses.

This is a very commonly encountered security pitfall where address parameters of constructors setters or public `external` functions are not input validated to not be Zero-addresses.

In the best case such scenarios only result in exceptional behavior at runtime, but in the worst case they could result in tokens getting burnt or the contract being locked.

4.50 Critical Addresses

Another security pitfall related to addresses is the aspect of changing values of Critical Addresses. Certain addresses within the context of the smart contract may be considered as critical, these may be special privileged roles such as

the owner address, which has special access to certain functions for updating critical parameters or doing other administrative aspects related to the smart contract.

Or these could also be addresses of other smart contracts that are used within the context of the application. As you can imagine there may be scenarios where such addresses would need to be changed, so for example the default owner of a contract could be the deployer of the contract and we may want to change this to another address later on.

Or, if the addresses correspond to other smart contracts, then we may want to change the value to another smart contract once we have updated it.

In such scenarios, the security pitfall is when this change is done in a single step, this may be using the Oracle library from `OpenZeppelin` where.

There is a transfer ownership function provided that transfers the ownership from the existing owner to a new owner that is provided as a parameter.

This happens in a single step where the `owner` variable is updated to the new address provided this single step change is prone to errors. If an incorrect address is used as a new address, then it may result in that contract getting locked forever, the reason for this is the address used may be an address for which we do not have the private key, so we can't sign any transactions from that address, which results in all the administrative functions or any of the address change functions being inaccessible. Thereafter the mitigation here or the best practice is to move away from a single step change and to move to what is known as a two-step change.

Where the first step grants or approves a new address as being the Owner or as being that changed address, the second step is a transaction from the new address that claims itself as being the new owner or as being the new address.

So this two-step change allows any errors that happen in the first step, where we `grant` or `approve` it to an incorrect address for which we do not have the key it allows us to recover from this mistake because the second transaction which claims itself as a new address can never be done if an incorrect address was used in the first step.

So this aspect of Critical Addresses being changed and allowing errors to be recovered by moving away from a single step to a two-step change is a critical aspect of mitigating the risk from incorrect address changes.

4.51 assert()

This security best practice is related to the use of asserts within smart contracts. Assert should be used only to check or verify program invariants within the smart contracts. They should not be used to make any state changes within their predicates and they should also not be used to validate any user inputs.

The reason for this is because, if any state changes are made as the side-effects of the predicates within asserts, then those could be missed both by developers during maintenance or when they are trying to do any testing.

They could also be missed by auditors because these state changes are not expected to happen within a search and similarly, asserts should not be used to validate user inputs because that should be done using `require()` statements which we'll see in the next slide.

As a general rule we do not expect to see any failures from asserts during normal contract functioning and therefore these best practices become very relevant.

4.52 assert() Vs require()

This best practice is related to the use of assert() versus `require()` and the specific conditions in which they should be used.

These two aspects are related, but they have different usages.

Asserts should be used to check or verify invariants where these invariants are expected to be held during normal contract functioning, so we do not expect any of these asserts to `fail` during the contract execution and any failures are critical panic errors that need to be caught and dealt with in a very serious manner.

On the other hand `require()` is meant to be used for input validation of arguments that are supplied by users to various public or `external` functions where we do expect failures to happen because the user provided values may be zero-addresses in some cases or maybe values that are out-of-range or do not make sense from the smart contracts perspective.

So this difference is something to be kept in mind, the best practice is to use `assert()` or `require()` appropriately as the situation demands. This had a more significant impact until `Solidity` compiler version `0.8.0`. Until then, `require()` used the `revert` opcode which refunded the remaining Gas and failure, whereas `assert` used the invalid opcode which consumed all the supplied Gas.

So until that version the usage of `assert()` or `require()` incorrectly, would result in different Gas semantics. This is because in one situation the remaining Gas would be refunded, whereas in the other case all of it would be consumed.

So this affected user experience as well, but this has changed since version `0.8.0` where both `require()` and `assert()` use the `revert` opcode and refund all the remaining Gas on failures.

4.53 Keywords

This security best practice is related to the use of duplicated keywords in `Solidity` over the different compiler versions. Different keywords have been deprecated to favor one over the other, so for example `msg.gas` has been deprecated to favor `msg.gasleft`, `throw` has been deprecated to favor the use of `reward`, `sha3` for `keccak-256`, `callcode` for `delegateCall`, `constant` Keyword for `view`, the `var` Keyword for using the actual type name instead.

So all such deprecated keywords they start initially as compiler warnings where the compiler wants us not to use these keywords and over the future versions these warnings could be converted into compiler errors in which case the compilation fails.

So the best practice here is to simply avoid the use of deprecated keywords even if they are compiling warnings, because these warnings can become errors in the future compiler versions.

4.54 Visibility

Remember that functions in `Solidity` have the notion of visibility where they could be either `public`, `external`, `internal` or `private`, this affects which users can call these functions.

So `public` and `external` functions are callable by anyone depending on the access control that is enforced additionally on top of that whereas `internal` and `private` can be called only from within the contracts or the derived contracts.

Until `Solidity` version `0.5.0` this visibility specifier was optional and they defaulted to `public`. This aspect led to vulnerabilities where the developer forgot to mention or specify the visibility in which case it became public by default and resulted in malicious users being able to call these functions and make unauthorized state changes completely unexpected by the developer or the smart partner.

So this optional specification of function visibility defaulting to `public` visibility was removed as of `Solidity` version `0.5.2`, so this was a big change when it came to increasing the security of smart contracts and since that version function visibility is required to be specified explicitly for every function.

4.55 Inheritance

Contracts that inherit from multiple contracts should be careful about the inheritance order because, if more than one such base contract defines an identical function, then the particular function implementation that gets included in the derived contract depends on this inheritance order.

The best practice is for this inheritance order to be from the more general implementation to the more specific implementation.

4.56 Inheritance pt.2

Another security pitfall related to inheritance is that of missing inheritance where a particular contract within an application might appear to inherit from another interface in that project or another abstract contract without actually doing so.

And it might appear, because of the contract name that is similar to the abstract contract or the interface name or also because of the functions that are defined within this contract their names the parameter types and, so on.

This appearance might give the notion that it is inheriting without actually inheriting this affects not only the readability and maintainability aspects for the developers of the project team, but it also affects the auditability because the security reviewer might look at this contract and assume certain aspects, thinking that it's inheriting from the similarly named interface or abstract contract whereas in fact it does not do so.

So the best practice here, is to make sure that the inheritance is done appropriately and. If there are similarly named contracts where they do not actually inherit from each other, then the name should be changed but, if they do in fact are meant to be inherited, then specifying that inheritance will help.

4.57 Gas Griefing

Gas briefing is a security concept that becomes interesting in the context of transaction relayers.

Remember that on Ethereum, users can submit transactions to the smart contracts on the blockchain or alternatively they can submit what are known as meta-transactions which are sent to the transaction relayers, where they do not need to be paid for Gas. The relayers in turn, forward such transactions to the blockchain with the appropriate amount of Gas.

In this scenario the users typically compensate the relays for the Gas out of that. In such situations it becomes necessary for the users, to trust the transaction relayers, to submit those transactions or forward their transactions with a sufficient amount of Gas, so that their transactions do not fail.

4.58 Reference Parameters

Remember that `Solidity` has value types and reference types. In this security pitfall is related to the use of reference types in function parameters when structs, arrays or mappings, which are the reference types, are passed as arguments to a function.

They may be passed by value or they may be passed by reference. This difference is dictated by the use of either the memory or the storage Keyword that specifies their data location. This was optional before `Solidity` version `0.5.0`, but since that version it is required to be specified explicitly.

This difference is critical from a security perspective, because pass by value, if you remember, makes a copy, so any changes to the copy does not affect the original value. But pass by reference, creates a pointer to the original variable, so any changes to the past value is actually modifying the original variable itself.

This, if not treated properly could lead to unexpected changes and modifications of the original variable or a copy which could have very different behavior and impact for the smart contract logic.

4.59 Arbitrary Jumps

Arbitrary jumps are possible within `Solidity`. `Solidity` supports many different types, one of which is a function type. These function type variables are not frequently encountered, but if they are indeed used especially within Assembly code in making arbitrary manipulations to variables of these types, then they could be used to change the control flow to switch to an arbitrary location in the code. This is something to be paid attention to and from a development perspective something to be avoided.

Assembly in general is very tricky to use and it bypasses many security aspects of `Solidity` such as type safety, so it's best to avoid the use of Assembly if

possible and definitely to avoid the use of function type variables and making arbitrary changes to it. This is because that could result in changes to control flow that is unexpected by the developers or the smart contract auditors.

4.60 Hash Collisions

Hash collisions are possible in certain scenarios where the `abi.encodePacked()` primitive, is used with multiple variable length arguments.

This happens because this primitive does not zero `pad` the arguments, and it also does not save any length information for those arguments as a result of which it allows packing, this `pad` encoding could lead to collisions in certain scenarios, which you can imagine can affect the security of the smart contract.

The best practice here is to avoid the use of the `pad` primitive where possible and use the `abi.encode()` primitive instead.

In scenarios where it can't be avoided, one should at least make sure that only one variable length argument is used in this parameter and certainly, users who can reach this primitive via function calls should not be allowed to write to the parameters used and tainted to force collisions from happening.

4.61 Dirty Bits

There is a security risk from Dirty High Order Bits in `Solidity`. Remember that the EVM word size is 256 bits or 32 points, and there are multiple types in `Solidity` whose size is less than 32 bytes using variables of such types may result in their higher order bits containing dirty values.

What this means is that they may contain values from previous rights to those bits, that have not been cleared or zeroed out. Such Dirty Order Bits are not a concern for variable operations because the compiler is aware of these Dirty Bits and takes care to make sure that they do not affect the values of variables.

By the way, if those variables end up getting used or passed around as message data, then that may result in them having the different values and causing malleability or non-uniqueness. This is a risk that needs to be kept in mind when looking at contracts that have variables of such types.

4.62 Incorrect Shifts

There is a security pitfall related to the use of incorrect shifts in `Solidity` Assembly specifically.

`Solidity` Assembly supports three different Shift operations: Shift left (`shl()`), Shift right (`shr()`) and Shift arithmetic right (`sar()`, all of which take two operands `x` and `y`. These operations Shift the `y` operand by `x` bits and not the other way around.

This can be confusing, understandably the developer may have used these two operands interchangeably in which case the Shift operation does something completely different from what the developer anticipated. This is something that needs to be checked when looking at Shift operations in `Solidity` Assembly.

4.63 Assembly

The use of Assembly in `Solidity` itself is considered as a security risk because Assembly bypasses multiple security checks such as type safety, that is enforced by `Solidity`.

Developers end up using `Solidity` Assembly to make the operations more optimized and efficient from a Gas perspective, but on the flip side this is very error-prone because the Assembly language Yul, is very different from `Solidity` itself and requires much greater understanding of the syntax and semantics of that Assembly language.

So the use of `Solidity` Assembly not only affects readability and maintainability, but also the auditability, because the auditors themselves might not be aware of the Yul language: the syntax and semantics.

All these aspects result in the recommended best practice of trying to avoid `Solidity` Assembly as much as possible or, if absolutely required, then the developers and the security review should double-check to make sure that they have been used appropriately in the context of the smart contracts.

4.64 RTLO

There is a security pitfall that arises because of the use of the Unicode Right-to-Left-Override control character (`U+202E`) in `Solidity` smart contracts causes the text to be rendered from right to left instead of the usual left to right.

This reverse rendering confuses the users as well as the security reviewers from understanding what the real intent is of that particular snippet of the smart contract.

The best practice here is to ensure that such confusing Unicode characters (the RTLO control character) is not used within smart contracts at all.

4.65 Constant

There is a best practice related to the use of the constant specifier for state variables in `Solidity`.

State variables whose values do not need to change for the duration of the lifetime of the contract can be declared as constant. This saves Gas because the compiler replaces all the occurrences of such state variables with the constant value. This effectively means that reading such state variables no longer requires the expensive `SLOAD` instructions.
So the best practice is to identify such state variables whose values do not need to change over the lifetime of the contract and declare them as constant. This also has an additional side effect on improving security because such state variables can no longer be accidentally changed within the different functions of the contract.

4.66 Variable names

There's a security best practice related to variable names. This ties to the programming style guidelines that we discussed in the `Solidity` module.

The names of variables should be as distinct and unique from each other as possible because if they are very similar (if they differ by only a few characters or one character), then it could be confusing both to the developer as well as to the security reviewer.

From a developer's perspective, it could lead to replaced usages where the developer uses a different variable than what was intended. As you can imagine, this can have disastrous effects to the functioning of the smart contract.

So variable naming affects readability it affects maintainability and auditability of the code. The best practice is to use very distinct names for the variables meaningful names for the variables, so that errors are avoided.

4.67 Uninitialized Variables

Another security pitfall related to variables is the use of uninitialized state or local variables. Remember that in `Solidity` the default values of uninitialized variables such as address, `bool` or `uint` is `0`, string is `""` and so on.

This results in address variables ending up as 0 addresses and `boolean` variables taking the value of `false` because 0 is effectively `false` (we have talked about the risks from 0 addresses and `bool`s being `false` by default will result in the conditionals taking a different branch than what was intended).

The best practice is to make sure that state and local variables are initialized with reasonable values, so that errors are avoided from having the default values being used.

4.68 Unused State/Local Variables

Another aspect that needs to be paid attention in the use of variables inside contracts is that these could be state variables or local variables. The specific aspect is that if these variables are declared but are never used within the contract.

It could be indicative of missing logic that is expected to be there (that uses these variables in certain ways). It may be missing because the developer forgot to add it or it could simply be indicative of some optimization opportunity where such variables can actually be removed and reduce the size of the byte curve, and therefore reduce the amount of Gas that is used either during deployment or during runtime.

The best practice here is to pay attention to all the variables that are declared (in functions, in the contract, state variables...), see if they are used and, if they are never used, then determine if they need to be removed for optimization, or if there is any logic that is missing that needs to be added that uses those variables.

4.69 Storage Pointers

There is a security pitfall related to the use of uninitialized storage pointers. Local storage variables that are uninitialized can point to unexpected storage locations within the contract.

This can lead to developers unintentionally modifying the contract state, which can lead to serious vulnerabilities. Given that this is so error-prone, `Solidity` compiler `0.5.0` started disallowing such pointers.

4.70 Function Pointers

There was a security risk in using uninitialized function pointers within constructors of contracts because of a compiler bug that resulted in unexpected behavior.

This compiler bug was present in versions `solc 0.4.5 - 0.4.26` and `solc 0.5.0 - 0.5.7` and has since been fixed.

4.71 Long Number Literals

There is a security risk in the use of long number literals within `Solidity` contracts. These number literals may require many digits to represent their high values (as constants) or many decimal digits of precision, which as you can imagine is error prone.

For example, if one were to define a variable representing Ether, then it would need to be assigned a number literal that has 18 zeros to represent the 18 decimals of precision.

So the developer may accidentally use an extra zero or miss a zero in which case the Ether precision is different, thus the logic using this variable will be broken.

The best practice here is to use the Ether or time suffixes supported by `Solidity` as applicable or to use the Scientific Notation which is also supported by `Solidity`.

4.72 Out-of-range Enum

Older versions of `Solidity` produced unexpected behavior with out-of-range enums. For example we had `enum Ea` (with a single member `a`) as shown here, then `E(1)` is out-of-range because, remember, indexing of `enum` members begins with 0.

So `E(1)` here is out-of-range because there's a single mapper. This out-of-range `enum` produced unexpected behavior in `Solidity < 0.4.5`. This was due to a compiler bug which has since been fixed.

The best practice until the fix was applied was to check the use of `enums` to make sure they are not out-of-range.

4.73 Public Functions

Remember that `Solidity` has the notion of visibility for functions, there are four visibility specifiers: `internal`, `private`, `public` and `external`. `public` functions consume more Gas than `external` functions.

The reason for this is because the arguments of `public` functions need to be copied from the call data component of the EVM to the memory component. This copying produces more byte code for such `public` functions which therefore consumes more Gas.

This copying is not required for `external` functions where their arguments can be left behind in the call data component of the EVM. This key difference leads to `public` functions consuming more Gas than `external` functions in `Solidity`.

So if there are functions in the contract that are never called from within the contracts themselves, then such functions should be declared with `external` visibility and not `public` visibility, which leads to better Gas efficiency.

4.74 Dead Code

Dead Code is any contract code that is unused from the contract's perspective or even unreachable from a control flow perspective.

This could be indicative of programmer error or missing logic that leads to the developer adding this code to the contract, but not adding the logic that actually makes use of this code. This is certainly an opportunity for optimization because dead code increases the code size of the contract which, during deployment, leads to increased Gas costs.

However, this also impacts readability, maintainability and auditability of the code, all of which affect security indirectly. Let's consider three scenarios in which dead code affects the security of smart contracts:

1. There is code in the contract that is in fact dead, but the developer or the smart contract auditor does not realize that this is dead code. If such code implements security checks, then we may assume that those checks are being enforced and improving the security, but in fact they are not effective because they are in dead code, so they reduce their security of the smart contracts again.

2. There is dead code in the smart contract and the developers are aware that this code is dead, but decide to leave it (without removing it). In such cases, such code may not be tested because the developers know that this is dead code and, because of this, they may end up with security vulnerabilities contained in them or they may contribute to such vulnerabilities. Later on, if someone else decides to use this dead code, the vulnerabilities contained by it (or affected by it) get manifested in the contract and affects the security negatively.

3. There is code that is actually used within the smart contracts, but the developers incorrectly determine that this is Dead Code (mistaken identity) and remove it. In such scenarios, if that code implemented security checks are actually improved security because of their logic, then removing it reduces the security of the code

Effectively, dead code contributes to the security of smart contracts indirectly in potentially significant ways. The best practice is for the developers to determine

if a particular piece of code is used or dead and, if it is dead, determine if it actually needs to be used. If it is not, then remove it from the contracts. If it needs to be used, then add logic that uses that code in the correct manner.

4.75 Unused Return Value

There are security risks associated with function `return` values in `Solidity`. Remember that in `Solidity`, functions may take arguments, implement logic that uses those arguments along with some local and global state, create some side effects due to all of that logic and then may `return` values that reflect the impact of that logic. For functions that return such values, the call sites are expected to look at those values and use them in some fashion.

The reason for this is because those `return` values could reflect some error codes that are indicative of some issues that happen during the processing within that function, or they could reflect the data that is produced as a side effect of that execution of the logic within the function.

If these `return` values are not used at the call sites, then that could be indicative of some missed error checking that needs to happen at the call sites. Or in cases where there was data that was being returned without any errors, not using that data at the call sites could result in unexpected behavior.

In both these scenarios, these could affect the security of the contract if that error checking or the missing logic (due to not using the data returned) affected the security aspects of the smart contract logic.

The best practice here is to see if a function needs to `return` values and if functions are returning values, then all their call sites should be checking those `return` values and using them in appropriate ways. If any of those call sites are not using the `return` values, and it does not affect the security, then the developers (or the auditors) need to evaluate if the functions need to return any value at all and remove the values from being returned from those functions.

4.76 Redundant Statements

Redundant statements are statements that either have no side effects or that do have side effects, but are made redundant because there are other statements that have the same side effect.

In either scenario these are indicative of programmer error or missing logic that needs to exist to make these statements not redundant, or they may just present an opportunity for optimization where these redundant statements

need to be removed.

Removal reduces the size of the contract and therefore makes it more Gas efficient at deploy or execution time.

The best practice here is to evaluate if statements are redundant and, if so, determine if they should indeed be having any side effects. If that's the case, add such side effects. If contrarily they are indeed redundant and do not affect the security in any way, then remove them.

The impact of such redundant statements could be indirect to security because of the errors (the logic that we talked about) or they could be direct, where such redundant statements are actually meant to enforce certain security checks. Because they are redundant those checks never get executed and directly impact the security of the contract in a negative way.

4.77 Storage array with signed Integers with ABIEncoderV2:

This specific compiler bug was related to storage arrays and signed integers, and their usage was enabled by the ABIEncoderV2, which was a `pragma` directive, that needed to be explicitly specified until the latest versions (as it is now used by default).

This specific bug arose when assigning an array of signed integers to a storage array of a different type `Type[]=int[]`. Under such assignments, it led to data corruption in that array.

This bug was present in versions `solc 0.4.7` until `solc 0.5.10` (which are much older versions than the latest one that we often encounter), so it's very unlikely that we'll look at smart contracts using these much older versions, but it is something to be kept in mind.

4.78 Dynamic constructor arguments clipped with ABIEncoderV2

A contract's constructor that takes structs or arrays that contain dynamically sized arrays (made possible because of ABIEncoderV2) reverted or decoded to invalid data.

This compiler bug was present in versions `solc 0.4.16` to `0.5.9`.

4.79 Storage array with multiSlot element with ABIEncoderV2

There was a compiler bug related to storage arrays in `Solidity`, specifically those with multi-slot elements, again made possible because of ABIEncoderV2.

Such storage arrays containing `struct` or other statically sized arrays were not read properly when they were directly encoded in external function calls or using the `abi.encode()` primitive.

This bug was present in versions `solc 0.4.16` to `0.5.10`.

4.80 Calldata structs with statically sized and dynamically encoded members with ABIEncoderV2

Another compiler bug was related to the `structs` type (specifically the call data structs) reading from call data structs that contained dynamically encoded, but statically sized members, could result in incorrect values being read.

This again was limited to the compiler versions `solc 0.5.6` to `0.5.11`.

4.81 Packed storage with ABIEncoderV2

There was a compiler bug related to packed storage. Storage structs and arrays with types that are smaller than 32 bytes when encoded directly from storage using ABIEncoderV2 could cause data corruption.

This occurred with compiler versions `solc 0.5.0` to `0.5.7`.

4.82 Incorrect loads with Yul optimizer and ABIEncoderV2

This is another compiler bug specifically coming from the Yul optimizer, part of it resulted in incorrect loads being done.

Yul, if you remember is `Solidity`'s Assembly language. When the experimental Yul optimizer was activated manually in addition to ABIEncoderV2, it resulted in memory loads and storage loads via `MLOAD` and `SLOAD` instructions to be replaced by values that were already written.

So effectively, the Yul optimizer replaced the `MLOAD` and `SLOAD` calls with stale values which as you can imagine, is a serious bug. This occurred with compiler versions `solc 0.5.14` to `0.5.15`.

4.83 Array slice dynamically encoded base type with ABIEncoderV2

There was a compiler bug specifically related to array slices, which there are views of the arrays that lets us access specific ranges of those arrays in a very efficient manner.

Accessing such array slices for arrays that had dynamically encoded base types resulted in invalid data being read for the compiler versions `solc 0.6.0` to `0.6.8`.

4.84 Missing escaping in formatting with ABIEncoderV2

This compiler bug was related to missed escaping. Escaping is relevant to string literals where certain characters can be escaped using the double backslash.

String literals that contained double backslash characters for escaping, that were passed directly to `external`, or encoding function calls, could result in a different string being used when ABIEncoderV2 was enabled.

Notice that this compiler bug was present across many compiler versions all the way from `solc 0.5.14` to `0.6.8`.

4.85 Double shift size overflow

If multiple conditions were true, then the shifting operations resulted in overflows resulting in unexpected values being output.

Some of those conditions were that the optimizer needed to be enabled. These had to be double bitwise shifts where large constants were being used whose sum overflowed 256 bits.

Under such conditions the shifting operations overflowed for the compiler versions `solc 0.5.5` to `0.5.6`.

4.86 Incorrect byte instruction optimization

This was a compiler bug originating from incorrect optimization of byte instructions.

The optimizer, when dealing with byte codes whose second argument was `31` or a constant expression that evaluated to `31`, incorrectly optimized it which resulted in unexpected values being produced.

This was possible when doing an index access on the `bytesNN` types (so all the types like `bytes1`, `bytes2` to `bytes32`) or when using the bytes opcode in assembly.

Unexpected values were produced when these conditions were met, from versions `solc 0.5.5` to `0.5.7`.

4.87 Essential assignments removed with Yul Optimizer

There was another compiler bug coming from the Yul optimizer. In this case, the Yul optimizer removed essential assignments for variables that were specifically declared inside `for` loops.

This would happen while using Yul's `continue` or `break` statements, and again limited to the compiler versions `solc 0.5.8/0.6.0` to `0.5.16/0.6.1`.

4.88 Private methods overriden

Remember that function visibilities in `Solidity` can be `private`, `internal`, `public` or `external`.

`private` functions are specific to the contract in which they are defined: they can't be called from any other contract, even those deriving from it.

While this is true, it was still possible for a derived contract to declare a function of the same name and type as a `private` function in one of the base contracts. And by doing so, change the behavior of the base contracts function.

What is interesting to note here, from a security perspective, is that this compiler bug was present across multiple versions all the way from `0.3.0` to `0.5.17`.

4.89 Tuple assignment multi stack slot components

Tuple assignments where the components occupied several stack slots, for example in the case of nested tuples, resulted in invalid values because of a compiler bug.

Notice again that this compiler bug lasted across 5 breaking versions: all the way from `0.1.6` to `0.6.6`.

4.90 Dynamic array cleanup

When dynamically sized arrays were being assigned with types whose size was at most 16 bytes in storage, it would cause the assigned array to shrink to reduce their slots.

However, some parts of the deleted slots were not being zeroed out by the compiler. This would lead to stale or dirty data being used. This bug was fixed in `0.7.3`.

4.91 Empty byte array copy

This bug is related to byte arrays, from `memory` or `calldata`, that were empty were copied to `storage` and they could result in data corruption.

This only occurred if the target array's length was subsequently increased, but without storing new data in it. Notice how specific the conditions are for this bug to be triggered. Nevertheless, this bug was discovered and fixed in version `0.7.4`.

4.92 Memory array creation overflow

When memory arrays were being created, if they were very large in size, then they would result in overlapping memory regions, which would lead to corruption.

In this case, this compiler bug was introduced in `0.2.0` and fixed in `0.6.5`.

4.93 Calldata `using for` compiler bug

Remember, `using for` primitive is used for calling library functions on specific types used within the smart contract.

In this case the bug was specific to when the parameters used in such function calls were in the calldata portion of the EVM. In such cases, the reading of such parameters would result in invalid data being read. This bug existed accross versions `solc 0.6.9` to `0.6.10`.

4.94 Free function redefinition

Remember, free functions in `Solidity` are functions that are declared outside contracts: they are declared at file level. This compiler bug allowed free functions to be declared with the same name and parameter types. This redefinition or collision was not detected by the compiler as an error. This bug was present in one of the recent versions: `0.7.1`, and fixed in `0.7.2`.

Compiler bugs should be taken very seriously because, unlike smart contracts that may differ from each other in the logic implemented, in the data structures or other aspects used, the compiler is a common dependency or perhaps a single point of failure for all the smart contracts compiled with that version.

Having said that, let's also recognize that a compiler is another software, so just like any software it is bound to have bugs and perhaps even more, because the compiler is significantly more complex than a smart contract or any other general software application.

From a security perspective, the things to be kept in mind when looking at a compiler version that's being used in smart contracts is to know which features of that compiler version are considered as being extensively used, and which are considered as experimental and perhaps staying away from them, so that one is not susceptible or vulnerable to any bugs in them.

It is also important to recognize the bugs that have been fixed in the compiler version, the bugs that have been reported and perhaps fixed in later versions (if those are available). These aspects should dictate the choice of the compiler version for the smart contracts and the specific features that are available within those compiler versions.

4.95 Proxy pitfalls: Initializers

The final set of security pitfalls and best practices that we'll discuss in this module are related to Proxy-based contracts.

Remember that Proxy-based architectures are used for upgradability and other aspects desired in smart contract applications. In this Proxy setup, there is typically a Proxy contract that does a `delegateCall` to a logic contract, and because of the `delegateCall`, the logic contract gets to implement logic that

executes on the state of the Proxy contract.

Under this specific setup, due to the `delegateCall` the data, the logic aspects has specific requirements that need to be met by both the Proxy as well as the logic contract. These lead to some security pitfalls and best practices.

In this particular pitfall, initializer functions should not be callable multiple times: they should be callable only once, and by the authorized Proxy contract as soon as the logic contract has been deployed.

Remember that under a Proxy setup, the implementation contract can't use a constructor to initialize its state, because it is working with the state of the Proxy contract that does a `delegateCall`. So instead, implementation contract is expected to declare an initializer function which does all the required initializations for it. Such functions need to have an `external` or `public` visibility because they need to be callable from an external contract (which is the Proxy contract).
The deployment is typically done from a deploy script or from a factory contract. Preventing multiple invocations is critical because such invocations could happen from unauthorized contracts (or unauthorized users). In those cases, they could re-initialize the contract with values that let them exploit some of the contract functionality.

The best practice here is to use `OpenZeppelin`'s OZ Initializable library, which provides an initializer modifier that can be applied to the initialize function, preventing it from being called multiple times.

4.96 Proxy pitfalls: State Variables

This is a pitfall related to the previous one discussed. This specifically applies to initializing state variables in the Proxy-based setup.

Constructors should not be used in the implementation (or logic contracts) to initialize its state, but using an initializer function instead. State variables in the implementation contract, similarly, should not be initialized in their declarations themselves because such initializations will not be reflected when the Proxy contract makes a `delegateCall` to this implementation.

So instead, the state variables should be initialized within the initializer function because otherwise they would not be set when the `delegateCall` happens.

4.97 Proxy pitfalls: Import Contracts

The contracts used in a Proxy setup may also derive from other libraries or other contracts within the project itself, which could be defined in other files. In this case they are imported to be used in the Proxy contract.

These imported contracts that the Proxy contracts derive from, should also adhere to the same rules discussed: the base contracts should also not use a constructor, they should be using an initializer function. In addition, such contacts should also not initialize state variables during declaration.

The best practice is to make sure that the imported contracts also follow those rules, because if not, the state would be uninitialized and using that state could result in undefined behavior or potentially even serious vulnerabilities.

4.98 Proxy pitfalls: selfdestruct

If a Data Proxy calls a logic implementation contract that has a `selfdestruct` call in it, then that logic contract would end up getting destroyed and thereafter all calls to that logic contract will end up delegating calls to an address without any code.

Similarly, the use of `delegateCall` may also cause issues because the logic implementation works with the state of the Data Proxy.

The best practice is to avoid entirely the use of `selfdestruct` or `delegateCalls` with Proxy-based contracts.

4.99 Proxy pitfalls: State Variables

In Proxy-based contracts, the order layout type and mutability of state variables declared in the proxy, the corresponding implementation (or different versions of the implementation), should be preserved exactly while upgrading. This is to prevent storage layout mismatch errors. These ones can lead to very critical errors if they are inherited.

The best practice is to make sure that these aspects of state variables are exactly the same in Proxy-based setups.

4.100 Proxy pitfalls: Function ID

Remember that `Solidity` and EVM have the notion of a function selector which is the `keccak256()` hash of the function signatures.

These selectors are used to determine which contact function is being called, so at runtime the function dispatcher in the contract byte code should determine (by looking at the function selector) if one of the functions in the proxy is being called or if this call needs to be delegated to the implementation contract.

In Proxy-based setups, a malicious Proxy contract may declare a function such that its function id collides (is the same as) with one of the Proxy Functions. So even though the call was targeting an implementation function, the malicious proxy, hijacks that call and could lead to the execution of a function that can cause an exploit.

The best practice here is to pay attention to the proxy, any trust assumptions related to the proxy and implementation contracts. Also, to check if the proxy has or can declare a function whose id might collide with one of the implementation contract functions.

4.101 Proxy pitfalls: Shadowing

Instead of the Proxy contract trying to hijack calls meant for the implementation by declaring functions whose IDs collide with the implementation contract functions, they could simply shadow the functions in the implementation contract.

This means that a Proxy contract can declare functions that have the same name, the same parameter numbers and types as functions in the implementation contract. In such a scenario, the function dispatcher would simply call the proxy contact function instead of forwarding it to the implementation contract.

This way, a malicious proxy can intercept (or hijack) calls instead of delegating it to the implementation contract and exploit this aspect to cause malicious behavior.

The best practice here is again to pay attention to the Proxy contract, the implementation contract, look at all the functions declared in both these contracts to see if any of the Proxy Functions are indeed Shadowing those in the implementation contract. If that is the case, recognize that such functions will be executed in the context of the proxy without being forwarded to the implementation contract.

Chapter 5

Security Pitfalls & Best Practices 201

5.1 ERC20 Transfer

This pitfall is specifically related to the `transfer` and `transferFrom` functions that allow transferring of `ERC20` tokens between addresses. According to the specification, these should return `bool` values, however not all token contracts adhere to the specification, so they may not return `bool` values: they may not return any value at all or they may return a different value of a different type. In such cases, callers that assume the `bool` values to be returned may fail, so the best practice here is for `ERC20` token contracts to make sure that they are returning `booln` values and for call sites to not make such assumptions: preferably sing `safeERC20` wrappers from `OpenZeppelin` that handle all the possible scenarios where `bools`, non-booleans or no values are being returned (and the contract simply revoked).

5.2 ERC20 Optional

The `ERC20` specification makes it optional for token contracts to implement `name`, `symbol` and `decimals` primitives. As a result, any contract that is interacting with `ERC20` contracts should make sure that these primitives are indeed present and implemented by those contracts. If they want to use them, the best practice is not to make an assumption that these perimeters will always be implemented by the `ERC20` contract because they are optional.

5.3 ERC20 Decimals

`ERC20` contracts have a notion of decimals which typically are 18 digits in precision, and therefore the token standard specifies using an `uint8` type to rep-

resent decimals, because that is sufficient to represent a value of 18. However, token contracts that do not adhere to the standard sometimes incorrectly use a `uint256` type for decimals. The best practice here is to check which type is being used by the `ERC20` contract and, if it is a `uint256` type, then have a further check to make sure that the decimal value is less than or equal to 255, because that is the maximum value that can fit within a `uint256` and a as required by the token standard.

5.4 ERC20 approve()

We have talked about the Race-condition risk from the `approve` function of `ERC20`. To summarize it again let's take a look at the same example that we discussed: we have a token Owner who has given an allowance of 100 tokens (`approve(100)`)to a spender, then wants to later decrease that allowance to 50 tokens (`approve(50)`). The spender may be able to observe this decrease operation and before that happens, it can frontrun in order to first spend the 100 tokens for which they already had the allowance from earlier. Then once `approve(50)` operation succeeds, they further spend those 50 tokens as well.

So effectively they have ended up spending 150 tokens while the Owner intended for them to only be able to spend 50 tokens. This is possible because of frontrunning (because of the Race-condition opportunity). The best practice here is to not use the `approve()` function, but instead use the `increaseAllowance()` or `decreaseAllowance()` functions that do not have this risk.

5.5 ERC777 Hooks

We have discussed the `ERC777` token standard which aims to improve some of what are considered as shortcomings of the `ERC20` standard and one of these improvements is the concept of hooks. These hooks get called before `send`, `transfer`, `mint`, `burn` and some other operations in these tokens. While they may enable a lot of interesting use cases, special care should be taken to make sure that these hooks do not make any external calls because such calls can result in reentrancy vulnerabilities. The best practice with `ERC777` tokens is to check for their hooks and make sure that external calls are not being made.

5.6 Token Deflation

There's a concept of token deflation that may happen in `ERC20` Token contracts. Some of these token contracts may take a fee when tokens are being transferred from one address to another. Because of this fee, the number of tokens across all the user addresses will reduce over time when they are transferred between those, so the number of tokens received by the target address may not be the same as the number of tokens sent by the sender. This depends on the amount

of fee and if the fee is being charged at all.

The best practices here with respect to token deflation is for token contracts to generally avoid the notion of a fee that causes deflation because that could break assumptions with the contracts that interact with this token contract. For smart contract applications that work with `ERC20` contracts, they should be aware if those `ERC20` contracts have this notion of deflation or not, and if so, make sure that their accounting logic takes care of this deflation. This is more of a concern in smart contact applications that allow their users to interact with them using arbitrary `ERC20` token contracts, and in such cases consider a guarded launch approach where the initial set of `ERC20` tokens that can be used with this contract does not have this notion of deflation.

5.7 Token Inflation

`ERC20` contracts could also have the opposite effect: token inflation. In this case, contracts generate interest for their token holders. This interest is distributed to holders while they make transfers. This effectively increases the number of tokens that are held by the user addresses over time, effectively meaning that when a token transfer happens, the recipient may receive more tokens than the amount originally sent that (reflecting the interest being distributed).

If the smart contract application is not aware of the `ERC20` contract generating interest, then those interest tokens may end up getting trapped in the `ERC20` contract without being realized. The best practice is again to avoid this notion of interest that causes inflation because their interacting contracts may make an assumption that no such thing is happening that could break a lot of the critical assumptions leading to vulnerabilities, or again such smart contact applications could consider a guarded launch approach where the `ERC20` tokens that they work with are known not to have this notion of interest and inflation.

5.8 Token Complexity

High token complexity is considered as a security risk. We have long known that complexity in general is very detrimental to security. The same aspect holds good for `ERC20` token contracts. These contracts should have a well-defined specification, they should be implementing a very simple contract because any unnecessary complexity could result in bugs: developers could make errors while developing these complex features. It is also much harder to reason about these complex features and definitely harder to find and fix bugs in such features, so the best practice is at a high level to avoid any unnecessary complexity when it comes to implementing token contracts.

5.9 Token Functions

In computer science, there is a notion of "*separation of concerns*" which says that in a computer application there should be different sections, each of which addresses a very specific concern. This applies to smart contact applications that work with `ERC20` token contracts as well. In this case what we mean is that a `ERC20` contract should only or mostly have functions that are relevant to `ERC20` tokens. They should not include any non-token related functions in them because that could introduce additional complexity, and like we just discussed complexity is detrimental to security because it could introduce bugs. At a high level one should avoid unnecessary complexity by bundling non-token related functions within a `ERC20` token contract because that increases likelihood of issues in general or in the worst case security vulnerabilities.

5.10 Token Address

`ERC20` contracts should be working with a single token address. What this means is that there should be a single address that maintains the balances of different users interacting with that contract, thus there is a single entry point for checking the balances of users. This is because multiple addresses within a contract can result in multiple entry points for the different balances that are held or maintained by those addresses, and not being aware of these multiple addresses and their balances can result in accounting bugs. The best practice is to make sure that an `ERC20` contract works with a single address.

5.11 Token Upgradeable

We have talked about upgradability in the context of the Proxy contact pattern. This upgradability is interesting in smart contract applications where the implementation part of the Proxy can be changed to a newer version to introduce new features or to fix any bugs in the previous versions. When it comes to `ERC20` token contracts, upgradability is a concern. The reason is because any change in functionality that is introduced by this upgreadeability is detrimental to the trust that the users place in these contracts. The rationale is that token functions in the contract are meant to be very simple: the `mint`, `burn` and `transfer` functions are required to adhere to the specifications so that all the contracts or all the users interacting with this token contract are assured of the functionality implemented by these functions. The best practice here is to make sure that these token contracts are not upgradable and, if an application is interacting with token contracts to check and verify that it is not upgradable.

5.12 Token Mint

Remember that in the context of token contracts, minting refers to the act of incrementing the account balances of addresses to which those new tokens are credited, and in this context of token minting one should be aware of the contract Owner having any extra capabilities over this functionality. If this is the case, then a malicious Owner could effectively mint an arbitrary number of tokens to any address of their choice, which as you can imagine is very detrimental to the security of the token contract because all the other users using this contract and maintaining balances of these tokens in that contract will be affected because their relative share of tokens will be much smaller. The best practice here is to be aware of any such extra capabilities over this printing functionality by the contact Owner because that could be abused.

5.13 Token Pause

Remember from the previous module that the ability to pause certain contract functionality is part of what is known as a guarded launch. However, when this guarded launch approach is applied to `ERC20` tokens, pausing some of their functionalities (like minting, burning or transferring) could be a concern because the authorized owners (who are allowed to pause and unpause such functionalities) or their addresses/accounts could be compromised (or they may even be malicious), resulting in pausing the contract functionality and trapping the funds of all the users interacting with that contract. The best practice here is to be aware of this risk and when interacting with token contracts to check and verify if those contracts are possible or not by certain authorized owners.

5.14 Token Blacklist

While the concept of blacklisting is commonly used in security for a long time to prevent malicious actors or actions from abusing the system this notion when applied to `ERC20` contracts is of concern the reason again is because authorized users who are allowed to create and maintain this blacklist by adding actors or actions into that list or by taking them out of that list those owners could be malicious or they could be compromised and in such scenarios where a token contract has this notion of a blacklist, then because of such malicious or compromised owners the users funds could get trapped, if their addresses are blacklisted, so the best practice is again to be aware of this risk and check and verify contracts to make sure that they do not have this notion of a blacklist and, if they do be aware of what can go wrong.

5.15 Token Team

Let's now talk about the deal behind the `ERC20` project and its implications on any security aspects. The team behind the `ERC20` project may be publicly known (we know who the project members are, what their past projects have been and how they are connected within the community) or this team could be anonymous (the project members are only known by their handles on github, twitter, telegram or discord... and we have very little information about what they have done in the past or about their real world identities and how they are connected within the social circles of the community).

In the latter context, there are two schools of thought:

1. One school of thought thinks that an anonymous team is riskier from a security perspective of the project because we do not have a good ways to evaluate what their reputation is within the social circles, with the community based on their past projects and so on... In this case the assumption is that there's a greater risk a security risk to the project because these anonymous teams could not be as concerned about security, because any security implications or exploits might not hurt the reputation from this project, or the team members could also be imagined to have left behind bugs or back doors within the project so that later they themselves could exploit the project (what is known as "*rugging*" the project. This school of thought believes that such anonymous teams should meet a higher bar when it comes to security (or security reviews on the flip side).

2. The other school of thought believes that anonymity (or pseudo-anonymity) should not matter to the security of the project. Any of your past projects based on who you are, what you have done and what your connections are within the community should not impact the security of a project. The project should be evaluated independently of who the project team members are.

Irrespective of which school of thought you may subscribe to, this is something to be kept in mind because privacy and anonymity are strong aspirational goals of Web 3 and any such team risk could potentially translate into a legal risk for someone who may review such projects or interact with it (as users).

5.16 Token Ownership

Token ownership refers to who owns the tokens and how many tokens they own. In scenarios where there are very few users who own a lot of those tokens, then such ownership situation will allow those owners to influence the price of those tokens, the liquidity of those tokens and any potential governance actions around those tokens, because those actions will be controlled by the token ownership. This is an aspect of risk from centralization because there are very few owners

holding a lot of tokens. This risk could manifest itself into a security risk as well.

5.17 Token Supply

It refers to the number of `ERC20` tokens that is supported by the token contract. This supply depends on what has been implemented for that particular contract and application. It could be either low or high. The concerning situation is when a particular `ERC20` contract has a very low supply of its tokens as this by implication means that the ownership may end up being concentrated within a few owners who own a significant part of the supply, in which case they have a significant influence over the price of those tokens their liquidity and therefore their volatility, so this scenario brings in an increased manipulation risk for such tokens with limited supply.

5.18 Token Listing

`ERC20` tokens get listed in various places to allow trading between users. These tokens may get listed on centralized exchanges or decentralized exchanges.

- Decentralized exchanges are expected to be more resilient to failures and therefore are expected to be up and accessible all the time.
- However, if token is listed on very few centralized exchanges and those exchanges happen to be inaccessible because they are down for maintenance or maybe in extreme situations where they are hacked, then a concern arises because majority of the tokens will now be inaccessible. This new low liquidity increase the price volatility of such tokens.

This is another aspect of centralization risk that one should be aware of when looking at tokens that are listed in very few exchanges.

5.19 Token Balance

Assumptions on token balances pose a security risk smart contract applications where the logic assumes that the balance of tokens that it is working with is always below a certain threshold. These applications stand the risk of those assumptions breaking if the balance exceeds those thresholds. This may be triggered by users who own a large number of tokens (typically known as whales), or it may also be triggered by what are known as flash loans.

A flash loan is a capability where a user is allowed to borrow a significant number of tokens without providing any collateral, but this loan has to be repaid or is forced to be repaid within the transaction itself. So by the end of the transaction, the flash loan capability makes sure that the tokens that were lent to the user

are paid back to that contract, but within that context of the transaction the user has access to a significant number of tokens as provided by that flash loan contract. Such a use of large funds or flash loans may be used by users or attackers to amplify arbitrage opportunities or exploit vulnerabilities where the logic incorrectly depends on load token balances. This risk from large funds or flash loans needs to be kept in mind because it could be manipulated.

5.20 Token Flash Minting

Similar to flash loans, there is the concept of flash minting that has similar concerns. Unlike flash loans, where the total amount of tokens that can be borrowed by a user is limited by the liquidity of tokens in that particular protocol, flash minting simply mints the new tokens that are handed to the user. These again are only available within the context of a transaction because at the end of the transaction, the flash minting mechanism will destroy all the tokens that were just minted and handed to the user. Similar to flash loans, if smart contracts that are working with such `ERC20` tokens make assumptions about the balances of those tokens that are available for a user, then they could lead to overflows or other serious security vulnerabilities, these again can be manipulated and there's a risk that needs to be kept aware of when dealing with external `ERC20` tokens.

5.21 ERC 1400 Addresses

So far we have looked at different security aspects of `ERC20` tokens let's now take a look at few other tokens that are nowhere as widely used as `ERC20` tokens, but introduce some concepts that are interesting from a security perspective.

One of such token standards is `ERC1400`. This token standard was driven by PolyMath and was related to the concept of security tokens (tokens that represent ownership in a financial security, and note that the security has nothing to do with the program or application security we are talking about). This token standard introduced the notion of permissioned addresses, which could block transfers from certain addresses. This is interesting from a security perspective because, if those addresses are malicious or if they can be compromised, then it leads to a denial of service (DoS) risk where transfers to and from such addresses can be blocked. This is a risk that we need to keep in mind if our smart contract application ever has to deal with `ERC1400` tokens.

5.22 ERC 1400 Transfers

Related to the notion of permissioned addresses, `ERC1400` also introduced the concept of forced transfers where there are trusted actors within the context of the standard that can perform unbounded transfers. These trusted actors can

transfer arbitrary amounts of funds to whichever addresses that they choose. This introduces a transfer risk that needs to be kept in mind when dealing with such tokens.

5.23 ERC 1644 Transfers

A related token standard to `ECR1400` is `ERC1644` that allows the concept of forced transfers that we just discussed. This is again in the context of a controller role, which is a trusted actor in this standard that is allowed to perform arbitrary transfers of funds to arbitrary addresses. The trusted actor, if malicious or compromised, can steal funds. In this `ERC` standard, there is a risk from the controller address that needs to be kept in mind.

5.24 ERC 621 totalSupply

`ERC621` token standard allows a different way to control the total supply of tokens. In this standard, there is a notion of trusted actors who can change the total supply after the contract is deployed. This is allowed using the `increaseSupply` and `decreaseSupply` functions that are specified by the standard. This introduces what is known as a token supply risk, where the token supply of such tokens can be changed arbitrarily after the contract is deployed.

5.25 ERC 884 Reissue

`ERC884` is another token standard that introduces yet another interesting security aspect. In this case, this token standard introduces the notion of cancelling and re-issuing. What this means is that the standard defines actors known as token implementers who can cancel addresses in the context of a contract that implements the standard. In that process, what these implementers do is that they move any tokens owned or held by those addresses to a new address while cancelling the older address. This, from a user's perspective, introduces token holding risk because if you are holding certain number of tokens in a particular address, then the token implementers could move those to a new address and cancel your existing address.

5.26 ERC884 Whitelisting

`ERC884` also introduces the concept of whitelisting addresses, where a certain set of addresses may be whitelisted by a contract implementing the standard. Token transfers are allowed only to such whitelisted addresses and not to addresses that don't exist in this whitelist. This again, as you can imagine, is a token transfer risk because a user might want to transfer tokens to a particular address but, if that is not whitelisted, then that token transfer is not allowed

5.27 Asset Limits

Let's now talk about a critical concept of guarded launch. This framework of ideas is widely used by almost every project in the ecosystem today in some form. It was put together and made popular by the team at Electric Capital. The fundamental idea is that when a new project is being launched, then there could be failures and vulnerabilities that have not been considered or discovered. Because of that, it makes a lot of security sense for the project to launch with minimal risk and over time increase the exposure as the project team gains more confidence in the normal functioning of the system. This idea is heavily inspired and motivated by a similar concept from the Web2 world: canary development. However in the Web3 world, because of notions of immutability, the difficulty of upgrading or updating code once it is deployed, the immediate nature, the extent of exploit possible make guarded launch in the Eeb 3 space somewhat different and perhaps difficult. It is nevertheless, more critical to implement and execute. There are multiple ways of doing guarded launches for smart contact applications. Let's take a look at the first such way of doing so.

The notion of asset limits: during launch the amount of assets that are managed by the smart contact application can be kept to a lower value than what is possible or desirable, and over time this asset value that is managed by the system can be increased gradually as we gain more confidence that this application does not have any further latent vulnerabilities that may get exploited and result in loss of these assets. The rationale again is that at launch time, there is likely a higher risk from latent bugs or failure modes that haven't been considered that could be exploited, so the best way to mitigate that is by introducing this notion of target launch and in this case specifically with asset limits.

5.28 Asset Types

Guarded launch can also be applied to asset types. smart contract applications can deal with multiple asset types (for example different types of `ERC20` tokens) and each of these asset types may be associated with a different level of risk. for example there could be `ERC20` tokens that are very widely used, well understood, time and battle tested that are generally considered as safer to use and there may also be newer or different `ERC20` token types that have slightly different behavior than those that are widely understood. Those come with a much greater risk because of the lower understanding for the lower use, so from a guarded launch perspective the idea is to launch with fewer asset types that are supported initially by the application and over time increase this in a certain manner. One way to do that is to first allow the use of known assets (those generally accepted as being safe by the community), then later on as the project matures and more confidence is gained, other asset types could be allowed within this application. This again mitigates the risk in a guarded launch approach.

5.29 User Limits

Limiting the number of users that can use a newly launched application (or new versions/features of the application). This is a widely used and time tested technique in the Eeb 2 world. The same concept applies to the Web3 world where, upon guarded launch using user limits, few trusted users are whitelisted or selected (based on certain criteria) and only these users are allowed to use or interact with the application. Over time, as the project team gains more confidence that these interactions by the selected group of users is as anticipated, they can open up the application to other users as well. The the outcome with this gradual approach is similar to the other ones where there is a higher risk initially, so limit it to a few trusted users and mitigate risk in that fashion, but then the idea is to gradually increase the exposure to other sets of users as well.

5.30 Usage Limits

Similar to user limits, we can also consider a guarded launch approach where there are usage limits: upon launch the usage is limited across certain criteria, then over time these limitations are removed to allow more extensive usage. This usage could be along the aspects of transaction size, volume of the transactions daily limits that are imposed on every user or even rate limiting per user or across all the users of that application. With these two guarded launch approaches of user limits and usage limits, it's easy to imagine how risk is mitigated because if something were to go wrong, then only that limited set of users (or limited set of transactions, or the limited set of value of the tokens or any other asset held by an application) is impacted.

5.31 Composability Limits

Composability is another aspect where we can apply the guarded launch approach. Remember that composability is a defining feature of Web3 where every application can expect to interact with or to be interacted with by any other application. So in this ecosystem this makes considering these applications as LEGOs that can be picked, chosen and combined in interesting ways to build applications that were originally unexpected. While this is a defining feature and even expected by design in the Web3 ecosystem, we have also talked about the security risks from unconstrained composability. Because applications can interact with and be composed with an arbitrary number and unknown applications, their differing assumptions, configurations, requirements and expectations could lead to failure modes that have not been considered or validated in the context of the application itself. Therefore, composability becomes critical from a guarded launch perspective.

One way to approach it is to, again, impose limits on composability where upon launch the application is only allowed to be composed (or interact) with

known applications (or protocol) and over time, extend this to arbitrary external smart contract applications that may pose an additional or increased risk. This gradual increase of exposure from whitelisted or trusted contracts extending to arbitrary untrusted contracts is another guarded launch approach.

5.32 Escrow

Another guarded launch approach is to use the familiar concept of Escrow from the traditional finance space. In this case, high value transactions (or high value operations) are escrowed where there are timelocks or specific governance capabilities that have the power to nullify or revert these transactions in case something unexpected happens with them. So the guarded launch approach is to first start off with an Escrow capability, which upon greater confidence in the system is removed.

5.33 Circuit Breaker

The next guarded launch approach is what is known as circuit breaker. This is perhaps the most widely used guarded launch approach by many of the smart contract applications that we see today. This is something that we discussed earlier where smart contracts allow certain authorized users to pause certain functions or functionalities of that smart contract when there is an emergency, and upon recovering from that emergency, there are unpause capabilities for those functionalities which again the authorized users can decide and trigger to recover from this emergency. This is something we discussed in the context of the `OpenZeppelin` pausable library that allows these capabilities to be applied selectively on different functions of a smart contract. So the guarded approach is to start off with this circuit breaker pause/unpause capabilities and later renounce to these capabilities, so that those authorized users need not be trusted with pausing and unpausing which, if abused, can lead to a DoS attack on those applications.

5.34 Emergency Shutdown

An extended or extreme version of the circuit breaker capability is what is known as emergency shutdown. In scenarios where simply pausing/unpausing the smart contract application does not help us recover from the issue at hand, and where there is something fundamentally broken or wrong with the smart contract application that needs to be fixed in a more involved manner, the emergency shutdown helps authorized users to turn off the smart contact applications from allowing users to further interact with it, and it also allows users to reclaim their assets that are held by that application and, where possible, this capability could also allow one to reset and restart such smart contact applications. From a guarded launch perspective, the idea is again to

launch an application with this emergency shutdown capability and over time, once we gain more confidence on the correct functioning of the system, remove this capability.

So far we have talked about removing these capabilities (the way that this is typically enforced within smart contracts is by removing the authorized users who can trigger those capabilities through setting the list of authorized addresses to an empty list or by setting them to Zero-addresses). this is yet another way or an extreme way to deal with the emergencies with this guarded launch approach.

5.35 System Specification

So far we have discussed security pitfalls and best practices focused on the `Solidity` language, the underlying EVM, the different token standards and so on... Now we are going to level up and discuss similar pitfalls and best practices, but focusing at the application level. These are software engineering best practices that have been developed and refined over decades, that apply specifically to smart contact applications as well. These application level aspects are arguably more important to discuss from a smart contract security auditing perspective because they can't be generalized across smart contact applications like we have done with `Solidity` or EVM level concepts. Because of that, there is a lack of tooling support for security pitfalls and best practices at this level, thus there is a greater dependency on manual analysis when it comes to security auditing. When that is insufficient (or incorrectly done) it has led to massive exploits that have resulted in losses of many millions of dollars.

With that context and motivation, let's talk about system specification. The design of any system or application starts with what is known as requirements gathering where such requirements are determined based on the target application category, the target market and the target users. Oncee those requirements are determined, they are translated (or coded) into a very detailed specification. This specification is required to describe in great detail how the different components of the system need to behave to achieve the design requirements and it's not just the "*how*" aspect, but also the "*why*" aspect: why is something being designed and specified the way it is being done. Without such a detailed specification, a system implementation will not have a baseline to be evaluated against the requirements that we have collected earlier. This is something critical for determining if the system behaves correctly, if the functions actually meet certain requirements that were designed (that were collected earlier). So to summarize, the design of a system begins with requirements, these requirements are translated into a very detailed specification which in future once we have an implementation allows us to evaluate, if the implementation actually meets the requirements system documentation.

5.36 System Documentation

System documentation is another critical component from a software engineering best practice. This is something that is often confused with specification. Remember that specification deals with design and requirements of the system whereas documentation deals with the actual implementation. The documentation describes what the different system components do to achieve the specification goals and how they do that. This has to cover various aspects related to the assets managed by that system, the actors within the context of that system and the various actions that these actors perform. It should also address the security specific aspects of the trust model and the threat model that are relevant to the system. So to summarize, in the design flow we start with the requirements that helps us create the specification which in turn helps us execute the implementation of that system. This implementation should be accompanied by extensive documentation that helps one evaluate it against the specification for correctness across various attributes.

5.37 Function Parameters

So with that high level view of system design let's now start discussing security aspects related to various application logic related constructs and concepts. The first one is function parameters. From a security perspective one should ensure that proper input validation has been performed for all function parameters. This is especially true if the visibility of such functions is `public` or `external`, because in these cases users who may potentially be untrusted can control the values that are assigned to these parameters and such tainted values can affect the control and data flow of the function and any logic thereafter. The best practice here is to make sure that there are valid sanity and threshold checks performed on these parameters, depending on what types they are. For example, if they are of the address type, then Zero-address validation should be performed because otherwise they could lead to exceptions during runtime or they could lead to tokens being burnt or access control being denied as we have discussed so far. The risk that we are trying to address here is from incorrect or invalid values being assigned to function parameters either accidentally or maliciously by users interacting with these functions.

5.38 Function Arguments

The arguments that are passed to functions, the call sites, that correspond to the function parameters are also something that need to be evaluated from a security perspective. At a high level, the arguments that are used at the call sites (the callers' arguments) should match the parameters that are required by the functions (or the callees). This matching should happen both in terms of their validity as well as their order, or in other words: the arguments at the call sites should be valid in that smart contract applications context to what

the function parameters expect. The order of such arguments should match the order of the function parameters as expected. These are the best practices that need to be followed when it comes to function arguments and the corresponding function parameters.

5.39 Function Visibility

We have discussed function visibility several times. This is something that is specific to the `Solidity` language that has four visibility specifiers. The order from maximum visibility to minimum visibility starts with `public`, `external`, `internal` then finally `private`. From a security perspective, to follow the principle of least privilege is critical to make sure that the strictest visibility is applied on the various functions. The reason is that, if a function is accidentally made `external` or `public` (when it should actually be `internal` or `private`) because of some critical functionality that should not be exposed to external, then this mistake can be exploited by users some of whom may be untrusted to invoke functionality that they are not supposed to have access to. This again is very relevant here because of the byzantine threat model.

5.40 Function Modifiers

Function modifiers are another interesting aspect of smart contracts written in `Solidity`. They are critical from a security perspective because modifiers are used to implement access control within the smart contracts and they're also used for different types of data validation in accounting and other application specific contents. Things to be kept in mind when analyzing modifiers is: to determine if any specific modifier is missing for the functions being analyzed, to check if they have been applied incorrectly on functions that either don't require these modifiers or that require these modifiers also.
If there are multiple modifiers used on a function, we have discussed how the ordering of the modifiers affects the logic implemented. Modifiers affect both control and data flow because from a control flow perspective, they could implement authorization checks that could revert if those checks fail, and therefore affect the control flow. They could also do different types of validation of the data that is being passed to the modifiers, in which case they do affect the data flow as well. The best practice with function modifiers is to ensure that correct modifiers have been used on the correct functions and in the correct order.

5.41 Function Returns

Smart contracts typically have multiple functions defined within them, and calls to such functions execute the logic within those functions, then return control back to the call sites. In many of these cases, the functions also return a value along with the control flow. Such return values should be analyzed to

make sure that the correct values are being returned. This is being done along all the paths within that function.

Another aspect to be checked is to ensure that for functions returning values, their call sites do indeed use those return values appropriately and do not ignore them. This is critical not only for the data flow aspect of the application logic context, but also from a security perspective. This is critical because this is the way that error conditions being returned by those function calls are caught and handled appropriately. Ignoring these could result in undefined behavior in the best cases and in the worst cases could result in serious vulnerabilities.

5.42 Function Timeliness

By timeliness we mean: when can these functions be called? Externally accessible functions (those with the `external` or `public` visibility) may be called at any time by users interacting with those smart contracts. On the flip side, they may never be called. The reason for this again is it could be accidental or it could be malicious, so it's not safe to assume that functions will be called in a very timely manner at specific system phases that make sense from the application logic context. Therefore, the implementation of functions within a contract should be very robust to track system state transitions, determine what state the system is currently in and in this state, which functions are expected (or make sense) to be called. For example, in the context of Proxy-based upgradable contracts where initialization functions are required to be used instead of constructors, such functions are meant to be called atomically along with contract deployment during construction to prevent anyone else from initializing those contracts with arbitrary values. Such initialization functions are not meant (or allowed) to be called after deployment.

5.43 Function Repetitiveness

Function repetitiveness is an aspect that refers to the number of times a function may be called. Again with `public` or `external` functions in a contract, they may be called any number of times by users. So it is not safe to assume that they will be called at all, called only once or a specific number of times as it makes sense to the application logic context. The function implementation and any state transitions happening within that function should not be making any assumptions on the number of times a particular function is called. They should be robust enough to track, prevent or ignore arbitrary repetitive invocations of functions or account for them in an idempotent way. Aagain, taking the example of Proxy-based upgradable contracts, initialization functions are meant to be called only once, which is why one of the security best practices is to use the initializer modifier from that `OpenZeppelin` library that we discussed earlier.

5.44 Function order

Along with timeliness and repetitiveness, the ordering of functions also matter. This refers to which function is called and when. `public`/`external` functions can be triggered by users in any order, so state transitions happening within those functions should not be making any assumptions on the order in which these functions are being called just because it makes sense from that application's context. The implementation should be robust enough to handle an arbitrary order of functions being called. This may again happen accidentally by users interacting with that application or it may be triggered maliciously. Again, taking the example of Proxy-based upgradable contracts and their requirement of initialization functions: such initialization functions are meant to be called before any other contract functions can be called, that ordering is critical because initialization functions initialize state variables. Allow any other contact function, that requires those state variables to be initialized, to be called would not make sense and could lead to vulnerabilities. So function ordering is something that needs to be paid attention to from a security perspective.

5.45 Function Inputs

Function inputs determine what data functions work with in the context of those particular function calls. `public` and `external` functions again can be called with any arbitrary input, so it is not safe for functions to make assumptions on the validity of the arguments that are being supplied to it. Without complete and proper validation on these inputs (these could be Zero-address checks, bound checks, sanity or threshold checks depending on the type of those arguments) we can't assume that these function inputs will comply with any assumptions being made about them in the function code.

5.46 Conditionals

Conditionals are used to affect the control flow aspects of the function implementation. Functions are rarely straight line code: they have different control flow constructs such as `if`, `else`, `for`, `while`, `do`, `break`, `continue` and `return`, within the `Solidity` smart contracts that are used to implement complex control flow to reflect the different conditions that these functions need to work with. Such conditionals have different predicates within them for the various checks that need to be enforced. Predicates involve the use of simple or complex expressions. These expressions involve operands or variables that are used along with operators. All these aspects of conditionals need to be checked to make sure that they enforce the control flow as anticipated by the developers. A common error is the use of the logical `or` (`||`) operator instead of the logical `and` (`&&`) operator within conditionals. These have caused serious security issues where they were being used to check for access control decisions. In such

cases the authorization checks would pass, if only one of the expressions in the predicate were `true` instead of requiring all of them to be `true`.

5.47 Access Control Spec

Access control deals with assets, actors and actions, or in other words which actors have access to which assets and how much of those assets and what actions can the actors use to access those assets. The access control specification should detail who can access what and why should they have that access, when can that access happen and how much of those assets can the actors access. All these aspects should be very accurately specified in great detail, so that they can be correctly implemented and enforced across the different contracts and functions, and across all the system transitions and flows that happen within those contracts and functions. This should help determine the trust, the threat models and any assumptions that are being made from this model. Without such an access control specification it will be very hard or even impossible to evaluate if the implementation actually enforces all these aspects.

5.48 Access Control Implementation

The implementation of access control should make sure that every aspect of the access control that was specified in the specification is implemented uniformly and accurately across all the actors on all the assets via all the actions possible. The implementation should make sure that none of the actors, assets and flow conditions within actions are missing or may be sidestepped. Such an implementation should help us evaluate if the access control enforcement has been done correctly according to the specification.

5.49 Access Control Modifiers

Access control is typically enforced in `Solidity` smart contacts by means of modifiers. Instead of implementing access control checks that are required for different functions multiple times in each of those functions, modifiers allow us to encapsulate those checks in one place and then these modifiers can be applied on any of those functions that require the access control checks implemented within them. While this encapsulation brings in the desired aspect of modularity, modifiers also impact auditability.

There's a school of thought that believes that modifiers are good for auditability: they make it easier because they implement all the checks in one place, so instead of reviewing the same checks multiple times in multiple functions these checks can be reviewed once within the modifier, then check if these modifiers are applied correctly to all the functions that require those checks, so that

makes auditability easier.

On the flip side, there's another school of thought that believes that modifiers are not as good for auditability as thought. The reason is that if there is a contract that has multiple modifiers and many functions that use those modifiers, then remember that the programming style guidelines recommend modifiers to be declared and defined at the beginning of the contract, and all the functions come thereafter so, if an auditor is reviewing functions deep down within the contract and it uses multiple modifiers, then they have to scroll up to the modifiers at the beginning of the contracts to check if the desired checks were implemented and if they were implemented correctly. This switching of context in the process of scrolling is believed to not lead to good auditability.

Nevertheless, modifiers are used extensively and reviewing these modifiers should make sure that they are indeed present on the functions that require the checks implemented by them, that modifiers implement valid checks in a correct manner and their order is also correctly specified for functions that use multiple modifiers.

5.50 Modifiers Implementation

Given the critical role of modifiers in access control, modifiers need to be implemented correctly. But what does that mean? Access control in smart contracts is enforced on different addresses that may be classified into different roles with differing privileges. Like we discussed in earlier modules, contracts may have a simple ownership based access control or a more flexible one based on RBAC. In such RBAC scenarios we need to check that modifiers are enforcing the correct checks on the correct roles, that such checks are composed correctly. Such a correct implementation is critical to access control which is the fundamental aspect of smart contract security and therefore needs to be reviewed very carefully.

5.51 Modifiers Usage

It is not sufficient to have the modifiers implemented correctly, but they should be used or applied correctly as well: the questions of which modifiers are used, why are they used, the how/what aspects, what are the parameters passed to them and what should they do with them, the order of modifiers when more than one is present, the when aspect (under what state transitions should they be applied), finally the where aspect (the functions where they're applied to). All such aspects of modifiers their functions and any parameters should have been considered correctly.

5.52 Access Control Changes

The access control implemented may need to be changed in some scenarios. In such cases, it is critical that the change is done correctly with respect to the assets actors or actions that are impacted. Using the wrong addresses for assets or actors, or allowing the changes to happen at the wrong times in the context of the application logic may lead to loss or locking of funds. Therefore, access control changes should be validated for correctness, use a two-step process to allow recovery from mistakes and also log changes for transparency and off-chain monitoring.

5.53 Comments

Code comments can be considered as part of documentation that is in line with the code. We should ensure that the code is well commented with the correct level of details and relevant information both with NatSpec and inline comments. This will help improve readability, maintainability and also auditability because comments can help document not only the functionality, but also the rationale behind it and any assumptions made, all of which can be analyzed while manually reviewing the code.

- The comments should accurately reflect what the corresponding code does.
- Discrepancies between code and comments should be addressed any to do's indicated by comments should also be addressed.
- Commented code and stale comments should also be removed

These are all the various aspects related to comments that need to be kept in mind while developing code or manually reviewing it.

5.54 Testing

Software testing or validation is a fundamental software engineering practice that is a critical contributor to improved security. Testing validates whether the system implementation meets the requirements as detailed by the specification. Unit tests, functional tests, integration and end-to-end tests should have been performed to achieve good test coverage across the entire code base. Changes introduced with any revisions should be validated with regression tests. Smoke testing indicates at a high level if the functionality works or not. Stress testing validates extreme scenarios with borderline cases to check if those have been considered correctly. Performance and security specific testing validates those aspects respectively. Any code or parameterization used specifically for testing should be removed from production code, which in smart contracts may apply differently to testnets vs. mainnet. Leaving test parameters or configurations behind may accidentally allow their usage resulting in unexpected maintenance

behavior or serious vulnerabilities, so overall we need to ensure that sufficient levels of testing have been performed across all these different categories we just mentioned.

5.55 Unused

Unused constructs may negatively impact security. This applies to any unused reports, inherited contracts, functions, parameters, variables, modifiers, events or return values; all of which should be removed or used appropriately after careful evaluation. Removing will not only reduce Gas costs, but also improve readability and maintainability of the code. Unused constructs may also be indicative of missing logic that may be a security concern, if that logic were to have implemented security related functionality, so one needs to either remove or use such unused constructs.

5.56 Redundant

Redundant constructs are also concerned. These are a kind of constructs that are not required either because there are equivalent constructs that implement the same functionality or because they are not relevant anymore. Such redundant code and comments can be confusing and should be removed or changed appropriately after careful evaluation. Similar to unused constructs, removing redundant constructs will not only reduce Gas costs but also improve readability and maintainability of the code. If redundant constructs are indicative of missing or incorrect logic, then they may be a security concern, if such logic were to have implemented security related functionality. So one needs to either remove such redundant constructs or make them relevant by adding or changing the corresponding logic.

5.57 ETH

Let's now talk about another fundamental aspect of smart contracts and Ethereum which is the way they handle Ether. Contracts that accept, manage or transfer Ether should take care of several things.

- They should ensure that functions handling Ether are using `msg.value` appropriately, remember that `msg.value` is a global variable in the context of a transaction which, for example when used or accounted multiple times (say inside loops) have led to critical vulnerabilities.
- They should ensure that logic that depends on Ether value accounts for either less or more Ether set via `payable` functions.
- Logic that depends on contract Ether balance, accounts for the different direct or indirect ways of receiving Ether such as `coinbase` transaction or `selfDestruct` recipient that we have discussed earlier.

- Logic that handles withdrawal balance and transfers does so correctly in any accounting logic.
- Transfers should be reentrancy safe.
- Ether can't accidentally get locked within a contract.

Functions handling Ether should also be checked extra carefully for access control input validation and error handling all these various aspects of Ether handling should be reviewed for correctness.

5.58 Tokens

Similar to Ether handling, contracts that accept manage or transfer `ERC` tokens should ensure several things:

- They should ensure that functions handling tokens, account for different types of `ERC` tokens such as `ERC20`, `ERC777` `ERC721`, `ERC1155`, etc...
- They should account for any deflationary or inflationary aspects of such tokens.
- Whether they are rebasing or not.
- Differentiate between trusted internal tokens and untrusted external tokens.

Different tokens could come with different peculiarities in terms of their decimals of precision, their return values, reverting behavio,r support for hooks, fungibility, supporting multiple token types or deviations from specifications. All of which could again result in susceptibility to re-entrances locking or even loss of funds. Therefore, functions handling tokens should be checked extra carefully for access control input validation and error handling to ensure that these aspects are handled correctly.

5.59 Actors

The aspirational goal in Web3 is for it to be a completely permissionless system where, ideally, there are no centralized trusted actors, such as admins, responsible for any aspect of smart contracts related to either development or management. Remember that Web3 aspires to be a zero trust system where no one needs to be trusted to use and not abuse the system, because everything is and should be verified. However, in guarded launch scenarios, the goal is to start with trusted actors/assets/actions and then progressively decentralize towards automated governance by the community. For the trusted phase, all the trusted actors (their roles and capabilities) should be clearly specified in the trust and threat models, implemented accordingly and documented for user information and any evaluation. This is a critical consideration in Web3's byzantine threat model.

5.60 Privileged Roles

Let's now talk about privileged roles. Trusted actors who have privileged roles in the context of the smart contact application with capabilities to deploy contracts modify critical parameters, pause and pause the system, trigger emergency shutdown, withdraw, transfer, drain funds and allow deny other actors, should ideally be addresses controlled by multiple independent and mutually distrusting entities.

They should not be controlled by private keys of externally owned accounts, but we are multiSig with the high pressure, say 5-7 or 9-11 depending on the criticality of the application, the value address, and eventually they should be governed by a community or a DAO (decentralized autonomous organization) of token holders. This is because an EOA is a single point of failure, if its key is compromised or the order is malicious multiSig on the other hand brings in the security design principle of privileged separation, which is tolerant to a few of the holders being malicious or compromised.

5.61 Privileged Roles pt.2

When such privileged roles within smart contracts are being changed it is recommended not to use a single step change because it is Error-prone. For example in a single step change, if the current admin accidentally changes the new admin to a Zero-address or an incorrect address that's where the private key is not available the system is left without an operational and the contract will have to be redeployed which is not easy or even entirely feasible in some scenarios.

Instead one should follow a two-step approach that we have discussed earlier, the current privileged role proposes a new address for the change and in the second step the newly proposed address, then claims the privileged role in a separate transaction. This two-step change mitigates risk by allowing accidental proposals to be corrected instead of leaving the system unoperational with no or malicious privileged.

5.62 Critical Parameters

When critical parameters of systems need to be changed it is recommended to enforce the changes after a time delay that is coupled and locked with that logic. This is to allow systems users to be aware of such critical changes and give them an opportunity to exit from that system, if they do not like the upcoming changes, or adjust their engagement in any other way with the system accordingly.
For example reducing rewards increasing fees or changing trust models in a system might not be acceptable to some users who may wish to withdraw their

funds before the change and exit. Such a time locked execution of delayed change enforcement needs to be combined with event emission to notify users of upcoming changes via off-chain interfaces or monitoring tools. So the best practice is a time delay change for critical parameters that is broadcasted using events to users monitoring via off-chain interfaces the goal is to surprise less be more transparent and fair.

5.63 Explicit Vs Implicit

As a general principle everything in security is about being explicit. Instead of being implicit, implicit assumptions, implicit trust or threat models implicit acceptance of assets actors actions lead to security vulnerabilities whereas, if they are explicitly specified implemented and documented they can be reasoned about and evaluated from a security perspective.

Even with the `Solidity` language it has progressively adopted explicit declarations of intent over the versions such as with function visibility and variable storage. So it's recommended to do the same at the application level where all requirements should be explicitly specified, so they're accurately implemented and lend themselves to validation. Implicit requirements specification and implementation assumptions should be explicitly documented and validated for correctness. Any latent implicit requirements and assumptions should be flagged as being dangerous.

5.64 Configuration

Security issues arise not only from implementation errors, but also from this configuration of system components, such as contracts, parameters, addresses and permissions all of which may lead to security issues. Such configuration aspects should be documented and validated test configurations should be clearly marked as such and separated appropriately from production configurations.

This is critical because testing is typically done with lower thresholds of different values to allow for faster or easier testing, they may also use more acceptable trust models or lower levels of thread than what is encountered in a production setting. So the best practice is to check configuration settings and make sure that they are correct relevant and validated for a production deployment.

5.65 initialization

Lack of initialization. Initializing with incorrect values or allowing untrusted actors to initialize system parameters may lead to security issues this is especially true for critical parameters, addresses, permissions and rules within the

system because the default or incorrect values may be used to exploit the system. Either technically or economically. The best practice therefore to avoid security pitfalls from initialization is to check that it is done and done correctly using the right values and done, so by only the authorized users.

5.66 Cleanup

Missing the cleaning up of old state or cleaning up incorrectly or insufficiently will lead to reuse of stale state which may lead to security issues. Cleaning could be in the context of using `Solidity`'s delayed primitive or even simply re-initializing variables to default values in the context of the application's logic.

For example this is applicable to contract state maintained in state variables within storage or even local variables within contact functions, where old scale values may lead to incorrect reads or rights in the context of the contract's logic. Cleaning up storage state using `delete` primitive provides Gas refunds with an EVM some of which has changed in recent upgrades, London upgrade for example reduced Gas refunds of s stores. Nevertheless, there are benefits besides security to this aspect of cleaning up.

5.67 Data Processing

At a very high and perhaps abstract level data, processing issues may lead to security issues in the application logic's context this could arise from several reasons such as while processing critical data or from processing of painted input data.

Processing could be missing or incorrectly implemented this could either resolve from a faulty specification or implementation without being caught during validation therefore all aspects of data processing should be reviewed for potential security impact

5.68 Data Validation

A specific aspect of data processing that we just discussed is data validation where contract functions check, if they receive data from external users or other contracts is valid, based on aspects of variable types, lower high thresholds or any other application logic specific context.

Validation issues very frequently lead to security issues. Missing validation of data or incorrectly insufficiently validating data especially tainted data from untrusted users will cause untrustworthy system behavior which may lead to security issues. Sanity and threshold checks are therefore critical aspects of data validation.

5.69 Numerical Issues

Another specific type of data processing is numerical processing, where the logic operates on numerical values incorrect numerical computation will almost always cause unexpected behavior some of which may lead to serious security issues. If not accounting miscalculations these may be related to overflow/underflow, precision handling, type casting, parameter return values, decimals, ordering of operations with multiplication/division and loop indices among other things.

The recommended best practice is to adopt widely used libraries for special mathematical support such as Fixed-point or floating point numbers and combine this with extensive testing using fuzzing and other tools meant to specifically test constraints and invariants for numerical issues.

5.70 Accounting Issues

Another specific type of data numerical processing is that related to accounting incorrect or insufficient tracking or accounting of business logic related aspects. Such as states phases permissions, rules, deposits, withdrawals of funds, mints. births, transfers of tokens or rewards penalties, fees within d5 applications all these may lead to serious security issues. We have seen numerous vulnerabilities specifically related to this aspect.

Therefore accounting aspects related to application logic states or transitions or numerical aspects as outlined earlier should be carefully reviewed to make sure they are correct and complete.

5.71 Access Control

We have discussed this multiple times, but this aspect of access control, warrants revisiting again because it's central and critical to security. Incorrect or insufficient access control or authorization related to system actors rules assets and permissions, may certainly lead to security issues.

Therefore the notion of assets actors actions in the context of trust and threat models should be reviewed with the utmost care to avoid such security issues.

5.72 Auditing & Logging

Recording or accessing snapshots or logs of important events, within a system is known as audit logging. The recorded events are called audit logs note that this auditing from a logging perspective is different from the concept of external reviews, which is also called auditing. Auditing and logging are important for

monitoring the security of an application.

In the context of smart contracts this applies to event emissions, the ability to query values of public state variables, exposed getter functions, and recording appropriate error strengths from requires, asserts and rewards. Incorrect or insufficient implementation of these aspects will impact off-chain monitoring and instant response capabilities which may lead to security issues. Correct and sufficient audit and logging is therefore something that also needs to be paid attention to for reasons of monitoring detecting and recovery aspects of security.

5.73 Cryptographic

Incorrect or insufficient cryptography, especially related to on-chain signature verification, or off-chain key management, will impact access control and may lead to security issues. So, aspects of keys accounts hashes signatures and randomness need to be paid attention to along with the fundamental concepts of ECDSA signatures and `keccak-256` hashes.

there are also other deeper and dual cryptographic aspects one will encounter in Ethereum applications or protocol upgrades with abbreviations such as `BLS`, `RANDAO`, and `VDF` and also zero knowledge (ZK) aspects. At a high level, cryptography is fundamental and critical to security and even a tiny mistake here can be disastrous surely leading to security vulnerabilities.

5.74 Error Reporting

Incorrect or insufficient detecting reporting and handling of error conditions will cause exceptional behavior to go unnoticed which may lead to security issues. At a high level security exploits almost always focus on exceptional behavior that is normally not encountered or validated or noticed.

Such exceptional behavior is what is anticipated caught and reported by error conditions. Any deviations from the specification are errors that should be detected reported and handled appropriately by the implementation.

5.75 DoS

Denial of service or DoS is also a security concern. Traditionally security has been considered as a triad referred to as CIA which stands for confidentiality integrity and availability. DoS affects availability and in this case that of the smart contract application. Preventing other users from successfully accessing system services by either modifying system parameters or shared state causes denial of service issues which affects the availability of the system.

The effects of this could cause users to have their funds locked reduce profits prevent from having their transactions included and therefore interactions with the contracts denied. Attackers may cause DoS without any apparent or immediate economic benefits to themselves and do, so by spending Ether on the Gas or any other tokens required for such duress causing interactions, which is typically referred to as griefing. So the best practices here are to recognize and minimize any such attributes in the smart contracts or application logic that could enable dos.

5.76 Timing

Timing issues can have a security impact. Incorrect assumptions on timing of user actions which can't be controlled. Triggering of system state transitions or dependencies on blockchain state blocks transactions may all lead to security issues depending on the application logic context. Therefore any timing attributes or logic within smart contact applications should be analyzed to check for such issues.

5.77 Ordering

Similar to timing issues incorrect assumptions on ordering of user actions or system state transitions may also lead to security issues. For example a user may accidentally or maliciously call a finalization function or other contract functions even before the initialization function has been called, if the system allows this to happen.

Attackers can front run or back run user interactions to force assumptions or ordering to fail Front-running is when the attackers race to finish their transaction or interaction before the user. Back-running is when they raise to be behind or right after the user's transaction of interaction.

Combining these two aspects can also be exploited in what are known as sandwich attacks where the user's transaction or attraction is sandwiched between those from the patent. So the best practice is to pay attention to the related aspects of timing and ordering attributes and evaluate, if they can be abused in any manner.

5.78 Undefined Behavior

Undefined behavior that is triggered accidentally or maliciously may lead to security issues. But what is undefined behavior? Any behavior that is not defined in the specification, but is allowed either explicitly or inadvertently in the implementation is undefined behavior.

Such behavior may never be triggered in normal operations but, if they are triggered accidentally in exceptional conditions that may result in rewards. However, if such behavior can also be exploited in some manner that leads to security issues in some cases it may not be clear, if such undefined behavior is a security concern or not, but nevertheless should be treated as such. The best practice is to make sure all acceptable behavior is detailed in the specification implemented accordingly and documented thoroughly.

5.79 Interactions

External interactions can have a security impact. Such interactions could be with assets actors or actions that are outside the adopted trust and threat models and hence external. Interacting with such external components for example tokens contracts or Oracles forces the system to trust or make assumptions about their correctness or availability which requires validation of their existence before interacting with them and any outputs from such interactions

Therefore such external interactions can have security implications and need to be considered carefully. Increasing dependencies and composability make this a significant challenge.

5.80 Trust

Trust is a fundamental concept in security. Thus minimization (or zero trust in the extreme case) is often the aspirational goal because trusted assets actors actions may be compromised or become malicious to subvert security. Trust minimization is a foundational value upon which Web3 is being picked, and one of the key tenets of decentralization where the notions of insiders and outsiders blurred and users may misuse the system under assumptions of Byzantine threat models.

So incorrect or insufficient trust assumptions about or among system actors and external entities may lead to privileged escalation or misuse, which may further lead to security issues the best practice therefore is to never trust, but always verify both the principle as well as in practice.

5.81 Gas

Remember that the notion of Gas and Ethereum stems from the need to bound computation because of the Turing completeness of the underlying EVM. Incorrect assumptions about Gas requirements especially for loops or external calls will lead to Out-of-Gas exceptions which may further lead to security issues such as failed transfers or locked funds. Gas usage must therefore be considered

while reviewing smart contracts to evaluate any assumptions leading to security implications of denial of service.

5.82 Dependency

Dependencies on external actors assets actions or software such as contracts, libraries, tokens, Oracles or Relayers will lead to trust correctness and availability assumptions which, if or when broken may lead to security issues. Dependencies therefore should be well documented and evaluated for such trust assumptions and threat models.

5.83 Constant

Issues may arise, if you assume certain aspects to be constant. That is they do not change for the duration of a transaction or even the contract's lifetime, but in fact they are not constant and change for some reason. Hard-coded assumptions could manifest for example in hard-coded contract configuration parameters. Example: Block times, block Gas Limits, opcode Gas prices, addresses, roles or permissions. Any such incorrect assumptions about system actors entities or parameters being constant may lead to security issues, if and when such factors change unexpectedly.

5.84 Fresh

Freshness of an object is an aspect that indicates, if it is the latest one in some relevant timeline or, if it is stale indicating that there is an updated value or version in that corresponding timeline. Using stale values and not the most recent values leads to freshness issues that could manifest into security issues.

Concrete examples are the use of nonsense in transactions to prevent replay attacks by repeating older transactions or the asset prices obtained from Oracles which, if stale can cause significant accounting issues leading to price manipulations and resulting vulnerabilities. Therefore increased assumptions about the status of or data from system actors being fresh because of lack of updation or availability may lead to security issues, if and when such factors have been updated and result resultant stale values being used instantly.

5.85 Scarcity

Scarcity refers to the notion that something is available in only few numbers. This may refer to assets or actors in the context of an application where assumptions may be made that, there are only a few assets or actors interacting with the application. Incorrect assumptions about such Scarcity say for

example tokens funds available to any system actor will lead to unexpected outcomes, if those assumptions are broken which may further lead to security issues.

For example susceptibility to flash loads or flash mints, related overflows is an example where the vulnerable contract makes a Scarcity related assumption and applies that to the size or type of variables used to maintain token balances. Which, if broken because of flash loans or mints can lead to overflows, if not mitigated appropriately. This is also related to civil attacks where an attacker subverts a system by creating a large number of identities and uses them to gain a disproportionately large influence where the system assumption on fewer unique identities is broken in some sense. Therefore one needs to evaluate if there are any Scarcity or abundance assumptions in an application that could cause security issues.

5.86 Incentive

Incentives are another fundamental aspect of blockchains and Web3. Mechanism design or crypto economics dictates almost everything in this space including infrastructure provisioning development and governance of systems. What incentives are provided and how much incentives are provided may be used or abused while interacting with smart contract applications.

Incentives could be either rewards or penalties, so for example incentives to liquidate positions in defile lending or applications of incentives to cause denial of service or briefing of a system. Incorrect assumptions about such incentives for system or external actors to either perform or not perform certain actions will lead to expected behavior not being triggered or unexpected behavior being triggered both of which may lead to security issues.

5.87 Clarity

Lack of clarity in assets actors or actions or system specification, documentation, implementation, user interface or user experience will lead to incorrect assumptions and unanticipated expectations or outcome which may lead to security issues. Therefore increasing the clarity by clearly thoroughly and accurately specifying implementing and documenting all security relevant aspects will help in mitigating risks from lack of clarity.

5.88 Privacy

Privacy and security are very closely related. In this context there could be privacy issues related to assets actors and their actions. Remember that data and transactions on the Ethereum blockchain are not private anyone can observe contract state and track transactions both included in the block, those pending

in the `mempool`. So incorrect assumptions about such privacy aspects of data or transactions that manifest in implementation or assumed trust and threat models can be abused leading to security issues.

5.89 Cloning

Cloning in this context refers to copy pasting code from other libraries contracts different parts of the same contract or from entirely different projects with minimal or no changes. The configurations context assumptions bugs and bug fixes for the original code may be ignored or used incorrectly in the context of the cloned code.
This may result in incorrect code semantics for the context being copied to copy over any vulnerabilities or miss any security fixes applied to the original code all of which may lead to security issues. There have been security vulnerabilities because of cloning incorrectly some of which have led to exploits as well. Cloning therefore is risky and has serious security implications.

5.90 Logic

The last concepts we've discussed are generalizations and higher level concepts related to application logic level issues that can't be specifically codified in tools or generalized because they differ across applications. These are perhaps much harder to reason and detect because it requires deep understanding of the application logic and hence there's mostly manual effort in security reviews. Such business logic which is application specific should have been translated from requirements to the specification, then implementation with all of it validated and documented accurately especially the security relevant aspects.

Without that security reviewers have to infer assumptions constraints program and variants trust and threat models which is not very effective or efficient. Application logic related vulnerabilities are perhaps the hardest to detect and have resulted in serious exploits. This is therefore of utmost importance to security and hopefully the 50 + concepts that we have discussed for the majority of this module will be helpful in doing so.

5.91 Principle #1

We will now discuss the 10 principles from Saltzer and Schroeder's secure design principles which are proposed by them in 1975 and have been widely cited and used in various aspects of information security ever since.

The first one is that of least privilege which states that every program and every user of the system should operate using the least set of privileges necessary to complete the job which means that we should ensure that various system actors

have the least amount of privilege granted as required by their roles to execute their specific tasks. Because granting excess privilege that what is absolutely required is prone to misuse or abuse when trusted actors misbehave or their access is hijacked by malicious entities privileges. Therefore should be need-based.

5.92 Principle #2

The second principle is about separation of privileges which states that where feasible a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.

This means that we should ensure critical privileges are separated across multiple actors, so that, there are no single points of failure or abuse. A good example of this is the use of a Multi-Sigs address versus an EOA for privileged actors such as contract Owner, admin or gobernement? who control key contract functionalities such as pause and pause shutdown, emergency fund, drain, upgradability of contracts, allow, deny lists and critical parameters.

The multisig address should be composed of entities that are different and mutually distrusting or verifying because such a privilege separation prevents single points of failure.

5.93 Principle #3

The third principle is that of least common mechanism. Which states that we should minimize the amount of mechanism common to more than one user and depended on by all users.

This means that we should ensure that only the least number of security critical modules or paths as required, are shared amongst the different actors of code, so that impact from any vulnerability or compromise and shared components is limited and contained to the smallest possible subset.

In other words common points or parts of failure are minimized, there are pros and cons of this approach that need to be made in depending on the context.

5.94 Principle #4

The fourth principle is that of fail-safe defaults which states that we need to base access decisions on permission rather than exclusion, so we need to ensure that variables or permissions are initialized to fail-safe default values which deny access by default, but can later be made more inclusive or permissive, if

and when necessary.

Instead of opening up the system to everyone by default which may include untrusted actors. We have discussed this in the context of guarded launch for assets actors and actions. Such fail-safe initial defaults could apply to function visibility critical parameter, initializations and permissions of assets actors and actions, there are again pros and cons of this approach that need to be considered as it applies to open or closed systems given the emphasis of Web3 on aspects of openness permissionless participation and composability among other things.

5.95 Principle #5

The fifth principle is that of complete mediation which states that every access to every object must be checked for authority. Which means that we should ensure that any required access control is enforced along all access paths to the object or function being protected. Examples are missing modifiers, permissive visibility or missing authorization flows. Complete mediation, therefore requires access control enforcement on every asset after action along all paths and at all times.

5.96 Principle #6

The sixth principle is that of economy of mechanism, which says keep the design as simple and small as possible. Which in this context can be applied to ensure that contracts and functions are not overly complex or large, so as to reduce readability maintainability or even auditability. This embodies the keep it simple and stupid or KISS Principle in some ways because complexity typically leads to insecurity and hence should be kept as low as possible.

5.97 Principle #7

The seventh principle is that of open design which states that the design should not be secret. This is especially relevant to the Web3 space as we have discussed earlier because smart contracts are expected to be open-sourced, verified and accessible to everyone for permissionless participation and composability. Security by obscurity of code or underlying algorithms is not an option. Security should be derived from the strength of the design and implementation under the assumption that Byzantine attackers will study their details and try to exploit them in arbitrary ways.

5.98 Principle #8

The eighth principle is that of psychological acceptability which states that it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Which in our context means that we need to ensure that security aspects of smart contract interfaces and system designs flows are human friendly and in queue them, so that we can program them or use them with ease and with minimal risk. This is a significant challenge in the web 3 space today where, there is a lot of early and experimental software undergoing rapid changes, but something to be kept in mind from a security perspective as things evolve and systems get more mass adoption.

5.99 Principle #9

The ninth principle is work factor, which recommends to compare the cost of circumventing the mechanism with the resources of a potential adapter. Which is very relevant and perhaps at an extreme in the case of smart contracts in Web3 because given the magnitude of value managed by smart contracts it is safe to assume that Byzantine attackers will risk the greatest amounts of the resources possible across intellectual social and financial capital to support such systems. And given the general state of current smart contracts the cost of circumventing is not very high, relative to hardened software or systems in the Web2 space for various reasons that we have discussed earlier.

The rewards from exploiting them are in tens or even hundreds of millions of dollars in some cases, so the risk versus reward is extremely skewed here. Therefore the mitigation mechanisms must appropriately factor in the highest levels of threat and risk.

5.100 Principle #10

The final tenth principle is about compromise recording which states that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

One way to interpret this is to say that achieving improving bug-free code is theoretically and practically impossible for real world smart contracts. Therefore one should strive for the best in performing all security due diligence and reduce the attack surface as much as possible. While in the same time, anticipate residual risk to exist in the deployed system. Anticipate that there will be potential incidents that exploit them and therefore have an instant response plan ready for that.

For doing that we can ensure that smart contracts and their accompanying operational infrastructure can be monitored and analyzed at all times for minimizing loss from any compromise due to vulnerabilities and exploits. As a concrete example critical operations in contracts should emit events to facilitate off-chain monitoring at runtime, where the available monitoring tools are used on smart contracts of interest to analyze not only such events, but also transactions interacting with them their Side-effects and potential security impacts.

Chapter 6

Audit Techniques & Tools 101

Intro

Welcome to the sixth module of the securium boot camp this one is about order techniques and tools 101, the central focus of this bootcamp on smart contact security auditing in this module we will cover the various technical and non-technical aspects of smart contact auditing starting with the high level `view` of what our audits, the entire context around that we will, then review the widely used tools in this space developed by teams from Trail of Bits consensus diligence and others. We will cover high level aspects of these tools without getting too much into their operational or technical details which is autoscope and for which I would highly encourage you to install these tools and experiment with them during this module finally we will review the audit process, the various aspects around that one will need to understand to become a smart contact security auditor, so let's dive in.

6.1 Audit

Audit is an external security assessment of a project code base. In contrast to a review or assessment done internally by the project team itself.
This external assessment performed by a third party external to the project is typically requested and paid for by the project team.
It's meant to detect and report security issues with their underlying vulnerabilities `severity` difficulty potential exploit scenarios and recommended fixes this includes both common security pitfalls and best practices and also deeper application logic and economic vulnerabilities in the context of smart contracts.
It may also provide subjective insights into code quality documentation and testing the scope depth format of audit reports varies across auditing teams, but they generally cover these similar aspects.

6.2 Scope

As for the ordered scope for Ethereum-based smart contract projects the scope is typically restricted to the on-chain smart contract code and sometimes includes the off-chain components that interact with the smart contracts as well
But for this module, the boot camp as a whole we are focusing only on smart contract security auditing.

6.3 Goal

The goal of audits is to assess project code along with any associated specification and documentation and alert the project team of potential security related issues that need to be addressed to improve the security posture, decrease the attack surface and mitigate risk.
This typically happens before smart contracts are deployed on the main net before launch, so that vulnerabilities can be fixed and verified to avoid exposure.

6.4 Non-Goal

Along with the goals we should also discuss what the non-goals of audits are, this is perhaps even more important to level set the expectations.
Audit is not a security warranty of bug-free code by any stretch of imagination. It is a best effort endeavoured by trained security experts who are operating within reasonable constraints of time understanding expertise and of course decidability, so just because the project has been audited does not mean that it will not have any vulnerabilities.
It should certainly have fewer vulnerabilities than before the audit assuming the reported vulnerabilities were fixed correctly.
The constraints are also critical and real, especially that of time and understanding. For now we can assume that most auditors are self-trained, with some help from peers with their experience in smart contact development or security in the web2 space being applied to web3.
The expertise of auditors also significantly affects the effectiveness of audits and we'll talk more about these and forthcoming slides.

6.5 Target

Who is the target for audits currently security firms or teams `execute` audits for their clients who pay for their services audit engagements are therefore geared or targeted towards priorities of clients that's the project owners and not project users or investors.
The goal of audits therefore is not to alert potential project users of any inherent risk that may be evaluated during the audit that is not the business or technical goal of audits.

This is often a point of discussion when it comes to audit firms their incentives and what they should be doing or not doing and also in the context of where potential project users should look for unbiased security risk posture of the projects that they're interested in. Nevertheless this is the current state of most audits today where their clients are projects and not users or investors of such projects.

6.6 Need

Let's start with the fundamental question of why we even have audits in the web3 space the reasons are simple, but multi-fold and mostly related to talent market supply demand and some unique characteristics of the web3 space.
Smart contract-based projects do not have sufficient in-house Ethereum smart contract security expertise and presumably not even the time to perform internal security assessments given the base of innovation in the space therefore they rely on external experts who have the domain expertise in those areas.
The reason most projects don't have that expertise is because the demand for it is orders of magnitude higher than the supply which itself is because we are still very early in the web3 life cycle, this is also the biggest motivation for this boot camp as even,
If projects have some expertise in house given the risk and value at stake they would still benefit from an unbiased external team with superior and either supplementary or complementary security skill sets that can review the assumptions design specification and implementation of the project code base. So these aspects hopefully justify at a high level the need for security audits in the current landscape.

6.7 Types

Now what are the types of audits there aren't any standard categories, but we can consider some broad classifications based on the nature of such audits audits depend on the scope nature status of projects and based on that they generally fall into the following categories.
We can think of new and repeat audits new audits are for new projects that are just being launched for the first time. Repeat audits on the other hand are for existing projects that have had an audit or two before, but is being revised there's a new version of this project coming up with new features or optimizations for which a repeat order is being performed.
Then, there is a fixed audit for reviewing the fixes made to the findings from a current or prior audit.
Then we can think of a retainer audit where the audit is constantly reviewing project updates or providing guidance in a continuous manner instead of discrete engagements.

Finally, we can think of incident audits which review and exploit incident root cause the incident identify the underlying vulnerabilities that led to the incident and propose fixes, this one is more of an instant response unlike the traditional audits described are also very likely other variants of these as well, but this should give a general idea of the types of audits which affect the scope and nature of engagements as well.

6.8 Timeline

The timeline or the time spread for audits depends on the scope nature status and more importantly the complexity of the project to be assessed and also the type of audit
This may vary from a few days for a fix or retainer audit to several weeks for a new repeat or instant audit that we discussed in the previous line this may even `require` months for projects with complex smart contracts with lots of external dependencies.
The timeline should certainly depend on the anticipated value at risk in those smart contracts and their criticality, but that is generally hard to guess ahead of time the timeline aspect is therefore a subjective one and there aren't reasonable objective measures to make decisions it's usually decided by simple metrics such as the number of files in that project the lines of code external dependencies or Oracles or complex mathematical libraries some measures of complexity of code or the application functionality in general and even familiarity of the auditing team with such contracts from earlier engagements.

6.9 Effort

The audit effort from a resources perspective typically involves more than one auditor simultaneously for getting independent, redundant or supplementary complementary assessments of the project the more than one approach is generally preferred to deal with any blind spots of individual auditors stemming from expertise their experience or even just luck.

6.10 Cost

The cost of an audit is an often discussed and debated topic it depends on the type and scope of audits and typically costs in the range of several thousands of dollars per week depending on the complexity of the project, the market demand and supply for audits at that point in time and certainly the strength and reputation of the auditing firm.

6.11 Pre-Reqs

The prerequisites for an audit are the things that should be factored in discussed agreed upon and made available before an audit begins.
This should typically include the following 10 things clear definition of the scope of the project to be assessed typically in the form of a specific commit hash of the project files on a GitHub repository
Which could be a public or a private repository, if the project is still in stealth mode
The team behind the project which could be public or anonymous and is engaged throughout this process
The specification of the project's design and architecture which is critical to security as we have discussed in earlier modules
The documentation of the project's implementation and associated business logic
And specifically from a security perspective the trust and threat models and specific areas of concern from the project team itself
It should also include all prior testing done tools used and reports from any other audits completed
The timeline effort and cost payments for the specific engagement must also be agreed upon
The engagement dynamics or channels for questions clarifications findings communication and reports should also be agreed upon to prevent surprises.
Finally, there should be single points of contact on both sides to make all this possible and seamless.

6.12 Limitations

Audits are generally considered as necessary for now at least for the reasons we have touched upon earlier, but audits are certainly not sufficient they can't guarantee `zero` vulnerabilities or exploits after the order they have limitations. This is because of three main reasons the first one is residual risk, there is risk reduction from an audit, but residual risk exists because of several factors such as the limited amount of audit time or effort limited insights into project specification implementation where in many cases there doesn't even exist a concrete written out specification, the documentation of the implementation itself doubles the specification residual risk could come from limited security expertise in the new and fast evolving technologies or the limited audit scope where an audit may not cover all the contracts or all the latest versions or their dependencies making the deployed contracts different from the ones audited residual risk could arise from significant project complexity and limitations of automated and manual analysis for all these reasons and maybe more audits can't and should not guarantee fully secure code that is free from any vulnerabilities or potential exploits such an expectation is unreasonable and any such positioning is misleading at best

Second not all audits are equal the quality of audits greatly depends on the expertise and experience of auditors effort invested given the project complexity quality and tools and processes used getting an order from a widely reputed security firm is not the same as getting it from someone else this affects residual risk to a great degree
Third, audits provide only a project security snapshot over a brief period of time this is typically a few weeks or sometimes even less however smart contracts need to evolve over time to add new features fix bumps or even optimize, this is sometimes done during or after an audit in code that is eventually deployed which reduces some of the benefits of the prior audit done because the changes introduced could have vulnerabilities themselves on the flip side relying on audits after every change is also impractical, so these tensions between security and shipping unfortunately exist even in web 3 similar to web 2, but arguably have a more significant impact in web 3 given the risk versus reward and other unique aspects of web 3 that we have discussed earlier.
So for these three broad reasons audits are considered necessary, but not sufficient by any means.

6.13 Reports

Audits typically end with a detailed audit report provided by the audit form to the project team. Projects sometimes publish such reports on their websites or GitHub repositories audit firms may also publish some of these with approval from the projects.
Such reports include details of the scope, goals, effort, timeline, approach use for, the audit tools and techniques used.
The finding summary the vulnerability details, if any found vulnerability classification as per the audit forms categorization because there isn't yet a standardized categorization vulnerability, severity, difficulty, likelihood as per OWASP or the firm's own rating and ranking, any potential exploit scenarios for the vulnerabilities which demonstrate how easy or hard it is for attackers and almost always the suggested fixes for the vulnerabilities.
They also include less critical informational notes recommendations suggestions on programming or software engineering best practices which may lead to security issues in certain scenarios.
Overall an audit report is a comprehensive structured document that captures a lot of these aspects in different levels of detail most audits provide a report at the end or there may even be interim reports shared as well depending on the duration and complex and while the format scope and level of details of these reports differ across audit firms they generally capture some or most of these categories of information and we'll go into the details of each of these concepts in the forthcoming slides.

6.14 Classification

The vulnerabilities found during the audit, if any are typically classified into different categories which make it helpful for the project team or even others to understand the nature of the vulnerability the potential impact severity impacted project components functionality and exploit scenarios and like we just discussed there isn't yet a standardized categorization and each audit form uses its own, so for example let's take a look at the classification used by Trail of Bits.

- There's access control which is related to authorization of users and assessment of rights auditing and logging related to
- Auditing of actions logging of problems
- Authentication related to the authentication of users in the context of the application
- Configuration of servers devices or software and in our case the smart contracts or off-chain components
- Cryptography related to protecting the privacy or integrity of data
- Data exposure related to unintended exposure of sensitive information
- Data validation related to improper reliance on the structure or values of data
- Denial of service or DoS related to causing system failure or inaccessibility
- Error reporting related to reporting of error conditions
- Patching related to keeping software up to date using patches in our case smart contracts that we have discussed earlier
- Session management related to identification of authenticated users.
- Finally, timing which is related to race conditions locking your order of operations

And, if none of these categories fit for the vulnerability, then it's typically categorized under undefined behavior that is figured by the program because of such a vulnerability we have broadly discussed these categories in the earlier modules of security and other audit forms may use a slightly different classification, but usually, there is a good overlap.

6.15 Difficulty

According to Owasp likelihood or difficulty which are semantically opposite terms by the way that's low likelihood is the equivalent of high difficulty.
This is a rough measure of how likely or difficult this particular vulnerability is to be uncovered and exploited by the attacker and OWASP proposes three likelihood levels of low medium and high some audit firms use OWASP, but others use their own terminology at ranking because does not apply very well to web 3 in general given the nature of risks vulnerabilities and even extent of impact from their exploits.
So Trailer of Bits for example classifies every finding into four difficulty levels
First one is low this means that the vulnerability may be easily exploited because public knowledge exists about this vulnerability type as it is related to a common security pitfall or a missing best practice at a `Solidity` or EVM level which further implies that it may be easily exploited
The second is medium which means that attackers typically need an in-depth knowledge of the complex system to exploit this vulnerability, this may be something application specific that is related to its business logic and not a commonly seen or known `Solidity` or EVM level vulnerability
The third one is high which means that an attacker must have privileged insider access to the system may need to know extremely complex technical details of that system or must discover some other weakness in order to exploit this issue this could imply that one of the trusted actors in the context of the application such as one of the privileged roles must be either malicious or compromised and potentially even with some insider details about some design or implementation to exploit this vulnerability
And finally, there is the undeterminate category which means that the difficulty of exploit was not determined during the engagement of the audit this could happen given the nature of the vulnerability the context of the application or even simply because the operational aspects of the audit engagement did not allow this to be determined irrespective of this subjective difficulty level determination the relative classification across the three or four categories is what is more important, this aspect should also be consistently applied to all the findings within the scope of the audit.

6.16 Impact

The other aspect of vulnerabilities that is important to recognize is impact and as per OWASP the impact of vulnerability estimates the magnitude of the technical and business impact on the system, if the vulnerability were to be exploited OWASP again proposes three levels of low medium and high, but this again needs to be revisited for web 3 because the impact from smart contract vulnerabilities and their exploits is generally very high and also the business or reputational aspects are very different in web 3 from a traditional web 2 sense

High impact is typically reserved for vulnerabilities causing loss of funds or locking of funds that may be triggered by any unauthorized user
Medium impact is reserved for vulnerabilities that affect the application in some significant way, but do not immediately lead to loss of funds anything else is considered a
Low impact these are again subjective in nature, but what matters more is that they make sense in a relative manner, so the high impact should be greater than a medium impact should be greater than a low impact in some reasonable justifiable way and also
This should be applied consistently across the audit these difficulty and impact ratings again are different across different audit forms with some of them being more stricter than others in classifying the vulnerabilities, this aspect of impact is perhaps the most noticed and discussed aspect as reported for vulnerabilities in the audit reports this is discussed and debated even between the audit firm, the project team given the subjective nature of this classification and something that gets paid a lot of attention even by the community at large when they are looking at high impact vulnerabilities reported in audits of the projects that they are interested in.

6.17 Severity

According to OWASP the likelihood and impact estimates are combined to calculate an overall severity for every risk, this is done by figuring out, if the likelihood and impact are low medium or high, then combining them into a three by three severity matrix.
So with the notation of likelihood hyphen impact is equal to severity the matrix looks like this a low likelihood and a low impact results in a severity that is an informational note a low likelihood a medium impact results in a low severity low high results in high medium low results in low and, so on as shown here,
so this is what is recommended by OWASP, but different firms end up using different severity levels payloads for example does not use this OWASP recommendation and uses five civility levels instead there's an informational severity where the issue does not pose an immediate risk, but is relevant to security best practices or helps with defensive depth,
there is a low severity where the risk is relatively small or is not a risk that the customer has indicated as being important medium risk where individual users information is addressed and exploitation would be bad for client reputation and, so on there's a high civility where it affects a large number of users it's very bad for the client's reputation and, so on.
Finally, there's an undetermined severity where the extent of this risk was not determined during the engagement on the other hand consensus diligence uses a different classification it uses minor to indicate that the issues are subjective in nature where, there are typically suggestions around best practices or readability medium severity are for issues that are objective in nature, but are not security vulnerabilities and major severity is for issues that are security vulnerabilities

that may not be directly exportable, but they require certain conditions in order to be exploited and finally, there are critical severities where the issues are directly exploitable security vulnerabilities that absolutely need to be fixed, so as we can see, there are clearly different severity considerations across firms, but again what matters more is the relative categorization consistency justification, the clarity.

6.18 Checklist

A checklist for projects to get ready for an audit is helpful, so that audit firms can assume some level of readiness from projects when audit starts. Trail of Bits for example recommends a checklist that has three broad categories test review and document
For test what is recommended is to enable an address every compiler warning and to also increase the unit and feature test coverage
For reviews what is recommended is for the project teams to perform an internal review to address common security pitfalls and best practices
For documentation what is recommended is one to describe what your product does who uses it why and how it delivers the functionality add comments about intended behavior in line with the code label and describe your tests and results both positive and negative tests and results four include past reviews and any bugs found five document steps to create a build environment six document external dependencies seven document build process including the debugging and test environment. Finally, eight document the deployment process and its environ.
Finally, having included the test review and document parts in a checklist what is also more critical is to communicate all the information in suitable ways to the audit form before an audit, so that they have all this information and do not waste their valuable time in discussing requesting duplicating or addressing these aspects.

6.19 Analysis Techniques

The analysis techniques used in audits involve a combination of different methods that are applied to the project core base along with any accompanying specification and documentation

1. Many are automated analysis performed with tools with some level of manual assistance and, there are generally eight broad categories
2. There's specification analysis that is completely manual
3. There's documentation analysis that's also manual
4. There's software testing which is automated

5. Static analysis again automated
6. Fuzzing
7. Combination
8. And automated technique symbolic checking that's also automated
9. And formal verification that is automated with some level of manual assistance
10. And finally, there is manual analysis that is entirely manual

Let's discuss each of these categories in some detail.

6.20 Specification

Specification as we have discussed earlier describes in detail the what and why aspects of the project and its components or in other words what is the project supposed to do functionally as part of its design and architecture as stemming from the requirements,
So from a security perspective it specifies what the assets are where they are held who are the actors in this context privileges of the actors who is allowed to access what and when trust relationships threat model potential attack vectors scenarios and mitigations.
Analyzing the specification of a project provides auditors with the above details and lets them evaluate any assumptions made and identify any shortcomings few smart contract projects have detailed specifications at their audit stage at west they have some documentation about what is implemented
And auditors end up spending a lot of time inferring specification from the documentation or implementation itself which leaves them with less time for deeper vulnerability assessment.

6.21 Documentation

Documentation is a description of what has been implemented based on the design and architectural requirements documentation should detail how something has been designed architected implemented without necessarily addressing the why aspects, the design requirement goals documentation in smart contract projects is typically in the form of README files in the GitHub repository describing individual contact functionality combined with the functional math Spec and individual code comments as discussed earlier documentation in many cases serves as a substitute for missing specification and provides critical insights into the assumptions requirements and goals of the project team understanding the documentation before looking at the code helps auditors save a lot of time in inferring the architecture of the project contract interactions program

constraints asset flow actors threat model and risk mitigation measures mismatches between the documentation, the court could indicate either stale or poor documentation software defects or security vulnerabilities therefore given this critical role of documentation the project team is highly encouraged to document thoroughly, so that auditors do not need to waste their time inferring all of our aspects by reading code instead.

6.22 Testing

Ectural requirements documentation should detail how something has been designed architected implemented without necessarily addressing the why aspects, the design requirement goals documentation in smart contract projects is typically in the form of README files in the GitHub repository describing individual contact functionality combined with the functional math Spec and individual code comments as discussed earlier documentation in many cases serves as a substitute for missing specification and provides critical insights into the assumptions requirements and goals of the project team understanding the documentation before looking at the code helps auditors save a lot of time in inferring the architecture of the project contract interactions program constraints asset flow actors threat model and risk mitigation measures mismatches between the documentation, the court could indicate either stale or poor documentation software defects or security vulnerabilities therefore given this critical role of documentation the project team is highly encouraged to document thoroughly, so that auditors do not need to waste their time inferring all of our aspects by reading code instead .

6.23 Static Analysis

Let's now talk about static analysis static analysis is a technique of analyzing program properties without actually executing the program this is in contrast to software testing where programs are actually executed or run with different inputs to examine their behavior and with smart contracts static analysis can be performed on the `Solidity` code directly or on the EVM byte code and is usually a combination of control flow and Data Flow analysis some of the widely used static analysis tools with smart contracts are Slither which is a static analysis tool from Trail of Bits and Maru which is a static analysis tool from ConsenSys Diligence both of which analyze intermediate representations derived from `Solidity` code of smart contracts.

6.24 Fuzzing

Fuzzing or first testing is an automated software testing technique that involves providing invalid unexpected or random data as inputs to software this is in contrast with software testing in general where chosen and valid data is used for

testing, so in first thing these invalid unexpected or random data are provided as inputs, then the program is monitored for exceptions such as crashes failing built-in code assertions or potential memory leaks Fuzzing is especially relevant to smart contracts because anyone can interact with them on the blockchain by providing random inputs without necessarily having a valid reason to do, so or any expectation from such an interaction this is in the context of arbitrary Byzantine behavior that we have discussed multiple times earlier the widely used Fuzzing tools for smart contracts are Echidna from Trail of Bits and Harvey from ConsenSys Diligence.

6.25 Symbolic Checking

Symbolic checking is a technique of checking for program correctness by using symbolic inputs to represent a set of states and transitions instead of using real inputs and enumerating all the individual states or transitions separately the related concept of model checking or property checking is a technique for checking whEther a finite state model of a system meets a given specification and in order to solve such a problem algorithmically both the model of the system and its specification are formulated in some precise mathematical language, the problem itself is formulated as a task in logic with the goal of solving that formula, there is decades of research and development in this domain and I would encourage anyone interested to explore the many references available here for smart contracts Manticore from taylor beds and mithril from consistent diligence are two widely used symbolic checkers which we will touch upon in later slides.

6.26 Formal Verification

Formal verification is the act of proving or disproving the correctness of algorithms underlying the system with respect to a certain formal specification of property using formal methods of mathematics formal verification is effective at detecting complex parts which are generally hard to detect manually or using simpler automated tools formal verification needs a specification of the program being verified and techniques to compare the specification with the actual implementation some of the tools in this space are sertora's proverb and change securities forex kEVM from runtime verification is a formal verification framework that models EVM semantics.

6.27 Manual Analysis

Manual analysis is complementary to automated analysis using tools it serves a critical need in smart contact audits today automated analysis using tools is cheap because it typically uses open source software that is free to use automated analysis is also fast deterministic and scalable, but however it's only as

good as the properties it is made aware of which is typically limited to those concerning `Solidity` and EVM related constraints manual analysis with humans on the other hand is expensive it's slow it's non-deterministic and it's not scalable because human expertise in smart contract security is a rare and expensive skill set today and we are slower prone to error and also inconsistent manual analysis however is the only way today to infer and evaluate business logic and application level constraints which is where a majority of the serious vulnerabilities are being found.

6.28 False Positives

Let's now talk about the concept of false positives and false negatives which are critical to understand in the context of smart contract audits or security in general compared to true positives which are findings that are indeed vulnerabilities false positives on the other hand are findings which flag the presence of vulnerabilities, but which in fact are not vulnerabilities and such false positives could be due to incorrect assumptions or simplifications in analysis which do not correctly consider all the factors required for the actual presence of vulnerabilities false positives require further manual analysis on findings to investigate, if they are indeed false positives or, if they are too positive and a high number of false positives increases the manual effort required in verification and also lowers the confidence in the accuracy of findings from the earlier automated analysis on the flip side true positives might sometimes be incorrectly classified as false positives which leads to such findings, the vulnerabilities behind those findings being ignored left behind in the code instead of being fixed and may end up getting exploited later.

6.29 False Negatives

On the other hand false negatives are missed findings that should have indicated the presence of other abilities, but which are in fact not reported at all such false negatives again could be due to incorrect assumptions or inaccuracies in analysis which did not correctly consider the minimum factors required for the actual presence of other abilities false negatives per definition are not reported or even realized unless a different analysis reveals their presence or the vulnerabilities are realized only when they're exploited a high number of false negatives lowers the confidence in the effectiveness of the earlier manual or automated analysis compared to this true negatives are missed findings or findings that are analyzed and dismissed which are in fact not vulnerabilities
So these concepts of true positives false positives true and false negatives come up often in smart contract auditing and in security in general and therefore this terminology, the distinction between these types should be well understood.

6.30 Audit Firms

Let's not talk about audit forms, there are several teams or firms that have security expertise with smart contracts and Ethereum and provide auditing services some have a web to origin from the traditional audit space where they provide other security services besides smart contact auditing while some others are specialized specifically in smart contract audits
There are a few others as well that are super specialized in certain formal verification privacy or cryptographic aspects within this space, there are at least 30 + audit firms that are widely cited in this space, this includes the boot camp partners ConsenSys Diligence Sigma Prime and Trail of Bits.

6.31 Security Tools

Having discussed audit techniques at a high level let's now talk a bit about the tooling that is used in this space smart contract security tools are critical in assisting both smart content developers as well as auditors with detecting potentially exploitable vulnerabilities highlighting dangerous programming styles or surfacing common patterns of misuse none of these however replace the need for manual review today to evaluate contract specific business logic and other complex control flow Data Flow and value flow aspects, so these tools at best complement manual analysis today.

6.32 Security Tools pt.2

We can think of tools in the space under different categories such as tools for testing test coverage linting static analysis symbolic checking Fuzzing formal verification visualization disassemblers.
Finally, monitoring and incident response tools let's now discuss some of the widely used tools in these categories we will only dive into a few of them in some detail and only touch upon the others.
Like I said earlier I would encourage you to explore these tools during this module, so install them most of them are open source and freely available play around with their options to understand how they work how effective they are and how they would fit within your toolbox when you start auditing smart contracts.

6.33 Slither Overview

So let's start with Slither which is a static analysis tool from Trail of Bits and one of the most widely used tools in this space Slither is a static analysis framework written in python 3 for analyzing smart contracts with an in `Solidity` it runs a suite of vulnerability detectors prints visual information about contact details.

Also provides an API to easily write custom analysis this helps developers and auditors find vulnerabilities enhances their code comprehension and also quickly prototype any custom analysis that they would like it implements 75 + detectors in the publicly available free version and we'll cover these different aspects of Slither in the forthcoming slides.

6.34 Slither Features

At a high level Slither implements vulnerability detectors and contact information printers it claims to have a low rate of false positives, the run time is typically less than one second per contract it is designed to integrate into ci cd frameworks.
It implements built-in printers that quickly report crucial smart contract information and also supports a detector API to write custom analysis in python it uses an intermediate representation known as slipped ir or Slither as it may be pronounced which enables simple and high precision analysis.

6.35 Slither Detectors

As mentioned Slither implements 75 + detectors each of which detects a particular type of vulnerability scissor can run on Truffle Embark dab Ether line or hard hat applications or on a single `Solidity` file and by default Slither runs all its detectors to run only selected detectors from within its suite.
There is a detect option to specify the names of detectors to run similarly to exclude certain detectors one can use the exclude option to specify the names of detectors to exclude two specific examples of detectors are reentrancy heat and unprotected-upgrade one can also exclude detectors based on the severity level associated with them
So for example to exclude all those detectors that are classified as informational or low severity one can use the exclude informational or exclude law options on this tool one can list all available detectors using the list detectors option, so I would encourage you to take a look at this tool, the various options and configurations that it supports.

6.36 Slither Printers

Besides the detectors like we mentioned Slither has a concept of printers that allow printing different types of contract information using the print options this helps in contract comprehension and gives us visibility into a lot of different aspects of the contract that's being analyzed the various print options include things like the control flow graph the call graph the contract summary data dependencies of variables summary of the functions inheritance relationships between contracts modifiers `require` and `assert` calls and storage order of the state variables

There are also many other details even from the Slither intermediate representation and at the EVM level all these could be very helpful in quickly understanding the contract structure getting a snapshot and zooming in on key aspects that are relevant from a security perspective.

6.37 Slither Upgradability

We've discussed in the security modules about how, there are many security challenges with Proxy-based upgradability and a lot of them were inspired by checks implemented by Slither along with documentation from OpenZeppelin on this topic Slither has a specific tool called the Slither check upgradeability, which reviews contracts that use the delegateCall Proxy pattern to detect potential security issues with upgradability.
These include initialized state variables missing or extra state variables and different state variable ordering between the Proxy and implementation contracts or different versions of the implementation contracts itself this also includes missing initialize function initialize function that is present, but that can be called multiple times because of the missing initializer modifier.
Finally, function id collision and shadow all these upgradeability aspects are conveniently packaged into a smaller tool which makes it very handy for checking that aspect.

6.38 Slither Code Similarity

Slither has a code similarity detector which can be used to detect similar `Solidity` functions this uses machine learning to detect similar and vulnerable `Solidity` functions it uses a pre-trained model using Etherscan verified contracts that is generated from more than 60 000 smart contracts and more than 850 000 functions this can be a useful tool to detect vulnerabilities from code clones forks or copies.

6.39 Slither Flat

Slither also has a contract flattening tool Slither flat which produces a flattened version of the code base and it supports three strategies most denied one file and local import most derived is for exporting all the most derived contracts one file helps us export all the contracts in one standalone file.
Finally, local import exports every contract in one separate file this tool handles circular dependency and also supports many compilation platforms such as Truffle hard hat Ether line and others.

6.40 Slither-Format

Slither also has a formatting tool, Slither format which automatically generates patches or fixes for a few of its detectors patches are compatible with git the detectors supported with this tool are a new state Saltzer version `pragma` naming convention `external` function constable states and `constant` function the patches generated by this tool should be carefully reviewed before applying just, so that you're comfortable with what those patches look like and, there are no bugs in it.

6.41 Slither ERC Conformance

Slither has an ERC conformance tool called Slither check ERC which takes conformance for various ERC standards such as `ERC20` `ERC721` `ERC777ERC165ERC223` . Finally, `ERC1820` some of which we have discussed in the security 201 module examples of these checks are to see, if functions are present return the correct type have `view` mutability and, if events are present emitted and parameters of such events are indexed as per the ERC Spec this is again handy for consolidating all ERC specific checks into one single two.

6.42 Slither-Prop

And finally Slither also has a property generation tool called the Slither prop which generates code properties or invariants that can, then be used for testing with unit tests or Echidna and completely automatically the `ERC20` scenarios that can be tested with this tool are things like checking for correct open transfer the possible functionality or that no one can incorrectly mint or burn tokens.

6.43 Slither New Detectors

Besides the various detectors printers and tools of Slither that we just discussed Slither also supports an extensible architecture that allows one to integrate new detectors into the tool the skeleton for such a detector implementation has things like arguments help impact confidence and link to the wiki for that detector. Finally, a placeholder for the most important part of the detector logic itself this extensible architecture can help with creating application specific detectors and also enables the community to contribute new detectors to the Slither codebase, so those are all the Slither features that we're going to cover here and as we see it is an extensive tool with support for 75 + detectors and multiple other helpful features as well for reasons of which it's widely referenced and used across projects in the space.

6.44 Manticore

Let's now move on to another tool from Trail of Bits called Manticore which is a symbolic execution tool this again helps with analysis of Ethereum smart contracts and complements Manticore can `execute` a program with symbolic inputs and explore all possible states it can reach it can automatically produce concrete inputs that result in any desirable program state it can detect crashes and other failures in smart contracts provide instrumentation capabilities. Finally, a programmatic interface to its analysis engine via python API similar to slytherin.

6.45 Echidna

Another tool from Trail of Bits is Echidna which is a Fuzzing tool and complements Slither and Manticore again for use with smart contract audience this is written in haskell and it performs grammar based Fuzzing campaigns based on a contracts ABI to falsify user defined predicates or even `Solidity` assertions in the smart contract code.

6.46 Echidna Features

Ekita has many notable features such as it generates inputs tailored to the actual code has an optional corpus collection of predefined campaigns it supports mutations and coverage guidance for deeper bugs it can be powered by the Slither prop tool to extract useful information before the Fuzzing campaign it has source code integration to help identify which lines are covered after the Fuzzing campaign it has support for multiple user interfaces automatic test case minimization for quick triage. Finally, seamless integration into the development workflow among other things.

6.47 Echidna Usage

As for Echidna's usage I would recommend looking up ekinda's documentation and available tutorials on trailer bit website for such details, but at a high level the usage involves three aspects first one is executing the test runner where the core Echidna functionality is part of an executable called a kidnap test that takes a contract and a list of invariants as inputs for each invariant it generates random call sequences to the contract and checks, if the invariant holds, if it can find some way to falsify the invariant it prints the call sequence that does so, these are typically referred to as outer examples in this terminology, if it can't find out examples, then we have some assurance that the contract is safe with respect to that invariant the second aspect is that of writing invariants invariants are expressed as `Solidity` functions with names that begin with Echidna underscore they have no arguments and they return a `boolean`, the

third aspect is that of collecting and visualizing coverage after finishing the Fuzzing campaign Echidna can save that coverage maximizing corpus in a special directory which will contain two entries a directory with JSON files that can be replayed by Echidna later and a plain text file that contains a copy of the source code with coverage annotations.

6.48 Eth Security Toolbox

Trailerbits has combined the three tools we just discussed into tools package which is a Docker container package called eat security toolbox where they are pre-installed and pre-configured and as you can imagine this makes it very handy and very easy to start off with using these tools, so the heat security toolbox has Slither Echidna and Manticore and besides these three also has Rattle and no tools which we will touch upon in coming slides.

6.49 Ethersplay

Ethersplay is a Binary Ninja plugin from trailer pets that enables an EVM disassembler and related analysis tools for those who aren't aware binary ninja is a widely used extensible reverse engineering platform which can disassemble a binary and display it in various ways, so Ethers play effectively extends that to work with EVM bytecode this takes EVM byte code in raw library format as input and generates a control flow graph of all functions it can also be used to display manticore's coverage.

6.50 PyEVMasm

Pi EVM asim is another security tool from Trail of Bits which provides an assembler and disassembler library for the Ethereum virtual machine EVM this includes a command line utility for doing the assembling and disassembling and also includes a python API for extensibility.

6.51 Rattle

Rattle is another security tool from trailer pits it is an EVM binary static analysis framework that is designed to work with deployed smart contracts it takes EVM byte strings as inputs and uses a flow sensitive analysis to recover the control flow graph in static analysis terminology flow sensitive refers to an analysis that considers the control flow of statements similarly, there is context sensitive and path sensitive analysis as well Rattle further converts the control flow graph into a single static assignment form or SSA form with infinite registers and optimizes this SSA by removing stacked instructions of dupes swaps pushes and pops remember that EVM is a stack based machine and. There are typically

many such stacked instructions in the bytecode as operands are pushed onto the stack and results are popped Rattle by converting the byte code instructions from a stack machine to SSA form removes more than 60 percent of all EVM instructions and because of that it presents a user-friendly interface for analyzing smart contract bytecode for anyone interested in programming language analysis I would encourage them to look up these concepts of SSA and sensitivity and, so on.

6.52 EVM CFG Builder

EVM CFG builder is another tool from Trail of Bits this is used to extract the control flow graph from EVM bytecode this tool helps us reliably recover control flow graph or CFG from the EVM byte code and also recovers function names and their attributes such as payable `view` `pure` etc it outputs the CFG to a DOT file, this EVM CFG builder tool is used by Ethers play Manticore and some other tools from Trail of Bits.

6.53 Crytic Compile

Critic compile is another tool from Trail of Bits it is a smart contract compilation library that is used in the security tools from trailer pets it supports Truffle mbar Ether scan Brownie Waffle Hardhat and other development environments, this plugin is used in the critic family of tools from Trail of Bits that includes Slither Echidna and.

6.54 Solc-Select

Select is a security helper tool gained from Trail of Bits, this is a script that is used to quickly switch between different `Solidity` compiler versions sourcing select manages installing and setting different salsi compiler versions using a wrapper around salsi which picks the right version according to what was said via solve `c` select `Solidity` compiler the solc binaries are downloaded from the official series language repository, this tool is very helpful while analyzing different smart contact projects because, there is often a need to switch between different `Solidity` compiler versions depending on which version is being used by the project that is being analyzed, so this tool is very handy in such situations and helps us work with other security tools that depend on the `Solidity` compiler version.

6.55 Etheno

Ethanol is a testing tool referred to as the Ethereum testing Swiss Army knife again from Trail of Bits it's a JSON RPC multiplexer analysis tool wrapper and

test integration tool all bundled into one for multiplexing it runs a JSON RPC server that can multiplex calls to one or more Ethereum clients with an API for filtering modifying such JSON RPC calls it enables differential testing by sending JSON RPC sequences to multiple Ethereum clients and further helps with the deployment and interaction with multiple networks at the same time for the analysis tool wrapping part it provides a JSON RPC client for advanced analysis tools such as maticore which makes it much easier to work with such tools because, there is now no need for custom scripts for those students and for integration with best frameworks such as Ganache and Truffle it helps run a local test network with a single command and enables the use of Truffle migrations to bootstrap anti-core analysis, so for all these reasons it is referred to as the Swiss Army knife for Ethereum testing.

6.56 MythX

Now moving on to tools from ConsenSys Diligence MythX may be considered as their flagship tool MythX is a powerful security analysis service that finds vulnerabilities in Ethereum smart contact code during the development lifecycle it is a paid API based service that uses several tools in the backend, these include Maru a static analyzer Mythril a symbolic analyzer, the third one being Harvey which is a gray box fuzzer and in combination among these three tools mithix implements a total of 46 + detectors while Maru and Harvey are closed source as of now vitriol is open source and we'll talk more about different aspects of MythX in the forthcoming slides.

6.57 MythX Process

So how does the MythX process work remember that widex is an API based service, so MythX does not run locally on the user's machines, but it runs in the cloud, so the first step is for the project to submit the code to the mythex service the analysis requests are encrypted with TLS, the code one submits can only be accessed by them and one is expected to submit both the source code, the compiled byte code of the smart contract for best results the second step is to activate the full suite of analysis techniques behind with `x` and here the longer MythX surrounds the more security weaknesses it can detect, this is because the precision of the symbolic checker, the Fuzzing components of with `x` can get better with more iterations the third and final step is to receive a detailed analysis report from the defects service, this report lists all the weaknesses found in the submitted code including the exact location of those issues, these reports that are generated can only be accessed by the submitter MythX here offers three scan modes quick standard and deep for differing levels of analysis depth and provides a user-friendly dashboard for analyzing the results returned.

6.58 MythX Tools

Now let's talk about the tools used by the MythX service when a project submits their code to the mythex API it gets analyzed by multiple microservices in parallel where three tools cooperate to return a more comprehensive set of results in the execution time decided by the type of scan chosen the first of the three tools is a static analyzer called Maru that passes the solarity asd for the project the second tool is a symbolic analyzer called Mythril that detects all the possible vulnerable states in the contract. Finally, the third tool is Harvey which is a grey box fuzzer that detects vulnerable execution paths in the smart contract compared to traditional black box Fuzzing gray box Fuzzing is guided by coverage information which is made possible by using program instrumentation to trace the code coverage reached by each input during Fuzzing, so these three tools are used in combination by the MythX service to provide a comprehensive analysis of the vulnerabilities within the smart contract being analyzed.

6.59 MythX Coverage

The coverage that is provided by MythX extends to most of the smart contract weaknesses found in the smart contract weakness registry referred to as the SWC registry which we will talk more about in one of the forthcoming slides, this comprehensive coverage addresses 46 + detectors as of today.

6.60 MythX SaaS

With `x` is based on a security as a service or sas platform with the premise that this sas approach is better because of three main reasons the first reason is that with this approach one can expect higher performance compared to running the security tools locally because the compute power in the cloud is typically much much higher than what may typically be expected at the user's end on a laptop or a desktop, the second aspect is that we can expect a higher vulnerability coverage with three tools than running any single standalone the third benefit is from continuous improvements to security analysis technology with new or improved security tests methodologies and tools that can be adopted as the smart contract security landscape evolves with different types of vulnerabilities and exploit vectors emerging as the compiler revisions change new coding patterns emerge new dependencies start getting used new protocols start getting used and even the Ethereum protocol upgrades over time, so for these three reasons the sas or API based approach of mythex is considered as being better than running any one of those tools locally on the user's end.

6.61 MythX Privacy

It's understandable that project teams may have concerns uploading their smart contract code to a sas service like with `x`, so methox provides a privacy guarantee the smart contract code submitted using their sas APIs first one is that the code analysis requests are encrypted with TLS and to provide comprehensive reports and improve performance the MythX service stores some of the contract data in its database including parts of the source code and byte code, but that data never leaves their secure server and is not shared with any outside parties it keeps the results of the analysis it retains them, so that it can be retrieved later, but the reports can be accessed only by the project team, the service enforces authorized access to such results.

6.62 MythX Performance

Performance is usually a concern with security tools that perform deep analysis such as with symbolic checking or Fuzzing because they may `require` a lot of compute resources and proportionately longer amounts of time for retrading through their analysis to get good coverage and position, so in this case with `x` can be configured for three types of scans depending on the time expectation quick scans run for five minutes standard scans run for 30 minutes while deep scans run for 90 minutes and as you can imagine standard scan gives better results than quick scans and deep scans better than standard ones, so one can customize this the type of scans according to the development phase and time available, so for example quick scans can be perhaps run by developers during their code comments and standard scans can be run at certain project milestones while deep scans that take a much longer time can be run on the nightly builds when you have more time.

6.63 MythX Versions

With `x` comes in different versions, so that it can be accessed via multiple ways, there is a command line interface version that provides a unified tool access to methox, there is with xjs which is a library to integrate detects in javascript or typescript projects, there is a python library pythex to integrate methods in python projects and finally, there is a visual studio code extension for mythex that allows a project to scan smart contracts and `view` the results directly from the code editor.

6.64 MythX Pricing

As for pricing MythX has four pricing plans the first one is an on-demand pricing plan that costs 9.99 for three scans and all three scan modes quick standard and deep are available as part of this plan the second pricing plan is a development

plan that costs 49 a month this gives access to quick and standard scan modes only and it allows 500 scans a month the third one is a professional plan which costs 249 a month and gives access to all scan mods and 10 000 scans a month and finally, there is an enterprise pricing plan that allows for custom pricing where custom plans can be decided between a project team and ConsenSys Diligence that meets the team's specific needs.

6.65 Scribble

Let's now move on to another tool from ConsenSys Diligence called Scribble scribble is a verification language and a runtime verification tool that translates high level specifications into `Solidity` code it allows one to annotate a `Solidity` smart contract with specific properties, there are four principles or goals with Scribble one that specifications should be easy to understand by developers and smart contract security auditors two specifications should be simple to reason about three specifications should be efficiently checked using off-the-shelf analysis tools and four a small number of core specification constructs should be sufficient to express and reason about more advanced constructs, so Scribble transforms annotations made within smart contract code using its specification language into concrete assertions, then with those instrumented contracts that are equivalent to the original ones one can use other tools from consistent diligent such as Mythril Harvey litex to leverage these assertions for performing deeper checks, so Scribble is a relatively newer tool from ConsenSys Diligence and sounds very powerful in its capabilities, so I would strongly encourage everyone to take a look at the documentation of Scribble take a look at the motivations, the underlying concepts driving this tool and also test it out to explore all its capabilities.

6.66 Fuzzing-as-a-Service

Fuzzing as a service is a service that has been recently launched by ConsenSys Diligence where projects can submit their smart contracts along with embedded inline specifications or properties written using the Scribble language that we just talked about these contracts are run through the Harvey fuzzer which uses the specified properties to optimize Fuzzing campaigns and any violations from such Fuzzing are reported back from the servers for the project to fix.

6.67 Karl

Call is another security tool from ConsenSys Diligence, this can be used to monitor the Ethereum blockchain for newly deployed smart contracts that may be vulnerable, this can be done in real time carl checks for security vulnerabilities using the victory detection engine also from consistent diligence, this can be an interesting monitoring tool for detecting vulnerable deployed smart contracts,

but not during security auditing or reviews for projects that have yet to be launched.

6.68 Theo

Another security tool from ConsenSys Diligence that is not specifically meant for auditing, but interesting nevertheless is Theo Theo is an exploitation tool with a Metasploit like interface and provides a python REPL console from where one can access a long list of interesting features such as automatic smart contact scanning which generates a list of possible exploits sending transactions to exploit a smart contract transaction pool monitoring Front-running backlining transactions and many others.

6.69 Visual Auditor

A tool that could be very handy in the manual analysis phase of smart contact auditing is the visual auditor this is a visual studio extension again from ConsenSys Diligence that provides security aware syntax and semantic highlighting for Soleil and Vyper languages examples of things that are highlighted include modifiers visibility specifiers security relevant built-ins such as a global transaction origin message data and, so on storage access modifiers indicating, if it is in memory or storage developer notes in comments such as to do's fix me hack etc invocations operations constructor fallback functions state variables it has support for review specific features such as audit annotations and bookmarks exploring dependencies inheritance function signature hashes it supports graph and reporting features such as interactive call graphs with call flow highlightingl diagrams and access to Surya features which we'll talk about in the next slide it also supports code augmentation features where additional information is displayed when hovering over Ethereum account addresses that allow one to download the byte code or source code or open it in the browser hovering over Assembly instructions to show the signatures and hovering over the state variables to show their declaration information, so overall the visual auditor is almost a must have tool while manually reviewing `Solidity` or Vyper code during audits.

6.70 Surya

Surya is a visualization tool from ConsenSys Diligence that helps auditors in understanding and visualizing `Solidity` smart contracts by providing information about their structure and generating call graphs and inheritance graphs that can be very useful it also supports querying the function call graph in many ways to help during the manual inspection of contracts, this is integrated with the visual auditor tool that we discussed in the previous slide studio supports

several commands such as graph function trace flatten inheritance dependencies parts generating a report in the markdown format etc.

6.71 SWC Registry

It is always helpful to have a registry of unique vulnerabilities, so that everyone can refer to a single source keep it updated and use them in interesting ways one such effort is the smart contract weakness classification registry or SWC registry this is an implementation of the weakness classification scheme proposed in eip1470 it is loosely aligned to the terminologies and structure used in the common weakness enumeration cwe from `web2` while being specific to smart contracts the goals of this project are three-fold first one is to provide a way to classify security issues in smart contract systems, the second one is to define a common language for describing security issues in smart contract systems architecture design and code. Finally, serve as a way to train and improve smart contact security analysis tools this repository is currently maintained by ConsenSys Diligence and contains 36 entries as of now.

6.72 Securify

Securify is a security scanner developed by chain security it's a static analysis tool for Ethereum smart contracts it's written in data log and supports 38 + vulnerabilities we won't go into the details of this tool over here, but I would encourage you to look it up.

6.73 VerX

Verax is a formal verification tool again from the chain security team that can automatically prove temporal safety properties of Ethereum smart contracts the verifier is based on a combination of three ideas one reduction of temporal safety verification to reachability checking two a symbolic execution engine used to compute precise symbolic states within a transaction and three the concept of delayed abstraction which approximates symbolic states at the end of transactions into abstract states the details of this tool or the concepts behind it are out of scope over here, but I would encourage you to look at their website for documentation and their academic paper for greater details behind the theory of this tool.

6.74 Smart Check

Smart check is a security tool from smart tech it is another static analysis tool for discovering vulnerabilities and other code issues in Ethereum smart contracts written in `Solidity` an interesting implementation aspect here is that it

translates `Solidity` source code into an xml based intermediate representation, then checks it against `x` part patterns for context expert stands for xml path language which uses a path notation for navigating through the hierarchical structure of an xml document.

6.75 K Framework

K-Framework is a verification framework from the runtime verification team it includes kEVM which is a model of EVM in the gray framework it is the first executable specification of the EVM that completely passes the official EVM test suites and, so could serve as a platform for building a wide range of verbal analysis tools for EVM again we won't go into any level of details for this framework here because we can't do much justice to such deep frameworks in one slide or a few minutes, but I would strongly encourage you to look at the documentation to get a better understanding of its capabilities.

6.76 Certora Prover

Certora Prover is a formal verification tool from Certora it checks that a smart contract satisfies a set of rules written in a language called specify each rule is checked on all possible transactions not by explicitly enumerating them of course, but rather through symbolic techniques the prover provides complete path coverage for a set of safety rules provided by the user for example a rule might want to check that a bounded number of tokens can be minted in an `ERC20` contract the prover either guarantees that such a rule holds on all paths and all inputs or produces a test input known as a counter example that demonstrates a violation of this rule, this problem addressed by certain approval is going to be undecidable which means that there will always be some pathological programs or rules for which the prover will time out without a definitive answer this proverb takes as input the smart contract either the byte code or the `Solidity` source code along with a set of rules written in the Certora's specification language specified the prover, then automatically determines whether or not the contract satisfies all the rules provided using a combination of two fundamental computer science techniques known as abstract interpretation and constraint solving I would encourage everyone to take a look at their website and documentation to understand how the prover tool works, the technology behind it.

6.77 HEVM

Tab hubs hEVM is an implementation of the EVM made specifically for unit testing and debugging smart contracts it can help run unit tests property tests and also help interactively debug contracts while showing the `Solidity` source code or also run arbitrary EVM code, so with this we have touched upon the

various security tools that you may come across in this space, there are likely others that we haven't covered here purely for constraints of time and scope and some like smt checker which we have covered in the soluti module earlier for all these tools the best way to understand their capabilities and specific use cases is to install and experiment with them and I would encourage everyone to do that with some of these tools at least.

6.78 CTFs

Let's now talk about a related concept called capture the flag or CTF as it is popularly known as CTFs are fun and educational challenges where participants have to hack different dummy smart contracts that have vulnerabilities in them they help understand the complexities around how such vulnerabilities may be exploited in the white, the popular CTFs in the space of Ethereum smart contracts include capture the Ether which is a set of 20 challenges created by steve marks which tests knowledge of Ethereum concepts of contracts accounts and math among other things then, there is Ethernot which is a `web3` or `Solidity` based war game from OpenZeppelin that is played in the Ethereum virtual machine and each level is a smart contract that needs to be hacked the game is completely open source and all levels are contributions made by players themselves, then we have dam vulnerable DEFI which is a set of eight DEFI related challenges created by tinker security researcher from open separate depending on the challenge one should either stop the system from working steal as much funds as they can or do some other unexpected things. Finally, we have the paradigm CTF which was a set of 17 challenges created by samson at paradigm, so CTFs can be a fun way to practically test out some of the things that you've learned in these modules, so I would encourage you to take a look at some of these and see how well you do with them.

6.79 Security Tools

In summary smart content security tools are useful in assisting auditors while reviewing smart contracts they automate many of the tasks that can be codified into rules with different levels of coverage correctness and precision these tools are fast cheap scalable and deterministic compared to manual analysis however they are also susceptible to false positives they are therefore especially well suited correctly to detect common security pitfalls and best practices at disability and EVM levels and with varying degrees of manual assistance they can also be programmed to check for application level business logic constraints.

6.80 Audit Process

Let's now talk about the audit process this is critical to understanding the different stages in the life cycle of an order from an auditor's perspective it helps

us understand what the auditors do at those different stages how do they focus their efforts at those different stages how do they interact with each other how do they interact with the project team and what the deliverables are at different stages of this audit life cycle, this process is going to be very different for every audit form and very different even perhaps for different audits generalizing, then an audit process can be thought of as a 10-step process as follows the first step is typically to read the specification and documentation of the project to understand the requirements design and architecture behind all the different aspects of the project, then run fast automated tools such as linters or static analyzers to investigate some of the common security pitfalls or missing smart contact best practices that we have discussed the third step would be to manually analyze the code to understand the business logic aspects and detect vulnerabilities in it this could be followed by running slower, but more deeper automated tools such as the symbolic checkers fuzzers or formal verification tools some of which we have discussed in this module, these typically `require` formulation of the properties or constraints beforehand hand holding during the analysis and even some post processing of the results these stages may involve auditors discussing with other auditors the findings from all these about tools or even the manual analysis to identify any false positives or remissing analysis the auditors may also convey the status to the project team for clarifying any questions on the business logic or the threat model or other aspects and all these aspects may be iterated as many times as possible within the duration of the order, so as to leave some time at the end for writing the report and writing the report itself involves summarizing all these about details on the findings and recommendations. Finally, the audit team delivers that report to the project team, then they discuss the findings the civilities, the potential fixes that are possible and there's also a step here where the audit team evaluates fixes from the project team for any of the findings reported, then they verify that those fixes indeed remove the vulnerabilities identified in those findings, so this is how a typical audit process may look like, the different stages in its life cycle let's now dive in to discuss some details about each of these 10 steps.

6.81 Read Spec/Docs

The first step in the audit process is typically reading the specification and documentation for projects that have a specification of the design and architecture of their smart contracts this is indeed the recommended starting point however very few new projects have a specification at least at the audit stage some of them have documentation in parts and to remember some of the key aspects of sophistication and documentation, the differences between the two specification starts with the project's technical and business goals and requirements it describes how the project's design and architecture help achieve those goals, the actual implementation of the smart contracts is a functional manifestation of these goals requirements specification design and architecture and understanding all these is critical in evaluating, if the implementation indeed

meets the goals and requirements documentation on the other hand is a description of what has been implemented based on the design and architectural requirements, so while specification answers the why aspect of something needs to be designed architected implemented the way it's been done documentation on the other hand answers the how aspect, if something has been designed architected implemented without necessarily addressing the y aspect and leaves it up to the auditors to speculate on the reasons and documentation remember is typically in the form of README files describing individual contact functionality combined with some functional math Spec and individual comments within the code itself encouraging projects to provide a detailed specification and documentation saves a lot of time and effort for the auditors in understanding the project's goal structure and prevents them from making the same assumptions as the implementation which is perhaps a leading cause of vulnerabilities, so the absence of both specification and documentation auditors are forced to infer those aspects such as the goals requirements design and architecture from reading the code itself and using tools such as Surya or the Slither printers that we discussed earlier identifying the key assets actors and actions in the application logic from the code base that is required for understanding the trust and threat models is a complex and involved task all this takes up a lot of time without the presence of a detailed and accurate specification leaving very less time for the auditors to perform deeper and more complex security analysis.

6.82 Fast Tools

Auditors typically also use some fast tools such as linters or static analyzers that perform their analysis and finish running within seconds automated tools such as these as we discussed help investigate common security pitfalls at the `Solidity` or EVM levels and detect missing smart contract best practices such tools implement control flow and Data Flow analysis on smart contracts in the context of their detectors which encode such common pitfalls and best practices evaluating their findings which are usually available within seconds or few minutes is a good starting point to detect common vulnerabilities based on well-known constraints or properties of `Solidity` language EVM or the Ethereum blockchain itself false positives are possible among some of the detector findings which need to be verified manually to check. If there are true or false posters, these tools can also miss certain findings leading to false negatives best examples of static analyzers in this space are Slither and Maru both of which we have touched upon in the earlier slides of this module.

6.83 Manual Analysis

Manual analysis is perhaps the most critical aspect of smart contract audits today manual code review as we've discussed is required to understand business logic and detect vulnerabilities in it automated analyzers can't understand ap-

plication level logic and infer their constraints and, so are limited to constraints and properties of the `Solidity` language EVM or the Ethereum blockchain itself manual analysis of the code is therefore required to detect security relevant deviations in the implementation from those captured in the specification or documentation and as we have discussed in the absence of specification or documentation auditors will be forced to infer business logic and their implied constraints directly from the code itself or from discussions with the project team and only thereafter evaluate, if those constraints or properties hold in all parts of the code base we'll take a deeper look at different approaches to manual analysis in the last part of this module.

6.84 Slow/Deep Tools

In contrast to the fast tools that we discussed earlier, there are also what may be thought of as slow or deeper tools these are tools in categories of Fuzzing symbolic checking or formal verification running such deeper automated tools fuzzers such as Echidna symbolic checkers such as magic Mythril tool suite such as MythX or formally verifying custom properties with Scribble or certain approval takes more understanding and preparation time to formulate such custom properties, but helps run deeper analysis which may take minutes to run, but helps discover edge cases in application level properties and mathematical errors among other things given that doing, so requires understanding of the project's application logic such tools are recommended to be used at least after an initial manual code review or sometimes after deeper discussions about the specification implementation with the project team itself also analyzing the output of these tools requires significant expertise with the tools themselves their domain specific language and sometimes even their inner workings to interpret their findings evaluating false positives is sometimes challenging with these tools, but the true positives they discover are typically significant and extreme corner cases even by the best manual analysis.

6.85 Discuss w/ Auditors

Brainstorming with other auditors is often helpful given enough eyeballs all bugs are shallow is a premise that is referred to as linus's law, this might apply with auditors too, if they brainstorm on the smart contract implementation assumptions findings and vulnerabilities while some audit firms encourage active or passive discussion there may be others whose approaches to let auditors separately perform the assessment to encourage independent thinking instead of group thinking the premise is that group thinking might bias the auditing to focus only on certain aspects and not others which may lead to missing detection of some vulnerabilities and therefore affects the effectiveness a hybrid approach may be interesting where the auditing initially brainstorms to discuss the project goals specification documentation and implementation, but later

firewall themselves to independently pursue the assessments. Finally, come togEther to compile their findings finding a balance between the overhead of such an approach, the benefits of such an overlapping effort may be an interesting consideration.

6.86 Discuss w/ Project Team

Discussion with the project team is another critical part of the audit process having an open communication channel with the project team is useful to understand their scope trust threat models any specific concerns to clarify any assumptions in specification documentation implementation or to discuss interim findings findings may also be shared with the project team immediately on a private repository to discuss impact fixes and other implications without waiting to discuss it at the end of the audit period, if the audit spans multiple weeks it may also help to have a weekly sync up call for such discussions and updating the status a counter point to this is to independently perform the entire assessment, so as to not get biased by the project teams inputs and opinions which may steer the auditors in certain directions potentially without letting them pay attention to other aspects.

6.87 Write Report

An audit report is a tangible deliverable at the end of an audit and therefore report writing becomes a very critical aspect of the entire audit process the audit report is a final compilation of the entire assessment and presents all aspects of the audit including the audit scope coverage timeline team effort summaries tools techniques findings exploit scenarios suggested fixes short-term long-term recommendations and any appendices with further details of tools and rationale an executive summary typically gives an overview of the audit report with highlights low lights illustrating the number type severity of vulnerabilities found and an overall assessment of risk it may also include a description of the smart contracts actors assets roles permissions access control interactions threat model and existing risk mitigation measures the bulk of the report focuses on the findings of the audit their type category likelihood impact severity justifications for these ratings potential exploit scenarios affected parts of smart contracts and potential remediations it may also address subjective aspects of code quality readability auditability and other software engineering best practices related to the documentation code structure function variable naming conventions test coverage etc that do not immediately pose a security risk, but are indicators of anti-patterns and processes influencing the interruption and persistence of security vulnerabilities the audit report should be articulate in terms of all these information and also actionable for the project team to address all raised concerns.

6.88 Deliver Report

The delivery of the audit report is another important aspect in the audit process and perhaps the final milestone when such a report is published and presented to the project unless interim findings or status is shared this will be the first time the project team will have access to the assessment details the delivery typically happens via a shared online document and is accompanied with the readout where the auditors present the report highlights to the project team for any discussion on the findings and their civility ratings the project team typically takes some time to review the audit report and respond back with any counterpoints on finding severities or suggested fixes depending on the prior agreement the project team, the audit form might release the audit report publicly after all required fixes have been made or the project may decide to keep it private for some reason.

6.89 Evaluate Fixes

Evaluating fixes is typically the final stage in the audit process and a very critical stage after the findings are reported to the project team they typically work on any required fixes, then request the audit firm for reviewing such fixes fixes may be applied for a majority of the findings, the review may need to confirm that applied fixes which in some cases could be different from what was recommended that these fixes indeed mitigate the risk reported by the findings some findings may also be contested as not being relevant outside the project's threat model or simply acknowledged as being within the project's acceptable risk model audit firms may evaluate the specific fixes applied and confirm or deny their risk mitigation and unless it is a fix or retainer type of audit this phase typically takes not more than a day or two because it would usually be outside the agreed upon duration of the audit, but most audit firms generally accommodate this to help ensure the security of the project, so these are the 10 steps of an audit process that you can expect to see within an audit or an audit form like mentioned earlier these are generalized opinions, the specifics of these different steps their order the level of effort that is put in each step, the philosophies behind them will surely differ across different audit forms, but nevertheless this is something that's very critical that needs to be paid attention to and understood to appreciate the different steps of the audit process.

6.90 Manual Review

In the final part of this module let's dive deeper into the many approaches to manual review as we mentioned a couple of times the manual review step is perhaps the most critical component of smart contract audits today, so auditors have different approaches to manually reviewing smart contracts for vulnerabilities they may be along the lines of starting with access control starting with

asset flow or control flow Data Flow inferring constraints understanding dependencies evaluating assumptions and evaluating security checklists auditors may start with one of these as their preferred approaches, then combine multiple of them for best results these are very subjective aspects, but we will explore them in some detail to understand what they make there.

6.91 Access Control

Starting with access control is very helpful because access control as we've discussed is the most fundamental security primitive it addresses who has authorized access to what or which actors have access to what assets well the overall philosophy might be that smart contracts are permissionless in reality they do indeed have different permissions or roles for different actors who or use them at least during their initial guarded launch the general classification is that of users and admins and sometimes even a role based access control privileged roles typically have control over critical configuration and application parameters including emergency transfers withdrawals of contact funds and such access control is typically enforced in modifiers as we have discussed in the earlier modules and also more generally we are the visibility of functions such as public external versus internal or private which were also discussed in the context of `Solidity` therefore starting with understanding the access control implemented by smart contracts and checking, if they have been applied correctly completely and consistently is a good approach to detecting violations which could be critical vulnerabilities.

6.92 Asset Flow

One can also start with the asset flow assets are Ether or `ERC20` `ERC721` or other tokens managed by smart contracts given that exploits target assets of value it makes sense to start evaluating the flow of assets into outside within and across smart contracts and their dependencies the questions of who when which why where what type and how much are the ones to be asked for who assets should be withdrawn deposited only by authorized specified addresses as per application logic for when assets should be withdrawn deposited only in authorized specified time windows or under authorized specified conditions as per application logic for which assets only those authorized specified types should be withdrawn deposited for why assets should be withdrawn deposited only for authorized specified reasons as per application logic for where assets should be withdrawn deposited only to authorized specified addresses as per the logic for what type assets only of authorized specified types should be withdrawn deposited as per the logic. Finally, for how much assets only in authorized specified amounts should be allowed to be withdrawn deposited again as per the application logic, so these are all the various aspects of asset flow that need to be evaluated.

6.93 Control Flow

Evaluating control flow is a fundamental program analysis approach control flow analyzes the transfer of control that is the execution order across and within smart contracts inter procedural control flow where the procedure is just under the name for a function is typically indicated by a polygraph which shows which functions or callers call which other functions or colleagues across or within smart contracts intra procedural control flow that's within a function is dictated by conditionals the, if else constructs loops for while do continue break constructs and return statements both intra and inter-procedural control flow analysis help track the flow of execution and data in smart contracts and therefore as you can imagine is a fundamental program analysis approach to evaluate security aspects.

6.94 Data Flow

Evaluating Data Flow is another fundamental aspect of program analysis which analyzes the transfer of data across and within smart contracts inter-procedural Data Flow is evaluated by analyzing the data that's the variables and constants used as argument values for function parameters at call sites and their corresponding `return` values intra procedural Data Flow on the other hand is evaluated by analyzing the assignment and use of variables or constants stored in storage memory stack all data locations along the control flow paths within functions both intra and inter procedural Data Flow analysis help track the flow of global or local storage memory changes in smart contracts and given that Data Flows where control flows they work togEther to help with program analysis of smart contracts in helping detect security vulnerabilities.

6.95 Inferring Constraints

Inferring constraints is an approach that is almost always required program constraints are basically rules that should be followed by the program `Solidity` level and EVM level security constraints are well known because they're part of the language and EVM specification however application level constraints are rules that are implicit to the business logic implemented and may not be explicitly described in the specification example of such a constraint may be to mint an `ERC721` token to an address when it makes a certain deposit of `ERC20` tokens to the smart contract and burn it when it withdraws the earlier deposit such business logic specific application level constraints may have to be inferred by auditors while manually analyzing the smart contract code another approach to inferring program constraints without having to understand the application logic is to evaluate what is being done on most program paths related to a particular logic and treat that as a constraint and, if such a constraint is missing on one or few program paths, then that could be an indicator of a vulnerability assuming that the constraint is securely related or those could simply mean

that such program paths are exceptional conditions where the constraints do not need to hold.

6.96 Dependencies

Understanding dependencies is another critical approach to manual analysis dependencies exist when the correct compilation or functioning of program code relies on code or data from other smart contracts that were not necessarily developed by the project team explicit program dependencies are captured in the import statements, the inheritance hierarchy for example many projects use the community developed audited and time tested libraries from OpenZeppelin for tokens access control Proxy security etc composability in `web3` as we have discussed is expected and even encouraged via smart contracts interfacing with other protocols and vice versa which results in emergent or implicit dependencies on the state logic of external smart contracts via Oracle's for example this is especially of interesting concern for DEFI protocols that rely on other related protocols for stable points yield generation borrowing lending derivatives Oracles etc assumptions on the functionality and correctness of such dependencies need to be reviewed for potential security impacts.

6.97 Assumptions

A meta level approach is that of evaluating assumptions many security vulnerabilities result from quality assumptions such as who can access what and when under what conditions for what reasons etc identifying the assumptions made by the program code and verifying, if they are indeed correct can be the source of many audit findings some common examples of faulty assumptions are only admins can call these functions initialization functions will only be called once by the contract Deployer which is relevant for upgradable contracts functions will always be called in a certain order as expected by the specification parameters can only have non-zero values or values within a certain threshold for example addresses will never be `zero` value certain addresses or data values can never be attack and control they can never reach program locations where they can be misused in program analysis literature this is known as state analysis or function calls will always be successful and, so checking for `return` values is not required.

6.98 Checklists

And the eighth and final approach to manual analysis is the one we are using in this bootcamp which is that of evaluating security checklists checklists are lists of itemized points that can be quickly and methodically followed and you will reference later by their list number to make sure all listed items have been processed according to the domain of relevance for some context for those who

aren't aware of the significance of checklists this checklist-based approach was made popular in the book the catalyst manifesto how to get things right by atul grande who is a noted surgeon writer and public health leader, this idea is best summarized in the review of his book by malcolm gladwell who writes that governor begins by making a distinction between errors of ignorance mistakes we make because we don't know enough and errors of ineptitude mistakes we make because we don't make proper use of what we do failure in the modern world he writes is about the second of these errors and he walks us through a series of examples from medicine showing how the routine tasks of surgeons have now become, so incredibly complicated that mistakes are one kind or another are virtually inevitable it's just too easy for an otherwise competent doctor to misstep or forget to ask a key question or in the stress and pressure of the moment to `fail` to pan properly for every eventuality, then visits with pilots, the people who build skyscrapers and comes back to the solution experts need checklists literally written guides that walk them through the key steps in any complex procedure in the last section of the book govante shows how his research team has taken this idea developed a safe surgery checklist and applied it around the world with staggering `success`, so this glorifying review should hopefully motivate a better appreciation for checklists to apply this to our context consider the mind-boggling complexities of the fast evolving Ethereum infrastructure new platforms new languages new tools new protocols, the risks associated with deploying smart contracts managing billions of dollars, there are, so many things to get right with smart contracts that it is easy to miss a few checks make incorrect assumptions or `fail` to consider potential situations checklists are known to increase retention and have a faster remember the hypothesis therefore is that smart contract security experts need checklist too smart for that security checklist such as the two security modules we have discussed earlier in this bootcamp will help in navigating the vast number of key aspects to be remembered remembered and applied with respect to the pitfalls and best practices they will help in going over the itemized features concepts pitfalls best practices and examples in a methodical manner without missing any items they will also help in referencing specific items of interest, so for example number 42 in security pitfalls and best practices 101 or number 98 which is this slide in audit techniques and tools 101.

6.99 Exploit Scenarios

Presenting proof of concept exploit scenarios could be a part of certain audits remember that exploits are incidents where vulnerabilities are triggered by malicious actors to misuse smart contracts resulting for example in stolen or frozen assets presenting proof of concept of such exploits either in code or written descriptions of hypothetical scenarios make audit findings more realistic and relatable illustrating specific exploit paths and justifying the severity of findings it goes without saying that horrified exploit should always be on a test net kept private and responsibly disclosed to project teams without any risk of being

actually executed on live systems resulting in real loss of funds or access descriptive exploit scenarios should make realistic assumptions on roles powers of actors practical reasons for their actions and sequencing of events that trigger vulnerabilities and illustrate the paths to exploitation.

6.100 likelihood & Impact

We have talked about estimating likelihood impact and severity of the findings likelihood indicates the probability of a vulnerability being discovered by malicious actors and triggered to successfully exploit the underlying weakness impact indicates a magnitude of implications on the technical and business aspects on the system, if the vulnerability were to be exploited severity as per olaf is a combination of likelihood and impact with reasonable evaluations of those two severity estimates from the wasp matrix should be straightforward however estimating, if likelihood or impact are no medium or high is not trivial in many cases, if the exploit can be triggered by a few transactions manually without requiring much resources or access example not being an admin and without assuming many conditions to hold true, then the likelihood is evaluated as high exploits that `require` deep knowledge of the system workings privileged roles large resources or multiple edge conditions to hold true are evaluated as medium likelihood others that `require` even harder assumptions to hold true such as minor collusion chain forks or insider collusion for example are considered as low likelihood. If there is any loss or locking up of funds, then the impact is evaluated as high exploits that do not affect funds, but disrupt the normal functioning of the system are typically evaluated as medium and anything else is of low impact some evaluations of likelihood and impact are contentious and debated sometimes between the audit and project teams and sometimes even the security community at large, this typically happens with security conscious audit teams pressing for higher likelihood and impact while the project teams downplay the risks.

6.101 Audit Summary

So finally to summarize audits are a time resource and expertise bounded effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible whose difficulty impact and severity levels might vary similar to what dijkstra once said about software testing audits can only show the presence of vulnerabilities, but not their absence, so with that we have come to the end of this module on audit techniques and tools 101 where we have touched upon 101 key concepts related to the various aspects of auditing as related to the tools and techniques used, so hopefully you have a much better understanding and appreciation for what it takes to do smart contract security audits now and like always I would again encourage you to look at the related references especially the various tools

to get a practical feel for their capabilities and challenges.

Chapter 7

Audit Findings 101

In this module, we will review 101 findings from public audit reports of leading audit firms to get a sense for the kinds of issues reported during audits and their suggestive fixes or recommendations these 101 findings range from medium severity to high and critical, which are of the highest concern as they could have led to loss of funds or significantly affected execution, if they had not been detected and fixed during audits.

7.1 Audit Finding #1

This finding was a ConsenSys Diligence audit of \verbAave V2|| protocol. It was a medium severity finding in the error handling category. Specifically this was about unhandled `return` values of `transfer` and `transferFrom` functions on `ERC20` tokens. Remember that these `ERC20` functions may `revert`, `return` a `bool` or not return any value at all, and therefore any code using such functions on external `ERC20` tokens should anticipate all such scenarios because `ERC20` implementations are not always consistent and adhere to the specification. As discussed in number 149 of `Solidity` 201 module, it is safer to instead use OpenZeppelin's `safeERC20` wrapper functions in such cases.

The specific recommendation made here by the audit team was to check the `return` value and `revert` on `0` or `false`, or use OpenZeppelin's `safeERC20` wrapper functions.

7.2 Audit Finding #2

This finding was a ConsenSys Diligence audit of the DeFi Saver protocol. It was a critical vulnerability of the reentrancy type which allowed for a random task execution in the context of the protocol. Specifically, in a scenario where a user took a flash loan, one of the functions gave the flash loan wrapper contract permission to execute functions on behalf of the users `DSProxy`, this permission

was revoked only after the entire recipe execution finished, which meant that in a case that any of the external calls along the recipe execution was malicious, it could perform a reentrancy attack by injecting any task of choice leading to users funds being transferred out or draining approved tokens.

This vulnerability was due to potential re-entrances from malicious external calls and therefore the recommendation was to add a reentrancy guard, such as the one from OpenZeppelin where the `NonReentrant` modifiers are used on functions that may be vulnerable to reentrances. We have discussed these aspects in number 157 of `Solidity` 201 module and number 13 of security pitfalls and best practices 101 module.

7.3 Audit Finding #3

Number three is again another finding from ConsenSys Diligence audit of the DeFi Saver protocol. This was a major severity finding in the input validation category, where tokens with more than 18 decimal points could have caused issues. The code assumed that the maximum number of decimals for each token was 18. Howeverm although this is uncommon, it is possible to have tokens with more than 18 decimals for example Yam V2 has 24 decimals, and interacting with such tokens could have resulted in broken code flow and unpredictable outcomes.

The specific recommendation was to make sure that the code won't fail in case the token's decimals were more than 18 and was fixed by using `SafeMath sub` function to revert to tokens that have greater than 18 decimals. We have discussed this aspect of `ERC20` decimals in number 104, 159 and 170 of security pitfalls and best practices 201 module.

7.4 Audit Finding #4

Number four is another finding from ConsenSys Diligence audit of the DeFi Saver protocol. This was a major severity in the error handling category, where error codes of a few compound protocol functions called by these contracts were not checked. Some of compound's protocols functions return an error code instead of reverting in case of failure, but DeFi Saver contracts never checked for error codes returned from such functions, causing them to not react to exceptional conditions in such function calls by reverting or other means necessary.

The specific recommendation was for the caller contract to revert in case the error code returned was not zero, indicating a failure. This was fixed accordingly. We've discussed this aspect of checking for function return values at number 142 and 175 of security pitfalls and best practices 201 module.

7.5 Audit Finding #5

Number five is yet another finding from ConsenSys Diligence audit of DeFi Saver protocol. This was a medium severity in our ordering category. This vulnerability was due to the use of a reversed order of parameters in the `allowance` function call, where the parameters that were used for the allowance function call were not in the same order as what was later used in the call to `safetransferFrom`.

The recommendation was to reverse the order of parameters in the `allowance` function call to fit the order in the `safetransferFrom` function call and was fixed by swapping the order of parameters. We have discussed the concepts of `ERC20` token allowance and `safeERC20` wrappers in number 148 and 149 of `Solidity` 201 module. This exact specific aspect of checking for ordering issues of function arguments in number 139 and other security impacts of broader ordering issues in number 145 and 178 of security pitfalls and best practices 201.

7.6 Audit Finding #6

Number six is a finding from ConsenSys Diligence audit of the DAOfi protocol where it was a critical severity finding in the input validation category. The finding here was that token approvals can be stolen in the `addLiquidity` function of the protocol where the function created the desired contract, if it did not already exist, then transferred tokens into the pair. However, there was no validation of the address to transfer tokens from and so, an attacker could have passed in any address with non-zero token approvals to the DAOfi V1 route. This could have been used to add liquidity to a pair contract for which the attacker was the pair Owner allowing the stolen funds to be retrieved using the `withdrawal` function.

The recommendation was to transfer tokens from `msg.sender` instead of `lp.sender`. We have discussed the importance of access control checks on correct addresses in number 148, 149, 160, 172, 180, 181 and 183 of security pitfalls and best practices 201 module, and also the importance of input validation specifically on function parameters tokens and addresses in 138 and 159 of security pitfalls and best practices 201 module.

7.7 Audit Finding #7

Number seven is another finding from ConsenSys Diligence audit of the DAOfi protocol where it was a major severity finding in the error handling category. The error was that `swapExactTokensForETH` checked the wrong return value instead of checking that the amount of tokens received from a swap was greater than the minimum amount expected from the swap. It calculated

the difference between the initial receiver's balance and the balance of the router.

The recommendation was to check the intended values. We have discussed this aspect of correctly checking function return values in number 142 and 175 of security pitfalls and best practices 201 module.

7.8 Audit Finding #8

Number eight is another finding from ConsenSys Diligence audit of the DAOfi protocol where it was a medium severity finding in the denial of service category. In this case the DAOfi pair `deposit` function accepted deposits of zero amounts blocking the pool thereafter. This was because the function allowed only a single deposit to be made and no liquidity could ever be added to a pool after the deposit variable was set to `true`. However, the `deposit` function did not check for a non-zero deposit amount and so, allowed a malicious user that did not hold any tokens to lock the pool by calling `deposit` without first transferring any funds to the pool.

The recommendation was to require a minimum deposit amount with non-zero checks. We have discussed denial of service in number 136 of security pitfalls and best practices 201 module and also the importance of input validation, specifically on function parameters in 138 and number 146 of security pitfalls and best practices 201 module.

7.9 Audit Finding #9

Number 9 is a finding from ConsenSys Diligence audit of the Fei protocol where it was a critical severity finding in the application logic where the `GenesisGroup.commit` function overrode previously committed values. The amount stored in the recipient's `committedFGEN` balance overrode any previously committed value, including allowing anyone to commit an amount of zero to any account, deleting their commitment entirely.

The recommendation was to ensure that the committed amount is added to the existing commitment instead of overwriting it. This finding is related to the numerical and accounting issues we discussed in number 170 and 171 or security pitfalls and best practices 201 module and also the general challenges of detecting application specific business logic issues in number 191 of that same module.

7.10 Audit Finding #10

Number 10 is another finding from ConsenSys Diligence audit of the Fei protocol where it was a critical severity finding related to the timing category.

Here, purchasing and committing was still possible after launch, which meant that even after the `GenesisGroup.launch` had successfully been executed, it was still possible to invoke `GenesisGroup.purchase` and commit functions.

The recommendation was to consider adding validation by ensuring that these functions could not be called after launch. This finding is related to the ordering issues we discussed in number 145 and 178 and also the timing issues discussed in 143 and 177 of the security pitfalls and best practices 201 module.

7.11 Audit Finding #11

11 is another finding from ConsenSys Diligence audit of Fei protocol where it was a major severity finding related to the data validation category. Fei performed some mint/burn operations via `UniswapIncentive.incentivize` function, which calculated buy/sell incentives using overflow prone `map`, then minted/burned from the target based on the results. Any overflows would have had unintended consequences on such minting or burning. The specific overflow prone `map` was because of unSafeCasting from a user-supplied `uint256` argument in the externally visible function to `int256`, which is a downcast and may have overflowed without appropriate checks.

The recommendation was to ensure that casts do not overflow, and was addressed by the use of OpenZeppelin's `SafeCast`. This finding is related to OpenZeppelin's `SafeCast` wrappers we discussed in number 177 of `Solidity` 201 module, dangers of integer overflow underflow we discussed in number 19 of security pitfalls and best practices 101 module, the broader aspect of the importance of input validation specifically on function parameters in 138 and 146 of security pitfalls and best practices 201 module.

7.12 Audit Finding #12

Number 12 is another finding from ConsenSys Diligence audit of Fei protocol where it was a medium severity finding related to the timing category (similar to number 10). Specifically, `BondingCurve` allowed users to acquire FEI before launch, where its allocate function could be called before genesis launch if the contract had a non-zero Ether balance. So by sending Ether one way, anyone could bypass the checks and mint FEI.

The recommendation was to prevent `allocate` from being allowed to be called before genesis launch. This finding is related to the ordering issues we discussed in number 145 and 178, the timing issues discussed in 143 and 177 of security pitfalls and best practices 201 module and also misuse of a contracts in their balance as discussed in number 26 of security pitfalls and best practices 101 module and number 158 of security pitfalls and best practices 201 module.

7.13 Audit Finding #13

Number 30 is another finding from ConsenSys Diligence audit of Fei protocol where it was a medium severity finding related to the error handling category. The issue was that `Timed.isTimeEnded` function returned `true` even, if the timer had not been initialized. `Timed.startTime` was set only when `_initTimed` was called. But before that was called, `Timed.isTimeEnded` function calculated remaining time using a start time of 0 and returned `true` even though the timer had not been started.

The recommendation was for `Timed.isTimeEnded` to return `false` or revert, if time had not been initialized. This finding is related to error handling in the context of the ordering issues we discussed in number 145 and 178 and also the timing issues discussed in 143 and 177 of security pitfalls and best practices 201 module.

7.14 Audit Finding #14

Number 14 is another finding from ConsenSys Diligence audit of Fei protocol where it was a medium severity finding related to the data validation category. This was specifically related to the code base using many arithmetic operations without the safe versions from `safeMap`. The reasoning was that all values and such operations were derived from actual Eth values, so they couldn't overflow.

The recommendation was that it was still safer to have those operations use safe mode arithmetic either by using `safeMap` or `Solidity` version greater than or equal to `0.8.0`. We have discussed this aspect of `Solidity` compiler `0.8.0` and OpenZeppelin `safeMap` in number 142, 146, and 175 for `Solidity` 201 module and number 19 of security pitfalls and best practices 101 module.

7.15 Audit Finding #15

Number 15 is another finding from ConsenSys Diligence audit of Fei protocol where it was a medium severity finding related to error handling category. In this case there was no checking for the return value of `IWETH.transfer` call. It's usually good to add a `require` statement that checks the return value or, as discussed in number 149 of `Solidity` 201 module, it's safer to use something like OpenZeppelin's `safeTransfer` mapper unless one is absolutely sure that the given token reverts in case of a failure.

The recommendation was to consider adding a required statement or using safe transfer which handles all possibilities of reward `boolean` and non-boolean `return` values we have discussed this aspect of correctly checking for function `return` values in number 142 and 175 of security pitfalls and best practices 201 module.

7.16 Audit Finding #16

Number 16 is another finding from ConsenSys Diligence audit of Fei protocol where it was a medium severity finding related to the timing category similar to number 10 and number 12. In this case `GenesisGroup.emergencyExit` function remained functional after launch. `GenesisGroup.emergencyExit` was intended as an escape mechanism for users in the event that the genesis `launch` method failed or froze and so, `emergencyExit` and `launch` functions were intended to be mutually exclusive, but were not because either of them remained callable after a successful call to the other. Thus may have resulted in edge cases in accounting.

The recommendation was to ensure that `launch` can't be called, if emergency exit has been called and vice versa this finding is therefore related to the ordering issues we discussed in number 145 and 178, the timing issues discussed in 143 and 177 of security pitfalls and best practices to one module.

7.17 Audit Finding #17

Number 17 is a finding from ConsenSys Diligence audit of bitbank protocol where it was a major severity finding related to error handling category. In this case, `ERC20` tokens that did not return a value would fail to transfer. Remember that although the `ERC20` standard suggests that a `transfer` should return `true` on success, some tokens may be non-compliant and in such a case the `transfer` call here would revert even if it were successful because of `Solidity` checking that the return data size matches the `ERC20` interface.

The recommendation was to consider using OpenZeppelin's `safeERC20` wrappers. We have specifically discussed this in number 149 `Solidity` 201 module and number 24 are security pitfalls and best practices 101 module.

7.18 Audit Finding #18

Number 18 is a finding from ConsenSys Diligence audit of MetaSwap protocol where it was a major severity finding related to re-entrancy. This reentrancy vulnerability was a `MetaSwap.swap` function where, if an attacker was able to reenter swap, they could execute their own trade using the same tokens and get all the tokens for themselves.

The recommendation was to add `ReentrancyGuard` such as the one from OpenZeppelin, where the `NonReentrant` modifiers are used on functions such as `MetaSwap.swap` that may be vulnerable to reentrances. We have discussed these aspects in number 157 of `Solidity` 201 and number 13 of security pitfalls and best practices 101 modules.

7.19 Audit Finding #19

19 is another finding from ConsenSys Diligence audit of MetaSwap protocol where it was a medium severity finding related to access control. A new malicious adapter could access users' tokens. For context, the purpose of the MetaSwap contract was to save users' Gas costs when dealing with a number of different aggregators. They could approve their tokens to be spent by MetaSwap and they could then perform trades with all supported aggregators without having to reapprove anything. The risk in this design was that an existing, or newly added, malicious or buggy adapter would have access to all users' tokens.

The recommendation was to redesign token approval architecture by making MetaSwap contract the only contract that receives token approval, where it moves tokens to another spender contract which in turn `delegateCall`s to the appropriate adapter. In this revised model, newly added adapters wouldn't be able to access users' funds. We have discussed access control aspects in number 4 of security pitfalls and best practices 101 module and in 148 and 149 of security pitfalls and best practices 201 module. This is also related to token handling aspect of number 159 and trust issues of 181 or security pitfalls and best practices 201 module.

7.20 Audit Finding #20

Number 20 is another finding from ConsenSys Diligence audit of MetaSwap protocol where it was a medium severity finding related to the timing category. In this case, a malicious or compromised Owner could front-run traders by updating adapters to steal from users. Due to the design of adapters, where they `delegateCall`ed to each other, users had to fully trust every adapter because just one malicious adapter could change the logic of all other adapters.

The recommendation was to disallow modification of existing adapters, but instead add new adapters and disable the old ones. This finding is related to the transaction order dependence aspect discussion number 21 of security pitfalls and best practices 101 module time delay change of critical contract aspects in number 163, trust aspects in number 181 and constant aspects in number 184 of security pitfalls and best practices 201 module.

7.21 Audit Finding #21

Number 21 is a finding from ConsenSys Diligence audit of mstable protocol where it was a major severity finding related to the timing category. In this case, it was the views of a sliding window where users could collect interest by only staking mTokens momentarily. For more context, when users deposited m-assets, in return for lending yield and swap fees users received credit tokens

at an exchange rate which was updated at every deposit. However, it enforced a minimum time frame of 30 minutes in which the interest rate would not be updated. A user who deposited shortly before the end of the time frame would receive credits at the stale interest rate and could immediately trigger an update of the rate and withdraw at the updated and more favorable rate after the 30 minute window. As a result, it would be possible for users to benefit from interest payouts by only staking m-assets momentarily.

The recommendation was to remove the 30 minutes window such that every deposit also updated the exchange rate between credits and tokens. This finding is therefore related to the timing issues discussed in number 143 and 177 of security pitfalls and best practices 201 module where system design has to anticipate and prevent abuse of timing aspects and assumptions of the protocol.

7.22 Audit Finding #22

Number 22 is a finding from ConsenSys Diligene audit of Bancor V2 protocol where it was a critical severity finding related to the typing category. This issue was about Oracle updates that could be manipulated to arbitrage rate changes by sandwiching the Oracle update between two transactions. The attacker could send two transactions at the moment the Oracle update appeared in the mempool. The first transaction sent with a higher Gas Price than the Oracle update transaction, so as to front run it would convert a very small amount to lock in the conversion rate, so that the stale Oracle price would be used in the following transaction. The second transaction sent at a slightly lower Gas Price than the transactions that updated the Oracle, so as to back run it and effectively sandwich it would perform a large conversion at the old scale weight, add a small amount of liquidity to trigger rebalancing and then convert back at the new rate. The attacker could obtain liquidity for step two using a flash loan and use that to deplete the reserves.

The recommendation was to not allow users to trade at a stale Oracle rate and trigger an Oracle update in the same transaction. This finding is related to the transaction order dependence aspect discussed in number 21 of security pitfalls and best practices 101 module, ordering aspect discussed in number 178 and freshness aspect discussed in number 185 of the security pitfalls and best practices 201.

7.23 Audit Finding #23

Number 23 is a finding from ConsenSys Diligence audit of Shell protocol where it was a major severity finding related to the input validation category, where certain functions lack input validation such as: `uint` should be larger than 0 when 0 is considered invalid, `uint` should be within constraints or thresholds,

`int` should be positive in some cases, length of arrays should match if more than one related arrays are sent as arguments, and addresses should not be zero-addresses.

The recommendation was to add input validation and incorporate tests that check if all parameters had indeed been validated. We've discussed the importance of input validation on function parameters and arguments in number 138 and 146 of security pitfalls and best practices 201 module.

7.24 Audit Finding #24

Number 24 is another finding from ConsenSys Diligence audit of Shell protocol where it was a major severity finding related to access control. There were several functions that gave extreme powers to the protocol administrator, of which the most dangerous was the one granting capability where the administrator could intentionally or accidentally deploy malicious or faulty code that could drain the whole pool or lock up the users' and LP's tokens. Also the function `safeApprove` allowed the administrator to move any of the users' tokens in the contract to any address which could be used as a bad door to completely drain the contract.

The recommendation was to remove the `safeApprove` function and improve users trust by making the code static and unchangeable after deployment. We discussed the importance of least privileged principle in number 192 and principle of separation of privilege in number 193 of security pitfalls and best practices 201 module, this also related to access control and trust issues we discussed in number 148, 149, 160 and 172 or security pitfalls and best practices 201 module.

7.25 Audit Finding #25

Number 25 is a finding from ConsenSys Diligence audit of Lien protocol where it was a critical severity finding related to denial of service, where a reverting fallback function would lock up all payouts in the context of the `transferEth` function. If any of the Ether recipients of such batch transfers were to be a smart contract that reverted, then the entire payout would fail and be unrecoverable.

The recommendation was to implement a pull-withdrawal pattern or ignore a failed transfer leaving the responsibility then up to the recipients. We have discussed denial of service in number 176 and concerns with Ether handling functions in number 158 of security pitfalls and best practices 201 module. We have discussed concerns with calls within loops leading to denial of service In number 43 oF security pitfalls and best practices 101 module. We've also reviewed OpenZeppelin's PullPayment library which specifically addresses this

pull versus push aspect of Ether transfers in number 158 of `Solidity` 201 module.

7.26 Audit Finding #26

26 is a finding from ConsenSys Diligence audit of the LAO protocol where it was a critical severity finding related to denial of service. The issue was related to `safeRagequit` and `ragequit` functions used for withdrawing funds from the LAO, The difference between them was that while `ragequit` function tried to withdraw all the allowed tokens, `safeRagequit` function only withdrew some subset of those tokens as defined by the user. The problem was that, even though one could quit, they would lose the remaining tokens. The tokens were not completely lost, but they would belong to the LAO and could potentially still be transferred to the user who quit. However, that required a lot of trust coordination and time and anyone could steal some of those tokens.

The recommendation was to implement a pull-pattern for token withdrawals. We have discussed denial of service in numbers 176 of security pitfalls and best practices 201 module.

7.27 Audit Finding #27

27 is another finding from the ConsenSys Diligence audit of the LAO protocol where it was another critical severity finding again related to denial of service. The issue was that, if someone submitted a proposal and transferred some amount of tribute tokens, these tokens were transferred back if the proposal was rejected. But if the proposal was not processed before the emergency processing, these tokens would not be transferred back. The proposal tokens were not completely lost, but belong to the LAO shareholders who may try to return that money back, but that required a lot of coordination and time and everyone who `ragequit` during that time would take a part of those tokens.

The recommendation again was to use a pull pattern for token transfers this is again related to the derivatives of this aspect we discussed in number 176 of security pitfalls and best practices 201 module.

7.28 Audit Finding #28

28 is another finding from ConsenSys Diligence audit of the LAO protocol where it was another critical severity finding again related to denial of service. The specific issue here was that emergency processing could be blocked. The rationale for emergency processing mechanism was that there was a chance that some token transfers may be blocked, and in such a scenario emergency processing would help by not transferring tribute tokens back to the user

and rejecting the proposal. The problem was that there was still a deposit transferred back to the sponsor, that could potentially be blocked too. So, if that were to happen, the proposal couldn't be processed and the allowed will be blocked.

The recommendation again was to use a pull-pattern for token transfers, this is again related to the denial of service aspect we discussed in number 176 of security pitfalls and best practices 201 module.

7.29 Audit Finding #29

Number 29 is another finding from Consensus Diligence audit of the LAO protocol where it was a major severity finding again related to denial of service. In this issue, token overflows might result in system halt or loss of funds because some functionality such as `processProposal` and `cancelProposal` would break due to `SafeMath` reverts. The overflows could happen because the supply of the token was artificially inflated.

The recommendation was to allow overflows for broken or malicious tokens to prevent system halt or loss of funds, but recognizing that in case such overflows occur, the balance of tokens would be incorrect for all token holders in the system. This is again related to the denial of service aspect we discussed in number 176 of security pitfalls and best practices 201e module, and also the dangers of integer overflow underflow we discussed in number 19 our security pitfalls and best practices 101 module.

7.30 Audit Finding #30

Number 30 is another finding from ConsenSys Diligence audit of the LAO protocol where it was another major severity finding again related to denial of service. The issue was that while iterating over all whitelisted tokens, if the number of tokens was too large, a transaction could run Out-of-Gas and all funds would be blocked forever.

The recommendation was to limit the number of whitelisted tokens or to add a function to remove tokens from the whitelist. This is related to the aspect of loop operations leading to denial of service from Out-of-Gas exception in number 42 and denial of service from block Gas Limit because of looping over a raise of unknown size we discussed in number 44 of security pitfalls and best practices 101 module, the broader aspects of denial of service and number 176 and Gas issues we discussed in number 182 of security pitfalls and best practices 201 module.

7.31 Audit Finding #31

Number 31 is another finding from ConsenSys Diligence audit of the LAO protocol where it was another major severity finding again related to denial of service. The issue was that the summoner could steal funds using the `bailout` functionality.The `bailout` function allowed anyone to transfer the kicked users' funds to the summoner if the user did not call `safeRangequit`. The intention was for the summoner to then transfer those funds to the kicked member afterwards. But the issue was that it required a lot of trust on the summoner for doing so, and they could deny such a transfer.

The recommendation again was to use a pull mechanism which would make the `bailout` function unnecessary. This denial of service aspect is something we discussed in number 176 of security pitfalls and best practices to one module.

7.32 Audit Finding #32

Number 32 is another finding from ConsenSys Diligence audit of the LAO protocol where it was another major severity finding related to timing and denial of service. If the proposal submission and sponsorship were done in two different transactions, it was possible to front run the sponsor proposal function by any member, so that they could block the proposal thereafter.

The recommendation of pull pattern for token transfers would solve the issue by making Front-running ineffective. This is related to the transaction order dependence aspect discussed in number 21 of security pitfalls and best practices 101 module and denial of service aspect that we discussed in number 176 of security pitfalls and best practices 201.

7.33 Audit Finding #33

Number 33 is another finding from the ConsenSys Diligence audit of the LAO protocol where it was another major severity finding related to timing and denial of service. Any member could front run another member's delegate key assignment where, if one tried to submit an address as the delegate key, someone else could try to assign that delegate address to themselves making it possible to block some address from being a delegate forever.

The recommendation was to make it possible to approve delegate key assignment or cancel the current delegation commit. Reveal methods could also be used to mitigate this attack. This is related to the transaction order dependence aspect discussed in number 21 or security pitfalls and best practices 101 module and denial of service aspect that we discussed in number 176 of security pitfalls and best practices 201.

An interesting point to note here is that the project decided to not fix this finding as reported in the audit report presumably because they did not see this as a major severity issue in the risk model. As discussed in the module on audit techniques and tools 101, the findings and recommendations reported by an audit team may not necessarily be fixed by the project team for different reasons, but usually because they're outside the threat model or within the trust model of the project.

7.34 Audit Finding #34

Number 34 is a finding from Trail of Bits audit of the Origin Dollar where it was a high severity finding related to denial of service. The specific issue was that queued transactions could never be cancelled. The governor contract contained special functions to set it as the admin of the `Timelock` and only the admin could call `Timelock.cancelTransaction`, but there were no functions in governor that called `Timelock.cancelTransaction` which made it impossible for `Timelock.cancelTransaction` to ever be called

The recommendation was in the short term to add a function to the governor that calls `Timelock.cancelTransaction` and in the long term considering letting governor inherit from `Timelock`, which would allow a lot of code to be removed and significantly lower the complexity of the two contracts. This is related to the denial of service aspect that we discussed in number 176 of security pitfalls and best practices 201 module.

7.35 Audit Finding #35

Number 35 is another finding from Trail of Bits audit of the Origin Dollar where it was a high severity finding related to access control. Missing access controls in the `Timelock.executeTransaction` function allowed proposal transactions to be executed separately, circumventing the governor `execute` function.

The recommendation was to allow only the admin to call `Timelock.executeTransaction`. This is related to access control aspects discussed in number 4 of security pitfalls and best practices 101 module and also in number 141, 148, 149, 150 and 172 or security pitfalls and best practices 201 module.

7.36 Audit Finding #36

Number 36 is another finding from Trail of Bits audit of the Origin Dollar where it was another high severity finding related to access control. The governor contract contained special functions to let the guardian queue a transaction to change the `Timelock`. However a regular proposal was also allowed to contain

a `setPendingAdmin` transaction to change the `Timelock.admin`, which posed an unnecessary risk in that an attacker could create a proposal to change the `Timelock.admin` themselves.

The recommendation was to add a check that prevented `setPendingAdmin` to be included in a regular proposal. This is related to access control aspects discussed in number 148, 149 and 172 and also the principle of complete mediation discussed in number 196 of security pitfalls and best practices 201 modules.

7.37 Audit Finding #37

Number 37 is another finding from Trail of Bits audit of the Origin Dollar where it was a high severity finding related to re-entrancing. Missing checks and no reentrancy prevention allowed untrusted contracts to be called from `mintMultiple` function which could be used by an attacker to drain the contract. The recommendation was in the short term to add checks to cause `mintMultiple` to revert if the amount was zero or the asset was not supported, and also to add a reentrancy guard to the `mint`, `mintMultiple`, `redeem` and `redeemAll` functions, and in the long term to make use of Slither and incorporate static analysis checks into the CI/CD pipeline, which would have flagged the reentrancy. Other recommendations were to add reentrancy guards to all non-view functions callable by anyone, ensure to always revert a transaction if an input is incorrect and to disallow calling untrusted contracts. We have discussed these aspects in number 157 of `Solidity` 201 and number 13 of security pitfalls and best practices 101 modules.

7.38 Audit Finding #38

Number 38 is another finding from Trail of Bits audit of the Origin Dollar where it was a high severity finding related to error handling. The issue was that several function calls did not check the return value without which the code is error-prone and may lead to unexpected results.

The recommendation was to check the return value of all function calls that return a value. We have discussed this in number 74 of security pitfalls and best practices 101 module and number 142 of security pitfalls and best practices 201.

7.39 Audit Finding #39

Number 39 is another finding from Trail of Bits audit of the Origin Dollar where it was a high severity finding related to denial of service. Several function calls were made in unbounded loops, which is Error-prone because it can trap the contracts due to Gas Limitations or failed transactions.

The recommendation was to review all the loops to allow iteration over part of the loop, or remove elements depending on Gas consumption to prevent denial of service. This is related to the aspect of making external calls within loops leading to denial of service from Out-of-Gas exceptions discussed in number 43 of security pitfalls and best practices 101 module and broader aspects of denial of service in number 176 and Gas issues we discussed in number 182 of security pitfalls and best practices 201.

7.40 Audit Finding #40

Number 40 is another finding from Trail of Bits audit of the Origin Dollar where it was a high severity finding related to data validation. Under certain circumstances the OUSD contract allowed users to transfer more tokens than they had in their balance, which is caused by a rounding issue.

The recommendation was in the short term to make sure the balance is correctly checked before performing all the arithmetic operations and in the long term to use the Trail of Bits tool Echidna to write properties on arithmetic invariants that ensure `ERC20` transfers are transferring the expected amount. This is related to aspects of token handling in number 159 data validation issues number 169 and numerical issues in number 170 we discussed in security pitfalls and best practices 201 module.

7.41 Audit Finding #41

Number 41 is another finding from the Trail of Bits audit of the Origin Dollar where it was another high severity finding related to data validation. OUSD total supply could be arbitrary and even smaller than user balances because the OUSD token contract allowed users to opt out of rebasing effects, at which point their exchange rate would be fixed and further rebases would not have any impact on token balances until the user opts back in.

The recommendation was to specify all common invariant violations for users and other stakeholders in the short term and redesign the system in the long run to preserve as many common invariants as possible. This is related to aspects of token handling in number 159, data validation issues in number 169 and numerical issues in 170 We discussed in security pitfalls and best practices 201 module.

7.42 Audit Finding #42

Number 42 is a finding from Trail of Bits audit of Yield protocol, where it was a medium severity finding leading to undefined behavior. The issue was that the flash minting feature from the fyDAI token could be used to redeem

an arbitrary amount of funds from a mature token in the context of the product.

The recommendation was effectively to disallow flash minting to prevent attackers from gaining leverage to manipulate the market and break internal invariants. This is related to aspect of flash minting in number 121 and Scarcity issues we discussed in number 186 of security pitfalls and best practices 201 module.

7.43 Audit Finding #43

Number 43 is another finding from Trail of Bits audit of Yield protocol, where it was a high severity finding related to access control in which a lack of `chainID` validation allowed signatures to be reused across forks YDAI implemented the draft `ERC2612` standard which allows a third party to transmit a signature from a token holder that modified the `ERC20` allowance for a particular user. These signatures used in calls to permit an `ERC20` permit did not account for chain splits and as a result, if the chain forked after deployment, the signed message would be considered valid on both forks.

The recommendation was to include the `chainID` opcode in the permit schema to make replay attacks impossible in the event of a post-deployment hard fork. This is related to aspects of cryptography issues we discussed in number 174 of security pitfalls and best practices 201 module.

7.44 Audit Finding #44

Number 44 is a finding from Trail of Bits audit of Hermez, where it was a high severity finding related to data validation in which lack of a contract existence check effectively allowed token theft.

The recommendation was to check for contract existence and `_safeTransferFrom` function before interacting with the contract remember that the `Solidity` documentation talks about low-level call `delegateCall` and `call` code returning success even if the called account is non-existent. This is related to number 87 of `Solidity` 101 module, number 38 of security pitfalls and best practices 101 module, the broader aspects of external interaction issues we discussed in number 174 of security pitfalls and best practices 201.

7.45 Audit Finding #45

Number 45 is another finding from Trail of Bits audit of Hermez where it was a medium severity finding related to timing, in which there was no incentive for bidders to vote earlier. Hermez relied on a voting system that allowed anyone to vote with any weight at the last minute and there was no incentive for users to bid tokens well before the voting ends. This allowed users to

build a large amount of tokens just before voting ended and so, anyone with a large fund could decide the outcome of the world. With all the words being public, users bidding earlier would be penalized because their bids would be known by other participants and an attacker could know exactly how many tokens would be necessary to change the outcome of the voting just before it ended.

The recommendation was to explore ways to incentivize users to vote earlier by considering a weighted bid with weight decreasing over time and also to recognize the many challenges of blockchain based online voting. This is related to timing issues of number 177 and incentive issues of number 187 that we discussed in security pitfalls and best practices 201 module.

7.46 Audit Finding #46

Number 46 is another finding from Trail of Bits audit of Hermez where it was a high severity finding related to access control. The system used the same account to change both frequently updated parameters and less frequent ones which is an Error-prone and risky design, because it increases the severity of any privileged account compromises.

The recommendation was to use a separate account to handle updating the less frequently changed parameters and in the long term to design and document the access control architecture carefully. This is related to access control aspects discussed in number 148, 149 and 172 and also the principle of least common mechanism we discussed in number 194 of the security pitfalls and best practices 201 module.

7.47 Audit Finding #47

Number 47 is another finding from Trail of Bits audit of Hermez where it was a high severity finding related to data validation. The issue was the one-step procedure for critical operations that is Error-prone and can lead to irrecoverable mistakes. For example the setter for the white hat group address sets the address to the provided argument, if the address is incorrect, then the new address would take on the functionality of that role immediately leaving it open to misuse by anyone who controlled that new address or, if the new address was one without an available private key, it would lock access to that role forever.

The recommendation was to use a two-step procedure for all such Critical Address updates to prevent irrecoverable mistakes. This is related to the two-step change of privilege rules we discussed in number 50 of security pitfalls and best practices 101 module and number 162 of the security pitfalls and best practices to one module.

7.48 Audit Finding #48

Number 48 is another finding from Trail of Bits audit of Hermez where it was a high severity finding related to configuration, in which Hermez auction protocol and withdrawal delayer had initialization functions that could be frontrun due to the use of the `delegateCall` Proxy pattern. These contracts could not be initialized with the constructor and had initializer functions, all of whom could be frontrun by an attacker, allowing them to initialize the contracts with malicious values.

The recommendation was to either use a factory pattern that would prevent front-running of the initialization or ensure deployment scripts are robust to prevent front-running attacks by atomically deploying and initializing. This was discussed in number 192 of `Solidity` 201, number 21 and number 95 of security pitfalls and best practices 101 module and number 143 and 166 of the security pitfalls and best practices 201 module.

7.49 Audit Finding #49

Number 49 is a finding from Trail of Bits audit of Uniswap V3 protocol where it was a medium severity finding related to data validation, in which missing validation of Owner argument in both the constructor and `setOwner` functions could permanently lock the Owner role, if a Zero-address or incorrect address were to be used. That would have forced an expensive redeployment of the factory contract followed by re-addition of pairs and liquidity which could have led to reputational damage among its users and potentially concurrent use of two versions of Uniswap.

The recommendations were

1. Designate `msg.sender` as initial Owner and transfer ownership to the chosen Owner after deployment.
2. Implement a two-step ownership change process through which the new Owner needs to accept ownership.
3. If it was needed, to be possible to sent the Owner to Zero-address, implement a separate `renounceOwnership` function.

This is related to the missing Zero-address validation we discussed in number 49 of security pitfalls and best practices 101 module and also the two-step change of privilege roles we discussed in number 50 of the same module along with number 162 of the security pitfalls and best practices 201 module.

7.50 Audit Finding #50

Number 50 is another finding from Trail of Bits audit of Uniswap V3 protocol where it was a high severity finding related, again, to data validation in which an incorrect comparison in the swap function allowed the swap to succeed even if no tokens were paired. This issue could be used to drain any pool of all of its tokens at no cost. The check inside one of the `require`s in that function was incorrect because it used `>=` instead `<=`.

The recommendation was to simply replace the `>=` with `<=` in that `require` statement. This is related to the aspect of conditionals we discussed in number 147, the broader aspect of data validation issues in number 169 of security pitfalls and best practices 201 body.

7.51 Audit Finding #51

Number 51 is another finding from Trail of Bits audit of Uniswap V3 protocol where it was a medium severity finding related to denial of service in the swap function, which relied on an unbounded loop that an attacker could exploit to disrupt swap operations by forcing the loop to go through too many operations and potentially trapping the swap due to Out-of-Gas exception.
The recommendation was to bound the loops and document the bounds. We have discussed concerns with calls within loops leading to denial of service in number 43 of security pitfalls and best practices 101 module and more broadly the deriver service issues in number 176 and Gas issues in 182 or security pitfalls and best practices to a one module.

7.52 Audit Finding #52

52 is another finding from Trail of Bits audit of Uniswap V3 protocol where it was a medium severity finding related to timing and access control where a frontrun on Uniswap V3 `poolInitialize` function allowed an attacker to set an unfair price and to drain assets from the first deposits. There were no access controls on the initialize function, so anyone could call it on a deployed pool, initializing a pool with an incorrect price allowed an attacker to generate profits from the initial liquidity providers deposits.

The recommendation was to

1. Move the price operations from initialize to the constructor or
2. Adding access controls to initialize or
3. Ensuring that the documentation clearly wanrs users about incorrect initialization.

This is discussed in number 192 of `Solidity` 201, number 21 and number 95 of security pitfalls and best practices 101 module and number 143 and 166 of the security pitfalls and best practices 201 module.

7.53 Audit Finding #53

Number 53 is another finding from Trail of Bits audit of Uniswap V3 protocol where it was a medium severity finding related to application logic, where swapping on a tick with `zero` liquidity enabled a user to adjust the price of one wei of tokens in any direction and, as a result, an attacker could set an arbitrary price at the pool's initialization or if when the liquidity providers withdrew all of the liquidity for a short time.

The recommendation was to ensure a design where pools don't end up in unexpected states or warn users of potential risks. This is related to business logic issues discussed in number 191 or security pitfalls and best practices 201 module.

7.54 Audit Finding #54

Number 54 is another finding from Trail of Bits audit of Uniswap V3 protocol where it was a high severity finding related to data validation, in which failed transfers may be overlooked due to lack of contract existence check. `TransferHelper.safeTransfer` performed a transfer with a low level call without confirming the contract's existence. As a result, if the tokens had not yet been deployed or had been destroyed, `TransferHelper.safeTransfer` would still have returned success even though no transfer had actually executed.

The recommendation was to check for contract existence before interacting with the contract. Remember that the solicited documentation warns about low level call `delegateCall` and `call` code returning success even if the called account is non-existent. This is related to number 87 or `Solidity` 101 module, number 38 of security pitfalls and best practices 101 module, the broader aspects of external interaction issues we discussed in number 174 of security pitfalls and best practices 201 module.

7.55 Audit Finding #55

Number 55 is a finding from Trail of Bits audit of DFX protocol where it was a high severity finding related to undefined behavior, in which the left hand side of an equality check had an assignment of the variable output amount, the right hand side of that check used the same variable. According to `Solidity`. documentation such a check constituted an instance of undefined behavior and as such, the behavior of that code was not specified and could change in future

releases of `Solidity`.

The recommendation was to rewrite the `if` statement such that it did not use and assign the same variable in an equality check and in general avoid undefined language usages. This is broadly related to the undefined behavior issues we discussed in number 179 of security pitfalls and best practices 201 module.

7.56 Audit Finding #56

Number 56 is another finding from Trail of Bits audit of DFX protocol where it was a high severity finding related to data validation, where certain functions returned raw values instead of converting them to numerical values that it used for its internal authority. Interchanging raw and human values would produce unwanted results and may have resulted in loss of funds for liquidity providers.

The recommendation was to change the semantics of such functions to return the numeric balance instead of raw balance and in the long term, use unit tests and fuzzic to ensure that all calculations return the expected values. This is related to the broad aspect of data validation issues we discussed in number 169 of security pitfalls and best practices 201 modules.

7.57 Audit Finding #57

57 is another finding from Trail of Bits audit of DFX protocol where it was a medium severity finding related to data validation, in which the system incorrectly assumed that one USDC is always worth one USD without using the USDC $\rightarrow$ USD Oracle provided by Chainlink, and therefore could result in exchange errors at times of deviation.

The recommendation was to replace the hard coded integer literal with the `getRate` method with a call to the relevant Chainlink Oracle, so as to ensure that the system is robust against changes in the price of any stablecoin. This is related to the broad aspect of data validation issues we discussed in number 169 and specifically the constant issues in number 184 of security pitfalls and best practices 201.

7.58 Audit Finding #58

Number 58 is another finding from Trail of Bits audit of DFX protocol where it was an undetermined severity finding related to patching, in which the system used a deprecated old version of the Chainlink price feed API aggregator interface throughout the contracts and the test. For example, the duplicated function `latestAnswer` was used, which is not present in the latest API reference aggregator interface V3. In the worst case scenario, the deprecated

API could cease to report the latest values which would very likely cause protocol liquidity providers to incur losses.

The recommendation was to use the latest stable versions of any external libraries or contracts used by the code panes as external dependencies. This is related to the broad aspects of configuration issues of number 165, external interaction issues of number 180 and dependency issues of 183. We discussed in security pitfalls and best practices 201 module.

7.59 Audit Finding #59

Number 59 is a finding from Trail of Bits audit of 0x protocol where it was a high severity finding related to data validation, in which the `cancelOrdersUpTo` function could cancel future orders, if it were called with a very large value such as `MAX_UINT256 - 1`.

The recommendation was to properly document this behavior, to warn users about the permanent effects of `cancelOrdersUpTo` on future orders or, alternatively, disallow the cancellation of future orders. This is related to the broad aspect of data validation issues we discussed in number 169 of security pitfalls and best practices 201 module.

7.60 Audit Finding #60

Number 60 is another finding from Trail of Bits audit of 0x protocol where it was a high severity finding related to specification, in which there was a specification called mismatch for asset Proxy Owner timelock period while the specification indicated that submitted transactions must pass a two week timelock before they were executed. The timelock period implementation did not enforce the two week period, but was instead configurable without any rain checks indicating that either the specification was outdated or that this was a serious flaw.

The recommendation was to implement necessary rain checks to enforce the timelock period described in the specification, or otherwise correct the specification to match the intended behavior. This is related to the broad aspect of system specification of numbers 136 and 155, and clarity issues of 188 we discussed in security pitfalls and best practices 201 module.

7.61 Audit Finding #61

Number 61 is another finding from Trail of Bits audit of 0x protocol where it was another high severity finding, again related to specification in which neither the 0x specification nor non-documentation stated clearly enough how

fillable orders were determined.

The recommendation was to define a proper procedure to determine if an order was available and detail it in the protocol specification and, if necessary, warn the users about potential constraints or others. This is related to the broad aspects of system specification of number 136 and 155, and clarity issues of 188 we discussed in the security pitfalls and best practices 201.

7.62 Audit Finding #62

Number 62 is another finding from Trail of Bits audit of 0x protocol where it was a medium severity finding related to timing, in which market makers had a reduced cost for performing Front-running attacks. For context, market makers received a portion of the protocol fee for each order failed. The protocol fee was based on the transaction Gas Price which meant that market makers could specify a higher Gas Price for a reduced overall transaction rate, using the refund they would receive upon disbursement of protocol fee ports.

The recommendation in the short term was to properly document that issue to make sure users were aware of that risk and in the long term, consider using an alternative fee that did not depend on the transaction gas price to avoid reducing the cost of performing Front-running attacks. This is related to the transaction order dependence aspect discussed in number 21 of security pitfalls and best practices 101 module and broader aspects of timing in 177 Gas and 182 and incentives in 187 from security pitfalls and best practices 201 module.

7.63 Audit Finding #63

Number 63 is another finding from Trail of Bits audit or 0x protocol where it was a medium severity finding, again related to timing in which if a validator has optimized a Race-condition in the signature, validator approval logic became exploitable. The `setSignatureValidatedApproval` function allowed users to delegate the signature validation to a contract but, if the validator was compromised, a Race-condition in this function could allow an attacker to validate any number of malicious transactions.

The recommendation was in the short term to document this behavior, to make sure users were aware of the inherent risks of using validators in the case of a compromise and in the long term to consider monitoring the blockchain for signature validator approval events to catch such Front-running attacks. This is related to the transaction order dependence aspect discussed in number 21 of security pitfalls and best practices 101 module and broader aspects of timing in 177 external interactions in 180 and trust issues in number 181 from security pitfalls and best practices 201 the principle of compromise recording from

number 201 of security pitfalls and best practices 201 module is also relevant here.

7.64 Audit Finding #64

Number 64 is another finding from Trail of Bits audit of 0x protocol where it was a medium severity finding related to deny of service, in which batch processing of transaction execution and order matching may lead to exchange griefing. For context, batch processing of transaction execution and order matching would iteratively process every transaction and order but, if one transaction or order failed because of insufficient allowance, the entire batch would revert and need to be resubmitted after removing the reverting transaction.

The recommendation was to implement `NoThrow` variants that is that do not revert for such bad processing and take into consideration the effect of malicious inputs when implementing functions that perform a batch of operations. We have discussed concerns with calls within loops which is representative of batch transactions leading to denial of service and number 43 of security pitfalls and press practices 101 module and more broadly the issues of error reporting in 175 and denial of service in 176 of security pitfalls and best practices 201 modules.

7.65 Audit Finding #65

Number 65 is another finding from Trail of Bits audit of 0x protocol where it was a medium severity finding related to data validation in which zero fee orders were possible, if a user performed transactions with a zero gas price.

The recommendation was to select a reasonable minimum value for the protocol fee for each order or transaction, and also to consider not depending on the gas price for the computation of protocol fees. We should also avoid giving miners an economic advantage in this protocol. This is related to the broad aspects of data validation issues in 169 Gas and 182 and incentives in 187 of security pitfalls and best practices 201 modules.

7.66 Audit Finding #66

Number 66 is another finding from Trail of Bits audit of 0x protocol where it was a medium severity finding related to data validation, in which calls to `setParents` may set invalid values and produce unexpected behavior in the staking contracts. `setParams` allows the Owner of the staking contracts to re-parameterize critical parameters, however reparametrization, lag, sanity, threshold or limit checks on all the parameters.

The recommendation was to add proper validation checks on all parameters in `setParams` and where the validation procedure was unclear or too complex to implement on-chain, document potential issues that could produce invalid values. This is related to system documentation in 137, function parameter validation in 138, function invocation arguments in 146 and broader aspect of data validation issues in 169 we discussed in security pitfalls and best practices 201.

7.67 Audit Finding #67

Number 67 is a finding from Sigma Prime's audit of synthetics EtherCollateral where it was a high severity finding related to data validation, in which there was improper enforcement of supply cap limitation. The `openLoan` function only enforced that the supply cap was not reached before the loan was opened without considering the loan amount being opened. As a result, any account could create a node that exceeded the maximum amount that could be issued by the EtherCollateral contract.

The recommendation was to add a `require` statement in the `openLoan` function to prevent the total cap from being exceeded by the loan to be open. This is related to token handling in number 159 and broader aspects of data validation in 169 and accounting issues in 171 that we discussed in the security pitfalls and best practices 201 module.

7.68 Audit Finding #68

Number 68 is another finding from Sigma Prime audit of synthetics EtherCollateral, where it was a high severity finding related to data validation and denial of service resulting from improper storage management. During opening loan accounts, when loans were opened, the associated account address got added to the accounts with open loans array regardless of whether it was already present in that array. Additionally, it was possible for a malicious attacker to create a denial of service condition exploiting the unbounded storage array in accounts synthetics.

The recommendation was to consider changing the `storeLoan` function to only push an account to the accounts with open loans array, if the loan to be stored was the first one for that particular account and to introduce a limit to the number of loans each account could have. This is related to the broad aspects of data validation in 169 accounting in 171 and denial of service issues in 176 that we discussed in security pitfalls and best practices 201 module.

7.69 Audit Finding #69

Number 69 is another finding from Sigma Prime's auditor synthetics Ether-Collateral where it was a medium severity finding related to configuration, in which the contract Owner could arbitrarily change minting fees and interest rates. The issue free rate and interest rate variables could both be changed by the intercollateral contact Owner anytime after loans had been opened.

The recommendation was to consider making the minting fee that's issue free rate to be a constant that can't be changed by the Owner. This is related to the time delayed change of critical parameters in number 163 configuration issues in 165 and constant issues in 184 that we discussed in security pitfalls and best practices to one body.

7.70 Audit Finding #70

Number 17 is a finding from Sigma Prime audit of Infinigold where it was a critical severity finding related to configuration, in which there was an incorrect Proxy implementation that prevented contract upgrades. The token implementation contract initialized: order, name, symbol and decimal state variables in a constructor instead of an initialize function. Therefore, when token Proxy made a `delegateCall` to token implementation, it would not be able to access any of the state variables of the token implementation contract. Instead, the token Proxy would access its local storage which would not contain the variables set in the constructor of the token implementation contract and so, the Proxy call to the implementation was made. Variables such as order would be uninitialized and effectively sent to their default values without access to the implementation state variables. The Proxy contract was rendered unusable.

The recommendation was one to set fixed constant parameters as constants because then, the Proxy contract wouldn't need to initialize anything to implement a standard Proxy implementation which uses an initializd function instead of a constructor and a few other recommendations as well. This is related to OpenZeppelin's OZ Initializable library in number 192 and other Proxy related aspects we discussed in `Solidity` 201 module, the aspect of initializing state variables in Proxy-based upgradable contracts in number 96 of security pitfalls and best practices 101 module along with the broader aspects of configuration in 165 and initialization in 166 that we discussed in security pitfalls and best practices 201 module.

7.71 Audit Finding #71

Number 71 is another finding from Sigma Prime audit of Infinigold where it was a high severity finding related to access control in which the `transferFrom`

function in the token implementation contract did not verify that the sender, that's the `from` address, is not blacklisted. As such, it was possible for a user to allow an account to spend a certain allowance regardless of their blacklisting status.

The recommendation was to use the `notBlacklisted` address modifier on the `from` address besides the `msg.sender` and two addresses. This is related to the aspect of access control in number 4 of security pitfalls and best practices 101 module and aspects of function modifiers in 141 and missing or incorrectly used modifiers and 150, 152 and broader aspects of access control in 172 that we discussed in security pitfalls and best practices 201 modules.

7.72 Audit Finding #72

Number 72 is a finding from Sigma Prime's audit of synthetics Unipool where it was a critical severity finding related to ordering, in which the wrong order of operations led to exponentiation of reward per token stored value because reward per token stored was mistakenly used in the numerator of a fraction instead of being added to the function. This would allow users to withdraw more funds than allocated to them or being unable to withdraw their funds at all because of insufficient SNX balance.

The recommendation was to fix the operand ordering in the expression. As expected, this is related to numerical issues of 170 and accounting issues of 171 that we discussed in the security pitfalls and best practices 201 modules.

7.73 Audit Finding #73

Number 73 is another finding from Sigma Prime's audit of synthetics Unipool where it was a high severity finding related to timing and ordering, in which staking before the initial notify reward amount led to disproportionate rewards. So, if a user successfully staked an amount of UNI tokens before the function notify, the word was called for the first time, their initial user reward per token paid would be set to zero, the staker would be paid out funds greater than their share of SNX demands.

The recommendation was to prevent state from being called before notified word amount was called for the first time. This is related to function invocation order in 145 and broader aspects of ordering in 178 and business logic issues in 191 that we discussed in security pitfalls and best practices 201 module.

7.74 Audit Finding #74

Number 74 is another finding from Sigma Prime's audit of synthetics Unipool where it was a high severity finding related to error handling, in which an external call reverting would block minting. For context, the function notify reward amount would revert if `block.timestamp` was less than period finish, but this function was called indirectly via the synthetics mint function which meant that a reward would cause the external call to fail and thereby halt the mint process.

The recommendation was to consider handling the case where the reward period had not elapsed without reverting. This is related to token handling in number 159, the broader aspect of error reporting issues in 175 that we discussed in security pitfalls and best practices 201 module.

7.75 Audit Finding #75

Number 75 is another finding from Sigma Prime's audit of synthetics Unipool where it was a medium severity finding related to timing and denial of service, in which gap between periods led to erroneous rewards. For context, the SNX rewards were earned each period based on reward and duration, as specified in the notify reward amount function and, if all stakers called get reward function, the contract would not have enough SNX balance to transfer out all the rewards and some stakers may not receive any rewards.

The recommendation was to force each period to start at the end of the previous period without any cap. This is related to function invocation timeliness in 143 token handling in 159, the broader aspect of denial of service in 175 and timing in 176 that we discussed in security pitfalls and best practices 201 module.

7.76 Audit Finding #76

76 is a finding from Sigma Prime audit of Chainlink where it was a high severity finding related to timing and denial of service, in which malicious users could DoS or hijack requests from changing contracts by replicating or Front-running legitimate requests. This is made possible because requests could specify their own callbacks which could be abused by an attacker to front run or force the failure of legitimate requests.

The recommendation was to consider restricting arbitrary callbacks by making them localized to the requester themselves. This is related to transaction order dependence aspect discussed in 21 of security pitfalls and best practices 101 and denial of service aspect of 176 and external interaction issues of 180 we discussed in security pitfalls and best practices 201 modules.

7.77 Audit Finding #77

77 is a finding from OpenZeppelin audit of UMA where it was a medium severity finding related to auditing and logging, in which event emission was missing after sensitive actions. The get latest funding rate function of the funding rate applied contract did not emit relevant events after executing the sensitive actions of setting the funding rate update time and proposal time and transferring the reports.

The recommendation was to consider emitting events after sensitive changes take place to facilitate tracking and notifying off-chain clients following the contracts activity. This is related to the missing events aspen discussed in 45 of security pitfalls and best practices 101, the broader auditing logging issues of 173 along with the principle of compromise recording of 201 we discussed in security pitfalls and best practices 201 module.

7.78 Audit Finding #78

78 is another finding from OpenZeppelin's audit of UMA where it was a medium severity finding related to specification, in which functions had unexpected side-effects. For example, the get latest funding rate function of the funding rate applied contract might also update the funding rate and send rewards. The get price function of the optimistic Oracle contract might also settle a price request. These side-effect actions were not clear in the name of functions, that is the names sounded like getter functions, but these were also setters and therefore were not expected which could lead to mistakes when the code is modified by new developers who weren't aware of all such project implementation.
The recommendation was to consider splitting such functions into separate getters and setters or alternatively, renaming those functions to describe all the actions that they perform. This is generally related to the programming style and naming conventions discussed in 97 of `Solidity` 101 module and broader system specification documentation and clarity issues of 136, 137 and 188 along with the principle of economy of mechanism of 197 we discussed in security pitfalls and best practices 201 module.

7.79 Audit Finding #79

79 is another finding from OpenZeppelin audit of 1inch where it was a medium severity finding related to denial of service, in which muri swap pairs could not be unpaused. For context, the morisot factory governance contract had a shutdown function that would be used to pause the contract and prevent any future swaps. However, there was no function to unpause the contract. There was also no way for the factory contract to redeploy a muri swap instance for a given pair of tokens. Therefore, if a murisoft contract were ever shut down or passed it would not be possible for that pair of tokens to ever be traded on the

muri swap platform again unless a new factory contract was deployed.

The recommendation was to consider adding unpauseability for murisoft contracts. This is related to OpenZeppelin's possible library we discussed in 156 of `Solidity` 201 module and guarded launch aspects of circuit breaker and emergency shutdown in 134 and 135 along with the broader aspects of denial of service of 176 we discussed in security pitfalls and best practices 201 module.

7.80 Audit Finding #80

80 is a finding from OpenZeppelin's audit to Futureswap V2 where it was a high severity finding related to timing and denial of service, in which attackers could prevent users from performing an instant withdrawal from the wallet contract. An attacker who observed an user's call to message processor instant withdraw in Ethereum's main tool could get the Oracle message and Oracle signature parameters from the user's transaction, then submit their own transaction to instant withdraw using the same parameters, but at a higher Gas Price to front run the user's transaction, but also carefully choosing the Gas Limit for the transaction such that the call would fail with an Out-of-Gas error at a certain point of the contracts flow. The result would be that the attackers instead withdraw would fail, but the user interaction number would have been successfully reserved and as a result the user's transaction would revert because it would be attempting to use a user interaction number that was no longer valid.

The recommendation was to consider adding an access control mechanism to restrict who could submit Oracle messages on behalf of the user. This is related to the transaction order dependence aspect discussed in 21 of security pitfalls and best practices 101 module and access control aspects of 148 149 and 172 along with derivative service aspects of 176 that we discussed in security pitfalls and best practices 201.

7.81 Audit Finding #81

81 is another finding from OpenZeppelin's audit of Futureswap V2 where it was a medium severity finding related to configuration, in which the code was not using a (great?) safe contract in fs token inheritance. For context, the fs token contract was intended to be an upgradable contract used behind a Proxy, however the contract's `ERC20snapshot`, `ERC20mintable` and `ERC20burnable` inherited from fs token were not imported from the upgrade safe OpenZeppelin library, but instead from their regular not upgrade safe counterparts that used constructors instead of initialized functions.

The recommendation was to use the upgrade safe libraries. This is related to OpenZeppelin's OZ Initializable library in 192 and other Proxy related aspects

we discussed in `Solidity` 201 module, the aspect of importing upgradable contracts in Proxy-based upgradable contracts of 97 of security pitfalls and best practices 101 module along with the broader aspects of configuration in 165 and initialization in 166 that we discussed in security pitfalls and best practices.

7.82 Audit Finding #82

82 is another finding from OpenZeppelin's audit of Futureswap V2 where it was a medium severity finding related to error handling, in which the output of the ECDSArecover function was unchecked. Remember that the ECDSArecover function from OpenZeppelin returns a Zero-address if the signature provided is invalid. This function was used twice in the Futureswap code. Once to recover an Oracle address from an Oracle signature and again to recover the user's address from their signature. If the Oracle signature were invalid, the Oracle address would be set to Zero-address and similarly, if the user signature were invalid, then the user message signer or the withdrawer would be set to a Zero-address. Either could result in unintended behavior.

The recommendation was to consider reverting if the output of ECDSA recover is never a Zero-address. This is related to OpenZeppelin's ECDSA library we discussed in 166 of `Solidity` 201 module missing Zero-address validation in 49 of security pitfalls and best practices 101 module along with the broader aspects of cryptography issues in 174 and error reporting in 175 that we discussed in security pitfalls and best practices 201.

7.83 Audit Finding #83

83 is a finding from OpenZeppelin's audit of Notional where it was a medium severity finding related to configuration, in which adding new variables to multi-level inherited upgradable contracts would break the storage layout. The Notional protocol used the open separate contracts to manage upgradability with the unstructured storage pattern. When using that upgradability approach and when working with multi-level inheritance, if a new variable were to be introduced in a parent contract that could potentially override the beginning of the storage layout of the child contract causing critical misbehavior in the system.

The recommendation was to consider preventing such scenarios by defining a storage gap in each upgradable parent contract at the end of all the storage variable definitions for future variable additions. In such an implementation, the size of the gap would be intentionally decreased each time a new variable was introduced thereby avoiding the overwriting of pre-existing storage values. This is related to the various open supplement Proxy aspects we discussed in 185 to 192 or `Solidity` 201 module state variables in Proxy-based upgradable

contracts and 99 of security pitfalls and press practices 101 module and broader aspects of configuration issues and 165 that we discussed in security pitfalls and best practices 201.

7.84 Audit Finding #84

84 is a finding from OpenZeppelin's audit of GEB where it was a medium severity finding related to data validation in which unsafe division was performed in `rdivide` and `wdivide` functions. For context, the functions `rdivide`, `wdivide` accepted the divisor `y` as an input parameter without checking if the value of `y` was zero. If that were to be the case, the call would revert due to division by 0.

The recommendation was to consider adding a `require` statement in the functions to ensure that `y` is greater than 0 or considered using the div functions provided in OpenZeppelin's safe map libraries. This is related to OpenZeppelin safe map library we discussed in 175 accelerating 201 module and function parameters in 138 along with the broader aspects of data validation issues in 169 that we discussed in security pitfalls and best practices 201 module.

7.85 Audit Finding #85

85 is another finding from OpenZeppelin's audit of 1inch where it was a medium severity finding related to timing, in which `safeApprove` was used incorrectly. The `safeApprove` function of OpenZeppelin `safeERC20` library prevents changing an allowance between non-zero values to mitigate a possible Front-running attack. Instead, the `safeIncreaseAllowance` and `safeDecreaseAllowance` functions should be used. However, the `uniERC20` library simply bypassed this restriction by first setting the allowance to zero that reintroduced the Front-running attack and undermined the value of using the `safeApprove` function.

The recommendation was to instead use `safeIncrease` allowance and `safeDecreaseAllowance` functions. This is related to OpenZeppelin `safeERC20` library we discussed in 149 of `Solidity` 201 module transaction order dependence in 21 and `ERC20` approved Race-condition in 22 of security pitfalls and best practices 101 module along with the broader aspects of timing issues in 177 that we discussed in security pitfalls and best practices 201.

7.86 Audit Finding #86

86 is a finding from OpenZeppelin's audit of Opyn Gamma where it was a high severity finding related to denial of service, in which Ether could get trapped in the protocol if a user sent more than the necessary Eth for a batch of actions in a specific context of the protocol. The remaining Eth was not returned to

the user and would get locked in the contract due to the lack of a withdrawal function.

The recommendation was to consider either returning all the remaining Eth to the user or creating a function that allowed the user to collect the remaining Eth. This is related to locked Ether in 29 of security pitfalls and best practices 101 module along with the broader aspects of numerical issues in 170 accounting issues in 171 and denial of service issues in 176 that we discussed in security pitfalls and best practices 201 module.

7.87 Audit Finding #87

87 is another finding from OpenZeppelin's audit of Opyn Gamma where it was a medium severity finding, again related to denial of service, in which the use of `Solidity`'s `transfer` primitive might render impossible. For context, when withdrawing Eth deposits the payable Proxy controller contract use `Solidity`'s `transfer` function which could fail. With a withdrawal smart contract, if you did not implement a payable fallback function or the payable fallback function uses more than 2300 Gas units for some reason.

The recommendation was to instead use send value function available in OpenZeppelin's address library, which can be used to transfer the Ether without being limited to 2300 Gas units and address any reentrancy risk from the use of this function by following the check effects, interactions, pattern and using OpenZeppelin's reentrancy guard line. This is related to receive and fallback functions in 33 and 34 and transfer function in 47 absolutely 101 module OpenZeppelin's address library in 159 absolutely 201 module avoid transfers sent as reiterancy mitigations in 15 and fallback versus received in 27 of security pitfalls and best practices 101 module along with the broader aspects of denial of service issues in 176 that we discussed in security pitfalls and best practices 201.

7.88 Audit Finding #88

88 is a finding from OpenZeppelin's audit of Endaoment where it was a high severity finding related to timing, in which a reentrancy vulnerability was present because of not following the checks effects interactions. For context, the finalized `grant` function of the fund contract was setting the `grantCompleteStorage` variable after a token transfer and could therefore be used to conduct a reentrancy attack leading to the contract funds being traded.

The recommendation was to always follow the check effects interactions pattern where the contract stay is modified before making any external call to other contracts and use reentrancy guards for such functions. This is related to OpenZeppelin's reentrancy guard library we discussed in 157 of `Solidity`

201 module re-entrance vulnerabilities in 13 of security pitfalls and best practices 101 module along with the broader aspects of timing issues in 177 that we discussed in security pitfalls and best practices 201.

7.89 Audit Finding #89

89 is a finding from OpenZeppelin's audit of Audius where it was a high severity finding related to auditing and logging in which no events were emitted while updating the governance registry and guardian addresses. For context, in the governance contract, the registry address, the guardian address are highly sensitive accounts which could be updated by calling set registry address and transfer guardianship respectively. However, these two functions did not emit any events because of which stakers who monitor the audio system would have to expect all transactions to detect, if any address they trusted was being replaced with an untrusted one instead of simply subscribing to events emitted.

The recommendation therefore was to consider emitting events when these addresses were updated to make it more transparent and easier for clients to subscribe to the events when they want to keep track of the status of the system. This is related to missing events in 45 of security pitfalls and best practices one-on-one module along with broader aspects of auditing logging issues in 173 and principle of compromise recording in 201 that we discussed in security pitfalls and best practices 201.

7.90 Audit Finding #90

90 is another finding from OpenZeppelin's audit of Audius where it was a high severity finding related to identification, in which the quorum requirement could be bypassed with civil account. For context, while the final vote on a proposal was determined via a token weighted vote, the quorum check in evaluate proposal outcome function could be bypassed by splitting one's tokens over multiple accounts and voting with each of those accounts. Each of those civil wars increased the proposal's numbers variable which meant that anyone could make the quorum check pass.

The recommendation was to consider measuring size by the percentage of existing tokens that have voted rather than the number of unique accounts that have loaded this is related to the broad aspects of token handling in 159 accounting in 171 and access control in 172 that we discussed in security pitfalls and best practices 201.

7.91 Audit Finding #91

91 is another finding from OpenZeppelin's audit of Audius where it was a medium severity finding related to timing from inconsistently checking initialization for context when a contract was initialized its initialized state variable was set to true and because interacting with uninitialized contracts would cause problems, the `requireIsInitialized` function was available to perform the step. However, this check was not used consistently. So for example, it was used in the get voting code function of the governance contract, but not used in the get registry address function of the same contract this could be misleading and cause uninitialized contacts to be called.

The recommendation was to consider calling `requireIsInitialized` consistently in all the functions of the contracts and, if there were a reason to not call it in some functions, consider documenting that or alternatively consider removing this check all together and preparing a deployment script that would ensure that all contracts were initialized in the same transaction that they were being deployed. This is related to the broad aspects of initialization issues in 166 and also the timing and ordering issues in 177 and 178 that we discussed in the security pitfalls at best practices 201 module.

7.92 Audit Finding #92

92 is another finding from OpenZeppelin's audit of Audius where it was a medium severity finding related to data validation in which the voting period and quorum could be set to zero. For context, when the governance contract was initialized, the values of voting period and voting quorum were checked to make sure that they were greater than zero. However, the corresponding sender functions, set voting period and set voting forum allow these variables to be reset to zero. Setting the voting period to zero would allow spurious proposals that can't be voted upon and setting the quorum to zero would allow proposals with zero words to be executed, which is very dangerous as you can imagine.

The recommendation was to consider adding validation to the setter functions this is related to function parameter validation in 138 function invocation arguments in 146 along with the broader aspects of data validation in 169 that we discussed in security pitfalls and best practices 201 module.

7.93 Audit Finding #93

93 is another finding from OpenZeppelin's audit of Audius where it was a medium severity finding related to configuration in which some state variables were not set during initialization. For context, the Audius contracts could be upgraded using the unstructured storage Proxy pattern, which requires the use of an initializer instead of the constructor to set the initial values of the state

variables. In some of the contracts the initializer was not initializing all the state variables.

The recommendation therefore was to consider setting all the required variables in the initializer and, if there were a reason for leaving them uninitialized, consider documenting that and adding checks on the functions that use those variables to ensure that they were not called before initialization. This is related to the various OpenZeppelin Proxy aspects we discussed in 185 to 192 of `Solidity` 201 module initializing state variables in Proxy-based upgradable contracts in 96 for security pitfalls and best practices 101 module and broader aspects of configuration issues in 165 that we discussed in security pitfalls and best practices 201.

7.94 Audit Finding #94

94 is a finding from OpenZeppelin's audit of Primitive where it was a medium severity finding related to timing, in which expired and/or paused options could still be traded. For context, option tokens could still be freely transferred when the option contract was either paused or expired or both. That would allow malicious option holders to sell paused or expired options that could not be exercised in the open market to exchanges, and users who did not take the necessary precautions before buying an option minted by the primitive protocol. The recommendation was to consider implementing the necessary logic in the auction contract to prevent such transfers of tokens during pause or after expiration or alternatively, if the described behavior was indeed intended to consider clearly documenting it to raise awareness among the option sellers and buyers this is generally related to OpenZeppelins possible in 156 absolutely 201 and broad aspects of timing and ordering issues in 177 178 system documentation and clarity issues in 137 and 188 and business logic issues in 191 that we discussed in security pitfalls and best practices 201 modules.

7.95 Audit Finding #95

Number 95 is a finding from OpenZeppelin's audit of AC0 protocol where it was a medium severity finding related to data validation, in which `ERC20` transfers could misbehave. For context, the `transferFromERC20` function was used throughout the AC0 token contract to handle transferring funds into the contract from a user. It was called within `mint`/`mintTo` and validate and burn functions where in each case the destination was the AC0 total contract. Such transfers may behave unexpectedly: if the external `ERC20` token contract charged fees, as an example, the popular USDT token does not presently charge any fees upon transfer, but it has a functionality to do so. in that case the amount received would be less than the amount sent. such deflationary tokens have the potential to lead to protocol insolvency when they are used to mint

new AC0 tokens. In the case of transfer `ERC20` similar issues could occur and cause users to receive less than expected when the collateral was transferred or when exercise assets were transferred.

The recommendation was to consider betting each token used within an AC0 options pair ensuring that failing transfer from and transfer calls would cause rewards with an AC0 token contract and additionally consider implementing some sort of sanity check which enforced that the balance of the AC0 token contract increases by the desired amount when calling transfers from `ERC20` this is related to token deflation vr fees in 107. guarded launch via asset types and 129 and broader aspects of token handling in 159 and data validation issues in 169 that we discussed in security pitfalls and best practices 201 module.

7.96 Audit Finding #96

96 is a finding from OpenZeppelin's audit of Compound protocol where it was a medium severity finding related to auditing and logging, in which there was incorrect event emission. For context, the Uniswap window update event of the Uniswap anchored new contract was being emitted in the poke window values function using incorrect values because it was being emitted before the relevant state changes were applied to the old observation and new observation variables and therefore causing the data logged by the event to be outdated.

The recommendation was to consider emitting the Uniswap window update event after changes were applied, so that all log data is up to date this is related to broader aspects of auditing logging issues at 173 ordering in 178 and freshness in 185 that we discussed in security pitfalls and best practices 201.

7.97 Audit Finding #97

97 is a finding from OpenZeppelin's audit of MCDEX Mai protocol where it was a critical severity finding related to access control, in which anyone could liquidate on behalf of another account. For context, the perpetual contract had a public liquidate from function that bypassed the checks in the liquidate function, which meant that it could be called to liquidate a position and with any user being able to set an arbitrary `from` address would cause a third party to confiscate an undercollateralized trader's position. So effectively, this meant that any trader could unilaterally rearrange another accounts position and also liquidate on behalf of the perpetual Proxy which could break down the automated market maker invariants.

The recommendation was to consider restricting liquidate `from` to internal visibility from public visibility this is related to aspects of function visibility specifiers in number 23 of `Solidity` 101 in current access control and number four

of security pitfalls and best practices 101 module and aspects of function visibility in 140 along with broader aspects of access control in 148 149 and 172 and trust issues in 181 that we discussed in security pitfalls and best practices 201 modules.

7.98 Audit Finding #98

98 is another finding from OpenZeppelin's audit of MCDEX Mai protocol where it was a critical severity finding related to denial of service, in which orders could not be cancelled because when a user or broker called cancel order, the cancel mapping was updated, but that had no subsequent effects because validate order param did not check, if the order had been cancelled.

The recommendation was to consider adding that check to order validation to ensure that cancelled orders would not be filled. This is related to broader aspects of data validation in 169 and denial of service issues in 176 that we discussed in security pitfalls and test practices 201 module.

7.99 Audit Finding #99

99 is another finding from OpenZeppelin's audit of MCDEX Mai protocol where it was a medium severity finding related to re-entrancing. For context, there were several examples of interactions preceding effects in the context of the checks effects interactions pattern or CEI pattern that we have discussed. The first example was in the deposit function of the collateral contract. The collateral was retrieved before the user balance was updated and an event was emitted. Also, in the withdrawal function of the collateral contract collateral was sent before the event was emitted. Finally, the same pattern occurred in deposit to insurance fund deposit Ether to insurance fund and withdraw from insurance fund functions of the perpetual contract. These reentrancy opportunities would affect the order and contents of emitted events which could confuse external clients about the state of the system.

The recommendation therefore was to consider always following the check effects interactions pattern or use reiteracy guard contract to protect those functions. This is related to OpenZeppelin's reentrancy guard library we discussed in 157 of `Solidity` 201 model reentrancy vulnerabilities and number 13 of security pitfalls and best practices one-on-one module along with broader aspects of auditing logging in 173 and timing issues in 177 that we discussed in security pitfalls and best practices 201 module.

7.100 Audit Finding #100

100 is another finding from OpenZeppelin's audit of MCDEX Mai protocol where it was a medium severity finding related to timing, in which governance parameter changes were enforced instantly. For context, many sensitive changes could be made via the function set governance parameter, such as the initial and maintenance margin rates or the lot size parameters. These new parameters would instantly take effect in the protocol, with important effects on protocol users some of which could be perceived as being as negative impacts.

The recommendation was to consider implementing a timelock mechanism for such changes to take place because by enforcing a delay between the signal of intent, the actual change users would have time to decide to continue engagement with the protocol or exit their positions as necessary. This is related to OpenZeppelin's TimelockController library which is this in 182 of `Solidity` 201 module and tight delay change of critical parameters in 163 along with broader aspects of timing in 177 and trust issues in 181 that we discussed in security pitfalls and best practices 201.

7.101 Audit Finding #101

101 is another finding from OpenZeppelin's audit of MCDEX Mai protocol where it was a high severity finding related to data validation, in which words could be duplicated. For context, the data verification mechanism used a commit reveals key to hide words during the voting period, where the intention was to prevent voters from simply voting with the majority. However, the design allowed voters to blindly copy each other's submissions because words were not cryptographically tied to the voter and so, undermined the objective of the comment reveals.

The recommendation was to consider including the voter address within the commitment to prevent votes from being duplicated and also consider including the relevant time stamp price identifier and round id as well to limit the applicability and reusability of commitment. This is related to broad aspects of data validation in 169 accounting in 171 cryptography in 174 and business logic issues in 191 that we discussed in security pitfalls and best practices.

Chapter 8

Audit Findings 201

Welcome to the eighth and final module of the securium bootcamp this module is about audit findings 201 where we will review 100 more findings from public audit reports of leading audit firms to get a sense for the kinds of issues reported during audits and their suggested fixes or recommendations the civility range of these 100 findings spans a spectrum, but mainly focuses on medium to low severity informational best practice guidelines and some of the more application level business logic and software engineering aspects to get critical perspectives different from those in the previous module to help level set expectations again we will only be able to touch upon key aspects of these findings in this video presentation and hope to discuss any particular ones of interest in other forums of the bootcamp the reason is that these findings `require` a lot of context from the deepest details of the protocol implementations which will certainly not have the time to do, so for all hundred findings this will need to be done by interested bootcamp participants in their own time by reviewing the audit reports and their corresponding protocol code bases to whatever depth possible I would strongly encourage everyone to do, so and at least read the audit reports covered here, if not the actual code basis of the audited projects themselves for each finding we will review the vulnerability category its finding summary, the proposed recommendation while relating some of these aspects to our learnings from the earlier models, so with that let's get started.

8.1 Audit Finding #102

One or two is a finding from ConsenSys audit of Umbra, related to access control and input validation in which potential edge cases for hook receiver contracts were not documented or validated.
For context there were very few constraints on arguments to certain external calls in the emperor contract. Anyone could force a call to a hook contract by transferring a small amount of tokens to an address that they controlled and withdrawing those tokens passing the target address as the hook receiver.

The recommendation was for the developers to document and validate such `external` function calls and untrusted parameters for potential edge cases.
This is related to validation of function parameter arguments in 138 and 139 and broad aspects of access control specification in 148 and tests in 155 that we discussed in security pitfalls and best practices 201 module.

8.2 Audit Finding #103

103 is another finding from ConsenSys audit of Umbra related to specification and documentation of token behavior restrictions.
For context as with any protocol that interacts with arbitrary `ERC20` tokens it is important to clearly document which tokens are supported.
This is best done by providing a specification for the behavior of the expected `ERC20` tokens and only relaxing the specification after careful review of a particular class of tokens and their interactions with the protocol.
The recommendation was that node deviations from normal `ERC20` behavior should be explicitly noted as not supported by Umbra protocol such as one deflationary or fee on transferred tokens.
These are tokens in which the balance of the recipient of a transfer may not be increased by the amount of the transfer there may also be some alternative mechanism by which balances are unexpectedly decreased.
While these tokens can be successfully sent the internal accounting of Umbra contract will be out of sync with the balance as recorded in the token contract resulting in loss of funds.
Second, inflationary tokens the opposite of deflationary tokens where Umbra contract provided no mechanism for claiming positive balance adjustments for such tokens.
Third, rebasing tokens a corporation of the above two cases of deflationary and inflationary tokens rebasing tokens are tokens in which an account's balance increases or decreases along with expansions or contractions in their supplies.
The contract provided no mechanism to obtain its internal accounting in response to these unexpected balance adjustments and funds may be lost as a result.
This is related to token deflation via fees in 107, total inflation via interest in 108, garden launch via asset types and 129, and broader aspects of token handling in 159, system specification and documentation in 136, 137 and accounting issues in 171, that we discussed in security pitfalls and best practices 201 model.

8.3 Audit Finding #104

104 is a finding from ConsenSys audit of DEFI Saver related to testing, the test suite was not complete and many of the tests failed to `execute`.
For complicated systems such as DEFI Saver which uses many different modules and interacts with different DEFI protocols it is crucial to have a full test

coverage that includes edge cases and `fail` scenarios which is critical for safer development and future upgrades.
As seen in some smart contact incidents a complete test suite could have prevented issues that might be hard to find with manual reviews.
So the recommendation was to add a full coverage test suite.
This is related to the broad aspect of testing in 155 that we discussed in security pitfalls and best practices 201 modules.

8.4 Audit Finding #105

105 is another finding from ConsenSys audit of DEFI Saver related to naming documentation and refactoring, where hyper get rates code was unclear because function names did not reflect their true functionalities, the code used some undocumented assumptions as well.
The recommendation was to refactor the code to separate getting rate functionality with get sell rate and get pi rate and also to explicitly document any assumptions in the code.
This is related to broad aspects of system documentation in 137 and clarity in 188 that we discussed in security pitfalls and best practices to one module.

8.5 Audit Finding #106

106 is another finding from ConsenSys audit of DEFI Saver related to error checking, where `return` value was not used for token utils withdraw tokens.
For context the `return` value of tokens withdraw tokens which represented the actual amount of tokens that were transferred was never used in the entire repository. This could cause discrepancy in the case where the requested transfer amount was uint256 max for some reason, in which case the amount actually transferred would be less than that requested and returned back, but never checked.
The recommendation therefore was to check the `return` value to validate the withdrawal and use that in the event committed.
This is related to function `return` values in 142 and accounting issues in 171 that we discussed in security pitfalls and best practices to one module.

8.6 Audit Finding #107

107 is another finding from ConsenSys auditor of Umbra related to access control and logging where there was missing access control for DEFISaverLogger.log, which was used as a logging aggregator within the system, but anyone could create logs the.
Recommendation was to add access control to all functions appropriately.

This is related to broad aspects of access control implementation in 149 and auditing and logging in 173 that we discussed in security pitfalls and best practices to one module.

8.7 Audit Finding #108

108 is a finding from ConsenSys audit of DAOfi related to documentation, where stale comments were present in the code base about storage slots.
The recommendation was to remove such stale compounds.
This is related to comments in 154 redundant constructs in 157 and broad aspects of system documentation in 137 that we discussed in security pitfalls and best practices to one module.

8.8 Audit Finding #109

109 is a finding from ConsenSys audit of `MStable` related to documentation, where there was a mismatch between what the code implemented and what the corresponding comment described.
The recommendation was to update the code or the comment to be consistent.
This is related to comments in 154 and broad aspects of system documentation in 137 and clarity issues in 188 that we discussed in security pitfalls and best practices 201 module.

8.9 Audit Finding #110

110 is another finding from ConsenSys audit of DAOfi related to unnecessary code or logic where there was an unnecessary call to DAOfiV1 factory formula function for context few to DAOfiV1 pair functions used a `external` function, which made a call to the factory to retrieve the immutable formula address set in the constructor. Instead such calls could simply be replaced with that immutable value.
The recommendation was therefore to remove such unnecessary calls and replace them with variable rates.
This is related to broad aspects of redundant constructs in 157, the principle of economy of mechanism in 197 that we discussed in security pitfalls and best practices 201 module.

8.10 Audit Finding #111

111 is another finding from ConsenSys audit of DAOfi related to testing, where increased testing of edge cases in complex mathematical operations, could have identified at least one issue raised in the report.

The recommendation was additional unit tests as well as Fuzzing or property based testing of curve related operations for more validation of mathematical operations.
This is related to the broad aspect of testing in 155 and numerical issues in 170 that we discussed in security pitfalls and best practices to one module.

8.11 Audit Finding #112

112 is a finding from ConsenSys audit of the fake protocol related to application logic where governor alpha proposals could be cancelled by the proposer, even after they had been accepted and queued.
For context governor alpha allows proposals to be cancelled via council, but a proposal could cancel proposals in any of pending active cancelled defeated succeeded queued or expired states.
The recommendation was to prevent proposals from being cancelled unless they were in pending or active states.
This is related to function invocation timeliness and order in 143 and 145, the broad aspects of ordering in 178 and business logic in 191 that we discussed in security pitfalls and best practices to one module.

8.12 Audit Finding #113

113 is a finding from ConsenSys audit of eRLC related to access control and timing.
For context the KYC admin had the ability to freeze the funds of any user at any time by revoking the KYC member role, the trust requirements from users could be slightly decreased by implementing a delay on granting this ability to new addresses such a delay could also help protect the development team, the system itself in the event of a private key compromise of the KYC ad.
The recommendation therefore was to use a TimelockController as the KYC default admin of the eRLC contract.
This is related to OpenZeppelin's `Timelock` controller library we discussed in 182 of `Solidity` 201 module and time delay change of critical parameters in 163, along with broader aspects of access control changes in 153, timing in 177 and trust issues in 181, that we discussed in security pitfalls and best practices 201 module.

8.13 Audit Finding #114

114 is a finding from ConsenSys Audit of 1inch related to documentation and testing, where the source code hardly contained any inline documentation which made it hard to reason about functions and how they were supposed to be used. Additionally the test coverage seemed to be limited whereas especially for a public facing exchange contract system test coverage should have been extensive

covering all functions especially those that could be directly accessed including potentially security relevant and edge cases. This would have helped in detecting some of the findings raised with the report.
The recommendations were to consider adding NATSpec format compliant inline code documentation, describe functions what they were used for and who was supposed to interact with them to document specific assumptions and to increase test coverage.
This is related to system documentation in 137 comments in 154 and broad aspects of testing in 155 and clarity issues in 188 that we discussed in security pitfalls and best practices 201 modules.

8.14 Audit Finding #115

115 is another finding from ConsenSys audit of 1inch related to configuration, where the compiler version `pragma` was unspecific in that it was floating or unlocked with caret `^0.6.0`.
Which that often makes sense for libraries to allow them to be included with multiple different versions of an application it may be a security risk for the application implementation itself a known vulnerable compiler version may accidentally be selected for deployment or security tools might fall back to an older compiler version ending up actually checking a different version from what is ultimately deployed in the blockchain.
The recommendation was to avoid floating parameters and pin a concrete compiler version, the latest without known security issues in at least the top level deployed contracts to make it unambiguous as to which compiler version was being used. The suggested rule of thumb was that a flattened source unit should have at least one non-floating concrete `Solidity` compiler version.
This is related to unlocked `pragma` in number two of security pitfalls and best practices 101 module and broader aspects of tests in 155 of security pitfalls and best practices 201 model.

8.15 Audit Finding #116

116 is another finding from ConsenSys audit of 1inch related to denial of service (DoS), where hard-coded Gas assumptions were pointed out as being problematic because Gas economics and Ethereum have changed in the past and made change again like with the recent EIP 1559 potentially rendering the contract system unusable in the future.
Recommendation was to be aware of this potential limitation and be prepared by documenting and validating for situations where Gas prices might change in a way that negatively affected the contracts.
This is related to Gas impact on denial of service in 42, 43 and 44 of security pitfalls and best practices 101 module and broader aspects of Gas issues in 182 of security pitfalls and best practices to a one module.

8.16 Audit Finding #117

117 is another finding from ConsenSys audit of 1inch related to access control and input data validation where it was a critical vulnerability, in which anyone could steal all the funds that belong to the referral fee receiver.
For context any token or ETH that belonged to the referee receiver was at risk and could be drained by any user by providing a custom Mooniswap pool contract that referenced existing token holdings because none of the functions in the refrigerate receiver verified that the user provided Mooniswap pool address was actually deployed by the linked Mooniswap factory.
The recommendations were:

1. to enforce that the user provided money swap contract was actually deployed by the link factory because other contracts can't be trusted
2. consider implementing token sorting and deduplication in the pool contract constructor as well.
3. consider employing a re-entrancy card to safeguard the contract from the ancestry attacks
4. improve testing because the vulnerable functions were not covered at all and
5. improve documentation and provide a specification that outlined how this contract was supposed to be used.

This is related to system specification and documentation in 136, 137 access control specification and implementation in 148, 149 and broader aspects of testing in 155 data validation issues in 169 and access control issues in 172 that we discussed in security pitfalls and best practices 201 model.

8.17 Audit Finding #118

180 is another finding from ConsenSys audit of 1inch related to privileged roles and timing, where there could be unpredictable behavior for users due to admin Front-running or in general.
For context administrators of contracts could update or upgrade things in the system without warning which could violate security goals of the system. Specifically privileged roles could use Front-running to make malicious changes just ahead of incoming transactions or purely accidental negative effects that could occur due to unfortunate timing of such changes.
The recommendation was to give users advanced notice of changes with the time lock, for example by making all system parameter and upgrades to `require` two steps, with a mandatory time window between them.
The first step would broadcast to users that a particular change was coming.

The second step would, then commit that change after a suitable waiting period this would allow users that do not want to accept such change to exit and others who are okay with those changes to continue engaging with the protocol.
This is related to OpenZeppelin's TimelockController library we discussed in 182 or `Solidity` 201 module and two-step change of privileged roles in 162 time delay change of critical parameters and 163 along with broader aspects of timing in 177 and trust issues in 181 that we discussed in security pitfalls and best practices 201 model.

8.18 Audit Finding #119

119 is the finding from ConsenSys audit of Growth DEFI related to specification and documentation, where the only documentation for growth DEFI was a single README file as well as some code comments,
A system's design specification and supporting documentation should be as important as the system's implementation itself because users rely on high level documentation to understand the big picture of how a system works.
Without spending time and effort to create such documentation a user's only resource is the code itself something the vast majority of users can't understand.
Security assessments depend on a complete technical specification to understand the specifics of how a system works when a behavior is not specified or is specified incorrectly security assessments must base their knowledge in assumptions leading to less effective review.
Also maintaining an updating code relies on supporting documentation to know why the system is implemented in a specific way, if code maintainers can't reference documentation they must rely on memory or assistance to make high quality changes.
The recommendation therefore was to improve system documentation and create a complete technical specification.
This is related to broad aspects of system specification and documentation in 136 and 137 undefined behavior issues in 179 and clarity issues in 188 that we discussed in security pitfalls and best practices 201 modules.

8.19 Audit Finding #120

120 is another finding from ConsenSys audit of Growth DEFI related to access control and lease privilege mechanism, where system states roles and permissions were not sufficiently restricted.
Smart contract code should strive to be strict where it behaves predictably is easier to maintain and increases the system's ability to handle non-ideal conditions. Whereas many of Growth DEFI states roles and permissions were loosely defined.
The recommendation was to document and monitor the use of administrative permissions and also specify strict operational requirements for each contract

as it pertains to roles and permissions.
This is related to access control specification implementation and changes in 148, 149 and 153 and broader aspects of access control issues in 172 and principle of least privilege in 192 that we discussed in security pitfalls and best practices to one module.

8.20 Audit Finding #121

121 is another finding from ConsenSys audit of Growth DEFI related to specification and access control where the concern was about tokens with non-standard behavior, such as `ERC777` callbacks which would enable an attacker to `execute` potentially arbitrary code during the transaction or inflationary deflationary and rebasing tokens.
The recommendation was to evaluate all tokens prior to inclusion in the system.
This is related to token deflation via fees in 107 token inflation we are interest in 108 garden launch we asset types in 129 and broader aspects of token handling in 159 system specification and documentation in 136 and 137 and accounting issues in 171 that we discussed in security pitfalls and best practices 201.

8.21 Audit Finding #122

122 is another finding from ConsenSys audit of Growth DEFI related to naming convention and readability in which the code base made use of many different contracts, abstract contracts, interfaces and libraries for inheritance and code reuse. Which is in principle a good practice to avoid repeated use of similar code, but with no descriptive naming conventions to indicate which files would contain meaningful logic the code pairs became difficult to navigate.
The recommendation was to use descriptive names for contracts and libraries.
This is related to broad aspects of programming style code layout and any convention in number 97 to 101 of `Solidity` 101 module and clarity issues in 188 principle of economy of mechanism at 197 and principle of psychological acceptability in 199 that we discussed in security pitfalls and best practices 201.

8.22 Audit Finding #123

123 is another finding from ConsenSys audit of Growth DEFI related to initialization and timing, in which many contracts allowed users to deposit or withdraw assets before the contracts were completely initialized, or while they were in a semi-configured state.
Because these contracts allowed interaction on semi-configured states the number of configurations possible when interacting with the system made it very difficult to determine whether the contracts behaved as expected in every scenario or even what behavior was expected from them the first place.

The recommendation was to prevent contract from being used before they were entirely initialized.
This is related to broad aspects of initialization issues in 166 and also the timing and ordering issues in 177 and 178 that we discussed in security pitfalls and best practices to one module.

8.23 Audit Finding #124

124 is another finding from ConsenSys audit of Growth DEFI related to denial of service (DoS) in which there was a potential for resource exhaustion by external calls performed within an unbounded loop.
For context request flash flow performed external calls in a potentially unbounded loop and in the worst case changes to system state could make it impossible to `execute` that code due to the block Gas the limit
The recommendation was to reconsider that logic or bound the loop.
This is related to the Gas impact on denial of service in 42, 43 and 44 of security pitfalls and best practices 101 module and broader aspects of denial of service in 176 and Gas issues in 182 of security pitfalls and best practices to one module.

8.24 Audit Finding #125

125 is a finding from ConsenSys audit of Paxos related to stale privileges and access control in which old owners could never be removed.
For context the intention of set owners was to replace the current set of owners with a new set of owners. However the is Owner mapping was never updated, which meant that any address that was ever considered an Owner was permanently considered as an Owner for purposes of signing transactions.
The recommendation was to change set owners such that before adding new owners it would loop through the current set of honors and clear, there is Owner `booleans`.
This is related to aspects of access control implementation and changes in 149 and 153 and broad aspects of access control issues in 172 that we discussed in security pitfalls and best practices 201 module.

8.25 Audit Finding #126

126 is a finding from ConsenSys audit of Aave V2 related to potential manipulation of interest rates using flash loans. Remember that flash loans allow users to borrow large amounts of liquidity from the protocol because of which it was possible in Aave to adjust the stable interest rate up or down by momentarily removing or adding large amounts of liquidity to reserves. Given that this type of manipulation is difficult to prevent especially when flash loans are available.
The recommendation was for Harvey to monitor the protocol at all times to make sure that interest rates were being rebalanced to same values.

This is related to aspects of flash loans in 120 and Scarcity in 186 and auditing and logging issues in 173 of security pitfalls and best practices 201 module.

8.26 Audit Finding #127

127 is a finding from ConsenSys audit of Aave governance down related to validation in which the concern was that because some protocol functionality relied on correct token behavior problems could arise. If a malicious token were to be white listed because it could block people from voting with that specific token or gain unfair advantage, if the balance could be manipulated, so.
The recommendation was to make sure to audit any new whitelisted asset.
This is related to aspects of carded launch via composability limit in 132 token handling in 159 external interactions in 180 and dependency issues in 183 of security pitfalls and best practices 201.

8.27 Audit Finding #128

128 is a finding from ConsenSys audit of Aave CPM related to specification and validation, where there was a risk of integer underflow, if token decimals were to be greater than 18 because in latest answer function an assumption was made that token decimals was less than 18
The recommendation was to add a simple check to the constructor to ensure that the added token had 18 decimals or less.
This is related to dangers of integer overflow underflow we discussed in number 19 of security pitfalls and best practices 101 module and system specification and documentation in 136, 137 and broader aspects of data validation in 169 and numerical issues in 170 of security pitfalls and best practices 201 body.

8.28 Audit Finding #129

129 is another finding from ConsenSys audit of Aave CPM related to testing of chain links performance at times of price volatility where.
The recommendation was that in order to understand the risk of changing Oracle deviating significantly it would help to compare historical prices on Chainlink, when prices were moving rapidly and see what the largest historical delta was compared to the live price on a large exchange.
This is related to the broad aspect of testing in 155 external interaction in 180 and freshness issues in 185 that we discussed in security pitfalls and best practices 201 module.

8.29 Audit Finding #130

130 is a finding from ConsenSys audit of Lien protocol related to configuration, where the concern was that the system had many components with complex functionality leading to a high attack surface, but no operand upgrade path, if any vulnerabilities were to be discovered after launch.
The recommendation was to identify which components were crucial for a minimum viable system, to focus efforts on ensuring the security of those components first, then moving on to others. Also to have a method for pausing and upgrading the system at least at the early phases of the project.
This is related to the various guarded launch approaches in 128 to 135, the broader principles of economy of mechanism and work factor in 197 and 200 of security pitfalls and best practices 201.

8.30 Audit Finding #131

131 is a finding from ConsenSys audit or Balancer protocol related to code factoring, where it is generally considered Error-prone to have repeated checks across the code base and therefore it was recommended to use modifiers for common checks within different functions, because that would result in less code duplication and increased readability.
This is related to function modifiers in 141 and broader aspects of clarity in 188 and cloning issues in 190 of security pitfalls and best practices 201 module.

8.31 Audit Finding #132

132 is another finding from ConsenSys audit of Balancer protocol related to ordering, where `BPool` functions used modifiers `_logs` and `_lock` in that order. Because `_lock` is a reentrancy guard it should have taken precedence over `_logs` in that order to prevent `_logs` from executing first before checking for re-entrancing.
The recommendation was to place `_lock` before other modifiers to ensure that it was the very first and very last thing to run when a function was called because we call that the order of execution is from left to right for modifiers.
This is related to function modifiers and 141 incorrectly used modifiers in 152 and broader aspects of ordering in 178 and business logic issues in 191 of security pitfalls and best practices 201 module.

8.32 Audit Finding #133

133 is a finding from ConsenSys audit of MCDEX V2 where the concern was code based fragility. Software is considered fragile when issues or changes in one part of the system can have Side-effects in conceptually unrelated parts of

the code base. Fragile software tends to break easily and may be challenging to maintain.
The recommendation was to prioritize two concepts: one follow the single responsibility principle for functions where one function does exactly one thing and nothing else and two reduce reliance on external systems.
This is related to broad aspects of external interactions in 180 dependency 183 clarity in 188 and principle of economy of mechanism in 197 of security pitfalls and best practices 201 module.

8.33 Audit Finding #134

134 is a finding from Trail of Bits audit of Liquidity protocol where the concern was that reentrancy could lead to incorrect order of emitted events. Because there were events emitted after external calls in some functions.
In the case of a reentrant call such events would be emitted in the incorrect order that's the event for the second operation would be emitted first followed by the event for the first operation causing any off-chain monitoring tools to have an inconsistent `view` of on-chain state.
The recommendation was to apply the checks effects interactions pattern and move the event emissions before the external calls to avoid any effects of potential re-entrancing.
This is related to reentrancy vulnerabilities in number 13 of security pitfalls and best practices 101 module along with broader aspects of timing issues in 177 and auditing logging in 173 that we discussed in security pitfalls and best practices 201 modules.

8.34 Audit Finding #135

135 is a finding from Trail of Bits audit of Origin Dollar where the concern was that variable Shadowing in ousd from `ERC20` could result in undefined behavior. For context osd inherited from `ERC20`, but redefined the allowances and total supply private state variables. Because of which accessing those variables could lead to returning different values from what was expected. Note that these were private state variables and, so the one in `ERC20` was not visible in OUSD, the concern was more of developer clarity than the variable scope and visibility.
The recommendation was to remove the shadowed variables in OUSD.
This is related to state variable Shadowing in 106 and 136 of `Solidity` 201 where we saw that `Solidity` version 0.6.0 made state variable Shadowing an error where the same named state variables were visible or accessible in both base and derived classes. This is also related to the broad aspect of clarity in 188 of security pitfalls and best practices 201 modules.

8.35 Audit Finding #136

136 is another finding from Trail of Bits audit of Origin Dollar where the concern was about error handling because world core rebase functions had no return statements.
For context world core rebase functions were declared to return and you end, but lacked a return statement. As a result these functions would always return the default value of 0 and were likely to cause issues for their callers. Third party code relying on the `return` values might therefore not have worked as intended.
The recommendation was therefore to add the missing return statements or remove the return type in those functions, then adjust the documentation as necessary.
This is related to function `return` values in 142 and error reporting issues in 175 of security pitfalls and best practices to one module.

8.36 Audit Finding #137

137 is another finding from Trail of Bits audit of Origin Dollar, where the concern was about multiple contracts missing inheritances. Multiple contracts where the implementations of their interfaces inferred based on their names and implemented functions, but did not inherit from them. This behavior is Error-prone and might lead the implementation did not follow the interface, if the code were to be updated.
The recommendation was to ensure that contracts inherit from their interfaces.
This is related to unused constructs in 156 and undefined behavior issues in 179 of security pitfalls and best practices to one body.

8.37 Audit Finding #138

138 is a finding from Trail of Bits audit of Yield protocol where the concern was that `Solidity` compiler optimizations could be dangerous.
Yield protocol had enabled optional compiler optimizations in `Solidity`, but there have been bugs with security implications related to such optimizations. `Solidity` compiler optimizations are disabled by default therefore it was unclear how many contracts in the wild actually used them and how well they were being tested and exercised.
So the short-term recommendation was to measure Gas savings from optimizations and evaluate the trade-offs against the possibility of an optimization related bug. And in the long term monitor the development and adoption of `Solidity` compiler optimizations to assess their maturity.
This is generally related to `Solidity` versions in number one and compiler bugs in 77 to 94 of `Solidity` 101 module and dependency issues in 183 of security pitfalls and best practices 201 module.

8.38 Audit Finding #139

139 is another finding from Trail of Bits audit of Yield protocol where the concern was that permission granting was too simplistic and not flexible enough. For context the yield protocol implemented several contracts that needed to call privileged functions from each other. However there was no way to specify which operation could be called for every privileged user. All the authorized addresses could call any restricted function, the Owner could add any number of them. Also the privileged addresses were supposed to be smart contracts, but there was no check for that and moreover once an address was added it could not be deleted therefore.
The recommendation was to rewrite the authorization system to allow only certain addresses to access certain functions.
This is related to access control and trust issues we discussed in 148, 149, 160 and 172 and principles of least privilege in 192 and principle of separation of privilege in 193 of security pitfalls and best practices 201 modules.

8.39 Audit Finding #140

140 is another finding from Trail of Bits audit of Yield protocol where the concern was that there was lack of validation when setting the maturity value. For context when fyDAI contract was deployed one of the deployment parameters was a maturity date passed as a unix timestamp. This was the date at which point fyDAI tokens could be redeemed for the underlying time.
The contract constructor however performed no validation of that timestamp to ensure that it was within an acceptable range. As a result it was possible to mistakenly deploy a wide-eyed contract that had a maturity date in the past or many years into the future which may not be immediately noticed.
The recommendation therefore was to add sanity and threshold checks to the wide eye contract constructor, to ensure that maturity timestamps were within an acceptable range, to prevent maturity dates from being mistakenly set in the past or too far into the future.
This is related to system documentation in 137 function parameter validation in 138 and broader aspect of data validation issues in 169 we discussed in security pitfalls and best practices 201 modules.

8.40 Audit Finding #141

141 is another finding from Trail of Bits audit of Yield protocol where the concern was about auditing and logging that delegates could be added or removed repeatedly to bloat logs.
For context when a user added or removed a delegate a corresponding event was emitted to log this operation. However there was no check to prevent the user from repeatedly adding or removing a delegation which could allow redundant events to be emitted repeatedly.

The recommendation was to add a `require` statement to check that the delegate address was not already enabled or disabled for the user to ensure that log messages are only emitted when a delegate is activated or deactivated to prevent bloated logs.
This is generally related to redundant constructs in 157 and broad aspects of auditing and logging in 173 that we discussed in security pitfalls and best practices to one module.

8.41 Audit Finding #142

142 is a finding from Trail of Bits audit of 0x protocol where the concern was about auditing and logging that there was a lack of events for critical operations. For context several critical operations did not trigger events which would make it difficult to review the correct behavior of the contracts once deployed. Users and blockchain monitoring systems would not be able to easily detect suspicious behaviors without events.
The recommendation was to add events where appropriate for all critical operations and in the long term to consider using a blockchain monitoring system to track any suspicious behavior in the contract.
This is related to the missing events aspect discussed in 45 of security pitfalls and best practices 101 module and broader auditing logging issues of 173 along with principle of compromise recording of 201 we discussed in security pitfalls and best practices to one module.

8.42 Audit Finding #143

143 is another finding from Trail of Bits audit of 0x protocol where the concern was about error handling, in that the function asserts taking pool exists should have returned a `boolean` to determine, if the staking pool existed or not, but it did not use a return statement and therefore would always return false or revert.
The recommendation was to add a return statement or remove the return type and change the documentation accordingly.
This is related to function `return` values in 142 and error reporting issues in 175 of security pitfalls and best practices to a one module.

8.43 Audit Finding #144

144 is a finding from Trail of Bits audit of DFX Finance where the concern was about specification in that the min and max family of functions had unorthodox semantics.
throughout the curve contract min target amount and max origin amount were used as open ranges that is ranges that exclude the value itself. This is unlike

the conventional interpretation of the terms minimum and maximum which are generally used to describe close ranges, so.
The recommendation was to make the inequalities in the required statements non-strict unless they are intended to be strict or alternatively document to convey that they are meant to be exclusive bonds. And in the long term ensure that mathematical terms such as minimum at least and at most are used in the typical way to describe values inclusive of minimums or maximums.
This is related to dangerous equalities in 28 now security pitfalls and best practices 101 module and broad aspects of system specification and documentation in 136 and 137 and numerical issues in 170 that we discussed in security pitfalls and best practices to one body.

8.44 Audit Finding #145

145 is another finding from Trail of Bits audit of DFX Finance, where the concern was about undefined behavior in that, if an operator attempted to create a new curve in the context of the protocol for a base and quote currency pair that already existed curve factory would return the existing curve instance without any indication of that causing a naive operator to maybe overlook this issue.
The recommendation was to consider rewriting that logic such that it reverted, if a base and code currency pair already existed and provide a `view` function to check for and retrieve existing curves prior to an attempt at curve creation.
This is related to the broad aspect of undefined behavior in 179 and clarity in 188 of security pitfalls and best practices to one module.

8.45 Audit Finding #146

146 is another finding from Trail of Bits audit of DFX Finance where the concern was about data validation in that few functions were missing Zero-address checks.
For example a Zero-address check should have been added to the router constructor to prevent the deployment of an invalid router which would `revert` upon a call to the `zero` items, so.
The recommendation was to review address type state variables to ensure that the code that sets the state variables performs Zero-address checks when necessary as a best practice.
This is related to missing Zero-address validation in 49 or security pitfalls and best practices 101 module and broader aspects of function parameters in 138 function invocation arguments and 146 two-step change of privileged roles in 162 and data validation issues in 169 of security pitfalls and best practices to one.

8.46 Audit Finding #147

147 is another finding from Trail of Bits audit of DFX Finance where the concern was about error handling in that the custom safe `approve` function did not check `return` values for approved call.
For context the router contract used OpenZeppelin's Safe `ERC20` library to perform safe calls to `ERC20`'s `approve` function, but the orchestrator library defined its own safe `approve` function.
This function check that a call to `approve` was successful, but did not check the return data to verify whether the call indeed returned true. In contrast OpenZeppelin's Safe `approve` function checks `return` values appropriately this issue could have resulted in uncaught `approve` errors in successful curve deployments causing undefined behavior.
The recommendation was to leverage OpenZeppelin's Safe save `approve` function wherever possible and also ensure that all low level calls have accompanying contract existence checks and `return` value checks where appropriate.
This is related to function `return` values in 142 error reporting issues and 175 and cloning issues in 190 of security pitfalls and best practices 201.

8.47 Audit Finding #148

148 is another finding from Trail of Bits audit of DFX Finance where the concern was about configuration. In that curve being an `ERC20` token implemented all six required `ERC20` methods balance of, total supply, allowance, transfer, `approve` and transfer from. But it did not implement the optional, but extremely common `view` methods for symbol name and decimals.
The recommendation was to implement simple name and decimals on curve contracts to ensure that contacts confirm to all required and recommended aspects of the `ERC20` specification.
This is related to `ERC20` name decimals and simple functions in 103 and configuration issues in 169 of security pitfalls and best practices 201 modules.

8.48 Audit Finding #149

149 is another finding from Trail of Bits audit of DFX Finance where the concern was about data validation in that although SafeMath was used throughout the code base to prevent underflows and overflows it was not used in a few calculations.
Although the audit could not prove that the lack of SafeMath would cause an arithmetic issue in practice, all calculations would benefit from the use of safe.
The recommendation was to review all critical arithmetic to ensure that it accounted for underflows, overflows and loss of precision by considering the use of SafeMath and safe functions of ABDKMath where possible to prevent any underflows and overflows.

This is related to dangers of integer overflow underflow we discussed in 19 of security pitfalls and best practices 101 module and broader aspects of data validation in 169 and numerical issues in 170 of security pitfalls and best practices 201.

8.49 Audit Finding #150

150 is another finding from Trail of Bits audit of DFX Finance where the concern was about timing and denial of service (DoS). In that the function set frozen could be used by the contract Owner to front run to deny deposits or swaps. The contract Owner could, then unfreeze them at a later time.
The recommendation was to consider rewriting the set frozen function, such that any contract freeze would not last long enough for a malicious Owner to easily `execute` an attack or alternatively consider implementing permanent freezes.
This is related to the transaction order dependence aspect discussed in 21 of security pitfalls and best practices 101 module denial of service in 176 and trust issues in 181 of security pitfalls and best practices 201.

8.50 Audit Finding #151

151 is a finding from Trail of Bits audit of Hermez network where the concern was about denial of service (DoS) by account creation span Hermez had no fees on about creation and, so an attacker could spam the network by creating the maximum number of accounts. Remember that Ethereum miners do not have to pay for Gas and, so they themselves could spam the network with account creation.
The recommendation was to add a fee for account creation and to also monitor account creation and alert users, if a malicious coordinator spam the system.
This is related to broad aspects of audit and logging in 173 denial of service in 176 and trust issues in 181 of security pitfalls and best practices 201.

8.51 Audit Finding #152

152 is another finding from Trail of Bits audit of Hermez network where the concern was about undefined behavior from using empty functions instead of interfaces because that leaves contracts errored.
For context withdrawal delayer interface was a contract meant to be an interface because it contained functions with empty bodies instead of function signatures, which might lead to unexpected behavior. Contract inheriting from withdrawal delayer interface would not `require` an override of those functions and, so would not benefit from the compiler checks on its correct interface.
The recommendation was to use an interface instead of a contract in withdrawal the layer interface, which would make derived contracts follow the interface properly and to also document the inheritance schema of the contracts.

This is related to unused constructs in 156, the undefined behavior issues in 179 or security pitfalls and best practices 201.

8.52 Audit Finding #153

153 is another finding from Trail of Bits audit of Hermez network where the concern was about data validation in that canceled transaction could be called on a non queued transaction.
Without a transaction existence check in cancel transaction an attacker could confuse monitoring systems because that emitted an event without checking that the transaction to be cancelled existed.
The recommendation was to check that the transaction to be cancelled existed in cancel transaction function to ensure that monitoring tools could rely on limited events.
This is related to data validation in 169 and auditing and logging in 173 that we discussed in security pitfalls and best practices 201 module.

8.53 Audit Finding #154

154 is another finding from Trail of Bits audit of Hermez network where the concern was about patching, in that contracts used as dependencies did not track upstream changes.
For context third-party contracts like concat storage were copy pasted into the Hermez repository, the code documentation did not specify the exact version used or, if it was modified.
This would make updates and security fixes on such dependencies unreliable since they would have to be updated manually specifically concat storage was borrowed from the `Solidity` Bytes Utils library which provided helper functions for bite-related operations and a critical vulnerability was discovered in that library's slice function, that allowed arbitrary rights for user supplied inputs.
The recommendation was to review the code base and document each dependency source and version and also include the third party sources as sub modules and git repository, so that internal path consistency could be maintained and dependencies could be updated periodically.
This is related to the broad aspect of configuration issues in 165 external interaction of 180 dependency of 183 and cloning issues in 190 that we discussed in security pitfalls at best practices 201 modules.

8.54 Audit Finding #155

155 is another finding from Trail of Bits audit of Hermez network where the concern was about access control in that the expected behavior regarding authorization for adding new tokens was unclear.

For context ad token allowed anyone to list a new token on Hermez which contradicted the online documentation that implied that only the governance should have had this authorization it was therefore unclear whether the implementation or the documentation was correct.
The recommendation was to update either the implementation or the documentation to standardize the authorization specification for adding new tokens.
This is related to the broad aspects of guarded launch via asset types and 129 system specification in 136 access control in 172 and clarity issues of 180a we discussed in security pitfalls and best practices to one module.

8.55 Audit Finding #156

156 is another finding from Trail of Bits audit to Hermez network where the concern was about undefined behavior in that contract name duplication left the code page error-prone.
The code base had multiple contracts that shared the same name which allowed Builder Waffle to generate incorrect JSON artifacts preventing third parties from using their tools. Builder Waffle did not currently support a code base with duplicate contact names, the compilation overwrote its artifacts and prevented the use of third-party tools such as Slither.
The recommendation was to avoid duplicate contact names change the compilation framework or use Slither which helps detect duplicate contract names.
This is related to broad aspects of programming style code layout and aiming convention in 97 to 101 percentage 101 module and clarity issues in 188 principle of economy of mechanism in 197 and principle of psychological acceptability in 199 that we discussed in security pitfalls and best practices to one module.

8.56 Audit Finding #157

157 is a finding from Trail of Bits audit of Advanced Blockchain where the concern was about patching, in that there was use of hard-coded addresses which may have caused errors.
For context each contract needed contract addresses in order to be integrated into other protocols and systems, these addresses were hard coded which could have cost errors and resulted in the code basis deployment with a correct asset. Using hard coded values instead of deploying provided values would have made these contracts difficult to test.
The recommendation was to set addresses when contacts were created rather than using hard coded values which would also facilitate testing.
This is related to tests in 155 configuration and initialization issues in 165 and 166 that we discussed in security pitfalls and best practices 201.

8.57 Audit Finding #158

158 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about configuration in that the borrow rate formula used an approximation of the number of blocks mined annually.
This number could change across different blockchains and years. The value assumed that a new block was mined every 15 seconds, but on Ethereum min net a new block is mined every 13 seconds approximately.
The recommendation was to analyze the effects of a deviation from the actual number of blocks mined annually in borrow rate calculations and document associated risks.
This is related to block values as time proxies in 18 of security pitfalls and best practices 101 module and configuration and initialization in 165 and 166 and `constant` issues at 184 that we discussed in security pitfalls and best practices 201 model.

8.58 Audit Finding #159

159 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about data validation. In that there were no lower upper bounds on the flash load rate implemented in the contract. This would therefore allow setting it to an arbitrarily high rate to secure higher fees.
The recommendation was to introduce lower and upper bound checks for all configurable parameters in the system to limit privileged users abilities.
This is related to function parameter validation in 138 function invocation arguments in 146 and broader aspect of data validation issues in 169 that we discussed in security pitfalls and best practices 201 modules.

8.59 Audit Finding #160

160 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about patching, in that the logic in the repositories contained a significant amount of duplicated code which increase the risk that new bugs would be introduced into the system as bug fixes must be copied and pasted into files across the system.
The recommendation was to use inheritance to allow code to be used across contracts and to minimize the amount of manual copying and pasting required to apply changes made in one file to other files.
This is related to programming style code layout and naming convention in 97 to 101 of `Solidity` 101 module and broad aspects of configuration in 165 clarity 188 cloning in 190 principle of economy of mechanism in 197 and principle of psychological acceptability in 199 that we discussed in security pitfalls and best practices 201 modules.

8.60 Audit Finding #161

161 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about insufficient testing. The repositories under review lacked appropriate testing which increased the likelihood of errors in the development process and made code more difficult to review.

The recommendation was to ensure that unit tests cover all public functions at least once as well as all known corner cases, and also to integrate coverage analysis tools into the development process and regularly review the coverage. This is related to broad aspect of testing in 155 that we discussed in security pitfalls and best practices 201.

8.61 Audit Finding #162

162 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about project dependencies containing vulnerabilities. Yarn audit identified off-chain dependencies with no vulnerabilities and due to the sensitivity of the deployment code and its environment it was important to ensure that dependencies were not malicious.

The recommendation was to ensure that dependencies were tracked verified patched and audited. This is related to the broad aspects of configuration in 165 external interaction in 180 and dependency of 183 that we discussed in security pitfalls and best practices to one module.

8.62 Audit Finding #163

163 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about the code base lacking code documentation, high level descriptions and examples, making the contracts difficult to review and increasing the likelihood of user mistakes.

The recommendation was to review and properly document the code base and also consider writing a formal specification of the protocol. This is related to broader aspects of system specification and documentation in 136 and 137, the principle of psychological acceptability in 199 that we discussed in security pitfalls and best practices 201.

8.63 Audit Finding #164

164 is another finding from Trail of Bits audit of Advanced Blockchain where the concern was about API encoder V2 not being production ready. At the time of this order, given that more than three percent of all GitHub issues for

the `Solidity` compiler were related to experimental features, primarily ABI encoder V2. ABI encoder V2 had been associated with more than 20 high severity bugs at that point in time.

The recommendation was to not use API encoder V2 by refactoring the code such that structs do not need to be passed to, or returned from functions, which is a feature enabled by it. This is related to compiler bugs in 77 to 94 of `Solidity` 101 module and dependency issues in 183 of security pitfalls and best practices 201 modules.

8.64 Audit Finding #165

165 is the finding from Trail of Bit's audit of dForce lending protocol where the concern was about the contract Owner having too many privileges compared to standard users of the protocol. Users could lose all of their assets if a contract owner's private keys were to be compromised. The contract Owner could, for example do the following:

1. Upgrade the system's implementation to steal funds.
2. Upgrade the tokens implementation to act maliciously.
3. Increase the amount of high tokens for remote distribution to such an extent that rewards could not be dispersed.
4. Arbitrarily update the interest model contracts.

Such concentration of privileges created a single point of failure and increased the likelihood that the Owner would be targeted by an attacker, especially given the insufficient protection on sensitive Owner private keys. Additionally it incentivized the Owner to act maliciously.

The recommendations were:

1. Clearly document the functions and implementations the Owner could change.
2. Split privileges to ensure that no one address had excessive ownership of the system.
3. Document the risks associated with privileged users and single points of failure.
4. Ensure that users were aware of all the risks associated with the system.

This is related to access control and trust issues we discussed in 148, 149, 160 and 172 and principle of least privilege in 192 and principle of separation of privilege in 193 of security pitfalls at best practices 201.

8.65 Audit Finding #166

166 is another finding from Trail of Bits audit of dForce lending protocol where the concern was about poor error handling practices in the test suite. For context, the test suite did not properly test expected behavior and certain components lacked error handling methods, which would cause failed tests to be overlooked. For example errors were silenced with a `try/catch` statement, which meant that there was no guarantee that a call had reverted for the right reason. As a result, if the test suite passed, it would have provided no guarantee that the transaction call had reverted correctly.

The recommendation was to test operations against a specific error message and ensure that errors were never silenced to check that a contact call had reverted for the right reason and overall follow standard testing best practices for smart contracts to minimize the number of issues during development. This is related to the broad aspect of testing in 155 of security pitfalls and best practices 201 modules.

8.66 Audit Finding #167

167 is a finding from Sigma Prime's audit of synthetix Ether collateral protocol where the concern was about redundant and unused code. For example, the `recordLoanClosure` function returned a `bool` which was never used by the calling function, and there were also some `if` statements that were redundant and unnecessary.

The recommendation was to remove such redundant constructs or use them in meaningful ways. This is related to redundant construction 157 of security pitfalls and best practices 201 module.

8.67 Audit Finding #168

168 is another finding from Sigma Prime's audit of synthetix Ether collateral protocol where the concern was that a single account could capture all the supply in the protocol. For context, the protocol did not rely on a `MAX_LOAN_SIZE` to limit the amount of Eth that can be locked for a loan. As a result, a single account could issue a loan that could reach the total minting supply.

The recommendation was to make sure that this behavior was understood and documented, and also considered introducing and enforcing a cap on the size of the loans allowed to be open. This is related to (youth?) handling in 158 data validation in 169 and numerical and accounting issues in 170 and 171 of security pitfalls and best practices to one module.

8.68 Audit Finding #169

169 is another finding from Sigma Prime's audit of synthetix Ether collateral protocol where the concern was about insufficient input validation, specifically in that the Ether collateral constructor did not check the validity of the addresses and other types provided as input parameters. This, for example, made it possible to deploy an instance of the contract with critical Addresses set to zero.

The recommendation was to consider introducing required statements to perform adequate input validation. This is related to missing zero-address validation in 49 of security pitfalls and best practices 101 module and broader aspects of function parameters in 138 function invocation arguments in 149 and data validation issues in 169 or security pitfalls and best practices 201.

8.69 Audit Finding #170

170 is another finding from Sigma Prime's audit of synthetix Ether collateral protocol where the concern was about unused event logs in that log events were declared, but never emitted.

The recommendation was to emit these events where required appropriately or remove them entirely. This is related to unused constructs in 156 and auditing and locking in 173 of security pitfalls and best practices 201 module.

8.70 Audit Finding #171

171 is a finding from Sigma Prime's audit of InfiniGold where the concern was about unintentional token burning in `transferFrom`. For context, InfiniGold allowed users to convert their PMGT tokens to gold certificates, which were digital artifacts effectively redeemable for actual gold. To do so, users were supposed to send their PMGT tokens to a specific burn address. however, the `transferFrom` function did not check its `to` address parameter against this burn address, which would allow users to accidentally send their tokens to the special burn address using the `transferFrom` function without triggering the emission of the burn event, which dictated how the gold certificates were created and distributed, so effectively users would lose their tokens without redeeming them for gold certificates.

The recommendation was to prevent sending tokens to the burn address in the `transferFrom` function by adding a `require` within `transferFrom`, which disallow the two address to be the burn address. This is generally related to missing zero-address validation in 49 of security pinfalls and best practices 101 module and broader aspects of function parameters in 138 function invocation arguments in 146 data validation in 169 accounting issues in 171 and error reporting issues in 175 of security pitfalls and best practices 201.

8.71 Audit Finding #172

172 is another finding from Sigma Prime's audit of InfiniGold where the concern was about denial derivative service from unbounded lists. For context, the `reset` function reset the role linked list by deleting all its elements. Calling the reset function would exceed the block Gas Limit, 8 million, at the time of the audit for more than 371 total elements in the role linked list. Similarly, other functions also looped through linked lists which meant that certain protocol actors could perform denial of service attacks on the lists they administered.

The recommendation was:

1. Either check that the linked list size is strictly less than 371 elements before adding a new element or
2. use the `gasLeft Solidity` primitive to make sure that traversing the linked list did not exceed the block Gas Limit at any point or
3. Change reset to take a specific number of elements as a function parameter

This is related to the Gas impact on DoS in 42, 43 and 44 of security pitfalls and best practices 101 module and broader aspects of denial of service in 176 and Gas issues in 182 of security pitfalls and best practices 201.

8.72 Audit Finding #173

173 is another finding from Sigma Prime's audit of InfiniGold where the concern was about the `ERC20 approve` function being vulnerable to Front-running.

The recommendation was to be aware of Front-running issues and approved and potentially use OpenZeppelin's library with `increaseAllowance()` and `decreaseAllowance()` functions with the caveat that deviating from the `ERC20` standard to address this issue could lead to backward incompatibilities with external third-party software. This is related to transaction order dependence in 21 and `ERC20` approved Race-condition in 22 of security pitfalls and best practices 101 module along with `ERC20` approved Race-condition in 105 and broader aspects of timing issues in 177 of security pitfalls and best practices 201 module.

8.73 Audit Finding #174

174 is another finding from Sigma Prime's audit of InfiniGold where the concern was about an unnecessary `require` statement in blacklistable contract which implemented a Zero-address check on the two address, when this check was also implemented in the transfer function of `ERC20` contract.

The recommendation was to consider removing the `require` statement for Gas saving purposes. This is related to redundant constructs in 157 now security pitfalls and best practices 201.

8.74 Audit Finding #175

175 is another finding from Sigma Prime's audit of InfiniGold where the concern was about the reward rate rounding to zero if duration was greater than reward. The reward rate value was calculated as reward divided by duration and due to the integer representation of these variables, if duration were to be larger than reward, the value of reward rate would round down to zero. Thus, stakers would not receive any reward for their stakes and there would be other implications as well such as collection of task tokens.

The recommendation was to be aware of the surrounding issue and also consider providing a way to claim the dust SNX rewards from rounding. This is generally related to divide before multiply in number 20 of security pitfalls and best practices 101 module and broadly related to data validation and numerical issues in 169 and 170 of security pitfalls and best practices 201 module.

8.75 Audit Finding #176

176 is another finding from Sigma Prime's auditor InfiniGold where the concern was about event log poisoning. For context, calling the withdrawal function would emit the withdrawal event where no UD tokens were required because this function could be called with 0. As a result, a user could continuously call this function creating a potentially infinite number of events which could lead to an event log poisoning situation where malicious external users could spam the unipool contract to generate arbitrary withdrawal events.

The recommendation was to consider adding a `require` or `if` statement to prevent the withdrawal function from emitting the withdrawal event when the amount variable was zero This is related to validation of function parameters in 138 and auditing and logging in 173 of security pitfalls and best practices 201 module.

8.76 Audit Finding #177

177 is a finding from OpenZeppelin's audit of HoldeFi where the concern was about insufficient incentives to liquidators for context the liquidation process is a very important part of every DEFI project because it addresses the problem of a system being under collateralized under critical conditions in the market and therefore needs a design that incentivizes speed of liquidation execution as per modify specification and implementation the liquidators would end up paying

for the expensive liquidation process without receiving any benefit other than buying potentially discounted collateral assets.
The recommendation was to consider improving the incentive design to give liquidators higher incentives to `execute` the liquidation process this is related to function invocation timeliness in 143 and incentive issues in 187 of security pitfalls and best practices 201 module.

8.77 Audit Finding #178

178 is another finding from OpenZeppelin's audit of HoldeFi where the concern was that markets could become insolvent. For context, when the value of all collateral is worth less than the value of all borrowed assets, a market is considered insolvent. The HoldeFi code base could do many things to reduce the risk of market insolvency, such as selection of collateral ratios, incentivizing third-party collateral liquidation, careful selection of tokens listed on the platform, etc... However, the risk of insolvency would not be entirely eliminated and there are numerous ways a market could still become insolvent. This risk is not unique to the HoldeFi project and all collateralized loans. Even non-blockchain ones have a risk of insolvency. However it was important to recognize that this risk does exist, that it could be difficult to recover from it.

The recommendation was therefore to consider adding more targeted tests for insolvency scenarios to better understand the behavior of the protocol and designing relevant mechanics to make sure the platform operated properly under such conditions, and also consider communicating the potential risk to the users if needed. This is related to garden launch via asset types and 129 and broader aspects of token handling in 159 system specification and documentation in 136 137 and accounting issues in 171 of security pitfalls and best practices 201.

8.78 Audit Finding #179

179 is another finding from OpenZeppelin's audit of HoldeFi where the concern was that the project re-implemented some of OpenZeppelin's libraries and copied them as is in some others, instead of importing the official ones. OpenZeppelin maintains a library of standard audited community reviewed and partly tested smart contracts. Re-implementing or copying them increases the amount of code that the whole defined team would have to maintain and missed all the improvements and bug fixes that the OpenZeppelin team was constantly implementing with the help of the community.

The recommendation was to consider importing the open zipline libraries instead of re-implementing or copying them and further extend them where necessary to add extra functionalities this is specifically related to cloning issues in 190 of security pitfalls and best practices 201.

8.79 Audit Finding #180

180 is another finding from OpenZeppelin's audit of HoldeFi where the concern was that there was a lack of in-text parameters in events throughout the whole device code base: none of the parameters in the events defined in the contracts were in text.

The recommendation was to consider indexing event parameters to facilitate off-chain services searching and filtering for specific events because remember that in-text event parameters are put into the topic part of the event log, which is faster to look up than the data part. This is specifically related to unindexed event parameters and 46 or security pitfalls and best practices 101 module and broadly related to auditing logging issues in 173 of security pitfalls and best practices 201 modules.

8.80 Audit Finding #181

181 is another finding from OpenZeppelin's audit of HoldeFi where the concern was that there was an inconsistent use of named return variables across the code base that affected explicitness and readability.

The recommendation was to consider removing all named return variables explicitly declaring them as local variables in the function body and adding the necessary explicit return statements where appropriate. This is related to function return values in 142 explicit over implicit in 164 and clarity issues in 188 of security pitfalls and best practices 201 module.

8.81 Audit Finding #182

182 is another finding from OpenZeppelin's audit of HoldeFi where the concern was that as part of some calculations and time checks, it used `block.timestamp` which is unreliable because timestamps can be slightly altered by miners to favor them in contracts that have logic depending strongly on them.

The recommendation was to consider taking into account this issue and warning users that such a scenario was possible and, if the alteration of time stamps couldn't affect the protocol in any way. to consider documenting that reasoning and writing tests to enforce that those guarantees would be preserved even if the code changed in future. This is related to weak PRNG and block values is type proxies in 17 and 18 or security pitfalls and best practices 201 module and broader aspects of trusted actors in 160 and timing issues in 177 of security pitfalls and best practices 201 module.

8.82 Audit Finding #183

183 is a finding from OpenZeppelin's audit of BarnBridge where the concern was about a `require` statement that made an assignment which deviates from standard usage and intention of `require` statements, and could lead to confusion.

The recommendation was to consider moving the assignment to its own line before the `require` statement. Then, using the `require` statement only for condition checking. This is related to `assert`/`require` state change in 51 of security pitfalls and best practices 101 module and broader aspects of error reporting in 175 and clarity issues in 188 of security pitfalls and best practices 201 module.

8.83 Audit Finding #184

184 there's another finding from OpenZeppelin's audit of BarnBridge where the concern was about commented code in that the code base had lines of code that had been commented up. This could lead to confusion and affected code readability and auditability.

The recommendation was to consider removing commented out lines of code that were no longer needed. This is related to comments in 154 and clarity issues in 188 of security pitfalls and best practices 201.

8.84 Audit Finding #185

185 is a finding from OpenZeppelin's audit of compound where the concern was about misleading reward messages. Error messages are intended to notify users about failing conditions and should provide enough information so that appropriate corrections needed to interact with the system can be applied. Uninformative error messages affect user experience.

The recommendation therefore was to consider reviewing the code pairs to make sure error messages were informative and meaningful and also reuse error messages for similar conditions. This is related to error reporting issues in 175 clarity issues in 188 and principle of psychological acceptability in 199 of security pitfalls and best practices 201 modules.

8.85 Audit Finding #186

186 is a finding from OpenZeppelin's audit of Fei protocol where the concern was about multiple outdated `Solidity` versions being used across contracts. The compiler options in the Truffle config file specified version `0.6.6` which was

released in april 2020, and throughout the code base there were also different versions of `Solidity` being used.

The recommendation was that given `Solidity`'s fast release cycle to consider using a more recent version of the compiler and specifying explicit `Solidity` versions in `pragma` statements to avoid unexpected behavior. This is related to `Solidity` versions unlocked `pragma@ and multiple \verbSolidity| pragma`s in 1, 2 and 3 of security pitfalls and best practices 101 module and explicit over implicit in configuration in 165 and dependency issues in 183 of security pitfalls and best practices 201.

8.86 Audit Finding #187

187 is another finding from OpenZeppelin's audit of Fei protocol where the concern was about test and production constants being in the same code base. For example, the core orchestrator contract defined the test mode `boole` variable which was then used to define several other test constants in the system. This decreased the legibility of production code and made the system's integral values more available.

The recommendation was to consider having different environments for production and testing with different contracts. This is related to tests in 155 and configuration issues in 165 of security pitfalls and best practices 201 module.

8.87 Audit Finding #188

188 is another finding from OpenZeppelin's audit of Fei protocol where the concern was about the use of unnecessarily smaller sized integer variables. In `Solidity`, using integers smaller than 256 bits (which is the EVM word size) tends to increase Gas Cost because the EVM must then perform additional operations to zero or mask out the unused bits in remaining parts of storage slots for such integers. This can be justified by savings and storage costs in some scenarios. However that was not the case for this code base.

The recommendation was to consider using integers of size 256 bits to improve Gas efficiency. This is related to system specification in 136 and principle of economy of mechanism in 197 of security pitfalls and best practices 201 module.

8.88 Audit Finding #189

189 is another finding from OpenZeppelin's audit of Fei protocol where the concern was the use of `uint` instead of `uint256` across the code base.

he recommendation was to consider replacing all instances of `uint` with `uint256` in favor of explicitness. This is related to explicit over implicit in 164 and clarity 188 of security pitfalls at best practices 201.

8.89 Audit Finding #190

190 is a finding from OpenZeppelin's audit of UMA protocol where the concern was about functions with unexpected Side-effects. For example, the `getLatestFundingRate` function of the funding rate applier contract might also update the funding rate and send rewards. The `getPrice` function of the optimistic Oracle might also settle a price request. These setter like Side-effect actions were not clear in the getter like names of functions and were thus unexpected and could lead to mistakes if the code were to be modified by new developers not experienced in all the implementation details of the project.

The recommendation was to consider splitting such functions into separate getters and setters, or alternatively consider renaming the functions to describe all the actions that they performed. This is related to broad aspects of programming style code layout and naming convention in 97 to 101 of `Solidity` 101 module and clarity in 188 principle of economy of mechanism in 197 and principle of psychological acceptability in 199 that we discussed in security pitfalls and best practices 201 module.

8.90 Audit Finding #191

191 is a finding from OpenZeppelin's audit of GEB protocol where the concern was about unSafeCasting. For example, one of the contracts used an unsigned integer which was cast to a signed integer, then negated however since `uint` could store higher values than `int`, it was possible that casting from `uint` to `int` may have created an overflow.

The recommendation was to consider verifying that values of such unsigned integers were within the acceptable range for signed integer time before casting, indicating and to consider using OpenZeppelin's SafeCast library which provides functions for safely casting between types. This is related to OpenZeppelin's SafeCast in 177 of solarity 201 module integer overflow underflow in 19 or security pitfalls and best practices 101 module and broader aspects of data validation and numerical issues in 169 and 170 of security pitfalls and best practices 201.

8.91 Audit Finding #192

192 is another finding from OpenZeppelin's audit of GEB protocol where the concern was about missing error messages in `require` statements. There were

many places where `require` statements were correctly followed by their error messages, clarifying what the triggered exception was. However, there were also places where `require` statements were not followed by corresponding error messages. If any of those required statements were to fail the check condition, the transaction would revert silently without an informative error message.

The recommendation was to consider including specific and informative error messages in all `require` statements. This is related to error reporting issues in 175 clarity issues in 188 and principle of psychological acceptability in 199 of security pitfalls and best practices to one module.

8.92 Audit Finding #193

193 is another finding from OpenZeppelin's audit of GEB protocol where the concern was about the use of Assembly in multiple contracts. While this did not pose a security risk per se, it is a complicated and critical part of the system. Remember that the use of Assembly discards several important safety features of `Solidity` which may render the code unsafe and more error-prone.

The recommendation therefore was to consider implementing thorough tests to cover all potential use cases of these functions to ensure that they behaved as expected and to consider including extensive documentation regarding the rationale behind its use, clearly explaining what every single Assembly instruction does, which would make it easier for users to trust the code, for reviewers to verify it and for developers to build on top of it or update it. This is related to Assembly usage in 63 of security pitfalls and best practices 101 module and broader aspects of system documentation in 137 tests in 155 clarity 188 and principle of psychological acceptability in 199 security pitfalls and best practices 201.

8.93 Audit Finding #194

194 is another finding from OpenZeppelin's audit of GEB protocol where the concern was about `catch` clause not being handled appropriately in a couple of functions. The `catch` clause of `Solidity`'s `try`/`catch` exception handling primitive was neither emitting events nor handling the error, but simply continuing the execution.

The recommendation was that, even if continuing execution after a possible fail was something explicitly wanted, to consider handling the `catch` clause by either emitting an appropriate event or registering the failed (tricol?) in the spirit of the failed early and loudly principle. This is related to error reporting in 175 clarity in 188 and principle of psychological acceptability in 199 of security pitfalls and best practices 201 modules

8.94 Audit Finding #195

195 is another finding from OpenZeppelin's audit of GEB protocol where the concern was about unnecessary event emission. For example, the pop dead from `q`(?) function of the accounting engine contract was emitting an unnecessary event whenever it was called with a dead block timestamp that had not been saved before.

The recommendation was to remove such event emits and prevent Gas wastage. This is related to redundant constructs in 157 of security pitfalls and best practices 201.

8.95 Audit Finding #196

196 is a finding from OpenZeppelin's audit of Opyn Gamma protocol where the concern was that the protocol's `0` token could be created with a non-white listed collateral asset. A product consisted of a set of assets and an auction time, and each product had to be whitelisted by the admin using the whitelist product function from the whitelist contract.

The recommendation was to consider validating if the assets involved in a product had already been whitelisted before allowing creation of `0` tokens. This is related to aspects of carded launch via composability limit in 132 token handling in 159 external interactions at 180 and dependency issues in 183 of security pitfalls and best practices 201.

8.96 Audit Finding #197

197 is another finding from OpenZeppelin's audit of Opyn Gamma protocol where the concern was about mismatches between contracts and interfaces. Interfaces define the exposed functionality of implemented contracts. However, in several interfaces there were functions from the counterpart contracts that were not defined.

The recommendation was to consider applying necessary changes in the mentioned interfaces and contracts, so that definitions and implementations fully match. This is related to system specification in 136 undefined behavior in 179 and clarity issues in 188 of security pitfalls and best practices 201.

8.97 Audit Finding #198

198 is another finding from OpenZeppelin's audit of Opyn Gamma protocol where the concern was about inconsistent state resulting from actions not executed. For context, the set asset pricing, set locking period and set dispute

period functions of the Oracle contract executed actions that were always expected to be performed atomically. Failing to do so, could lead to inconsistent states in the system.

The recommendation was therefore to consider implementing an additional function that calls all three set asset pricing, set locking period and set dispute period functions, so that all these actions could be executed atomically in a single transaction. This is related to function invocation timeliness in 143 configuration and initialization in 165 and 166 timing in 177 and undefined behavior issues in 179 of security pitfalls and best practices 201.

8.98 Audit Finding #199

199 is another finding from OpenZeppelin's audit of Opyn Gamma protocol where the concern was about using a deprecated Chainlink API. The Chainlink pricer was using multiple functions from the deprecated Chainlink V2 API, such as latest answer and get timestamp. Such functions might certainly stop working if Chainlink stops supporting deprecated APIs.

The recommendation was to consider using the latest Chainlink V3 API. This is related to external interaction in 180 and dependency issues in 183 of security pitfalls and best practices to one body.

8.99 Audit Finding #200

200 is a finding from OpenZeppelin's audit of PoolTogether V3 protocol where the concern was that funds could be lost. For context the sweep timelock balances function accepted a list of users with unlocked balances to distribute. However, if there were duplicate users in the list, their balances would be counted multiple times while calculating the total amount of withdrawal from the yield service, which could lead to loss of their funds.

The recommendation was to consider checking for duplicate users when calculating the amount to withdraw. Rhis is related to token handling in 159 and numerical and accounting issues in 170 and 171 of security pitfalls and best practices 201 module.

8.100 Audit Finding #201

201 is a finding from OpenZeppelin's audit of Set Protocol where the concern was about clearing address variables by setting them to zero-addresses instead of using `delete`.

The recommendation was to consider replacing assignments of zero with `delete` statements because `delete` better conveyed the intention and was also more idiomatic. This is related to explicit over implicit in 164 cleanup in 167 and clarity issues in 188 of security pitfalls and best practices 201 module.

With that, hopefully we now have a much better appreciation for such application level business logic software engineering best practices and guidelines that are not immediately identifiable as vulnerabilities leading to fund loss, lock, etc... but are nevertheless critical to adhere to, so that they do not translate into critical vulnerabilities unknowingly or at a later time under some assumptions considerations interactions threat trust models not considered by the developers or the auditors.

Chapter 9

Self-Assessment

9.1 Ethereum 101 Quiz

1. Which of the following EVM components is/are non-volatile across transactions?

A. Stack

B. Memory

C. Storage

D. Calldata

Find the solution here.

2. The number of transactions in a Ethereum block depends on

A. Nothing. It is a constant

B. Gas used by transactions

C. Block gas limit

D. Block difficulty

Find the solution here.

3. EVM Stores

A. Most significant byte in the smallest memory address

B. Most significant byte in the largest memory address

C. In Big-endian order

D. In Little-endian order

Find the solution here.

4. Ethereum's threat model is characterised by

A. Trusted miners and users

B. Trusted users, untrusted miners

C. Trusted miners, untrusted users

D. Everyone is untrusted

Find the solution here.

5. Ethereum smart contracts do not run into halting problems because

A. EVM is not Turing Complete

B. EVM is Turing Complete

C. EVM is Turing Complete but is bounded by gas sent in transaction

D. EVM is Turing Complete but is bounded by the stack depth

Find the solution here.

6. Which of the following operation(s) touch(es) storage?

A. `SWAP`

B. `SLOAD`

C. `DUP`

D. `PUSH`

Find the solution here.

7. The most gas-expensive operation is

A. `SLOAD`

B. `SSTORE`

C. `CREATE`

D. `SELFDESTRUCT`

Find the solution here.

8. Transaction `T1` attempts to write to storage values `S1` and `S2` of contract `C`. Transaction `T2` attempts to read the same storage values `S1` and `S2`. However, `T1` reverts due to an exception after writing `S2`. Which or the following is/are TRUE?

A. `T2` reads the value of `S1` updated by `T1`

B. `T2` reads the value of `S1` prior to `T1`'s attempted update

C. `T2` also reverts because of the dependency on `T1`

D. This scenario is not possible

Find the solution here.

9. The gas tracking website https://etherscan.io/gastracker says that Low gas cost is 40 gwei. This affects

A. The transaction `gasPrice`

B. The transaction `gasLimit`

C. The transaction `value`

D. Both B & C

Find the solution here.

10. Security of Ethereum DApps depends on

A. Security of their smart contracts

B. Security of their off-chain components

C. Security of Ethereum

D. None of the above

Find the solution here.

11. A `nonce` is present in

A. Ethereum transaction

B. Ethereum account

C. Both A & B

D. Neither A nor B

Find the solution here.

12. Miners are responsible for setting

A. Transaction gas price

B. Block gas limit

C. Both A & B

D. Neither A nor B

Find the solution here.

13. Which of the following information CANNOT be obtained in the EVM?

A. Block difficulty

B. Transaction logs

C. Balance of an account

D. Block hash of any block

Find the solution here.

14. Miners are incentivized to validate and create new blocks by

A. Block rewards

B. Altruism

C. Transaction fees

D. Their belief in decentralization

Find the solution here.

15. Smart contracts on Ethereum

A. May be deployed by anyone

B. May be deployed only through the DApp store

C. May have some form of access control

D. Are guaranteed to be secure

Find the solution here.

16. Hardfork on Ethereum

A. Has never happened

B. Happened only once after the DAO attack

C. Happens with backwards-incompatible protocol changes

D. Happens when developers and miners disagree on changes

Find the solution here.

17. Which call instruction could be used to allow modifying the caller account's state?

A. `CALL`

B. `CALLCODE`

C. `DELEGATECALL`

D. `STATICCALL`

Find the solution here.

18. The length of addresses on Ethereum is

A. 256 bits

B. 20 bytes

C. Depends on the Externally-Owned-Account or Contract address

D. Configurable

Find the solution here.

19. Which of the following statements is/are TRUE about gas?

A. Unused gas is returned to the transaction destination account

B. Gas used by the transaction is credited to the beneficiary address in block header

C. Unused gas is credited to the beneficiary address in block header

D. Both A & B

Find the solution here.

20. The high-level languages typically used for writing Ethereum smart contracts are

A. `Go`

B. `C++`

C. `Viper`

D. `Solidity`

Find the solution here.

21. Ethereum nodes talk to each other via

A. Peer-to-Peer network

B. Client-Server network

C. Satellite network

D. None of the above

Find the solution here.

22. EVM is not a von Neumann architecture because

A. It was co-founded by Vitalik Buterin, not John von Neumann

B. Program instructions are stored separately from data

C. Program instructions are stored in a ROM not RAM

D. It is quasi Turing complete

Find the solution here.

23. User `A` sends transaction `T1` from address `A1` with `gasPrice` `G1` and later transaction `T2` from address `A2` with `gasPrice` `G2`

A. `T1` will be always included in an earlier block than `T2`

B. Inclusion/Ordering of these transactions depends only on gas prices `G1` and `G2`

C. Inclusion/Ordering of these transactions depends only on network congestion

D. Inclusion/Ordering of these transactions depends on miners

Find the solution here.

24. The types of accounts on Ethereum are

A. All Accounts are the same

B. Permissioned Accounts and Permissionless Accounts

C. Externally-Owned-Accounts and Contract Accounts

D. User Accounts and Admin Accounts

Find the solution here.

25. The number of decimals in Ether is

A. 0

B. 1

C. 18

D. Configurable

Find the solution here.

26. The difference(s) between Bitcoin and Ethereum is/are

A. The underlying tokens: Bitcoin vs Ether

B. Smart contract support

C. UTXO vs Accounts

D. Nakamoto Consensus

Find the solution here.

27. Security Audits for smart contracts are performed because

A. They are required for listing DApp on the DApp store

B. They are required for deployment on Ethereum

C. They help remove vulnerabilities and reduce risk

D. They are required by exchanges to list tokens

Find the solution here.

28. The number of modified Merkle-Patricia trees in an Ethereum block is

A. One

B. Three

C. Three plus number of contract accounts

D. Three plus number of transactions included in the block

Find the solution here.

29. EVM opcodes

A. Are multi-byte instructions

B. Are single byte instructions

C. Take operands in registers

D. Take operands on stack

Find the solution here.

30. Gas for EVM opcodes

A. Is constant and the same for all opcodes

B. May be changed over time to prevent DoS attacks

C. Depend on the gas price

D. Depend on the miners

Find the solution here.

31. Ethereum Virtual Machine is a

A. Register-based virtual machine

B. Stack-based virtual machine

C. Heap-based virtual machine

D. Stackless virtual machine

Find the solution here.

32. Which of the following statement(s) is/are FALSE?

A. EVM can get the block number only of the current block

B. EVM can get the block hash only of the current block

C. EVM can get the account balance only of the current account

D. EVM can get the code hash of only the current account

Find the solution here.

9.2 Solidity 101 Quiz

1. User from EOA `A` calls Contract `C1` which makes an external call (`CALL` opcode) to Contract `C2`. Which of the following is/are true?

A. `tx.origin` in `C2` returns `A`'s address

B. `msg.sender` in `C2` returns `A`'s address

C. `msg.sender` in `C1` returns `A`'s address

D. `msg.value` in `C2` returns amount of wei sent from `A`

Find the solution here.

2. Which of the following is/are true for `call`/`delegatecall`/`staticcall` primitives?

A. They are used to call contracts

B. They only revert without returning success/failure

C. `Delegatecall` retains the `msg.sender` and `msg.value` of caller contract

D. `Staticcall` reverts if the called contract reads contract state of caller

Find the solution here.

3. The gas left in the current transaction can be obtained with

A. `tx.gas()`

B. `gasleft()`

C. `msg.gas()`

D. `block.gaslimit()`

Find the solution here.

4. The default value of

A. `bool` is `false`

B. `address` is `0`

C. Statically-sized array depends on the underlying type

D. `enum` is its first member

Find the solution here.

5. Which of the following is/are true about events?

A. Events are meant for off-chain applications

B. Events can be accessed only by the emitting contract

C. Indexing event parameters creates searchable topics

D. A maximum of three events can have indexed parameters

Find the solution here.

6. Function `foo()` uses `block.number`. Which of the following is/are always true about `foo()`?

A. It should be marked as `pure`

B. It should be marked as `view`

C. It should be marked as `payable`

D. Cannot determine mutability based only on this information

Find the solution here.

7. `Solidity` functions

A. Can be declared only inside contracts

B. Can have named return variables

C. Can have unnamed parameters

D. Can be recursive

Find the solution here.

8. Conversions in `Solidity` have the following behavior

A. Implicit conversions are never allowed

B. Explicit conversions of `uint16` to `uint8` removes the higher-order bits

C. Explicit conversion of `uint16` to `uint32` adds lower-order padding

D. Explicit conversions are checked by compiler for safety

Find the solution here.

9. When Contract `A` attempts to make a `delegatecall` to Contract `B` but a prior transaction to Contract `B` has executed a `selfdestruct`

A. The `delegatecall` reverts

B. The `delegatecall` returns a failure

C. The `delegatecall` returns a success

D. This scenario is not practically possible

Find the solution here.

10. If `a = 1` then which of the following is/are true?

A. `a += 1` makes the value of `a = 2`

B. `b = ++a` makes the value of `b = 1`

C. `a -= 1` makes the value of `a = 1`

D. `b = a--` makes the value of `b = 1`

Find the solution here.

11. `transfer` and `send` primitives

A. Are used for Ether transfers

B. Trigger the `receive()` or `fallback()` functions of `address`

C. Always return a value to be checked

D. Provide only 2300 gas

Find the solution here.

12. A contract can receive Ether via

A. `msg.value` to `payable` functions

B. `selfdestruct` destination

C. `coinbase` transaction

D. `receive()` or `fallback()` functions

Find the solution here.

13. Structs in `Solidity`

A. Are user-defined type

B. Are reference types

C. Can contain or be contained in arrays and mappings

D. None of the above

Find the solution here.

14. The following is/are true about `ecrecover` primitive

A. Takes a message hash and ECDSA signature values as inputs

B. Recovers and returns the public key of the signature

C. Is susceptible to malleable signatures

D. None of the above

Find the solution here.

15. Which of the following is/are valid control structure(s) in `Solidity` (excluding `YUL`)?

A. `if`

B. `else`

C. `elif`

D. `switch`

Find the solution here.

16. `address` types

A. Can always receive Ether

B. Have members for `balance`, `call`, `code`

C. Can be converted to `uint160` or `contract` types

D. Can be added and subtracted

Find the solution here.

17. If the previous block number was 1000 on Ethereum mainnet, which of the following is/are true?

A. `block.number` is 1001

B. `blochhash(1)` returns 0

C. `block.chainID` returns 1

D. `block.timestamp` returns the number of seconds since last block

Find the solution here.

18. If we have an array then its data location can be

A. memory and its persistence/scope will be the function of declaration

B. storage and its persistence/scope will be the entire contract

C. calldata and it will only be readable

D. None of the above

Find the solution here.

19. `delete varName;` has which of the following effects?

A. `varName` becomes `0` if `varName` is an integer

B. `varName` becomes `true` if `varName` is a boolean

C. No effect if `varName` is a mapping

D. Resets all struct members to their default values irrespective of their types

Find the solution here.

20. Which of the following is/are valid function specifier(s)?

A. `internal`

B. `pure`

C. `payable`

D. `immutable`

Find the solution here.

21. Function visibility

A. Goes from `private-internal-external-public` in decreasing restrictive order (i.e. `private` being the most restrictive)

B. Goes from `internal-private-external-public` in decreasing restrictive order (i.e. `internal` being the most restrictive)

C. May be omitted to default to internal in the latest `0.8.0+` compiler versions

D. None of the above

Find the solution here.

22. For error handling

A. `require()` is meant to be used for input validation

B. `require()` has a mandatory error message string

C. `assert()` is meant to be used to check invariants

D. `revert()` will abort and revert state changes

Find the solution here.

23. Which of the following is/are true?

A. Constant state variables can be initialized within a constructor

B. Immutable state variables are allocated a storage slot

C. Gas costs for constant and immutable variables is lower

D. Only value types can be immutable

Find the solution here.

24. Integer overflows/underflows in `Solidity`

A. Are never possible because of the language design

B. Are possible but prevented by compiler added checks (version dependent)

C. Are possible but prevented by correctly using certain safe math libraries

D. Are possible without any mitigation whatsoever

Find the solution here.

25. Which of the following is true about mapping types in `mapping(_KeyType => _ValueType)`?

A. `_KeyType` can be any value or reference type

B. `_ValueType` can be any value or reference type

C. Can only have storage (not memory) as data location

D. Can be iterated over natively (i.e. without implementing another data structure)

Find the solution here.

26. `receive()` and `fallback()` functions

A. Can rely only on 2300 gas in the worst case

B. May receive Ether with `payable` mutability

C. Are mandatory for all contracts

D. Must have `external` visibility

Find the solution here.

27. In `Solidity`, `selfdestruct(address)`

A. Destroys the contract whose address is given as argument

B. Destroys the contract executing the selfdestruct

C. Sends address's balance to the calling contract

D. Sends executing contract's balance to the address

Find the solution here.

28. Which of the following is/are correct?

A. `Solidity` file with `pragma solidity ^0.6.5;` can be compiled with compiler version `0.6.6`

B. `Solidity` file with `pragma solidity 0.6.5;` can be compiled with compiler version `0.6.5`

C. `Solidity` file with `pragma solidity ^0.6.5;` can be compiled with compiler version `0.7.0`

D. `Solidity` file with `pragma solidity >0.6.5 <0.7.0;` can be compiled with compiler version `0.7.0`

Find the solution here.

29. The impact of data location of reference types on assignments is

A. storage assigned to storage (local variable) makes a copy

B. memory assigned to memory makes a copy

C. memory assigned to storage creates a reference

D. None of the above

Find the solution here.

30. Which of the below are value types?

A. `address`

B. `enum`

C. `struct`

D. `contract`

Find the solution here.

31. Arrays in `Solidity`

A. Can be fixed size or dynamic

B. Are zero indexed

C. Have `push`, `pop` and `length` members

D. None of the above

Find the solution here.

32. `Solidity` language is

A. Statically typed

B. Object-oriented

C. Supports inheritance

D. Supports inline assembly

Find the solution here.

9.3 Solidity 201 Quiz

1. `OpenZeppelin SafeERC20` is generally considered safer to use than `ERC20` because

A. It adds integer overflow/underflow checks

B. It adds return value/data checks

C. It adds pause/unpause capability

D. It adds race-conditon checks

Find the solution here.

2. The OpenZeppelin library that provides `onlyOwner` modifier

A. Is Ownable

B. Provides role based access control

C. Provides a function to renounce ownership

D. None of the above

Find the solution here.

3. `OpenZeppelin ECDSA`

A. Implements functions for signature creation & verification

B. Is susceptible to signature malleability

C. Both A & B

D. Neither A nor B

Find the solution here.

4. Which of the following is/are true about `Solidity` compiler `0.8.0`?

A. ABI coder v2 is made the default

B. No opt-out primitives for default checked arithmetic

C. Failing to `assert` returns the gas left instead of consuming all gas

D. Exponentiation is made right associative

Find the solution here.

5. EVM memory

A. Is linear and byte-addressable

B. Is reserved by `Solidity` until `0x7f`

C. Can be accessed in bytes using `MLOAD8`/`MSTORE8`

D. Is non-volatile or persistent

Find the solution here.

6. Dappsys provides

A. A proxy implementation

B. A floating-point implementation with `wad` & `ray`

C. A flexible authorization implementation

D. All of the above

Find the solution here.

7. Libraries are contracts

A. That cannot have state variables

B. That cannot be inherited

C. That always require a `delegatecall`

D. That are not meant to receive Ether

Find the solution here.

8. ERC20 `transferFrom(address sender, address recipient, uint256 amount)` (that follows the `ERC20` spec strictly)

A. Transfers token amount from `sender` to `recipient`

B. `sender` must have given caller (`msg.sender`) approval for at least `amount` or more

C. Deducts amount from `sender`'s allowance

D. Deducts amount from caller's (`msg.senders`'s) allowance

Find the solution here.

9. `ERC777` may be considered as an improved version of `ERC20` because

A. Hooks allow reacting to token `mint`/`burn`/`transfer`

B. It can help avoid separate `approve` and `transferFrom` transactions

C. It can help prevent tokens getting stuck in contracts

D. It removes reentrancy risk

Find the solution here.

10. WETH is

A. An `ERC20` pre-compile for Wrapped Ether built into Ethereum protocol

B. Warp Ether for super-fast Ether transfers

C. Wrapped Ether to convert Ether into an `ERC721` NFT

D. None of the above

Find the solution here.

11. `OpenZeppelin SafeCast`

A. Prevents underflows while downcasting

B. Prevents overflows while downcasting

C. Prevents underflows while upcasting

D. Prevents overflows while upcasting

Find the solution here.

12. `OpenZeppelin ERC20Pausable`

A. Adds ability to pause token transfers

B. Adds ability to pause token minting and burning

C. Provides modifiers `whenPaused` and `whenNotPaused`

D. None of the above

Find the solution here.

13. `CREATE2`

A. Deploys two contracts proxy and implementation concurrently

B. Deploys contract at an address that can be predetermined

C. Uses a salt and contract `creationCode`

D. None of the above

Find the solution here.

14. Name collision error with inheritance happens when the following pairs have the same name within a contract

A. Function & modifier

B. Function & event

C. Function & function

D. Event & modifier

Find the solution here.

15. `OpenZeppelin`**'s (role-based)** `AccessControl` **library**

A. Provides support only for two specific: `owner` and `user`

B. Provides support for different roles with different authorization levels

C. Provides support for granting and revoking roles

D. None of the above

Find the solution here.

16. `OpenZeppelin`**'s proxy implementations**

A. Typically have a proxy contract and an implementation contract

B. Use `delegatecall`s from proxy to implementation

C. Cannot support upgradable proxies

D. None of the above

Find the solution here.

17. Which of the following is/are true for a function `f` that has a modifier `m`?

A. Function `f` cannot have another modifier because every function function can have at most one modifier

B. Function `f`'s code is inlined at the point of '`_`' within modifer `m`

C. Function `f` reverts if '`_`' is not executed in the modifier `m`

D. None of the above

Find the solution here.

18. Zero address check is typically recommended because

A. The use of zero address for transfers will trigger an EVM exception

B. Ether/tokens sent to zero address will be inaccessible

C. Ether/tokens sent to zero address can be accessed by anyone

D. Address 0 is the Ethereum Masternode account and is forbidden for access

Find the solution here.

19. `Solidity` supports

A. Multiple inheritance

B. Polymorphism

C. Contract overloading

D. Function overloading

Find the solution here.

20. `OpenZeppelin ERC721`

A. Implements the NFT standard

B. `safeTransferFrom()` checks for zero-addresses

C. `approve()` is susceptible to race-condition just like `ERC20`

D. `setApprovalForAll(address operator, bool _approved)` approves/removes `operator` for all of caller's tokens

Find the solution here.

21. For `contract A {uint256 i; bool b1; bool b2; address a1;}` the number of storage slots used is:

A. 4

B. 3

C. 2

D. 1

Find the solution here.

22. Which of the following is/are generally true about storage layouts?

A. The number of slots used for a contract depends on the ordering of state variable declarations

B. The slots for struct elements are consecutive

C. The slot `s` for dynamic array contains the length with individual elements stored consecutively in slots starting at `keccak256(s)`

D. The slot `s` for mapping is empty with individual values stored consecutively in slots starting at `keccak(h(k).s)` where `k` is the first key and `h` is a hash function that depends on type of `k`

Find the solution here.

23. Assuming all contracts `C1`, `C2` and `C3` define explicit constructors in `contract C1 is C2, C3` and both `C2` and `C3` don't inherit contracts, the number & order of constructor(s) executed is/are

A. One, that of `C1`

B. Three, in the order `C2`, `C3`, `C1`

C. One, that of `C3`

D. Three, in the order `C1`, `C2`, `C3`

Find the solution here.

24. `OpenZeppelin SafeMath`

A. Prevents integer overflow/underflow at compile-time

B. Is not required if using `Solidity` compiler version $\geq$ `0.8.0`

C. Both A & B

D. Neither A nor B

Find the solution here.

25. Which of the following is/are not allowed?

A. Function overriding

B. Function overloading

C. Modifier overloading

D. Modifier overriding

Find the solution here.

26. Which of the following is/are true about abstract contracts and interfaces?

A. Abstract contracts have at least one function undefined

B. Interfaces can have some functions defined

C. Unimplemented functions in abstract contracts need to be declared virtual

D. All functions are implicitly virtual in interfaces

Find the solution here.

27. Storage layout

A. Refers to the layout of state variables in storage

B. Is organized in 256-byte slots

C. Is packed for value types that use less than 32 bytes

D. Always starts on a new slot for reference types

Find the solution here.

28. Which of the following EVM instruction(s) do(es) not touch EVM storage?

A. `SLOAD`

B. `MSTORE8`

C. `SSTORE`

D. `SWAP`

Find the solution here.

29. Proxied contracts

A. Should use constructors in implementation contract to initialize the proxy's state variables

B. Should use an external/public `initialize()` function

C. Should have their `initialize()` function called only once

D. All of the above

Find the solution here.

30. `OpenZeppelin`'s `ReentrancyGuard` library mitigates reentrancy risk in a contract

A. For all its functions by simply deriving/inheriting from it

B. Only for functions that apply the `nonReentrant` modifier

C. By enforcing a checks-effects-interactions pattern in its functions

D. None of the above

Find the solution here.

31. EVM inline assembly has

A. Its own language `Yul`

B. Safety checks just like `Solidity`

C. Access to all variables in the contract and function where present

D. References to variables as their addresses not values

Find the solution here.

32. If `OpenZeppelin`'s `isContract(address)` returns `false` for an address then

A. Address is guaranteed to not be a contract

B. Codesize at address is 0 at time of invocation

C. Both A & B

D. Neither A nor B

Find the solution here.

9.4 Security Pitfalls & Best Practices 101 Quiz

1. The use of `pragma` in the given contract snippet

```
pragma solidity ^0.6.0;

contract test {
   // Assume other required functionality is correctly implemented
   // Assume this contract can work correctly without modifications
        across 0.6.x/0.7.x/0.8.x compiler versions
}
```

A. Is incorrect and will cause a compilation error

B. Allows testing with `0.6.11` but accidental deployment with buggy `0.6.5`

C. Is illustrative of risks from using a much older Solidity version (assume current version is `0.8.9`)

D. None of the above

Find the solution here.

2. The given contract snippet has

```
pragma solidity 0.8.4;

contract test {

   // Assume other required functionality is correctly implemented

   function kill() public {
      selfdestruct(payable(0x0));
   }
}
```

A. Unprotected call to `selfdestruct` allowing anyone to destroy this contract

B. Dangerous use of zero address leading to burning of contract balance

C. A compiler error because of the use of the `kill` reserved keyword

D. None of the above

Find the solution here.

3. The given contract snippet has

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    modifier onlyAdmin() {
        // Assume this is correctly implemented
        _;
    }

    function transferFunds(address payable recipient, uint amount)
        public {
        recipient.transfer(amount);
   }
}
```

A. Missing use of `onlyAdmin` modifier on `transferFunds`

B. Missing `return` value check on `transfer`

C. Unprotected withdrawal of funds

D. None of the above

Find the solution here.

4. In the given contract snippet

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    mapping (uint256 => address) addresses;
    bool check;

    modifier onlyIf() {
        if (check) {
            _;
        }
    }

    function setAddress(uint id, address addr) public {
        addresses[id] = addr;
    }

    function getAddress(uint id) public onlyIf returns (address) {
        return addresses[id];
    }
}
```

A. `getAddress` returns the expected addresses if `check` is `true`

B. `getAddress` returns zero address if `check` is `false`

C. `getAddress` reverts if `check` is `false`

D. None of the above

Find the solution here.

5. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    modifier onlyAdmin {
     // Assume this is correctly implemented
        _;
    }

    function delegate (address addr) external {  addr.delegatecall(
        abi.encodeWithSignature("setDelay(uint256)"));
    }
}
```

A. Potential controlled `delegatecall` risk

B. `delegatecall` return value is not checked

C. `delegate()` may be missing `onlyAdmin` modifier

D. `delegate()` does not check for contract existence at `addr`

Find the solution here.

6. The vulnerability/vulnerabilities present in the given contract snippet is/are

```
pragma solidity 0.7.0;
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
// which works with 0.7.0

contract test {

 // Assume other required functionality is correctly implemented
 // For e.g. users have deposited balances in the contract
 // Assume nonReentrant modifier is always applied

    mapping (address => uint256) balances;

    function withdraw(uint256 amount) external nonReentrant {
        msg.sender.call{value: amount}("");
        balances[msg.sender] -= amount;
    }
}
```

A. Reentrancy

B. Integer underflow leading to wrapping

C. Missing check on user balance in `withdraw()`

D. All of the above

Find the solution here.

7. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

 // Assume other required functionality is correctly implemented

    uint256 private constant secret = 123;

    function diceRoll() external view returns (uint256) {
        return (((block.timestamp * secret) % 6) + 1);
    }
}
```

A. `diceRoll()` visibility should be `public` instead of `external`

B. The private variable `secret` is not really hidden from users

C. `block.timestamp` is an insecure source of randomness

D. Integer overflow

Find the solution here.

8. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented
    // Contract admin set to deployer in constructor (not shown)
    address admin;

    modifier onlyAdmin {
        require(tx.origin == admin);
        _;
    }

    function emergencyWithdraw() external payable onlyAdmin {
        msg.sender.transfer(address(this).balance);
    }
}
```

A. Incorrect use of `transfer()` instead of using `send()`

B. Potential man-in-the-middle attack on admin address authentication

C. Assumption on contract balance might cause a revert

D. Missing event for critical `emergencyWithdraw()` function

Find the solution here.

9. The given contract snippet is vulnerable because of

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    uint256 private constant MAX_FUND_RAISE = 100 ether;
    mapping (address => uint256) contributions;

    function contribute() external payable {
        require(address(this).balance != MAX_FUND_RAISE);
        contributions[msg.sender] += msg.value;
    }
}
```

A. Integer overflow leading to wrapping

B. Overly permissive function visibility of `contribute()`

C. Incorrect use of `msg.sender`

D. Use of strict equality (`!=`) may break the `MAX_FUND_RAISE` constraint

Find the solution here.

10. In the given contract snippet, the require check will

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    function callMe (address target) external {
        (bool success, ) = target.call("");
        require(success);
    }
}
```

A. Pass only if target is an existing contract address

B. Pass for a non-existent contract address

C. Pass always

D. Fail always

Find the solution here.

11. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented
    // Assume admin is set correctly to contract deployer in
        constructor
    address admin;

    function setAdmin (address _newAdmin) external {
        admin = _newAdmin;
    }
}
```

A. Missing access control on critical function

B. Missing zero-address validation

C. Single-step change of critical address

D. Missing event for critical function

Find the solution here.

12. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    address admin;
    address payable pool;

   constructor(address _admin) {
        admin = _admin;
    }

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function setPoolAddress(address payable _pool) external
        onlyAdmin {
        pool = _pool;
    }

    function addLiquidity() payable external {
        pool.transfer(msg.value);
    }
}
```

A. Uninitialized pool storage variable which assumes `setPoolAddress()` will be called before `addLiquidity()`

B. Incorrect use of modifier `onlyAdmin` on `setPoolAddress()`

C. Missing zero-address validation for `_pool` in `setPoolAddress()`

D. Transaction order dependence risk from admin front-running with pool address change

Find the solution here.

13. The security concern(s) in the given proxy-based implementation contract snippet is/are

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/proxy/utils/Initializable.sol";

contract test is Initializable {

    // Assume other required functionality is correctly implemented

    address admin;
    uint256 rewards = 10;

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function initialize (address _admin) external {
        require(_admin != address(0));
        admin = _admin;
    }

    function setRewards(uint256 _rewards) external onlyAdmin {
        rewards = _rewards;
    }
}
```

A. Imported contracts are not upgradeable

B. Multiple `initialize()` calls possible which allows anyone to reset the admin

C. rewards will be 0 in the proxy contract before `setRewards()` is called by it

D. All the above

Find the solution here.

14. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob
    /master/contracts/token/ERC20/IERC20.sol";

contract test {

    // Assume other required functionality is correctly implemented

    address admin;
    address token;

    constructor(address _admin, address _token) {
        require(_admin != address(0));
        require(_token != address(0));
        admin = _admin;
        token = _token;
    }

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function payRewards(address[] calldata recipients, uint256[]
        calldata amounts) external onlyAdmin {
        for (uint i; i < recipients.length; i++) {
            IERC20(token).transfer(recipients[i], amounts[i]);
        }
    }
}
```

A. Potential out-of-gas exceptions due to unbounded external calls within loop

B. `ERC20 approve()` race condition

C. Unchecked return value of `transfer()` (assuming it returns a boolean/other value and does not revert on failure)

D. Potential reverts due to mismatched lengths of recipients and amounts arrays

Find the solution here.

15. The vulnerability/vulnerabilities present in the given contract snippet is/are

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob
    /master/contracts/security/ReentrancyGuard.sol";

contract test {
```

```
5
6  // Assume other required functionality is correctly implemented
7  // For e.g. users have deposited balances in the contract
8
9      mapping (address => uint256) balances;
10
11     function withdrawBalance() external {
12         msg.sender.call{value: balances[msg.sender]}("");
13         balances[msg.sender] = 0;
14     }
15 }
```

A. Reentrancy

B. Integer overflow leading to wrappingProvides support for different roles with different authorization levels

C. Integer underflow leading to wrapping

D. None of the above

Find the solution here.

16. The security concern(s) in the given contract snippet is/are

```
1  pragma solidity 0.8.4;
2
3  contract test {
4  // Assume other required functionality is correctly implemented
5  // Assume that hash is the hash of a message computed elsewhere in
       contract
6  // Assume that the contract does not make use of chainID or nonces
       in its logic
7
8  function verify(address signer, bytes32 memory hash, bytes32 sigR,
       bytes32 sigS, uint8 sigV) internal view returns (bool) {
9     return signer == ecrecover(hash, sigV, sigR, sigS);
10 }
```

A. Signature malleability risk of `ecrecover`

B. Missing use of nonce in message hash may allow replay attacks across transactions

C. Missing use of `chainID` in message hash may allow replay attacks across chains

D. Missing zero-address check for `ecrecover` return value may allow invalid signatures

Find the solution here.

9.5 Security Pitfalls & Best Practices 201 Quiz

All questions in this quiz are based on the `InSecureumToken` contract shown below.
This is the same contract you see for all the 8 questions in this quiz.
The `InSecureumToken` contract implements a token contract which allows users to buy tokens by depositing Ether (at a certain conversion ratio), and transfer tokens.

```
pragma solidity 0.7.0;

contract InSecureumToken {

    mapping(address => uint) private balances;

    uint public decimals = 10**18; // decimals of the token
    uint public totalSupply; // total supply
    uint MAX_SUPPLY = 100 ether; // Maximum total supply

    event Mint(address indexed destination, uint amount);

    function transfer(address to, uint amount) public {
        // save the balance in local variables
        // so that we can re-use them multiple times
        // without paying for SLOAD on avery access
        uint balance_from = balances[msg.sender];
        uint balance_to = balances[to];
        require(balance_from >= amount);
        balances[msg.sender] = balance_from - amount;
        balances[to] = safeAdd(balance_to, amount);
    }

    /// @notice Allow users to buy token. 1 ether = 10 tokens
    /// @dec Users can send more ether than token to be bought, to
        donate a fee to the protocol team
    function buy (uint desired_tokens) public payable {
        // Check if enough ether has been sent
        uint required_wei_sent = (desired_tokens / 10) * decimals;
        require(msg.value >= required_wei_sent);

        // Mint tokens
        totalSupply = safeAdd(totalSupply, desired_tokens);
        balances[msg.sender] = safeAdd(balances[msg.sender],
            desired_tokens);
        emit Mint(msg.sender, desired_tokens);
    }

    /// @notice Add two values. Revert if overflow
    function safeAdd(uint a, uint b) pure internal returns(uint) {
        if (a + b < a) {
            revert();
        }
        return a + b;
    }
}
```

1. The `InSecureumToken` contract strictly follow the specification of

A. `ERC20`.

B. `ERC777`.

C. `ERC721`.

D. None of the above.

Find the solution here.

2. To avoid lock of funds, the following feature(s) MUST be implemented before contract deployment

A. A `transferFrom()` function.

B. A `burn()` function.

C. A way to withdraw/exchange/use Ether from the contract.

D. None of the above.

Find the solution here.

3. Which of the following assertion(s) is/are true (without affecting the security posture of the contract)?

A. `buy()` does not need `payable` keyword.

B. `balances` must be `private`.

C. `transfer()` can be `external`.

D. `safeAdd()` can be `public`.

Find the solution here.

4. The total supply is limited by

A. `10**18`.

B. `100 * 10**18`.

C. `100`.

D. None of the above.

Find the solution here.

5. The following issue(s) is/are present in the codebase

A. An integer underflow allows one to drain Ether.

B. Unsafe rounding allows one to receive new tokens for free.

C. A division by zero allows one to trap/freeze the system.

D. None of the above.

Find the solution here.

6. The following issue(s) is/are present in the codebase

A. A front-running allows one to pay less than expected for tokens.

B. A lack of access control allows one to receive tokens for free.

C. Incorrect balance update allows one to receive new tokens for free.

D. None of the above.

Find the solution here.

7. The following issue(s) is/are present in the codebase

A. A reentrancy allows one to drain Ether.

B. A reentrancy allows one to drain the tokens.

C. A reentrancy allows one to receive new tokens for free.

D. None of the above.

Find the solution here.

8. The following issue(s) is/are present in the codebase

A. An integer overflow allows one to drain Ether.

B. An integer overflow allows one to receive new tokens for free.

C. An integer overflow allows one to trap/freeze the system.

D. None of the above.

Find the solution here.

9.6 Audit Techniques & Tools 101 Quiz

1. Which of the below is/are accurate?

A. Audits identify all security vulnerabilities and guarantee bug-free code.

B. Audits cover only smart contracts but never the offchain code.

C. Audits suggest fixes for issues identified and aim to reduce risk.

D. None of the above.

Find the solution here.

2. Audit reports from audit firms tipically include

A. Finding likelihood/difficulty, impact and severity.

B. Exploit scenarios and recommended fixes.

C. Formal verification of all findings with proofs and counterexamples.

D. All of the above.

Find the solution here.

3. These audit techniques are especially well-suited for smart contracts (compared to Web2 programs)

A. Formal verification because contracts are relatively smaller with specific properties.

B. Fuzzing because anyone can send random inputs to contracts on blockchain.

C. Static source-code analysis because contracts are expected to be open source.

D. High-coverage testing because contract states and transitions are relatively fewer.

Find the solution here.

4. The following kinds of findings may be expected during audits

A. True positives after confirmation from the project team.

B. False positives due to assumptions from missing specification and threat model.

C. False positives due to limitations of time and expertise.

D. None of the above.

Find the solution here.

5. Which of the following is/are true?

A. Audited projects always have clear/complete specification and documentation of all contract properties.

B. Manual analysis is typically required for detecting application logic vulnerabilities.

C. Automated tools like Slither and MythX have no false negatives.

D. The project team always fixes all the findings identified in audits.

Find the solution here.

6. Automated tools for smart contract analysis

A. Are sufficient therefore making manual analysis unnecessary.

B. Have no false positives whatsoever.

C. Are best-suited for application-level vulnerabilities.

D. None of the above.

Find the solution here.

7. Which of the following is/are true?

A. Slither supports detectors, printers, tools and custom analyses.

B. Echidna is a symbolic analyzer tool.

C. MythX is a combination of static analysis, symbolic checking and fuzzing tools.

D. None of the above.

Find the solution here.

8. Which of the following is/are correct about false positives?

A. They are findings that are not real concerns/vulnerabilities after further review.

B. They are real vulnerabilities but are falsely claimed by auditors as benign.

C. They are possible with automated tools.

D. None of the above.

Find the solution here.

9. Audit findings

A. May include both specific vulnerabilities and generic recommendations.

B. May not all be fixed by the project team for reasons of relevancy and acceptable trust/threat model.

C. Always have demonstrable proof-of-concept excploit code on mainnet.

D. None of the above.

Find the solution here.

10. Which of the following is/are typicial manual review approach(es)?

A. Asset flow.

B. Symbolic checking.

C. Inferring constraints.

D. Evaluating assumptions.

Find the solution here.

11. Access control analysis is a crytical part of manual review for the reason(s) that

A. It is the easiest to perform because smart contracts never have access control

B. It is the fastest to perform because there are always only two roles: users and admins.

C. It is fundamental to security because privileged roles (of which there may be many) may be misused/compromised.

D. None of the above.

Find the solution here.

12. Which of the following is/are true about vulnerability difficulty and impact?

A. Difficulty indicates how hard it was for auditors to detect the issue.

B. Difficulty is an objective measure that can always be quantified.

C. Impact is tipically classified as High if there is loss/lock of funds.

D. None of the above.

Find the solution here.

13. Application-level security constraints

A. Are always clearly/completely specified and documented.

B. Have to be typically inferred from the code or discussions with project team.

C. Typically require manual analysis.

D. None of the above.

Find the solution here.

14. Which of the following is/are typically true?

A. Static analysis analyzes program properties by actually executing the program.

B. Fuzzing uses valid, expected and deterministic inputs.

C. Symbolic checking enumerates individual states/transitions or efficient state space traversal.

D. None of the above.

Find the solution here.

15. Which of the following is/are generally true about asset flow analysis?

A. Analyzes the flow of Ether or tokens managed by smart contracts.

B. Assets should be withdrawn only by authorized addresses.

C. The timing aspects of asset withdrawals/deposits is irrelevant.

D. The type and quantity of asset withdrawals/deposits is irrelevant.

Find the solution here.

16. Which of the following is/are generally true about control and data flow analyses?

A. Interprocedural control flow is typically indicated by a call graph.

B. Intraprocedural control flow is dictated by conditionals (if/else), loops (for/while/do/continue/break) and return statements.

C. Interprocedural data flow is evaluated by analyzing the data used as argument values for function parameters at call sites.

D. Intraprocedural data flow is evaluated by analyzing the assignment and use of variables/constants along control flow paths within functions

Find the solution here.

9.7 Audit Findings 101 Quiz

All questions in this quiz are based on the `InSecureumDAO` contract snippet. This is the same contract snippet you will see for all the 8 questions in this quiz. The `InSecureumDAO` contract snippet illustrates some basic functionality of a Decentralized Autonomous Organization (DAO) which includes the opening of the DAO for memberships, allowing users to join as members by depositing a membership fee, creating proposals for voting, casting votes, etc... Assume that all the other functionality (that is not shown or represented by "...") is implemented correctly.

```
pragma solidity 0.8.4;
import 'https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol';
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/Pausable.sol";

contract InSecureumDAO is Pausable, ReentrancyGuard{

    // Assume that all functonality represented by ... below is implemented as expected

    address public admin;
    mapping (address => bool) public members;
    mapping (uint256 => uint8[]) public votes;
    mapping (uint256 => uint8) public winningOutcome;
    uint256 memberCount = 0;
    uint256 membershipFee = 1000;

    modifier onlyWhenOpen() {
        require(address(this).balance > 0, 'InSecureumDAO: This DAO is closed');
        _;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin);
        _;
    }

    modifier voteExists(uint256 _voteId) {
        // Assume this correctly checks if _voteId is present in votes
        // ...

        _;
    }

    constructor (address _admin) {
        require(_admin == address(0));
        admin = _admin;

    }

    function openDAO() external payable onlyAdmin {
```

```
40          // Admin is expected to open DAO by making a notional
                deposit
41          // ...
42      }
43
44      function join() external payable onlyWhenOpen nonReentrant {
45          require(msg.value == membershipFee, 'InSecureumDAO:
                Incorrect ETH amount');
46          members[msg.sender] = true;
47          // ...
48      }
49
50      function createVote(uint256 _voteId, uint8[] memory
            _possibleOutcomes) external onlyWhenOpen whenNotPaused {
51          votes[_voteId] = _possibleOutcomes;
52          // ...
53      }
54
55      function castVote(uint256 _voteId, uint8 _vote) external
            voteExists(_voteId) onlyWhenOpen whenNotPaused {
56          // ...
57      }
58
59      function getWinningOutcome(uint256 _voteId) public view returns
             (uint8) {
60          // Anyone is allowed to view winning outcome
61          // ...
62          return(winningOutcome[_voteId]);
63      }
64
65      function setMembershipFee(uint256 _fee) external onlyAdmin {
66          membershipFee = _fee;
67      }
68
69      function removeAllMembers() external onlyAdmin {
70          delete members[msg.sender];
71      }
72  }
```

1. Based on the comments and code shown in the `InSecureumDAO` snippet

A. The DAO is meant to be opened only by the `admin` by making an Ether deposit to the contract.

B. DAO can be opened by anyone by making an Ether deposit to the contract.

C. DAO requires an exact payment of `membershipFee` to join the DAO.

D. None of the above.

Find the solution here.

2. Based on the comments and code shown in the `InSecureumDAO` snippet

A. Guarded launch via circuit breakers has been implemented correctly for all state modifying functions.

B. Zero-address check(s) has/have been implemented correctly.

C. All critical privileged-role functions have events emitted.

D. None of the above.

Find the solution here.

3. Reentrancy protection only on `join()` (assume it's correctly specified) indicates that

A. Only `payable` functions require this protection because of handling `msg.value`.

B. `join()` likely makes untrusted external call(s) but not the other contract functions.

C. Both A and B.

D. Neither A nor B.

Find the solution here.

4. Access control on `msg.sender` for DAO membership is required in

A. `createVote()` to prevent non-members from creating votes.

B. `castVote()` to prevent non-members from casting votes.

C. `getWinningOutcome()` to prevent non-members from viewing winning outcomes.

D. None of the above.

Find the solution here.

5. A commit/reveal scheme (a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden from others, with the ability to reveal the committed value later) is relevant for

A. `join()` to not disclose `msg.sender` while joining the DAO.

B. `createVote()` to not disclose the possible outcomes during creation.

C. `castVote()` to not disclose the vote being cast.

D. All the above.

Find the solution here.

6. Security concern(s) from missing input validation(s) is/are present in

A. `createVote()` for duplicate `_voteId`.

B. `castVote()` for existing `_voteId`.

C. `getWinningOutcome()` for existing `_voteId`.

D. `setMembershipFee()` for sanity/threshold checks on `_fee`.

Find the solution here.

7. `removeAllMembers()` function

A. Will not work as expected to remove all the members from the DAO.

B. Will work as expected to remove all the members from the DAO.

C. Is a critical function missing an event emission.

D. None of the above.

Find the solution here.

8. `InSecureumDAO` will *not* be susceptible to something like the 2016 "DAO exploit"

A. Because it derives from `ReentrancyGuard.sol` which protects all contract functions by default.

B. Only if it does not have a withdraw Ether function vulnerable to reentrancy and makes no external calls.

C. Because Ethereum protocol was fixed after the DAO exploit to prevent such exploits.

D. Because Solidity language was fixed after the DAO exploit to prevent such exploits.

Find the solution here.

9.8 Audit Findings 201 Quiz

All questions in this quiz are based on the `InSecureumNFT` contract snippet. `InSecureumNFT` is a `NFT` project that aims to distribute `CrytptoSAFU NFTs` to its community where most of them are fairdropped based on past contributions and a few are sold. `CryptoSAFUs` with lower `IDs` have more unqiue traits, may be valued higher and therefore require a random distribution for fairness. Assume thar all strictly required `ERC721` functionaity (not shown) and any other required functionality (not shown) are implemented correctly. Only functionality specific to the sale and minting of NFTs is shown in this contract snippet.

```
pragma solidity 0.8.0;

interface ERC721TokenReceiver{function onERC721Received(address
    _operator, address _from, uint256 _tokenId, bytes calldata
    _data) external returns(bytes4);}

// Assume that all strictly required ERC721 functionality (not
    shown) is implemented correctly
// Assume that any other required functionality (not shown) is
    implemented correctly
contract InSecureumNFT {
    bytes4 internal constant MAGIC_ERC721_RECEIVED = 0x150b7a02;
    uint public constant TOKEN_LIMIT = 10; // 10 for testing, 13337
         for production
    uint public constant SALE_LIMIT = 5; // 5 for testing, 1337 for
         production

    mapping (uint256 => address) internal idToOwner;
    uint internal numTokens = 0;
    uint internal numSales = 0;
    address payable internal deployer;
    address payable internal beneficiary;
    bool public publicSale = false;
    uint private price;
    uint public saleStartTime;
    uint public constant saleDuration = 13*13337; // 13337 blocks
        assuming 13s block times
    uint internal nonce = 0;
    uint[TOKEN_LIMIT] internal indices;

    constructor(address payable _beneficiary) {
        deployer = payable(msg.sender);
        beneficiary = _beneficiary;
    }

    function startSale(uint _price) external {
        require(msg.sender == deployer || _price != 0, "Only
            deployer and price cannot be zero");
        price = _price;
        saleStartTime = block.timestamp;
        publicSale = true;
    }

    function isContract(address _addr) internal view returns (bool
        addressCheck) {
```

```
37          uint256 size;
38          assembly { size := extcodesize(_addr) }
39          addressCheck = size > 0;
40      }
41
42      function randomIndex() internal returns (uint) {
43          uint totalSize = TOKEN_LIMIT - numTokens;
44          uint index = uint(keccak256(abi.encodePacked(nonce, msg.
                sender, block.difficulty, block.timestamp))) %
                totalSize;
45          uint value = 0;
46          if (indices[index] != 0) {
47              value = indices[index];
48          } else {
49              value = index;
50          }
51          if (indices[totalSize - 1] == 0) {
52              indices[index] = totalSize - 1;
53          } else {
54              indices[index] = indices[totalSize - 1];
55          }
56          nonce += 1;
57          return (value + 1);
58      }
59
60      // Calculate the mint price
61      function getPrice() public view returns (uint) {
62          require(publicSale, "Sale not started.");
63          uint elapsed = block.timestamp - saleStartTime;
64          if (elapsed > saleDuration) {
65              return 0;
66          } else {
67              return ((saleDuration - elapsed) * price) /
                    saleDuration;
68          }
69      }
70
71      // SALE_LIMIT is 1337
72      // Rest i.e. (TOKEN_LIMIT - SALE_LIMIT) are reserved for
            community distribution (not shown)
73      function mint() external payable returns (uint) {
74          require(publicSale, "Sale not started.");
75          require(numSales < SALE_LIMIT, "Sale limit reached.");
76          numSales++;
77          uint salePrice = getPrice();
78          require((address(this)).balance >= salePrice, "Insufficient
                funds to purchase.");
79          if ((address(this)).balance >= salePrice) {
80              payable(msg.sender).transfer((address(this)).balance -
                    salePrice);
81          }
82          return _mint(msg.sender);
83      }
84
85      // TOKEN_LIMIT is 13337
86      function _mint(address _to) internal returns (uint) {
87          require(numTokens < TOKEN_LIMIT, "Token limit reached.");
```

```
88          // Lower indexed/numbered NFTs have rare traits and may be
                considered
89          // as more valuable by buyers => Therefore randomize
90          uint id = randomIndex();
91          if (isContract(_to)) {
92              bytes4 retval = ERC721TokenReceiver(_to).
                    onERC721Received(msg.sender, address(0), id, "");
93              require(retval == MAGIC_ERC721_RECEIVED);
94          }
95          require(idToOwner[id] == address(0), "Cannot add, already
                owned.");
96          idToOwner[id] = _to;
97          numTokens = numTokens + 1;
98          beneficiary.transfer((address(this)).balance);
99          return id;
100     }
101 }
```

1. Missing zero-address check(s) in the contract

A. May allow anyone to start the sale.

B. May put the NFT sale proceeds at risk.

C. May burn the newly minted NFTs.

D. None of the above.

Find the solution here.

2. Given that lower indexed/numbered `CryptoSAFU` NFTs have rarer traits (and are considered more valuable as commented in `_mint`), the implementation of `InSecureumNFT` is susceptible to the following exploits

A. Buyers can repeatedly mint and revert until they receive desired NFT.

B. Buyers can generate addresses to mint until they receive desired NFT.

C. Miners can manipulate `block.timestamp` to facilitate minting of desired NFT.

D. None of the above.

Find the solution here.

3. The `getPrice()` function

A. Is expected to reduce the mint price over time after sale starts.

B. Allows free mints after ĩ3337 blocks from when `startSale()` is called.

C. Visibility should be changed to `external`.

D. None of the above.

Find the solution here.

4. `InSecureumNFT` contract is

A. Not susceptible to reentrancy given the absence of external contract calls.

B. Not susceptible to integer overflow/wrapping given the compiler version used and the absence of `unchecked` blocks.

C. Susceptible to reentrancy during minting.

D. Perfectly safe for production.

Find the solution here.

5. Assuming `InSecureumNFT` contract is deployed in production (i.e. live for users) on mainnet without any changes to shown code

A. Use of evident test configuration wil cause fewer NFTs to be minted than expected in production.

B. Illustrates the lack of best-practice for test parametrization to be removed or kept separate from production code.

C. It will behave as documented in code to mint the expected number of NFTs in production.

D. None of the above.

Find the solution here.

6. The function `startSale()`

A. May be succesfully called/executed by anyone.

B. May be succesfully called/executed with `_price` of 0.

C. Must be called for minting to happen succesfully.

D. None of the above.

Find the solution here.

7. The minting of NFTs

A. Requires an exact amount of ETH to be paid by the buyer.

B. Refunds excess ETH paid by buyer back to the buyer.

C. Transfers the NFT `salePrice` to the `beneficiary` address.

D. May be optimized to prevent any zero ETH transfers in its refund mechanism.

Find the solution here.

8. The NFT sale

A. May be restarted by anyone any number of times.

B. Can be started exactly once by `deployer`.

C. Is missing an additional check on `publicSale`.

D. Is missing an event emit in `startSale`.

Find the solution here.

9.9 RACE 4 Quiz

All 8 questions in this quiz are based on the `InSecureum` contract. This is the same contract you will see for all the 8 questions in this quiz. `InSecureum` is adapted from a widely used `ERC20` contract.

```
pragma solidity 0.8.10;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Context.sol";

contract InSecureum is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;
    uint256 private _totalSupply;
    string private _name;
    string private _symbol;

    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    function name() public view virtual override returns (string memory) {
        return _name;
    }

    function symbol() public view virtual override returns (string memory) {
        return _symbol;
    }

    function decimals() public view virtual override returns (uint8) {
        return 8;
    }

    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view virtual override returns (uint256) {
        return _balances[account];
    }

    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
```

```
42      }
43
44      function allowance(address owner, address spender) public view
            virtual override returns (uint256) {
45          return _allowances[owner][spender];
46      }
47
48      function approve(address spender, uint256 amount) public
            virtual override returns (bool) {
49          _approve(_msgSender(), spender, amount);
50          return true;
51      }
52
53      function transferFrom(
54          address sender,
55          address recipient,
56          uint256 amount
57      ) public virtual override returns (bool) {
58          uint256 currentAllowance = _allowances[_msgSender()][sender
                ];
59          if (currentAllowance != type(uint256).max) {
60              unchecked {
61                  _approve(sender, _msgSender(), currentAllowance -
                        amount);
62              }
63          }
64          _transfer(sender, recipient, amount);
65          return true;
66      }
67
68      function increaseAllowance(address spender, uint256 addedValue)
             public virtual returns (bool) {
69          _approve(_msgSender(), spender, _allowances[_msgSender()][
                spender] + addedValue);
70          return true;
71      }
72
73      function decreaseAllowance(address spender, uint256
            subtractedValue) public virtual returns (bool) {
74          uint256 currentAllowance = _allowances[_msgSender()][
                spender];
75          require(currentAllowance > subtractedValue, "ERC20:
                decreased allowance below zero");
76          _approve(_msgSender(), spender, currentAllowance -
                subtractedValue);
77          return true;
78      }
79
80      function _transfer(
81          address sender,
82          address recipient,
83          uint256 amount
84      ) internal virtual {
85          require(sender != address(0), "ERC20: transfer from the
                zero address");
86          require(recipient != address(0), "ERC20: transfer to the
                zero address");
```

```
87          uint256 senderBalance = _balances[sender];
88          require(senderBalance >= amount, "ERC20: transfer amount
                exceeds balance");
89          unchecked {
90              _balances[sender] = senderBalance - amount;
91          }
92          _balances[recipient] += amount;
93          emit Transfer(sender, recipient, amount);
94      }
95
96      function _mint(address account, uint256 amount) external
            virtual {
97          _totalSupply += amount;
98          _balances[account] = amount;
99          emit Transfer(address(0), account, amount);
100     }
101
102     function _burn(address account, uint256 amount) internal
            virtual {
103         require(account != address(0), "ERC20: burn from the zero
                address");
104         require(_balances[account] >= amount, "ERC20: burn amount
                exceeds balance");
105         unchecked {
106             _balances[account] = _balances[account] - amount;
107         }
108         _totalSupply -= amount;
109         emit Transfer(address(0), account, amount);
110     }
111
112     function _approve(
113         address owner,
114         address spender,
115         uint256 amount
116     ) internal virtual {
117         require(spender != address(0), "ERC20: approve from the
                zero address");
118         require(owner != address(0), "ERC20: approve to the zero
                address");
119         _allowances[owner][spender] += amount;
120         emit Approval(owner, spender, amount);
121     }
122 }
```

1. `InSecureum` **implements**

A. Atypical `decimals` value

B. Non-standard `decreaseAllowance` and `increaseAllowance`

C. Non-standard `transfer`

D. None of the above

Find the solution here.

2. In `InSecureum`

A. `decimals()` can have `pure` state mutability instead of `view`

B. `_burn()` can have `external` visibility instead of `internal`

C. `_mint()` should have `internal` visibility instead of `external`

D. None of the above

Find the solution here.

3. `InSecureum transferFrom()`

A. Is susceptible to an integer underflow

B. Has an incorrect allowance check

C. Has an optimisation indicative of unlimited approvals

D. None of the above

Find the solution here.

4. In `InSecureum`

A. `increaseAllowance` is susceptible to an integer overflow

B. `decreaseAllowance` is susceptible to an integer overflow

C. `decreaseAllowance` does not allow reducing allowance to zero

D. `decreaseAllowance` can be optimised with `unchecked{}`

Find the solution here.

5. `InSecureum _transfer()`

A. Is missing a zero-address validation

B. Is susceptible to an integer overflow

C. Is susceptible to an integer underflow

D. None of the above

Find the solution here.

6. `InSecureum _mint()`

A. Is missing zero-address validation

B. Has an incorrect event emission

C. Has an incorrect update of account balance

D. None of the above

Find the solution here.

7. `InSecureum _burn()`

A. Is missing a zero-address validation

B. Has an incorrect event emission

C. Has an incorrect update of account balance

D. None of the above

Find the solution here.

8. `InSecureum _approve()`

A. Is missing a zero-address validation

B. Has incorrect error messages

C. Has an incorrect update of allowance

D. None of the above

Find the solution here.

9.10 RACE-X: Certora Quiz

This RACE-X is composed of 24 questions split into 3 sections:

- Part 1: Propositional Logic (Q1-Q10)
- Part 2: Properties of DeFi Systems (Q11-Q22)
- Part 3: Questions on (earlier shared) Certora Video (Q23-Q24)

This RACE-X uses well known logic symbols as shown in the below table:

Symbol	Free Language	Logic Meaning
$\wedge$	And	And
$\neg$	Not	Not
$\vee$	Or	Or
$\Rightarrow$	Imply/Implies	Implication
$\Leftrightarrow$	Iff/Equivalent	Equivalency

1. When is the expression $p \Rightarrow q$ false?

A. $\neg p \wedge \neg q$

B. $p \wedge q$

C. $\neg p \wedge q$

D. None of the above

Find the solution here.

2. Is the following expression true? $p \wedge q \Rightarrow p$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

3. Is the following expression true? $(p \wedge (q \vee \neg p)) \wedge \neg q$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

4. Is the following expression true? $\neg(\neg p \vee \neg q \vee \neg r) \Leftrightarrow p \wedge q \wedge r$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

5. Is the following expression true? $\neg((\neg p \vee \neg q) \vee (\neg p \wedge \neg q))$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

6. Is the following expresison true? $\neg((p \wedge q) \vee (\neg p \wedge \neg q))$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

7. Given the expression: $M : p \vee \neg p$
Which of the following expressions implies the above given expression M**?**

A. $p \vee \neg p$

B. $p \wedge \neg p$

C. $(p \wedge \neg p) \vee \neg p$

D. $(\neg p \vee p) \wedge \neg p$

Find the solution here.

8. Given the expression: $M : p \wedge \neg p$
Which of the following expressions implies the above given expression M**?**

A. $p \vee \neg p$

B. $p \wedge \neg p$

C. $(p \wedge \neg p) \vee \neg p$

D. $(\neg p \vee p) \wedge \neg p$

Find the solution here.

9. Is the following expression true? $(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow r)$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

10. Is the following expression true? $(()p \Rightarrow q) \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$

A. True in all cases

B. False in all cases

C. True for some cases, false for others

D. None of the above

Find the solution here.

11. Given the below four properties for OpenZeppelin's ERC20 implementation, answer questions 11 to 13

P1: Forall user: `balanceOf(user)` can only change on `mint()`, `burn()`, `transfer()`, `transferFrom()`
P2: `totalSupply()` is the sum of `balanceOf()` over all users
P3: Forall user: `balanceOf(user) <= totalSupply()`
P4: Forall user: `balanceOf(user)` can only change on operation performed when `msg.sender == user` or when `allowance(user, msg.sender)` is not zero

Which of the below properties are correct properties of `ERC20`?

A. Property P1

B. Property P2

C. Property P3

D. Property P4

Find the solution here.

12. Given the four properties for OpenZeppelin's ERC20 implementation shown in question 11,
and given the following buggy version of transferFrom:

```
function transferFrom(address _from, address _to, uint256 _value)
    public returns (bool) {
   require(_to != address(0));
   require(_value <= balances[_from]);
   require(_value <= allowed[_from][msg.sender]);
   balances[_from] = balances[_from].add(_value);
   balances[_to] = balances[_to].sub(_value);
   allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(
       _value);
   emit Transfer(_from, _to, _value);
   return true;
}
```

Which of the below properties are violated?

A. Property P1

B. Property P2

C. Property P3

D. None of the above

Find the solution here.

13. Given the four properties for OpenZeppelin's ERC20 implementation shown in question 11,
Is there an implication between P3 and P2?

A. Yes, P2 $\Rightarrow$ P3

B. Yes, P3 $\Rightarrow$ P2

C. Yes, P2 $\Leftrightarrow$ P3

D. No

Find the solution here.

14. Assuming a correct implementation of `transferFrom`, as in OpenZeppelin's ERC20 implementation, and given the following pseudo unit-test code

```
uint256  bFrom = balances[from];
uint256 bTo = balances[to];
transferFrom(from, to, x);
assert($exp$);
```

Which of the following expressions are valid choices (should always hold) for `$exp$`?

A. `balances[from] ==  bFrom - x`

B. `from != to  ||  balances[to] == bTo`

C. `balances[to] + balances[from] == bFrom + bTo`

D. None of the above

Find the solution here.

15. The below contract (same contract for questions 15 to 17) has a bug:

```
contract test {
   // Assume other required functionality is correctly implemented
   uint256 private constant MAX_FUND_RAISE = 100 ether;
   mapping (address => uint256) contributions;

    function contribute() external payable {
       require(address(this).balance != MAX_FUND_RAISE);
       contributions[msg.sender] += msg.value;
   }
}
```

Which of the following invariants should hold on a correct implementation of the contribute function?

P1: Forall user: `address(this).balance <= contributions[user]`
P2: Forall user: `contributions[user] <= address(this).balance`
P3: Forall user: `contributions[user] <= MAX_FUND_RAISE`
P4: `address(this).balance <= MAX_FUND_RAISE`

A. P1

B. P2

C. P3

D. P4

Find the solution here.

16. Following the same contract of question 15, which of the following invariant(s) that is/are supposed to hold is/are violated due to the buggy implementation?
P1: Forall user: `address(this).balance <= contributions[user]`
P2: Forall user: `contributions[user] <= address(this).balance`
P3: Forall user: `contributions[user] <= MAX_FUND_RAISE`
P4: `address(this).balance <= MAX_FUND_RAISE`

A. P1

B. P2

C. P3

D. P4

Find the solution here.

17. Following the same contract of question 15, the revert characteristic (conditions in which the function should revert) of a correct implementation of `contribute` is

A. `msg.value == 0`

B. `address(this).balance + msg.value > MAX_FUND_RAISE`

C. `msg.value == MAX_FUND_RAISE`

D. None of the above

Find the solution here.

18. In the below contract (same contract for questions 18 & 19)

```
pragma solidity 0.7.0;

contract InSecureumToken {

mapping(address => uint) private balances;
uint public decimals = 10**18; // decimals of the token
uint public totalSupply; // total supply
uint MAX_SUPPLY = 100 ether; // Maximum total supply
event Mint(address indexed destination, uint amount);

function balanceOf(address u) public returns (uint256) {
   return balances[u];
}

function ethBalance(address u) public returns (uint256) {
   return u.balance;
}

function transfer(address to, uint amount) public {
   // save the balance in local variables
   // so that we can re-use them multiple times
   // without paying for SLOAD on every access
   uint balance_from = balances[msg.sender];
   uint balance_to = balances[to];
   require(balance_from >= amount);
   balances[msg.sender] = balance_from - amount;
   balances[to] = safeAdd(balance_to, amount);
}

/// @notice Allow users to buy a token. 1 ether = 10 tokens
/// @dev Users can send more ether than token to be bought, to
    donate a fee to the protocol team.
function buy(uint desired_tokens) public payable {
    // Check if enough ether has been sent
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(msg.value >= required_wei_sent);
    // Mint the tokens
    totalSupply = safeAdd(totalSupply, desired_tokens);
    balances[msg.sender] = safeAdd(balances[msg.sender],
        desired_tokens);
```

```
39      emit Mint(msg.sender, desired_tokens);
40  }
41
42  /// @notice Add two values. Revert if overflow
43  function safeAdd(uint a, uint b) pure internal returns(uint) {
44      if (a + b < a) {
45          revert();
46      }
47      return a + b;
48  }
49  }
```

Given the following two properties:
P1: `totalSupply()` is the sum of `balanceOf()` over all users.
P2: Monotonicity of `totalSupply` vs the contract's ether balance:

(a) `totalSupply` is increased iff (⇔) `this.balance` is increased and

(b) `totalSupply` is decreased iff (⇔) `this.balance` is decreased

Which of the existing issues in the code violates which property?

A. An issue in `buy()` violates P1

B. An issue in `buy()` violates P2

C. An issue in `transfer()` violates P1

D. An issue in `transfer()` violates P2

Find the solution here.

19. Following the same contract of question 18 and assuming a correct implementation of `buy()` and `transfer()`, which properties should hold?

A. The order of operation `buy()` and `transfer()` is not important, i.e. first calling `buy` and then `transfer` has the same outcome as first calling `transfer` and then `buy`.

B. `transfer` is additive i.e. performing `transfer` in two steps:

```
transfer(to,x);
transfer(to,y);
```

is equivalent yo performing it in one step:

```
transfer(to,x+y);
```

C. `buy` is additive i.e. performing `buy` in two steps:

```
buy(x1){value:x};
buy(y1){value:y};
```

is equivalent yo performing it in one step:

```
buy(x1+y1){value:x+y};
```

D. None of the above

Find the solution here.

20. In OpenZeppelin's implementation of ERC721, which of the following properties are correct specification assuming `user != 0`?

A. `ownerOf(tokenId) == user` $\Leftrightarrow$ `balanceOf(user) == tokenId`

B. `ownerOf(tokenId) == user` $\Leftrightarrow$ `balanceOf(user) == 1`

C. `ownerOf(tokenId) == user` $\Rightarrow$ |balanceOf(user) $>= 1$|

D. None of the above

Find the solution here.

21. In OpenZeppelin's implementation of ERC721, which of the following is necessarily correct after a successful (non-reverting) call to `transferFrom(from, to, tokenId)`?

A. `ownerOf(tokenId) == to`

B. `ownerOf(tokenId) == from`

C. `balanceOf(to) >= 1`

D. `balanceOf(from) >= balanceOf(to)`

Find the solution here.

22. In OpenZeppelin's implementation of ERC721 Enumerable, which of the following expressions is true?

A. `tokenByIndex(i) == j` $\Rightarrow$ `i < totalSupply()`

B. `i < totalSupply()` $\Rightarrow$ `tokenByIndex(i)` $\neq$ `0`

C. `(u != o && tokenOfOwnerByIndex(o, i) == j)` $\Rightarrow$ `ownerOf(j)` $\neq$ `u`

D. None of the above

Find the solution here.

23. Based on the lecture on "Auditing and Formal Verification: Better together" by Certora's CEO Mooly Sagiv, which of the following is generally accepted?

A. Formal verification eliminates the need for auditing

B. Auditing eliminates the need for formal verification

C. Auditing may find bugs after a project been formally verified

D. Formal verification may find bugs after a project has been audited

Find the solution here.

24. Based on the lecture on "Auditing and Formal Verification: Better together" by Certora's CEO Mooly Sagiv, the takeaways are

A. Spec is the law

B. Writing correct spec is challenging

C. Spec should be audited

D. None of the above

Find the solution here.

9.11 RACE 5 Quiz

All 8 questions in this quiz are based on the `InSecureum` contract. This is the same contract you will see for all the 8 questions in this quiz.

```
pragma solidity ^0.8.0;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/IERC1155.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/IERC1155Receiver.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/extensions/IERC1155MetadataURI.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Context.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/introspection/ERC165.sol";

contract InSecureum is Context, ERC165, IERC1155, IERC1155MetadataURI {

    mapping(uint256 => mapping(address => uint256)) private _balances;
    mapping(address => mapping(address => bool)) private _operatorApprovals;
    string private _uri;

    constructor(string memory uri_) {
        _setURI(uri_);
    }

    function supportsInterface(bytes4 interfaceId) public view virtual override(ERC165, IERC165) returns (bool) {
        return
            interfaceId == type(IERC1155).interfaceId ||
            interfaceId == type(IERC1155MetadataURI).interfaceId ||
            super.supportsInterface(interfaceId);
    }

    function uri(uint256) public view virtual override returns (string memory) {
        return _uri;
    }

    function balanceOf(address account, uint256 id) public view virtual override returns (uint256) {
        require(account != address(0), "ERC1155: balance query for the zero address");
        return _balances[id][account];
    }

    function balanceOfBatch(address[] memory accounts, uint256[] memory ids)
        public view virtual override returns (uint256[] memory) {
        uint256[] memory batchBalances = new uint256[](accounts.length);
```

```
38          for (uint256 i = 0; i < accounts.length; ++i) {
39              batchBalances[i] = balanceOf(accounts[i], ids[i]);
40          }
41          return batchBalances;
42      }
43
44      function setApprovalForAll(address operator, bool approved)
            public virtual override {
45          _setApprovalForAll(_msgSender(), operator, approved);
46      }
47
48      function isApprovedForAll(address account, address operator)
            public view virtual override returns (bool) {
49          return _operatorApprovals[account][operator];
50      }
51
52      function safeTransferFrom(
53          address from,
54          address to,
55          uint256 id,
56          uint256 amount,
57          bytes memory data
58      ) public virtual override {
59          require(
60              from == _msgSender() || isApprovedForAll(from,
                    _msgSender()),
61              "ERC1155: caller is not owner nor approved"
62          );
63          _safeTransferFrom(from, to, id, amount, data);
64      }
65
66      function safeBatchTransferFrom(
67          address from,
68          address to,
69          uint256[] memory ids,
70          uint256[] memory amounts,
71          bytes memory data
72      ) public virtual override {
73          require(
74              from == _msgSender() || isApprovedForAll(from,
                    _msgSender()),
75              "ERC1155: transfer caller is not owner nor approved"
76          );
77          _safeBatchTransferFrom(from, to, ids, amounts, data);
78      }
79
80      function _safeTransferFrom(
81          address from,
82          address to,
83          uint256 id,
84          uint256 amount,
85          bytes memory data
86      ) public virtual {
87          address operator = _msgSender();
88          uint256 fromBalance = _balances[id][from];
89          unchecked {
90               fromBalance = fromBalance - amount;
```

```
91          }
92          _balances[id][from] = fromBalance;
93          _balances[id][to] += amount;
94          emit TransferSingle(operator, from, to, id, amount);
95          _doSafeTransferAcceptanceCheck(operator, from, to, id,
                amount, data);
96      }
97
98      function _safeBatchTransferFrom(
99          address from,
100         address to,
101         uint256[] memory ids,
102         uint256[] memory amounts,
103         bytes memory data
104     ) internal virtual {
105         require(to != address(0), "ERC1155: transfer to the zero
                address");
106         address operator = _msgSender();
107         for (uint256 i = 0; i < ids.length; ++i) {
108             uint256 id = ids[i];
109             uint256 amount = amounts[i];
110             uint256 fromBalance = _balances[id][from];
111             fromBalance = fromBalance - amount;
112             _balances[id][to] += amount;
113         }
114         emit TransferBatch(operator, from, to, ids, amounts);
115         _doSafeBatchTransferAcceptanceCheck(operator, from, to, ids
                , amounts, data);
116     }
117
118     function _setURI(string memory newuri) internal virtual {
119         _uri = newuri;
120     }
121
122     function _mint(
123         address to,
124         uint256 id,
125         uint256 amount,
126         bytes memory data
127     ) internal virtual {
128         require(to != address(0), "ERC1155: mint to the zero
                address");
129         address operator = _msgSender();
130         _balances[id][to] += amount;
131         emit TransferSingle(operator, address(0), to, id, amount);
132         _doSafeTransferAcceptanceCheck(operator, address(0), to, id
                , amount, data);
133     }
134
135     function _mintBatch(
136         address to,
137         uint256[] memory ids,
138         uint256[] memory amounts,
139         bytes memory data
140     ) internal virtual {
141         address operator = _msgSender();
```

```
142         require(operator != address(0), "ERC1155: mint from the
                zero address");
143         for (uint256 i = 0; i < ids.length; i++) {
144             _balances[ids[i]][to] += amounts[i];
145         }
146         emit TransferBatch(operator, address(0), to, amounts, ids);
147         _doSafeBatchTransferAcceptanceCheck(operator, address(0),
                to, ids, amounts, data);
148     }
149 
150     function _burn(
151         address from,
152         uint256 id,
153         uint256 amount
154     ) internal virtual {
155         require(from != address(0), "ERC1155: burn from the zero
                address");
156         address operator = _msgSender();
157         uint256 fromBalance = _balances[id][from];
158         _balances[id][from] = fromBalance - amount;
159         emit TransferSingle(operator, from, address(0), id, amount)
                ;
160     }
161 
162     function _burnBatch(
163         address from,
164         uint256[] memory ids,
165         uint256[] memory amounts
166     ) internal virtual {
167         require(from != address(0), "ERC1155: burn from the zero
                address");
168         address operator = _msgSender();
169         for (uint256 i = 0; i < ids.length; i++) {
170             uint256 id = ids[i];
171             uint256 amount = amounts[i];
172             uint256 fromBalance = _balances[id][from];
173             require(fromBalance >= amount, "ERC1155: burn amount
                    exceeds balance");
174             unchecked {
175                 _balances[id][from] = fromBalance - amount;
176             }
177         }
178         emit TransferBatch(operator, from, address(0), ids, amounts
                );
179     }
180 
181     function _setApprovalForAll(
182         address owner,
183         address operator,
184         bool approved
185     ) internal virtual {
186         require(owner != operator, "ERC1155: setting approval
                status for self");
187         _operatorApprovals[owner][operator] = approved;
188         emit ApprovalForAll(owner, operator, approved);
189     }
190 
```

```
191     function _doSafeTransferAcceptanceCheck(
192         address operator,
193         address from,
194         address to,
195         uint256 id,
196         uint256 amount,
197         bytes memory data
198     ) private {
199         if (isContract(to)) {
200             try IERC1155Receiver(to).onERC1155Received(operator,
                    from, id, amount, data) returns (bytes4 response) {
201                 if (response == IERC1155Receiver.onERC1155Received.
                        selector) {
202                     revert("ERC1155: ERC1155Receiver rejected
                            tokens");
203                 }
204             } catch Error(string memory reason) {
205                 revert(reason);
206             } catch {
207                 revert("ERC1155: transfer to non ERC1155Receiver
                        implementer");
208             }
209         }
210     }
211 
212     function _doSafeBatchTransferAcceptanceCheck(
213         address operator,
214         address from,
215         address to,
216         uint256[] memory ids,
217         uint256[] memory amounts,
218         bytes memory data
219     ) private {
220         if (isContract(to)) {
221             try IERC1155Receiver(to).onERC1155BatchReceived(
                    operator, from, ids, amounts, data) returns (
222                 bytes4 response
223             ) {
224                 if (response != IERC1155Receiver.
                        onERC1155BatchReceived.selector) {
225                     revert("ERC1155: ERC1155Receiver rejected
                            tokens");
226                 }
227             } catch Error(string memory reason) {
228                 revert(reason);
229             } catch {
230                 revert("ERC1155: transfer to non ERC1155Receiver
                        implementer");
231             }
232         }
233     }
234 
235     function isContract(address account) internal view returns (
            bool) {
236         return account.code.length == 0;
237     }
238 }
```

1. `InSecureum balanceOf`

A. May be optimised by caching state variable in local variable

B. May be optimised by changing state mutability from `view` to `pure`

C. May be optimised by changing its visibility to `external`

D. None of the above

Find the solution here.

2. In `InSecureum`, array lengths mismatch check is missing in

A. `balanceOfBatch()`

B. `_safeBatchTransferFrom()`

C. `_mintBatch()`

D. `_burnBatch()`

Find the solution here.

3. The security concern(s) with `InSecureum _safeTransferFrom()` is/are

A. Incorrect visibility

B. Susceptibility to an integer underflow

C. Missing zero-address validation

D. None of the above

Find the solution here.

4. The security concern(s) with `InSecureum _safeBatchTransferFrom()` is/are

A. Missing array lengths mismatch check

B. Susceptibility to an integer underflow

C. Incorrect balance update

D. None of the above

Find the solution here.

5. The security concern(s) with `InSecureum _mintBatch()` is/are

A. Missing array lengths mismatch check

B. Incorrect event emission

C. Allows burning of tokens

D. None of the above

Find the solution here.

6. The security concern(s) with `InSecureum _burn()` is/are

A. Is missing zero-address validation

B. Susceptibility to an integer underflow

C. Incorrect balance update

D. None of the above

Find the solution here.

7. The security concern(s) with `InSecureum _doSafeTransferAcceptanceCheck()` is/are

A. `isContract` check on incorrect address

B. Incorrect check on return value

C. Call to incorrect `isContract` implementation

D. None of the above

Find the solution here.

8. The security concern(s) with `InSecureum isContract()` implementation is/are

A. Incorrect visibility

B. Incorrect operator in the comparison

C. Unnecessary because Ethereum only has Contract accounts

D. None of the above

Find the solution here.

Chapter 10

Self-Assessment Solutions

10.1 Ethereum 101 Quiz Solutions

1. Which of the following EVM components is/are non-volatile across transactions?

Solution: C.
A stack machine is a computer processor or a virtual machine in which the primary interaction is moving short-lived temporary values to and from a push down stack.

The EVM is a simple stack-based architecture consisting of the stack, volatile memory, non-volatile storage with a word size of 256-bit (chosen to facilitate the Keccak256 hash scheme and elliptic-curve computations) and Calldata.

Calldata is a read-only byte-addressable space where the data parameter of a transaction or call is held.

2. The number of transactions in a Ethereum block depends on

Solution: B & C.
Block gas limit is set by miners and refers to the cap on the total amount of gas expended by all transactions in the block, which ensures that blocks can't be arbitrarily large. Blocks therefore are not a fixed size in terms of the number of transactions because different transactions consume different amounts of gas.

3. EVM Stores

Solution: A & C.
EVM uses big-endian ordering where the most significant byte of a word is stored at the smallest memory address and the least significant byte at the

largest

4. Ethereum's threat model is characterised by

Solution: D.
Given the aspirational absence of trusted intermediaries, everyone and everything is meant to be untrusted by default. Participants in this model include developers, miners/validators, infrastructure providers and users, all of whom could potentially be adversaries.

5. Ethereum smart contracts do not run into halting problems because

Solution: C.
Turing-complete systems face the challenge of the halting problem i.e. given an arbitrary program and its input, it is not solvable to determine whether the program will eventually stop running. So Ethereum cannot predict if a smart contract will terminate, or how long it will run. Therefore, to constrain the resources used by a smart contract, Ethereum introduces a metering mechanism called gas.

6. Which of the following operation(s) touch(es) storage?

Solution: B.
Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. `PUSH`, `POP`, `SWAP`, `DUP` etc... Storage is a 256-bit to 256-bit key-value store. This is accessed with `SLOAD`/`SSTORE` instructions.

7. The most gas-expensive operation is

Solution: C.
`SLOAD` is 2100 gas and `SSTORE` is 20000 gas to set a storage slot from 0 to non-0 and 5000 gas otherwise. `CREATE` is 32000 gas and `SELFDESTRUCT` is 5000 gas.

8. Transaction `T1` attempts to write to storage values `S1` and `S2` of contract `C`. Transaction `T2` attempts to read the same storage values `S1` and `S2`. However, `T1` reverts due to an exception after writing `S2`. Which or the following is/are TRUE?

Solution: B.
Of the transaction properties: atomicity (it is all or nothing i.e. cannot be divided or interrupted by other transactions).

A transaction reverts for different exceptional conditions such as running out of gas, invalid instructions etc... In which case all state changes made so far are discarded and the original state of the account is restored as it was before this transaction executed.

9. The gas tracking website `https://etherscan.io/gastracker` says that Low gas cost is 40 gwei. This affects

Solution: A.
The `gasPrice` is the price a transaction originator is willing to pay in exchange for gas. The price is measured in wei per gas unit. The higher the gas price, the faster the transaction is likely to be confirmed on the blockchain. The suggested gas price depends on the demand for block space at the time of the transaction.

In addition, `gasLimit` is the maximum amount of gas the originator is willing to pay for this transaction value: the amount of Ether (in wei) to send to the destination.

10. Security of Ethereum DApps depends on

Solution: A, B & C.
On-chain vs Off-chain: Smart contracts are “on-chain" Web3 components and they interact with “off-chain" components that are very similar to Web2 software. So the major differences in security perspectives between Web3 and Web2 mostly narrow down to security considerations of smart contracts vis-a-vis Web2 software.

11. A `nonce` is present in

Solution: C.
One of the fields in an Ethereum account is the `nonce`, which is a counter used to make sure each transaction can only be processed once.

Furthermore, a transaction is a serialized binary message that also contains a `nonce`, which is again a sequence number, issued by the originating EOA, used

to prevent message replay.

12. Miners are responsible for setting

Solution: B.
Gas price: The price a transaction originator is willing to pay in exchange for gas. The price is measured in wei per gas unit. The higher the gas price, the faster the transaction is likely to be confirmed on the blockchain. The suggested gas price depends on the demand for block space at the time of the transaction.

Block gas limit is set by miners and refers to the cap on the total amount of gas expended by all transactions in the block, which ensures that blocks can't be arbitrarily large.

13. Which of the following information CANNOT be obtained in the EVM?

Solution: Correct is B & D.
Recall the following opcode: `0x40 BLOCKHASH 1 1`. It gets the hash of one of the 256 most recent complete blocks (so, not any block). In addition, there's no operation to access transaction logs.

14. Miners are incentivized to validate and create new blocks by

Solution: A & C.
Miners are rewarded for blocks accepted into the blockchain with a block reward in ether (currently 2 ETH). A miner also gets fees which is the ether spent on gas by all the transactions included in the block.

15. Smart contracts on Ethereum

Solution: A & C.
Web3 is a permissionless, trust-minimized and censorship-resistant network for transfer of value and information.

16. Hardfork on Ethereum

Solution: C.
A hard fork is planned soon, to introduce an exponential difficulty increase, and to motivate a transition to PoS when ready.

17. Which call instruction could be used to allow modifying the caller account's state?

Solution: B & C.
Recall the fllowing opcodes

```
0xf1 CALL 7 1 Message-call into an account
0xf2 CALLCODE 7 1 Message-call into this account with an
    alternative account's code
0xf4 DELEGATECALL 6 1 Message-call into this account with an
    alternative account's code, but persisting the current values
    for sender and value
0xfa STATICCALL 6 1 Static message-call into an account
```

Another variant of `CALL` is `DELEGATECALL`, which essentially runs the code of another contract inside the context of the execution of the current contract.

18. The length of addresses on Ethereum is

Solution: B.
Ethereum state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts.

19. Which of the following statements is/are TRUE about gas?

Solution: B.
Any unused gas in a transaction (`gasLimit` minus gas used by the transaction) is refunded to the sender's account at the same `gasPrice`. Ether used to purchase gas used for the transaction is credited to the beneficiary address (specified in the block header), the address of an account typically under the control of the miner. These are the transaction "fees" paid to the miner.

20. The high-level languages typically used for writing Ethereum smart contracts are

Solution: C & D.
`Solidity` language continues to dominate smart contracts without much real competition (except `Viper` perhaps).

21. Ethereum nodes talk to each other via

Solution: A.
Ethereum node/client: A node is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum nodes.

22. EVM is not a von Neumann architecture because

Solution: B.
EVM does not follow the standard von Neumann architecture. Rather than storing program code in generally accessible memory or storage, it is stored separately in a virtual ROM accessible only through a specialized instruction.

In computer science, a universal Turing machine (UTM) is a Turing machine that simulates an arbitrary Turing machine on arbitrary input. This principle is considered to be the origin of the idea of a stored-program computer used by John von Neumann in 1946 for the "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture.

23. User `A` sends transaction `T1` from address `A1` with `gasPrice` `G1` and later transaction `T2` from address `A2` with `gasPrice` `G2`

Solution: D.
Transaction inclusion is not guaranteed and depends on network congestion and `gasPrice` among other things. Miners determine inclusion.

Transaction order is not guaranteed either, and depends on network congestion and `gasPrice` among other things. Miners determine order as well.

24. The types of accounts on Ethereum are

Solution: C.
Ethereum has two different types of accounts: externally Owned Accounts (EOAs) controlled by private keys, and contract Accounts controlled by their contract code.

25. The number of decimals in Ether is

Solution: C.
Ether is subdivided into smaller units, being the smallest unit named wei. 10^{18} wei is 1 Ether.

26. The difference(s) between Bitcoin and Ethereum is/are

Solution: A, B & C.
Ethereum uses Bitcoin's consensus model: Nakamoto Consensus.

27. Security Audits for smart contracts are performed because

Solution: C.
Secure Software Development Lifecycle (SSDLC) processes for Web2 products have evolved over several decades to a point where they are expected to meet some minimum requirements of a combination of internal validation, external assessments (e.g. product/process audits, penetration testing) and certifications depending on the value of managed assets, anticipated risk, threat model and the market domain of products (e.g. financial sector has stricter regulatory compliance requirements).

28. The number of modified Merkle-Patricia trees in an Ethereum block is

Solution: C.
Blocks contain block header, transactions and ommers' block headers. Block header contains `stateRoot`, `transactionsRoot` and `receiptsRoot` are 256-bit hashes of the root nodes of modified Merkle-Patricia trees.

29. EVM opcodes

Solution: B & D.
The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes (hence called bytecode), where each byte represents an operation.

Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. `PUSH`, `POP`, `SWAP`, `DUP` etc.

30. Gas for EVM opcodes

Solution: B.
Gas costs for different instructions are different depending on their computational/storage load on the client.

A hard fork to change the gas calculation for certain I/O-heavy operations and to clear the accumulated state from a denial-of-service (DoS) attack that exploited the low gas cost of those operations.

A hard fork to address more DoS attack vectors, and another state clearing. Also, a replay attack protection mechanism.

31. Ethereum Virtual Machine is a

Solution: B.
The EVM is a simple stack-based architecture consisting of the stack, volatile memory, non-volatile storage with a word size of 256-bit (chosen to facilitate the Keccak256 hash scheme and elliptic-curve computations) and Calldata.

32. Which of the following statement(s) is/are FALSE?

Solution: B, C & D.

```
0x31 BALANCE 1 1 Get balance of the given account
0x3f EXTCODEHASH 1 1 Get hash of an account's code
0x40 BLOCKHASH 1 1 Get the hash of one of the 256 most recent
    complete blocks
0x43 NUMBER 0 1 Get the block's number
```

10.2 Solidity 101 Quiz Solutions

1. User from EOA `A` calls Contract `C1` which makes an external call (`CALL` opcode) to Contract `C2`. Which of the following is/are true?

Solution: A & C.
Block and Transaction Properties:

- `msg.sender` (`address`): sender of the message (current call)
- `msg.value` (`uint`): number of wei sent with the message
- `tx.origin` (`address`): sender of the transaction (full call chain)

The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every external function call. This includes calls to library functions.

2. Which of the following is/are true for `call`/`delegatecall`/`staticcall` primitives?

Solution: A & C.
`Call`/`Delegatecall`/`Staticcall`: In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single bytes memory parameter and return the success condition (as a bool) and the returned data (bytes memory).

With `delegatecall`, only the code of the given address is used but all other aspects (`storage`, `balance`, `msg.sender`, etc...) are taken from the current contract. The purpose of `delegatecall` is to use library/logic code which is stored in callee contract but operate on the state of the caller contract.

With `staticcall`, the execution will revert if the called function modifies the state in any way

3. The gas left in the current transaction can be obtained with

Solution: B.
Block and Transaction Properties:

- `block.gaslimit` (`uint`): current block `gaslimit`
- `tx.gasprice` (`uint`): gas price of the transaction
- `gasleft()` returns (`uint256`): remaining gas

The function `gasleft()` was previously known as `msg.gas`, which was deprecated in version `0.4.21` and removed in version `0.5.0`.

4. The default value of

Solution: A, B, C & D.
A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the typical "zero-state" of whatever the type is.

For example, the default value for a `bool` is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, `bytes` and `string`, the default value is an empty array or `string`. For the `enum` type, the default value is its first member.

5. Which of the following is/are true about events?

Solution: A & C.
Events are an abstraction on top of the EVM's logging functionality. Emitting events cause the arguments to be stored in the transaction's log (a special data structure in the blockchain). These logs are associated with the address of the contract, are incorporated into the blockchain and stay there as long as a block is accessible. The Log and its event data is not accessible from within contracts (not even from the contract that created them). Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Adding the attribute indexed for up to three parameters adds them to a special data structure known as "topics" instead of the data part of the log. If you use arrays (including string and bytes) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes). All parameters without the indexed attribute are ABI-encoded into the data part of the log. Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

6. Function `foo()` uses `block.number`. Which of the following is/are always true about `foo()`?

Solution: D.
Because it wouldn't always be true unless you know what other things `foo()` uses too.

7. Solidity functions

Solution: B, C & D.
Functions that are defined outside of contracts are called "free functions" and always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.

Function return variables are declared with the same syntax after the `returns` keyword. The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their default value and have that value until they are (re-)assigned.

Function parameters are declared the same way as variables, and the name of unused parameters can be omitted. Function parameters can be used as any other local variable and they can also be assigned to.

Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc..., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

8. Conversions in `Solidity` have the following behavior

Solution: B.
An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. Implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

However, if the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behavior and allows you to bypass some security features of the compiler e.g. `int` to `uint`. If an integer is explicitly converted to a smaller type, higher-order bits are cut off. If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end).

9. When Contract `A` attempts to make a `delegatecall` to Contract `B` but a prior transaction to Contract `B` has executed a `selfdestruct`

Solution: C.
The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

10. If `a = 1` then which of the following is/are true?

Solution: A & D.
Operators Involving LValues (i.e. a variable or something that can be assigned to):

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=` and `^=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1` and `a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change

11. `transfer` and `send` primitives

Solution: A, B & D.
The `receive` function is executed on a call to the contract with empty `calldata`. This is the function that is executed on plain Ether transfers via `.send()` or `.transfer()`. In the worst case, the receive function can only rely on 2300 gas being available (for example when `send` or `transfer` is used), leaving little room to perform other operations except basic logging.

`.transfer(uint256 amount)` sends given `amount` of wei to `address`, reverts on failure, forwards 2300 gas stipend, not adjustable.

`.send(uint256 amount)` sends given amount of wei to `address`, returns false on failure, forwards 2300 gas stipend, not adjustable.

12. A contract can receive Ether via

Solution: A, B, C & D.
A contract without a `receive` Ether function can receive Ether as a recipient of a `coinbase` transaction (aka miner block reward) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the `receive` Ether function).

A contract can have at most one `receive` function, declared using `receive() external payable` without the `function` keyword. This is the function that is executed on plain Ether transfers via `.send()` or `.transfer()`.

Similarly, a contract can have at most one `fallback` function, declared using either

```
fallback() external // it can also have the payable specifier
```

or

```
fallback (bytes calldata _input) external returns (bytes memory
    _output) // it can also have the payable specifier
```

both without the `function` keyword. This function must have `external` visibility. The `fallback` function always receives data, but in order to also receive Ether it must be marked `payable`. In the worst case, if a `payable fallback` function is also used in place of a `receive` function, it can only rely on 2300 gas being available.

An `Error(string)` exception (or an exception without data) is generated in the following situations: if your contract receives Ether via a `public` function without `payable` modifier (including the `constructor` and the `fallback` function).

13. Structs in `Solidity`

Solution: Correct is A, B & C.
Struct Types are custom defined types that can group several variables of same/different types together to create a custom data structure. The `struct` members are accessed using "`.`" e.g.:

```
struct s {address user; uint256 amount}
```

where `s.user` and `s.amount` access the struct members.

Mappings define key-value pairs and are declared using the syntax `mapping(_KeyType => _ValueType) _VariableName`. The `_KeyType` can be any built-in value type, `bytes`, `string`, or any `contract` or `enum` type. Other user-defined or complex types, such as `mappings`, `structs` or array types are not allowed. `_ValueType` can be any type, including `mappings`, arrays and `structs`.

14. The following is/are true about `ecrecover` primitive

Solution: A & C.
Although internally it first recovers the public key from the signature, it actually returns the address derived from the public key.

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (
    address)
```

recovers the address associated with the public key from elliptic curve signature or return 0 on error. The function parameters correspond to ECDSA values of the signature: r = first 32 bytes of signature, s = second 32 bytes of signature, v = final 1 byte of signature. `ecrecover` returns an `address`, and not an

`address payable`.

If you use `ecrecover`, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. This is usually not a problem unless you require signatures to be unique or use them to identify items. `OpenZeppelin` has a ECDSA helper library that you can use as a wrapper for `ecrecover` without this issue.

15. Which of the following is/are valid control structure(s) in `Solidity` (excluding `YUL`)?

Solution: A & B.
`elif` is specific to `Python`, and `switch` has not been implemented in `Solidity` yet. `Solidity` has `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from `C` or `JavaScript`.

16. `address` types

Solution: B & C.
The `address` type comes in two types:

1. `address`: Holds a 20 byte value (size of an Ethereum address)
2. `address payable`: Same as `address`, but with the additional members `transfer` and `send`. `address payable` is an address you can send Ether to, while a plain `address` cannot be sent Ether.

Members of Address Type:

- `address.balance (uint256)`: balance of the Address in Wei
- `address.code (bytes memory)`: code at the Address (can be empty)
- `address.call(bytes memory) returns (bool, bytes memory)`: issue low-level `CALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

Implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(address)`. Explicit conversions to and from `address` are allowed for `uint160`, integer literals, `bytes20` and `contract` types.

When attempting to add or subtract addresses, one gets

```
TypeError: Operator - not compatible with types address and
    int_const 1. Arithmetic operations on addresses are not
    supported. Convert to integer first before using them.
```

17. If the previous block number was 1000 on Ethereum mainnet, which of the following is/are true?

Solution: A, B & C.
Block number is the number of the block that is currently being mined, the next one. Block number 1 was too long ago and its hash can no longer be accessed due to scaling reasons. Mainnet ID Chain is 1.
Recall Block and Transaction Properties

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent, excluding current, blocks
- `block.chainid (uint)`: current chain id
- `block.number (uint)`: current block number
- `block.timestamp (uint)`: current block timestamp as seconds since unix epoch

18. If we have an array then its data location can be

Solution: A, B & C.
Every reference type has an additional annotation - the data location where it is stored. There are three data locations: memory, storage and calldata.

- whose lifetime is limited to an external function call
- whose lifetime is limited to the lifetime of a contract and the location where the state variables are stored
- which is a non-modifiable, non-persistent area where function arguments are stored and behaves mostly like memory. It is required for parameters of external functions but can also be used for other variables

19. `delete varName;` has which of the following effects?

Solution: A & C.
`delete a` assigns the default value (whose byte-representation is all zeros) for the type to `a`.

For integers it is equivalent to `a = 0`.
For bools, it is equivalen to `a = false`.
For structs, it assigns a struct with all members reset.
`delete` has no effect on mappings.
If you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

20. Which of the following is/are valid function specifier(s)?

Solution: A, B & C.
Function Visibility Specifiers: `public`, `external`, `internal` or `private`.
Function Mutability Specifiers: `pure` or `view`.
In addition, there's also the `payable` specifier, which allows a public function to receive Ether.

State Variables are the ones which can be declared as `constant` or `immutable`.

21. Function visibility

Solution: A.
Default visibility is `public` in current `Solidity` versions.

- Public functions are part of the contract interface and can be either called internally or via messages
- External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function f cannot be called internally (i.e. `f()` does not work, but `this.f()` works)
- Internal functions can only be accessed internally from within the current contract or contracts deriving from it
- Private functions can only be accessed from the contract they are defined in and not even in derived contracts

22. For error handling

Solution: A, C & D.

- `assert(bool condition)`: causes a `Panic` error and thus state change reversion if the condition is not met - to be used for internal errors
- `require(bool condition)`: reverts if the condition is not met - to be used for errors in inputs or external components.
- `require(bool condition, string memory message)`: reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.
- `revert()`: abort execution and revert state changes
- `revert(string memory reason)`: abort execution and revert state changes, providing an explanatory string

The `assert` function creates an error of type `Panic(uint256)`. Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input.

23. Which of the following is/are true?

Solution: C & D.
For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared.
The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.
Compared to regular state variables, the gas costs of constant and immutable variables are much lower.

24. Integer overflows/underflows in `Solidity`

Solution: B & C.
Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is 0 up to $2^{32} - 1$. There are two modes in which arithmetic is performed on these types: The "wrapping" or "unchecked" mode and the "checked" mode. By default, arithmetic is always "checked", which means that if the result of an operation falls outside the value range of the type, the call is reverted through a failing assertion. You can switch to "unchecked" mode using `unchecked`. This was introduced in compiler version `0.8.0`.

25. Which of the following is true about mapping types in `mapping(_KeyType => _ValueType)`?

Solution: B & C.
The `_KeyType` can be any built-in value type, `bytes`, `string`, or any `contract` or `enum` type. Other user-defined or complex types, such as mappings, structs or array types are not allowed.

`_ValueType` can be any type, including mappings, arrays and structs. They can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions.

You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that.

26. `receive()` and `fallback()` functions

Solution: A, B & D.
A contract can have at most one `fallback` function, declared using either

```
fallback() external // it can also have the payable specifier
```

or

```
fallback (bytes calldata _input) external returns (bytes memory
    _output) // it can also have the payable specifier
```

both without the `function` keyword. This function must have `external` visibility. The `fallback` function always receives data, but in order to also receive Ether it must be marked `payable`. In the worst case, if a `payable fallback` function is also used in place of a `receive` function, it can only rely on 2300 gas being available.

27. In `Solidity`, `selfdestruct(address)`

Solution: B & D.
`selfdestruct(address payable recipient)`: Destroy the current contract, sending its funds to the given Address and end execution.

28. Which of the following is/are correct?

Solution: A & B.
The version pragma indicates the specific `Solidity` compiler version to be used for that source file and is used as follows: `pragma solidity x.y.z;` where `x.y.z` indicates the version of the compiler.

Using the version pragma does not change the version of the compiler. It also does not enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.

A "`^`" symbol prefixed to `x.y.z` in the pragma indicates that the source file may be compiled only from versions starting with `x.y.z` until `x.(y+1).z`. For e.g., `pragma solidity ^0.8.3;` indicates that source file may be compiled with compiler version starting from `0.8.3` until any `0.8.z` but not `0.9.z`. This is known as a "floating pragma".

Complex pragmas are also possible using "`>`", "`>=`", "`<`" and "`<=`" symbols to combine multiple versions e.g. `pragma solidity >=0.8.0 <0.8.3;`.

29. The impact of data location of reference types on assignments is

Solution: D.
In reality, they all do the opposite.

Assignments between storage and memory (or from calldata) always create an independent copy.
Assignments from memory to memory only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
Assignments from storage to a local storage variable also only assign a reference.
All other assignments to storage always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

30. Which of the below are value types?

Solution: A, B & D.
Value Types are the ones passed by value, i.e. they are always copied when they are used as function arguments or in assignments - Booleans, Integers, Fixed Point Numbers, Address, Contract, Fixed-size Byte Arrays (bytes1, bytes2, ..., bytes32), Literals (Address, Rational, Integer, String, Unicode, Hexadecimal), Enums, Functions.

On the other hand, reference types can be modified through multiple different names. Arrays (including Dynamically-sized bytes array bytes and string), Structs, Mappings.

31. Arrays in `Solidity`

Solution: A, B & C.
Arrays can have a compile-time fixed size, or they can have a dynamic size. Indices are zero-based.

Array members:

- `length`: returns number of elements in array
- `push()`: appends a zero-initialised element at the end of the array and returns a reference to the element
- `push(x)`: appends a given element at the end of the array and returns nothing
- `pop`: removes an element from the end of the array and implicitly calls delete on the removed element

32. `Solidity` **language is**

Solution: A, B, C & D.
Inline assembly support is given by the `YUL` syntax.
`Solidity` is statically typed, supports inheritance, libraries and complex user-defined types. It is a fully-featured high-level language. The syntax and OOP concepts are from `C++`.

10.3 Solidity 101 Quiz Solutions

1. `OpenZeppelin SafeERC20` is generally considered safer to use than `ERC20` because

Solution: B.
`OpenZeppelin SafeERC20` consists of wrappers around `ERC20` operations that throw on failure when the token contract implementation returns `false`. Tokens that return no value and instead revert or throw on failure are also supported with non-reverting calls assumed to be successful. Adds `safeTransfer`, `safeTransferFrom`, `safeApprove`, `safeDecreaseAllowance` and `safeIncreaseAllowance`.

2. The OpenZeppelin library that provides `onlyOwner` modifier

Solution: A & C.
`OpenZeppelin Ownable` provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions. By default, the owner account will be the one that deploys the contract. This can later be changed with `transferOwnership`. This module is used through inheritance. It will make available the modifier `onlyOwner`, which can be applied to your functions to restrict their use to the owner.

3. `OpenZeppelin ECDSA`

Solution: D.
`OpenZeppelin ECDSA` provides functions for recovering and managing Ethereum account ECDSA signatures. These are often generated via `web3.eth.sign`, and are a 65 byte array (of type bytes in `Solidity`) arranged the following way:

```
[[v (1)], [r (32)], [s (32)]]
```

The data signer can be recovered with `ECDSA.recover`, and its address compared to verify the signature. Most wallets will hash the data to sign and add the prefix '`\x19Ethereum Signed Message:\n`', so when attempting to recover the signer of an Ethereum signed message hash, you'll want to use `toEthSignedMessageHash`.

Externally Owned Accounts (EOA) can sign messages with their associated private keys, but currently contracts cannot.

4. Which of the following is/are true about `Solidity` compiler `0.8.0`?

Solution: A, C & D.
Some of the breaking changes of `Solidity v.0.8.0`:

ABI coder v2 is activated by default. You can choose to use the old behaviour using `pragma abicoder v1;`. The pragma `pragma experimental ABIEncoderV2;` is still valid, but it is deprecated and has no effect. If you want to be explicit, please use `pragma abicoder v2;` instead.

Arithmetic operations revert on underflow and overflow. You can use `unchecked` to use the previous wrapping behaviour.

Failing assertions and other internal checks like division by zero or arithmetic overflow do not use the invalid opcode but instead the revert opcode. More specifically, they will use error data equal to a function call to `Panic(uint256)` with an error code specific to the circumstances. This will save gas on errors while it still allows static analysis tools to distinguish these situations from a revert on invalid input, like a failing `require`.

Exponentiation is right associative, i.e., the expression `a**b**c` is parsed as a^{b^c} (`a**(b**c)`). Before `0.8.0`, it was parsed as $\left(a^b\right)^c$ (`(a**b)**c`). This is the common way to parse the exponentiation operator.

5. EVM memory

Solution: A & B.
EVM memory is linear and can be addressed at byte level and accessed with `MSTORE`/`MSTORE8`/`MLOAD` instructions. All locations in memory are initialized as zero.

`Solidity` reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) : `0x60 - 0x7f` (32 bytes): zero slot (The zero slot is used as initial value for dynamic memory arrays and should never be written to)

6. `Dappsys` provides

Solution: A & C.
`Dappsys DSProxy` implements a proxy deployed as a standalone contract which can then be used by the owner to execute code.

`Dappsys DSMath` provides arithmetic functions for the common numerical primitive types of `Solidity`. You can safely add, subtract, multiply, and divide uint numbers without fear of integer overflow. You can also find the minimum and maximum of two numbers. Additionally, this package provides arithmetic

functions for two new higher level numerical concepts called `wad` (18 decimals) and `ray` (27 decimals). These are used to represent fixed-point decimal numbers.

`Dappsys DSAuth` provides a flexible and updatable auth pattern which is completely separate from application logic.

7. Libraries are contracts

Solution: A, B & D.
Libraries are deployed only once at a specific address and their code is reused using the `DELEGATECALL` opcode. This means that if library functions are called, their code is executed in the context of the calling contract. They use the `library` keyword.

Libraries are restricted in the following ways: They cannot have state variables, they cannot inherit nor be inherited, they cannot receive Ether.

Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless.

8. ERC20 `transferFrom(address sender, address recipient, uint256 amount)` (that follows the `ERC20` spec strictly)

Solution: A, B & D.
`transferFrom(address sender, address recipient, uint256 amount)` moves amount tokens from `sender` to `recipient` using the allowance mechanism. `amount` is then deducted from the caller's allowance. Returns a boolean value indicating whether the operation succeeded. Emits a Transfer event.

9. `ERC777` may be considered as an improved version of `ERC20` because

Solution: A, B & C.
Like `ERC20`, `ERC777` is a standard for fungible tokens with improvements such as getting rid of the confusion around decimals, minting and burning with proper events, among others, but its killer feature is receive hooks.

A hook is simply a function in a contract that is called when tokens are sent to it, meaning accounts and contracts can react to receiving tokens. This enables a lot of interesting use cases, including atomic purchases using tokens (no need to do `approve` and `transferFrom` in two separate transactions), rejecting reception of tokens (by reverting on the hook call), redirecting the received tokens to other addresses, among many others.

Furthermore, since contracts are required to implement these hooks in order to receive tokens, no tokens can get stuck in a contract that is unaware of the

ERC777 protocol, as has happened countless times when using ERC20s.

10. WETH is

Solution: D.
WETH stands for Wrapped Ether.
For protocols that work with ERC20 tokens but also need to handle Ether, WETH contracts allow converting Ether to its ERC20 equivalent WETH (called wrapping) and vice-versa (called unwrapping). WETH can be created by sending Ether to a WETH smart contract where the Ether is stored and in turn receiving the WETH ERC20 token at a 1:1 ratio. This WETH can be sent back to the same smart contract to be "unwrapped" i.e. redeemed back for the original Ether at a 1:1 ratio. The most widely used WETH contract is WETH9 which holds more than 7 million Ether for now.

11. OpenZeppelin SafeCast

Solution: B.
OpenZeppelin SafeCast are Wrappers over Solidity's uintXX/intXX casting operators with added overflow checks. Downcasting from uint256/int256 in Solidity does not revert on overflow. This can easily result in undesired exploitation or bugs, since developers usually assume that overflows raise errors. SafeCast restores this intuition by reverting the transaction when such an operation overflows.

12. OpenZeppelin ERC20Pausable

Solution: A, & B.
Not C, because it inherits these modifiers from Pausable and doesn't implement them.

OpenZeppelin ERC20Pausable implements ERC20 token with pausable token transfers, minting and burning. Useful for scenarios such as preventing trades until the end of an evaluation period, or having an emergency switch for freezing all token transfers in the event of a large bug.

The emergency stop mechanism using functions pause and unpause that can be triggered by an authorized account. This module is used through inheritance. It will make available the modifiers whenNotPaused and whenPaused, which can be applied to the functions of your contract. Only the functions using the modifiers will be affected when the contract is paused or unpaused.

13. `CREATE2`

Solution: B & C.
`OpenZeppelin Create2` makes usage of the `CREATE2` EVM opcode easier and safer. `CREATE2` can be used to compute in advance the address where a smart contract will be deployed.

```
deploy(uint256 amount, bytes32 salt, bytes bytecode) -> address \\
    Deploys a contract using CREATE2
```

14. Name collision error with inheritance happens when the following pairs have the same name within a contract

Solution: A, B & D.
Name Collision Error is an error when any of the following pairs in a contract have the same name due to inheritance

- a function and a modifier
- a function and an event
- an event and a modifier

15. `OpenZeppelin`**'s (role-based)** `AccessControl` **library**

Solution: B & C.
`OpenZeppelin AccessControl` provides a general role based access control mechanism. Multiple hierarchical roles can be created and assigned each to multiple accounts. Roles can be used to represent a set of permissions. `hasRole` is used to restrict access to a function call. Roles can be granted and revoked dynamically via the `grantRole` and `revokeRole` functions which can only be called by the role's associated admin accounts.

16. `OpenZeppelin`**'s proxy implementations**

Solution: A & B.
`OpenZeppelin Proxy` is an abstract contract that provides a fallback function that delegates all calls to another contract using the EVM instruction `delegatecall`. We refer to the second contract as the implementation behind the proxy, and it has to be specified by overriding the virtual `_implementation` function.

`OpenZeppelin ERC1967Proxy` implements an upgradeable proxy. It is upgradeable because calls are delegated to an implementation address that can be changed.

17. Which of the following is/are true for a function `f` that has a modifier `m`?

Solution: B.
Function Modifiers can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function. The function's control flow continues after the "`_`" in the preceding modifier. Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented. The modifier can choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body. The '`_`' symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

18. Zero address check is typically recommended because

Solution: B.
`address(0)` which is 20-bytes of 0's is treated specially in `Solidity` contracts because the private key corresponding to this address is unknown. Ether and tokens sent to this address cannot be retrieved and setting access control roles to this address also won't work (no private key to sign transactions). Therefore zero addresses should be used with care and checks should be implemented for user-supplied address parameters.

19. `Solidity` supports

Solution: A, B & D.
There's no such thing as "contract overloading".

`Solidity` supports multiple inheritance including polymorphism. There's also function overloading because a contract can have multiple functions of the same name but with different parameter types.
20. `OpenZeppelin ERC721`

Solution: A, B & D.
C is not correct since approval can only give or take away one single token, so changing approval doesn't allow stealing more than what was already approved.

`OpenZeppelin ERC721` omplements the popular `ERC721` Non-Fungible Token Standard.

`safeTransferFrom()`: Safely transfers `tokenId` token from `from` to `to`, checking first that contract recipients are aware of the `ERC721` protocol to prevent tokens from being forever locked. One of the requirements is that `from` cannot be the zero address.

`setApprovalForAll(address operator, bool _approved)`: Approve or remove `operator` as an operator for the caller. Operators can call `transferFrom` or `safeTransferFrom` for any token owned by the caller.

21. Function visibility

Solution: C.
The `uint256` takes a full slot, the bools (each 1 byte) and the address (20 bytes) can packed into the same slot.

State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0.

For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible. If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot.

22. Which of the following is/are generally true about storage layouts?

Solution: A, B & C.
In the case of mappings, the slots are unique for each key. They're not consecutive.

Ordering of storage variables and struct members affects how they can be packed tightly. For example, declaring your storage variables in the order of `uint128`, `uint128`, `uint256` instead of `uint128`, `uint256`, `uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

The elements of structs and arrays are stored after each other, just as if they were given as individual values.
In the case of Dynamic Arrays, if the storage location of the array ends up being a slot p after applying the storage layout rules, this slot stores the number of elements in the array (byte arrays and strings are an exception). Array data is located starting at `keccak256(p)` and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes.

For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations. The value corresponding to a mapping key k is located at `keccak256(h(k).p)` where '`.`' is concatenation and '`h`' is a function that is

applied to the key depending on its type.

23. Assuming all contracts `C1`, `C2` and `C3` define explicit constructors in `contract C1 is C2, C3` and both `C2` and `C3` don't inherit contracts, the number & order of constructor(s) executed is/are

Solution: B.
The constructors of all the base contracts will be called following the C3-linearization rules.

24. `OpenZeppelin SafeMath`

Solution: B.
It is not A, because it does not prevent them at compile, but at runtime. It can be argued it's not B since it's a recommendation and not a requirement.

`OpenZeppelin SafeMath` provides mathematical functions that protect your contract from overflows and underflows.

Until `Solidity` version `0.8.0` which introduced checked arithmetic by default, arithmetic was unchecked and therefore susceptible to overflows and underflows which could lead to critical vulnerabilities. The recommended best-practice for such contracts is to use `OpenZeppelin`'s `SafeMath` library for arithmetic.

25. Which of the following is/are not allowed?

Solution: C.
Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header.

A contract can have multiple functions of the same name but with different parameter types. This process is called "overloading".

Function modifiers can override each other. This works in the same way as function overriding (except that there is no overloading for modifiers).

26. Which of the following is/are true about abstract contracts and interfaces?

Solution: A, C & D.
Note that A isn't necessarily correct since abstract classes can have all functions defined.

Contracts need to be marked as `abstract` when at least one of their functions is not implemented. They use the `abstract` keyword.

Interfaces cannot have any functions implemented.

Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered virtual. Functions with `private` visibility cannot be virtual.

27. Storage layout

Solution: A, C & D.
Not B, because it's 256-bit slots, not Byte.

Storage is a key-value store that maps 256-bit words to 256-bit words and is accessed with EVM's `SSTORE`/`SLOAD` instructions. All locations in storage are initialized as zero.

For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible. Value types use only as many bytes as are necessary to store them.

28. Which of the following EVM instruction(s) do(es) not touch EVM storage?

Solution: B & D.
Storage is a key-value store that maps 256-bit words to 256-bit words and is accessed with EVM's `SSTORE`/`SLOAD` instructions. All locations in storage are initialized as zero.

EVM memory is linear and can be addressed at byte level and accessed with `MSTORE`/`MSTORE8`/`MLOAD` instructions. All locations in memory are initialized as zero.

The Stack is made up of 1024 256-bit elements. EVM instructions can operate with the top 16 stack elements. Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. `PUSH`, `POP`, `SWAP`, `DUP`, etc...

29. Proxied contracts

Solution: B & C.
`OpenZeppelin Initializable` aids in writing upgradeable contracts, or any kind of contract that will be deployed behind a proxy. Since a proxied contract cannot have a constructor, it is common to move constructor logic to an external initializer function, usually called `initialize`. It then becomes necessary to protect this initializer function so it can only be called once. The

`initializer` modifier provided by this contract will have this effect.

To avoid leaving the proxy in an uninitialized state, the `initializer` function should be called as early as possible by providing the encoded function call as the `_data` argument. When used with inheritance, manual care must be taken to not invoke a parent initializer twice, or to ensure that all initializers are idempotent. This is not verified automatically as constructors are by `Solidity`.

30. `OpenZeppelin`'s `ReentrancyGuard` library mitigates reentrancy risk in a contract

Solution: B.
`OpenZeppelin ReentrancyGuard` prevents reentrant calls to a function. Inheriting from `ReentrancyGuard` will make the `nonReentrant` modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

31. EVM inline assembly has

Solution: A, C & D.
Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of `Solidity`. You should only use it for tasks that need it, and only if you are confident with using it. The language used for inline assembly in `Solidity` is called `Yul`.
You can access `Solidity` variables and other identifiers in inline assembly by using their name. Local variables of value type are directly usable in inline assembly. Local variables that refer to memory/calldata evaluate to the address of the variable in memory/calldata and not the value itself.

32. If `OpenZeppelin`'s `isContract(address)` returns `false` for an address then

Solution: B.
Returns `true` if account is a contract. It is unsafe to assume that an address for which this function returns false is an externally-owned account (EOA) and not a contract. Among others, `isContract` will return `false` for the following types of addresses:

- an externally-owned account
- a contract in construction
- an address where a contract will be created
- an address where a contract lived, but was destroyed

10.4 Security Pitfalls & Best Practices 101 Quiz Solutions

1. The use of `pragma` in the given contract snippet

```
pragma solidity ^0.6.0;

contract test {
   // Assume other required functionality is correctly implemented
   // Assume this contract can work correctly without modifications
        across 0.6.x/0.7.x/0.8.x compiler versions
}
```

Solution: B & C.
Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using '`^`' in `pragma solidity 0.5.10`) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs.

There were bugs that were fixed in versions between `0.6.5` and `0.6.11` which means those fixes were absent in `0.6.5`. Choice B was about these aspects.

Illustrative means "serving as an example or explanation". So the use of `^0.6.0` when the latest available version is `0.8.9` (as mentioned in choice C) is an example of using a much older compiler version when newer versions with bug fixes and more security features e.g. built-in overflow checks in `^0.8.0` are available.

2. The given contract snippet has

```
pragma solidity 0.8.4;

contract test {

   // Assume other required functionality is correctly implemented

   function kill() public {
      selfdestruct(payable(0x0));
   }
}
```

Solution: A & B.
Unprotected call to `selfdestruct`: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions.

Zero Address Check: `address(0)` which is 20-bytes of 0's is treated specially in Solidity contracts because the private key corresponding to this address is unknown. Ether and tokens sent to this address cannot be retrieved and setting access control roles to this address also won't work (no private key to sign transactions). Therefore zero addresses should be used with care and checks

should be implemented for user-supplied address parameters.

Reserved Keywords: These keywords are reserved in `Solidity`. They might become part of the syntax in the future: `after`, `alias`, `apply`, `auto`, `case`, `copyof`, `default`, `define`, `final`, `immutable`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `unchecked`.

3. The given contract snippet has

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    modifier onlyAdmin() {
        // Assume this is correctly implemented
        _;
    }

    function transferFunds(address payable recipient, uint amount)
        public {
        recipient.transfer(amount);
   }
}
```

Solution: A & C.
Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. `owner`, `controller`, etc...) typically in modifiers. Missing checks allow attackers to control critical logic.

Unprotected (`external`/`public`) function calls sending Ether/tokens to user-controlled addresses may allow users to withdraw unauthorized funds.

`transferFunds()` clearly lets anyone withdraw any amount to any address. The only hint in the question is the `onlyAdmin` modifier. While some other access control may also have been acceptable, the focus is on the code snippet provided and hence A is true.

`transfer` (unlike `send`) does not return a success/failure return value. It reverts on failure. So there is nothing to be checked. Note that `ERC20`'s `transfer()` returns a boolean which should be checked.

4. In the given contract snippet

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    mapping (uint256 => address) addresses;
    bool check;

    modifier onlyIf() {
        if (check) {
            _;
        }
    }

    function setAddress(uint id, address addr) public {
        addresses[id] = addr;
    }

    function getAddress(uint id) public onlyIf returns (address) {
        return addresses[id];
    }
}
```

Solution: A, & B.
If a modifier does not execute '_' or reverts, the function using that modifier will return the default value causing unexpected behavior.

Remember that function modifiers can be used to change the behavior of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function. The function's control flow continues after the "_" in the preceding modifier. Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented. The modifier can choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body. The '_' symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

5. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    modifier onlyAdmin {
     // Assume this is correctly implemented
        _;
    }

    function delegate (address addr) external {  addr.delegatecall(
        abi.encodeWithSignature("setDelay(uint256)"));
    }
}
```

Solution: A, B, C & D.
`delegatecall()` or `callcode()` to an address controlled by the user allows execution of malicious contracts in the context of the caller's state. Ensure trusted destination addresses for such calls.

Ensure that return values of low-level calls (`call`/`callcode`/`delegatecall` /`send`/etc...) are checked to avoid unexpected failures.

Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. `owner`, `controller`, etc...) typically in modifiers. Missing checks allow attackers to control critical logic.

Low-level calls `call`/`delegatecall`/`staticcall` return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed.

6. The vulnerability/vulnerabilities present in the given contract snippet is/are

```
pragma solidity 0.7.0;
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
// which works with 0.7.0

contract test {

 // Assume other required functionality is correctly implemented
 // For e.g. users have deposited balances in the contract
 // Assume nonReentrant modifier is always applied

    mapping (address => uint256) balances;

    function withdraw(uint256 amount) external nonReentrant {
        msg.sender.call{value: amount}("");
        balances[msg.sender] -= amount;
    }
}
```

Solution: B & C.
The code in this question was unintentionally missing inheritance from the `ReentrancyGuard` Contract. While there's a lot of discussion about the correct meaning of the term "underflow", this is how it is used in the `Solidity` Documentation and other related literature.

Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards.

Not using `OpenZeppelin`'s `SafeMath` (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. `Solc v0.8.0` introduced default overflow/underflow checks for all arithmetic operations.

I hope that the comment

```
// Assume nonReentrant modifier is always applied
```

implied that the intent was to apply the modifier "correctly" (i.e. with successful compilation), which further implies reentrancy risk mitigation i.e. A is not a correct choice. Also, if A were to also be correct then that would again result in the "All of the above" ambiguity (A + B + C or D or A + B + C + D).

7. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

 // Assume other required functionality is correctly implemented

    uint256 private constant secret = 123;

    function diceRoll() external view returns (uint256) {
        return (((block.timestamp * secret) % 6) + 1);
    }
}
```

Solution: B & C.
Marking variables `private` does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain.

PRNG relying on `block.timestamp`, `now` or `blockhash` can be influenced by miners to some extent and should be avoided.

Making it public in this case should not affect gas given that there are no function arguments to copy over (if there were parameters/arguments, making it public would increase gas). Even otherwise, making it public from external should not affect the attack surface of the contract because it will only further allow (trusted) contract functions to call it.

In addition, the logic of `diceRoll()` is broken as it returns only 1 or 4.

8. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented
    // Contract admin set to deployer in constructor (not shown)
    address admin;

    modifier onlyAdmin {
        require(tx.origin == admin);
        _;
    }

    function emergencyWithdraw() external payable onlyAdmin {
        msg.sender.transfer(address(this).balance);
    }
}
```

Solution: B & D.
Although `transfer()` and `send()` have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. Use `call()` instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection.

Use of `tx.origin` for authorization may be abused by a MITM malicious contract forwarding calls from the legitimate user who interacts with it. Use `msg.sender` instead.

Regarding C, 0 transfers should not revert. Even if they did, in this context, it wouldn't be considered a "security" concern because there would be nothing to withdraw and so a revert wouldn't be a concern w.r.t. any locked funds as such.

Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain.

9. The given contract snippet is vulnerable because of

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    uint256 private constant MAX_FUND_RAISE = 100 ether;
    mapping (address => uint256) contributions;

    function contribute() external payable {
        require(address(this).balance != MAX_FUND_RAISE);
        contributions[msg.sender] += msg.value;
    }
}
```

Solution: D.
`external` or `public` is required for a function to be `payable`. This use of `msg.sender` is very common and correct.

Use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using `>=` or `<=` instead of `==` for such variables depending on the contract logic.

A contract can receive Ether via `payable` functions, `selfdestruct()`, `coinbase` transaction or pre-sent before creation. Contract logic depending on `this.balance` can therefore be manipulated.

Given the compiler version, even if there is an attempted integer overflow at runtime, it will revert before overflowing (because of inbuilt checks) with an exception but will not wrap. So A is not a vulnerability. While this is true in general, this snippet cannot be overflowed because of its dependence on `msg.value`.

10. In the given contract snippet, the require check will

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    function callMe (address target) external {
        (bool success, ) = target.call("");
        require(success);
    }
}
```

Solution: B.
Low-level calls `call`/`delegatecall`/`staticcall` return `true` even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed.

11. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented
    // Assume admin is set correctly to contract deployer in
        constructor
    address admin;

    function setAdmin (address _newAdmin) external {
        admin = _newAdmin;
    }
}
```

Solution: A, B, C & D.
Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic.

Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever.

Changing critical addresses in contracts should be a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) replaces the old address with the new one (i.e. claims ownership). This gives an opportunity to recover from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible.

Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain.

12. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly implemented

    address admin;
    address payable pool;

   constructor(address _admin) {
        admin = _admin;
    }

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function setPoolAddress(address payable _pool) external
        onlyAdmin {
        pool = _pool;
    }

    function addLiquidity() payable external {
        pool.transfer(msg.value);
    }
}
```

Solution: A, C & D.
Externally accessible functions (`external`/`public` visibility) may be called in any order (with respect to other defined functions). It is not safe to assume they will only be called in the specific order that makes sense to the system design or is implicitly assumed in the code. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called before other system functions can be called.

Uninitialized state/local variables are assigned zero values by the compiler and may cause unintended results e.g. transferring tokens to zero address. Explicitly initialize all state/local variables.

Setters of `address` type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever.

Race conditions can be forced on specific Ethereum transactions by monitoring the mempool. For example, the classic `ERC20 approve()` change can be

front-run using this method. Do not make assumptions about transaction order dependence.

13. The security concern(s) in the given proxy-based implementation contract snippet is/are

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/proxy/utils/Initializable.sol";

contract test is Initializable {

    // Assume other required functionality is correctly implemented

    address admin;
    uint256 rewards = 10;

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function initialize (address _admin) external {
        require(_admin != address(0));
        admin = _admin;
    }

    function setRewards(uint256 _rewards) external onlyAdmin {
        rewards = _rewards;
    }
}
```

Solution: B & C.
There are no imported contracts that need to be made upgradable (by implementing `Initializable`).

Contracts imported from proxy-based upgradeable contracts should also be upgradeable where such contracts have been modified to use initializers instead of constructors.

Proxy-based upgradeable contracts need to use public initializer functions instead of constructors that need to be explicitly called only once. Preventing multiple invocations of such initializer functions (e.g. via `initializer` modifier from `OpenZeppelin`'s `Initializable` library) is a must.

Initializing state-variables in proxy-based upgradeable contracts should be done in initializer functions and not as part of the state variable declarations in which case they won't be set.

14. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob
    /master/contracts/token/ERC20/IERC20.sol";

contract test {

    // Assume other required functionality is correctly implemented

    address admin;
    address token;

    constructor(address _admin, address _token) {
        require(_admin != address(0));
        require(_token != address(0));
        admin = _admin;
        token = _token;
    }

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function payRewards(address[] calldata recipients, uint256[]
        calldata amounts) external onlyAdmin {
        for (uint i; i < recipients.length; i++) {
            IERC20(token).transfer(recipients[i], amounts[i]);
        }
    }
}
```

Solution: A, C & D.
There's no guarantee that the passed arrays are of same length, so if one would be longer than the other one it can cause an `Out Of Bounds` error, which is why D is correct.

Calls to external contracts inside a loop are dangerous (especially if the loop index can be user-controlled) because it could lead to DoS if one of the calls reverts or execution runs out of gas. Avoid calls within loops, check that loop index cannot be user-controlled or is bounded.

`ERC20 transfer()` does not return boolean, so contracts compiled with `solc >= 0.4.22` interacting with such functions will revert. Use `OpenZeppelin`'s `SafeERC20` wrappers.

This is `ERC20` token transfer and not Ether transfer (which throws on failure). `ERC20` transfer is typically expected to return a boolean but non-`ERC20`-conforming tokens may return nothing or even revert which is typically why `SafeERC20` is recommended.

15. The vulnerability/vulnerabilities present in the given contract snippet is/are

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob
    /master/contracts/security/ReentrancyGuard.sol";

contract test {

 // Assume other required functionality is correctly implemented
 // For e.g. users have deposited balances in the contract

    mapping (address => uint256) balances;

    function withdrawBalance() external {
        msg.sender.call{value: balances[msg.sender]}("");
        balances[msg.sender] = 0;
    }
}
```

Solution: A.
Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards.

Not using `OpenZeppelin`'s `SafeMath` (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. `Solc v0.8.0` introduced default overflow/underflow checks for all arithmetic operations.

16. The security concern(s) in the given contract snippet is/are

```
pragma solidity 0.8.4;

contract test {
// Assume other required functionality is correctly implemented
// Assume that hash is the hash of a message computed elsewhere in
    contract
// Assume that the contract does not make use of chainID or nonces
    in its logic

function verify(address signer, bytes32 memory hash, bytes32 sigR,
    bytes32 sigS, uint8 sigV) internal view returns (bool) {
   return signer == ecrecover(hash, sigV, sigR, sigS);
}
```

Solution: A, B, C & D.
The `ecrecover` function is susceptible to signature malleability which could lead to replay attacks. Consider using `OpenZeppelin`'s `ECDSA` library.

The vulnerability occurs when a digital signature is valid for multiple transactions, which can happen when one sender (say Alice) sends money to multiple recipients through a proxy contract (instead of initiating multiple transactions). In the proxy contract mechanism, Alice can send a digitally signed message off-chain (e.g., via email) to the recipients, similar to writing personal checks in the real world, to let the recipients withdraw money from the proxy contract via transactions. To assure that Alice does approve a certain payment, the proxy contract verifies the validity of the digital signature in question. However, if the signature does not give the due information (e.g., `nonce`, proxy contract address), then a malicious recipient can replay the message multiple times to withdraw extra payments. This vulnerability was first exploited in a replay attack against smart contracts. This vulnerability can be prevented by incorporating the due information in each message, such as a `nonce` value and timestamps.

Indistinguishable Chains: This vulnerability was first observed from the cross-chain replay attack when Ethereum was divided into two chains, namely, ETH and ETC. Recall that Ethereum uses `ECDSA` to sign transactions. Prior to the hard fork for EIP-155, each transaction consisted of six fields (i.e., `nonce`, `recipient`, `value`, `input`, `gasPrice` and `gasLimit`). However, the digital signatures were not chain-specific, because no chain-specific information was even known back then. As a consequence, a transaction created for one chain can be reused for another chain. This vulnerability has been eliminated by incorporating `chainID` into the fields.

`ecrecover() returns (address)` recovers the address associated with the public key from elliptic curve signature or return zero on error.

10.5 Security Pitfalls & Best Practices 201 Quiz Solutions

1. The `InSecureumToken` contract strictly follow the specification of

Solution: D.
Since decimals was defined as `uint` (same as `uint256`) and not as `uint8` as `ERC20` or `ERC777` standardized, it can't strictly be either of them. And since this is clearly a fungible token contract, it can't be `ERC721`.

2. To avoid lock of funds, the following feature(s) MUST be implemented before contract deployment

Solution: C.
Contracts that accept Ether via `payable` functions but without withdrawal mechanisms will lock up that Ether. Remove `payable` attribute or add a withdraw function.

3. Which of the following assertion(s) is/are true (without affecting the security posture of the contract)?

Solution: C & D.
`buy()` cannot function without being `payable`.
There's no reason the visibility of balances needs to be `private`.
`transfer()` can be `external` since it's not called internally.
`safeAdd()` can be `public` since it is a `pure` function.

4. The total supply is limited by

Solution: D.
It would be B, but `MAX_SUPPLY` isn't actually used anywhere in the code.

5. The following issue(s) is/are present in the codebase

Solution: B.
It's impossible to get any Ether out of this contract, so no draining.
It divides `desired_tokens` first and only then multiplies by the decimals, this causes any amount of tokens below 10 to result in 0 `required_wei_sent`.
There are no divisions here that could allow a division by 0.

6. The following issue(s) is/are present in the codebase

Solution: C.
No requests made before/after a function call would be able to change the token price.
All of the functions are intended to be used by users, so no "access control" would be possible without excluding users.
A user can send all of their tokens to themselve, which will double their balance due to the pre-loaded variable reuse.

7. The following issue(s) is/are present in the codebase

Solution: D.
No reentrancies are possible since no external calls are made.

8. The following issue(s) is/are present in the codebase

Solution: D.
While it is indeed possible to exploit an overflow at the multiplication (`(desired_tokens / 10) * decimals`), it doesn't allow you to receive FREE tokens (although it makes them a bargain).

10.6 Audit Techniques & Tools 101 Quiz Solutions

1. Which of the below is/are accurate?

Solution: C.
Audit is not a security guarantee of "bug-free" code by any stretch of imagination but a best-effort endeavour by trained security experts operating within reasonable constraints of time, understanding, expertise and of course, decidability.

For Ethereum-based smart-contract projects, the scope is typically the on-chain smart contract code and sometimes includes the off-chain components that interact with the smart contracts.

The goal of audits is to assess project code (with any associated specification, documentation) and alert project team, typically before launch, of potential security-related issues that need to be addressed to improve security posture, decrease attack surface and mitigate risk.

The audit detects and describes (in a report) security issues with underlying vulnerabilities, severity/difficulty, potential exploit scenarios and recommended fixes.

2. Audit reports from audit firms tipically include

Solution: A & B.
Audit Reports include details of the scope, goals, effort, timeline, approach, tools/techniques used, findings summary, vulnerability details, vulnerability classification, vulnerability severity/difficulty/likelihood, vulnerability exploit scenarios, vulnerability fixes and informational recommendations/suggestions on programming best-practices.

The vulnerabilities found during the audit are typically classified into different categories which helps to understand the nature of the vulnerability, potential impact/severity, impacted project components/functionality and exploit scenarios.

3. These audit techniques are especially well-suited for smart contracts (compared to Web2 programs)

Solution: A, B, C & D.
Smart contract testing has a similar motivation but is arguably more complicated despite their relatively smaller sizes (in lines of code) compared to Web2 software.

Fuzzing is especially relevant to smart contracts because anyone can interact with them on the blockchain with random inputs without necessarily having a valid reason or expectation (arbitrary byzantine behaviour).

It is natural for this question to be open to interpretation because it is a 100k feet level question comparing techniques "especially well-suited" to web3 smart contracts vis-a-vis web2 software, which covers a lot of ground. It is not about what is possible or what is expected but about suitability. All these techniques can be and are performed on web2 applications but generalized aspects of size, scope, nature of user-interactions, source-code availability/expectations and reduced states/transitions of smart contracts make the techniques more suitable for them compared to web2 software.

4. The following kinds of findings may be expected during audits

Solution: A, B & C.
Findings may be contested as not being relevant, outside the project's threat model or simply acknowledged as being within the project's acceptable risk model.

False Positives are findings which indicate the presence of vulnerabilities but which in fact are not vulnerabilities. Such false positives could be due to incorrect assumptions or simplifications in analysis which do not correctly consider all the factors required for the actual presence of vulnerabilities.

False Negatives: are missed findings that should have indicated the presence of vulnerabilities but which are in fact are not reported at all. Such false negatives could be due to incorrect assumptions or inaccuracies in analysis which do not correctly consider the minimum factors required for the actual presence of vulnerabilities.

Auditors generally collate all findings from their review into a report which is handed to the project team. At this point, the assumption from the auditors is that all the findings in their report are true positives. However, depending on the differing threat/trust models or different assumptions made between the audit & project teams, some of the findings may be treated as false positives by the project team which thereafter may choose to ignore such findings, recognize but not act (via fixes) on them, etc...

Auditors internally may bring up their individual findings with other team members to discuss if they are indeed true/false. They may also bring up doubtful findings with the project team during an audit (interim discussions/clarifications). Findings which are deemed by everyone as false positives, i.e. irrelevant, will not be included in the report. There may also be disagreements between the auditors & project teams about the threat/trust models, assumptions or difficulty/severity levels, which may lead to opposing viewpoints

which are sometimes documented in the reports. But, in general, many of the reported findings are "confirmed" by the projects, after which we can think of them as true positives.

5. Which of the following is/are true?

Solution: B.
Very few smart contract projects have detailed specifications at their first audit stage. At best, they have some documentation about what is implemented. Auditors spend a lot of time inferring specification from documentation/implementation which leaves them with less time for vulnerability assessment.

Manual analysis is however the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found.

Automated analysis using tools is cheap (typically open-source free software), fast, deterministic and scalable (varies depending on the tool being semi-/fully-automated) but however is only as good as the properties it is made aware of, which is typically limited to `Solidity` and EVM related constraints.

Findings may be contested as not being relevant, outside the project's threat model or simply acknowledged as being within the project's acceptable risk model

6. Automated tools for smart contract analysis

Solution: D.
Manual analysis is however the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found.

Automated analyzers do not understand application-level logic and their constraints. They are limited to constraints/properties of `Solidity` language, EVM or Ethereum blockchain.

Smart contract security tools are useful in assisting auditors while reviewing smart contracts. They automate many of the tasks that can be codified into rules with different levels of coverage, correctness and precision. They are fast, cheap, scalable and deterministic compared to manual analysis. But they are also susceptible to false positives. They are especially well-suited currently to detect common security pitfalls and best-practices at the `Solidity` and EVM level. With varying degrees of manual assistance, they can also be programmed to check for application-level, business-logic constraints.

7. Which of the following is/are true?

Solution: A & C.
Slither is a `Solidity` static analysis framework written in `Python3`. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses.

Echidna is a Haskell program designed for fuzzing/property-based testing of Ethereum smart contracts.

MythX is a powerful security analysis service that finds `Solidity` vulnerabilities in your Ethereum smart contract code during your development life cycle. It is a paid API-based service which uses several tools on the backend including a static analyzer (Maru), symbolic analyzer (Mythril) and a greybox fuzzer (Harvey) to implement a total of 46 detectors.

8. Which of the following is/are correct about false positives?

Solution: A & C.
False positives require further manual analysis on findings to investigate if they are indeed false or true positives.

Smart contract security tools are useful in assisting auditors while reviewing smart contracts. They automate many of the tasks that can be codified into rules with different levels of coverage, correctness and precision. They are fast, cheap, scalable and deterministic compared to manual analysis. But they are also susceptible to false positives.

9. Audit findings

Solution: A & B.
Detect and describe (in a report) security issues with underlying vulnerabilities, severity/difficulty, potential exploit scenarios and recommended fixes.

They also provide subjective insights into code quality, documentation and testing.

Findings may be contested as not being relevant, outside the project's threat model or simply acknowledged as being within the project's acceptable risk model.

Codified exploits should always be on a testnet, kept private and responsibly disclosed to project teams without any risk of being actually executed on live

systems resulting in real loss of funds or access

10. Which of the following is/are typicial manual review approach(es)?

Solution: A, C & D.
Symbolic checking is automated. Auditors have different approaches to manual reviewing smart contract code for vulnerabilities. Starting with access control, asset flow, inferring constraints and evaluating assumptions.

11. Access control analysis is a crytical part of manual review for the reason(s) that

Solution: C.
While the overall philosophy might be that smart contracts are permissionless, in reality, they do indeed have different permissions/roles for different actors who interact/use them.

Access control is the most fundamental security primitive which addresses 'who' has authorised access to 'what'. (In a formal access control model, the 'who' refers to subjects, 'what' refers to objects and an access control matrix indicates the permissions between subjects and objects.)
The general classification is that of users and admin(s). For purposes of guarded launch or otherwise, many smart contracts have an admin role that is typically the address that deployed the contract. Admins typically have control over critical configuration and application parameters including (emergency) transfers/withdrawals of contract funds.

12. Which of the following is/are true about vulnerability difficulty and impact?

Solution: C.
Per OWASP, likelihood or difficulty is a rough measure of how likely or difficult this particular vulnerability is to be uncovered and exploited by an attacker.

Many likelihood and impact evaluations are contentious and debatable between the audit and project teams, typically with security-conscious audit teams pressing for higher likelihood and impact and project teams downplaying the risks.

If there is any loss or locking up of funds then the impact is evaluated as High. Exploits that do not affect funds but disrupt the normal functioning of the system are typically evaluated as Medium. Anything else is of Low impact.

13. Application-level security constraints

Solution: B & C.
Auditors may need to infer business logic and their implied constraints directly from the code or from discussions with the project team and thereafter evaluate if those constraints/properties hold in all parts of the codebase.

However, application-level constraints are rules that are implicit to the business logic implemented and may not be explicitly described in the specification e.g. mint an `ERC721` token to the address when it makes a certain deposit of `ERC20` tokens to the smart contract and burn it when it withdraws the earlier deposit. Such constraints may have to be inferred by the auditors while manually analyzing the smart contract code.

14. Which of the following is/are typically true?

Solution: D.
Static analysis is a technique of analyzing program properties without actually executing the program.

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.

Instead of enumerating reachable states one at a time, the state space can sometimes be traversed more efficiently by considering large numbers of states at a single step.

15. Which of the following is/are generally true about asset flow analysis?

Solution: B & C.
Assets are Ether or `ERC20`/`ERC721`/other tokens managed by smart contracts. Given that exploits target assets of value, it makes sense to start evaluating the flow of assets into/outside/within/across smart contracts and their dependencies.

'Who': Assets should be withdrawn/deposited only by authorised/specified addresses as per application logic.

'When': Assets should be withdrawn/deposited only in authorised/specified time windows or under authorised/specified conditions as per application logic (when).

'What type': Assets, only of authorised/specified types, should be withdrawn/deposited as per application logic.

'How much': Assets, only in authorised/specified amounts, should be withdrawn/deposited as per application logic

16. Which of the following is/are generally true about control and data flow analyses?

Solution: A & B.
Interprocedural (procedure is just another name for a function) control flow is typically indicated by a call graph which shows which functions (callers) call which other functions (callees), across or within smart contracts.

Intraprocedural (i.e. within a function) control flow is dictated by conditionals (if/else), loops (for/while/do/continue/break) and return statements.

Interprocedural data flow is evaluated by analyzing the data (variables/constants) used as argument values for function parameters at call sites.

Intraprocedural data flow is evaluated by analyzing the assignment and use of (state/memory/calldata) variables/constants along the control flow paths within functions.

10.7 Audit Findings 101 Quiz Solutions

1. Based on the comments and code shown in the `InSecureumDAO` snippet

Solution: A, B & C.
While the payable `openDAO()` function is protected by the correctly implemented `onlyAdmin` modifier, it is always possible to force send Ether into a contract via `selfdestruct()`. The `onlyWhenOpen()` modifier only checks for the contracts own balance which can be bypassed by doing that. The payable `join()` function indeed checks for the `msg.value` to exactly match `membershipFee`.

2. Based on the comments and code shown in the `InSecureumDAO` snippet

Solution: D.
All state modifying functions, that can be accessed by users other than admin, are indeed correctly "protected" by the `onlyWhenOpen` modifier, but that modifier is, as explained in the previous answer, not correctly implemented itself. The only zero-address check is made during construction, but it's made against a state variable that will be in zero-state already (default value). The zero-address check should instead be made against the `_admin` variable which is to be set as the admin. There are several functions that sound more than critical enough to have events (e.g. `removeAllMembers`), but this contract isn't using events at all.

3. Reentrancy protection only on `join()` (assume it's correctly specified) indicates that

Solution: B.
A simply sounds like nonesense. Since it says that we should assume that reentrancy protection has been used correctly, and a reentrancy vulnerability requires making untrusted external calls, we can assume that it is at least likely, although not certain, that other functions do not.

4. Access control on `msg.sender` for DAO membership is required in

Solution: A & B.
It wouldn't make much sense to pay a membership fee if you are allowed to create and cast votes without it. There's no clear reason to prevent non-members from accessing winning outcomes though, since they'd be publicly readable on the blockchain anyway.

5. A commit/reveal scheme (a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden from others, with the ability to reveal the committed value later) is relevant for

Solution: C.
It's not possible to hide the `msg.sender` using a simple commit/reveal scheme and it wouldn't make much sense to try, since there's no clear advantage from temporarily hiding your membership. It also wouldn't make much sense to not disclose possible outcomes of a new vote, unless you want to make your members vote blindly on the options. It does make sense to hide what you are voting for until voting closes, since this makes it impossible to calculate how many members voted for a specific option and how many fake/sibyl members you'd exactly need to create in order to manipulate the vote.

6. Security concern(s) from missing input validation(s) is/are present in

Solution: A & D.
The `createVote()` function currently allows overwriting existing votes by specifying a previously used `_voteId`. It would probably be better to use an array instead of a mapping here and simply push new votes into it. `castVote()` has a modifier in place ensuring that a vote can only be cast on existing votes. Since function body should be assumed as correctly implemented, we should also assume there are no security concerns in regards to validation either. Without sanity/threshold checks when setting fees in `setMembershipFee()`, admins could practically close the DAO off, preventing new members from joining, which could certainly be considered a security concern for the protocol.

7. `removeAllMembers()` function

Solution: A & C.
What the function actually does is only removing the admin as member from the DAO, he'd still stay the admin though. Properly implementing this function would actually be rather difficult, since a simple delete on the mapping variable without specifying a key would not actually delete any of its values. Assuming it would work as advertised by its name you can certainly say it would be a critical function that should emit an event, which it currently does not.

8. `InSecureumDAO` will *not* be susceptible to something like the 2016 "DAO exploit"

Solution: B.
The 2016 "DAO exploit" was indeed a reentrancy issue caused by an external call within an Ether withdrawal function. But simply inheriting from `ReentrancyGuard.sol` will not prevent them since you actually have to apply

the `nonReentrant` modifier to relevant functions. There have indeed been some efforts to prevent reentrancy issues with changes made in Ethereum and `Solidity`, but none of them can be considered a "fix".

A recurrence of the 2016 "DAO exploit" is indeed still possible, although more unlikely since, thanks to all the attention it got, this anti-pattern is now widely known and rarely found anymore during audits.

10.8 Audit Findings 201 Quiz Solutions

1. Missing zero-address check(s) in the contract

Solution: B.
While the require statement in `startSale()` states that only the deployer may call the function AND the price needs to be not zero, the actual code uses OR which allows anyone to start the sale as long as they specify a valid price - but that can't be fixed by adding a zero-address check.

All proceeds appear to be intended to go to the benificiary and since there's no validation of the `_benificiary` address when it is set during construction, a zero-address could indeed put the sale proceeds to risk. In the given code, the internal `_mint(_to)` function is always called with `msg.sender` as `_to` value which can't be a zero-address.

2. Given that lower indexed/numbered `CryptoSAFU NFTs` have rarer traits (and are considered more valuable as commented in `_mint`), the implementation of `InSecureumNFT` is susceptible to the following exploits

Solution: A, B & C.
The index of a CryptoSAFU NFT depends on a `nonce` that increases after every mint and has an `internal` visibility preventing contracts to read its current value easily, which would allow them to predict an index for the current block. But a prediction is not necessary since a contract can simply call `_mint()` repeatedly every block and revert if the result is not desired, ensuring a refund.

The `msg.sender` is indeed also a variable for the "random" index generation, although it's very effective exploiting it, since you'd still have to pay the full price for each of those attempts because the nonce will change after each buy. There's also no need to generate a new address, you can just keep buying using the same address until you receive the desired NFT.

A miner would indeed be able to pre-calculate a desirable index off-chain by picking a specific `block.timestamp` and adding their mint-transaction to the beginning of their block.

3. The `getPrice()` function

Solution: A & B.
The price is multiplied by `saleDuration - elapsed` and while `saleDuration` stays constant, `elapsed` will increase over time, making the multiplicator value and therefore the price lower over time.

There's indeed a possibility for a free mint when `saleDuration` and `elapsed` have exactly the same value, which is not a very likely scenario though. Once `elapsed` is larger than `saleDuration` the subtraction won't underflow since that is handled by `if (elapsed > saleDuration)`. Since this function is called internally, it wouldn't make much sense to change its visibility to `external`.

4. `InSecureumNFT` contract is

Solution: B & C.
There are multiple external contract calls. The compiler version and absence of `unchecked` blocks should indeed prevent integer overflows/wrapping, but instead will cause reverts which could lead to Denial of Service.

The fact that `mint()` keeps checking the current balance instead of the actual `msg.value` and that the `onERC721Received` hook is called before the balance is transferred to the beneficiary, can indeed be exploited using a reentrancy attack.

5. Assuming `InSecureumNFT` contract is deployed in production (i.e. live for users) on mainnet without any changes to shown code

Solution: A & B.
Multiple comments throughout the code show a discrepancy between the configuration expected in production and the actual configuration that is currently implemented. A much better way to do it, would be parameterization by setting these values during construction, which allows using the same code without changes for both mainnet and testnets.

6. The function `startSale()`

Solution: A, B & C.
While the `require` statement in `startSale()` states that only the deployer may call the function AND the price needs to be not zero, the actual code uses OR, which allows anyone to start the sale as long as they specify a valid price. This also means that a price of 0 can be successful if the caller is the deployer. The `mint()` function requires `publicSale` to be `true`, which can only happen by calling `startSale()`.

7. The minting of NFTs

Solution: B, C & D.
Thanks to the refund mechanism after the actual price has been determined, the buyer does not need to send an exact amount of ETH. After refunding the buyer, what is left in the contracts balance and sent to the benificiary should indeed be the `salePrice`. The refund mechanism can be optimized by skipping

transfers when the current balance equals the price exactly.

8. The NFT sale

Solution: A, C & D.
`startSale()` is not checking whether `publicSale` is already `true`, allowing `saleStartTime` to be reset and also overwriting the `price` and can indeed be called by anyone since the authentication can be bypassed by simply specifying a `_price` $\neq 0$. The start of the sale would certainly be a good point to log an event, but events are currently completely missing from the contract.

10.9 RACE 4 Quiz Solutions

1. `InSecureum` **implements**

Solution: A & B.
The decimals value follows the standard but it typically returns `18` (`8` is atypical), imitating the relationship between Ether and Wei.

The `decreaseAllowance` and `increaseAllowance` functions were introduced in the `OpenZeppelin ERC20` implementation to mitigate frontrunning issues of the standard approve, but they are not part of the `ERC20` standard.

The `transfer` function is part of the standard though.

2. In `InSecureum`

Solution: A & C.
Since `decimals()` returns a constant hardcoded value without accessing storage other non-calldata information it can indeed be declared as `pure`.

Generally, functions prefixed with underscores should be internal or should not have the prefix. Making `_burn()` external would currently allow anyone to burn anyone else's balance. And the fact that `_mint()` is currently external allows anyone to mint as many InSecureum tokens as they wish.

3. `InSecureum transferFrom()`

Solution: A, B & C.
The subtraction within the `unchecked` block effectively allows anyone to steal anyone else's full token balance since subtracting from an allowance of 0 will cause an integer underflow and the allowance value will wrap since `0 - 1 == type(uint256).max`).

The fact that the function won't revert when subtracting from the allowance due to the `unchecked` block, can by itself be seen as an incorrect allowance check. The other check, skipping allowance subtraction when an "infinite approval" was given by setting the allowance to the maximum value of `uint256`, appears to be correct. Since the special handling for unlimited approvals prevents unnecessary storage updates, it is indicative of an optimization.

4. In `InSecureum`

Solution: C & D.
Neither function make use of the `unchecked` block which would allow integer overflows to happen in this version of `Solidity`.

The `decreaseAllowance` function does indeed not allow reducing the allowance to zero since the requirement enforces that the `subtractedValue` must always be smaller than `currentAllowance`. It would be better to use `>=` here to allow allowance reductions to zero. That requirement does make `Solidity`'s own integer underflow check for `currentAllowance - subtractedValue` redundant, so it could indeed be optimised with `unchecked{}`.

5. `InSecureum _transfer()`

Solution: D.
All of the addresses `_transfer()` uses are checked to make sure they're not zero-addresses. Neither integer overflows nor underflows are possible with this `Solidity` version without the use of `unchecked{}`.

6. `InSecureum _mint()`

Solution: A & C.
The `_mint` function is currently not ensuring that the receiving address is non-zero. The event emission appears to be correctly following `IERC20`:

```
event Transfer(address indexed from, address indexed to, uint256
    value);
```

This mint implementation overwrites the account's current balance instead of adding to it.

7. `InSecureum _burn()`

Solution: B.
It correctly applies zero-address validation on the account to burn from. The event permission is incorrect, `from` and `to` need to be switched around to follow the `IERC20` interface:

```
event Transfer(address indexed from, address indexed to, uint256
    value);
```

The balance update is correct and although an `unchecked` block is used, no underflow can happen thanks to the requirement before.

8. `InSecureum _approve()`

Solution: B & C.
Although no zero-address validation is missing the error messages have been confused with each other. The update of allowances is currently incorrect since it only adds the amount to the current allowance instead of setting it to the amount overwriting the old value.

10.10 RACE-X: Certora Quiz Solutions

1. When is the expression $p \Rightarrow q$ false?

Solution: D.
2. Is the following expression true? $p \wedge q \Rightarrow p$

Solution: C.
3. Is the following expression true? $(p \wedge (q \vee \neg p)) \wedge \neg q$

Solution: B.
4. Is the following expression true? $\neg(\neg p \vee \neg q \vee \neg r) \Leftrightarrow p \wedge q \wedge r$

Solution: A.
5. Is the following expression true? $\neg((\neg p \vee \neg q) \vee (\neg p \wedge \neg q))$

Solution: C.
6. Is the following expresison true? $\neg((p \wedge q) \vee (\neg p \wedge \neg q))$

Solution: C.
7. Given the expression: $M : p \wedge \vee \neg p$
Which of the following expressions implies the above given expression M?

Solution: A, B, C & D.

8. Given the expression: $M : p \wedge \wedge \neg p$
Which of the following expressions implies the above given expression M**?**

Solution: A, B, C & D.
9. Is the following expression true? $(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow r)$

Solution: C.
10. Is the following expression true? $((()p \Rightarrow q) \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$

Solution: A.
11. Given the below four properties for OpenZeppelin's ERC20 implementation, answer questions 11 to 13

P1: Forall user: `balanceOf(user)` can only change on `mint()`, `burn()`, `transfer()`, `transferFrom()`
P2: `totalSupply()` is the sum of `balanceOf()` over all users
P3: Forall user: `balanceOf(user) <= totalSupply()`
P4: Forall user: `balanceOf(user)` can only change on operation performed when `msg.sender == user` or when `allowance(user, msg.sender)` is not zero

Which of the below properties are correct properties of `ERC20`**?**

Solution: A, B & C.
12. Given the four properties for OpenZeppelin's ERC20 implementation shown in question 11,
and given the following buggy version of transferFrom:

```
function transferFrom(address _from, address _to, uint256 _value)
    public returns (bool) {
   require(_to != address(0));
   require(_value <= balances[_from]);
   require(_value <= allowed[_from][msg.sender]);
   balances[_from] = balances[_from].add(_value);
   balances[_to] = balances[_to].sub(_value);
   allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(
       _value);
   emit Transfer(_from, _to, _value);
   return true;
 }
```

Which of the below properties are violated?

Solution: D.
13. Given the four properties for OpenZeppelin's ERC20 implementation shown in question 11,
Is there an implication between P3 and P2?

Solution: Correct is A.

14. Assuming a correct implementation of `transferFrom`, as in OpenZeppelin's ERC20 implementation, and given the following pseudo unit-test code

```
uint256  bFrom = balances[from];
uint256 bTo = balances[to];
transferFrom(from, to, x);
assert($exp$);
```

Which of the following expressions are valid choices (should always hold) for `$exp$`?

Solution: B & C.

15. The below contract (same contract for questions 15 to 17) has a bug:

```
contract test {
   // Assume other required functionality is correctly implemented
   uint256 private constant MAX_FUND_RAISE = 100 ether;
   mapping (address => uint256) contributions;

    function contribute() external payable {
       require(address(this).balance != MAX_FUND_RAISE);
       contributions[msg.sender] += msg.value;
   }
}
```

Which of the following invariants should hold on a correct implementation of the contribute function?

P1: Forall user: `address(this).balance <= contributions[user]`
P2: Forall user: `contributions[user] <= address(this).balance`
P3: Forall user: `contributions[user] <= MAX_FUND_RAISE`
P4: `address(this).balance <= MAX_FUND_RAISE`

Solution: B, C & D.

16. Following the same contract of question 15, which of the following invariant(s) that is/are supposed to hold is/are violated due to the buggy implementation?

P1: Forall user: `address(this).balance <= contributions[user]`
P2: Forall user: `contributions[user] <= address(this).balance`
P3: Forall user: `contributions[user] <= MAX_FUND_RAISE`
P4: `address(this).balance <= MAX_FUND_RAISE`

Solution: C & D.

17. Following the same contract of question 15, the revert characteristic (conditions in which the function should revert) of a correct implementation of `contribute` is

Solution: B.

18. In the below contract (same contract for questions 18 & 19)

```
pragma solidity 0.7.0;

contract InSecureumToken {

mapping(address => uint) private balances;
uint public decimals = 10**18; // decimals of the token
uint public totalSupply; // total supply
uint MAX_SUPPLY = 100 ether; // Maximum total supply
event Mint(address indexed destination, uint amount);

function balanceOf(address u) public returns (uint256) {
   return balances[u];
}

function ethBalance(address u) public returns (uint256) {
   return u.balance;
}

function transfer(address to, uint amount) public {
   // save the balance in local variables
   // so that we can re-use them multiple times
   // without paying for SLOAD on every access
   uint balance_from = balances[msg.sender];
   uint balance_to = balances[to];
   require(balance_from >= amount);
   balances[msg.sender] = balance_from - amount;
   balances[to] = safeAdd(balance_to, amount);
}

/// @notice Allow users to buy a token. 1 ether = 10 tokens
/// @dev Users can send more ether than token to be bought, to
    donate a fee to the protocol team.
function buy(uint desired_tokens) public payable {
    // Check if enough ether has been sent
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(msg.value >= required_wei_sent);
    // Mint the tokens
    totalSupply = safeAdd(totalSupply, desired_tokens);
    balances[msg.sender] = safeAdd(balances[msg.sender],
        desired_tokens);
    emit Mint(msg.sender, desired_tokens);
}

/// @notice Add two values. Revert if overflow
function safeAdd(uint a, uint b) pure internal returns(uint) {
    if (a + b < a) {
        revert();
    }
    return a + b;
}
```

```
49 }
```

Given the following two properties: P1: `totalSupply()` is the sum of `balanceOf()` over all users.
P2: Monotonicity of `totalSupply` vs the contract's ether balance:

(a) `totalSupply` is increased iff (⇔) `this.balance` is increased and

(b) `totalSupply` is decreased iff (⇔) `this.balance` is decreased

Which of the existing issues in the code violates which property?

Solution: B & C.
Ethereum state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts.

19. Following the same contract of question 18 and assuming a correct implementation of `buy()` and `transfer()`, which properties should hold?

Solution: B.
20. In OpenZeppelin's implementation of ERC721, which of the following properties are correct specification assuming `user != 0`?

Solution: C.
21. In OpenZeppelin's implementation of ERC721, which of the following is necessarily correct after a successful (non-reverting) call to `transferFrom(from, to, tokenId)`?

Solution: A & C.

22. In OpenZeppelin's implementation of ERC721 Enumerable, which of the following expressions is true?

Solution: A & C.

23. Based on the lecture on "Auditing and Formal Verification: Better together" by Certora's CEO Mooly Sagiv, which of the following is generally accepted?

Solution: C & D.

24. Based on the lecture on "Auditing and Formal Verification: Better together" by Certora's CEO Mooly Sagiv, the takeaways are

Solution: A, B & C.

10.11 RACE 5 Quiz Solutions

1. `InSecureum balanceOf`

Solution: D.
Since the `_balances` state variable is only accessed once and immediately returned, caching doesn't make sense.

State mutability can't be changed to `pure` since the function accesses a state variable, that requires at least `view`.

It can't be changed to `external` because it is currently being called internally by the `balanceOfBatch()` function.

2. In `InSecureum`, array lengths mismatch check is missing in

Solution: A, B, C & D.
The public function `balanceOfBatch()` receives a list of `accounts` and a list of `ids`, both of which items get passed on to `balanceOf(accounts[i], ids[i]);`. To ensure that neither array is accessed out-of-bounds, it should be checked whether both lists are of the same length.

Neither the internal function `_safeBatchTransferFrom()` nor its public caller function `safeBatchTransferFrom()` check the length of passed ids and amounts. Therefore the check is missing.

The internal functions `_mintBatch()` and `_burnBatch()` are currently never called, but a contract extending `InSecureum` might. It would make sense to check the lengths of passed `ids` and `amounts` in them, so that public functions calling them do not need to remember to do so.

3. The security concern(s) with `InSecureum _safeTransferFrom()` is/are

Solution: A, B & C.
It is prefixed with an underscore, which is usually an indication of an `internal` visibility, and it's also called by a similarly named public `safeTransferFrom()` function that applies more input validation before calling it. This validation ensures that the sender actually has approval for the transfer of funds, which would be bypassed by this function being `public`. It should instead be `internal` allowing an inheriting contract to internally call it.

The new `fromBalance` is calculated within an `unchecked{}` block, bypassing integer underflow prevention measures of `Solidity` version `0.8.0^`. Since the `fromBalance` isn't checked for whether there's a sufficient balance for a transfer, this effectively allows sending unlimited amounts to the specified

recipient.

Neither `safeTransferFrom()` nor `_safeTransferFrom()` are checking whether the to address is non-zero, making it possible to accidentally burn tokens.

4. The security concern(s) with `InSecureum _safeBatchTransferFrom()` is/are

Solution: A & C.
Array length mismatch check is missing.

There's no usage of an `unchecked{}` block, therefore an integer underflow cannot happen with this `Solidity` version.

The new value of `fromBalance` is calculated but it's never actually updated in storage. This effectively allows sending the same tokens unlimited amount of times.

5. The security concern(s) with `InSecureum _mintBatch()` is/are

Solution: A, B & C.
Array length mismatch check is missing.

Comparing the emission of the `TransferBatch` event to other occurrences, it appears that `id`s and `amount`s have been accidentally swapped.

The zero-address check incorrectly ensures that the sender is non-zero (which would never be possible anyway) instead of ensuring that the receiving account is non-zero. This effectively allows minting to the zero-address, burning all minted tokens immediately.

6. The security concern(s) with `InSecureum _burn()` is/are

Solution: D.
The zero-address validation exists and is correctly checking the value of `from`.

There's no usage of an `unchecked{}` block, therefore an integer underflow cannot happen with this `Solidity` version.

The balance appears to be correctly updated after subtraction.

7. The security concern(s) with `InSecureum` `_doSafeTransferAcceptanceCheck()` is/are

Solution: B & C.
The `isContract()` function is correctly called on `to`, which is the receiving address (that is potentially a contract) that this function is supposed to check support of `ERC1155`, before tokens are sent to it, since they'd otherwise be stuck in a contract not supporting this standard.

Comparing `_doSafeTransferAcceptanceCheck()` and `_doSafeBatchTransferAcceptanceCheck()` shows a clear discrepancy when checking the return value, with the batch function's implementation correctly checking support for the `ERC1155` standard. This function is in fact currently doing the opposite, ensuring that tokens are only sent to contracts that do NOT support it.

The `isContract()` function currently returns `true` if the passed address is in fact NOT a contract (has a code length of 0). It should instead return true only when the address has a code length larger than 0, showing that there's currently a contract residing at account.

8. The security concern(s) with `InSecureum` `isContract()` implementation is/are

Solution: B & C.
A visibility of `internal` allowing inheriting contracts to use it appears appropriate.

The comparison should indeed be "bigger-than-zero" instead of "equals-zero". That is so because when an address is not a contract, it has a code length equal to 0. However, when the code length is bigger than 0, that means that a contract is currently residing in that address.
Ethereum not only has Contract accounts but also Externally Owned Accounts (EOA), which do not have any contract code but an off-chain public-private keypair instead.

Bibliography

[1] Blockchair: Universal blockchain explorer and search engine. https://blockchair.com/.

[2] Blockscout evm explorer. https://blockscout.com/.

[3] The chainlink docs. https://docs.chain.link/.

[4] The dappsys docs. https://github.com/dapphub/dappsys.

[5] Etherchain: Ethereum blockchain explorer. https://etherchain.org/.

[6] Ethereum improvement proposals. https://eips.ethereum.org/.

[7] Etherscan: The ethereum blockchain explorer. https://etherscan.io/.

[8] Ethplorer: Ethereum tokens explorer. https://ethplorer.io/.

[9] The openzeppelin docs. https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts.

[10] Solidity. https://en.wikipedia.org/wiki/Solidity.

[11] The solidity language. https://docs.soliditylang.org/.

[12] Uniswap v2. https://uniswap.org/blog/uniswap-v2/.

[13] Uniswap v3. https://uniswap.org/blog/uniswap-v2/.

[14] Weth9 smart contract. https://etherscan.io/address/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2#code.

[15] Andreas M. Antonopoulos. *Mastering Ethereum*. Stanford University Press, 2021.

[16] Vitalik Buterin. Ethereum whitepaper. https://ethereum.org/en/whitepaper/.

[17] Jacques Dafflon and Jordi Baylina. Eip-777: Token standard. https://eips.ethereum.org/EIPS/eip-777, Nov 2017.

[18] The Ethereum Foundation. Ethereum development documentation. https://ethereum.org/en/developers/docs/.

[19] The Ethereum Foundation. Ethereum eips. https://github.com/ethereum/EIPs/tree/master/EIPS.

[20] The Ethereum Foundation. Ethereum glossary. https://ethereum.org/en/glossary/.

[21] The Ethereum Foundation. Solidity changelog. https://github.com/ethereum/solidity/blob/develop/Changelog.md.

[22] Preethi Kasireddy. How does ethereum work, anyway? https://preethikasireddy.medium.com/how-does-ethereum-work-anyway-22d1df506369.

[23] Takenobu T. Ethereum evm illustrated. https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf.

[24] Gavin James Wood. Ethereum yellowpaper. https://ethereum.github.io/yellowpaper/paper.pdf.

[25] Gavin James Wood and The Ethereum Foundation. Language influences on solidity. https://docs.soliditylang.org/en/v0.8.9/language-influences.html.

[26] Gavin James Wood and The Ethereum Foundation. The solidity language documentation. https://docs.soliditylang.org/en/v0.8.9/.

Appendix A

Historical block Gas limits as of 4/10/2021

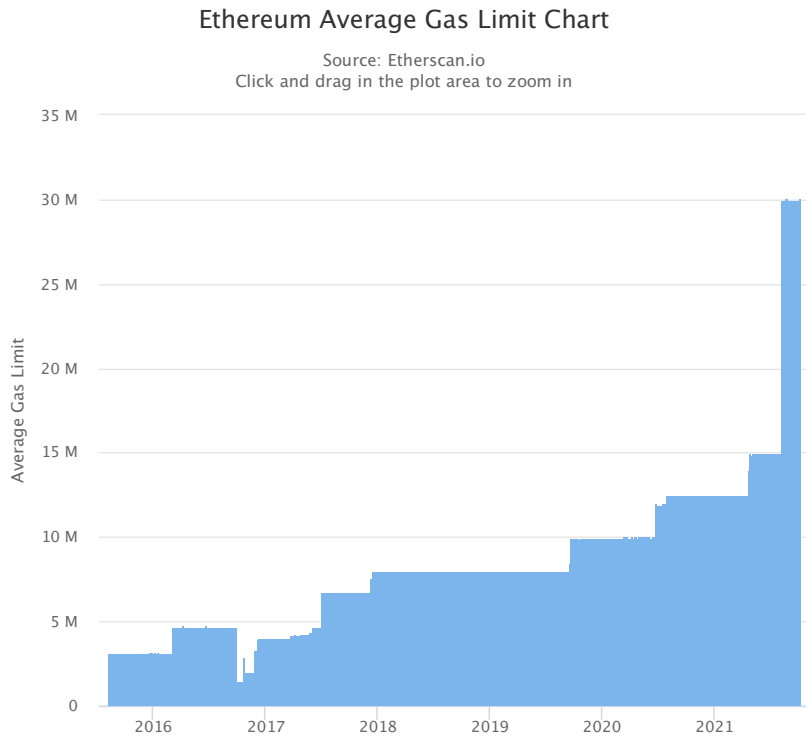

Figure A.1: Historical block Gas limits as of 4/10/2021

Appendix B

Chapter Summaries as Keypoints

B.1 Ethereum 101

(See https://secureum.substack.com/p/Ethereum-101 for the original article.)

1. Ethereum is "A Next-Generation Smart Contract and Decentralized Application Platform"
2. Ethereum is a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions.
3. Ethereum is an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system's state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs. It is often described as "the world computer."
4. The Ethereum platform enables developers to build powerful decentralized applications with built-in economic functions. While providing high availability, auditability, transparency, and neutrality, it also reduces or eliminates censorship and reduces certain counterparty risks.
5. Ethereum's purpose is not primarily to be a digital currency payment network. While the digital currency ether is both integral to and necessary for the operation of Ethereum, ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer.

6. Unlike Bitcoin, which has a very limited scripting language, Ethereum is designed to be a general-purpose programmable blockchain that runs a virtual machine capable of executing code of arbitrary and unbounded complexity. Where Bitcoin's Script language is, intentionally, constrained to simple true/false evaluation of spending conditions, Ethereum's language is Turing complete, meaning that Ethereum can straightforwardly function as a general-purpose computer.

7. The original blockchain, namely Bitcoin's blockchain, tracks the state of units of bitcoin and their ownership. You can think of Bitcoin as a distributed consensus state machine, where transactions cause a global state transition, altering the ownership of coins. The state transitions are constrained by the rules of consensus, allowing all participants to (eventually) converge on a common (consensus) state of the system, after several blocks are mined. Ethereum is also a distributed state machine. But instead of tracking only the state of currency ownership, Ethereum tracks the state transitions of a general-purpose data store, i.e., a store that can hold any data expressible as a key-value tuple.

8. Ethereum's core components:

    1. **P2P network**: Ethereum runs on the Ethereum main network, which is addressable on TCP port 30303, and runs a protocol called ÐΞVp2p.
    2. **Transactions**: Ethereum transactions are network messages that include (among other things) a sender, recipient, value, and data payload.
    3. **State machine**: Ethereum state transitions are processed by the Ethereum Virtual Machine (EVM), a stack-based virtual machine that executes bytecode (machine-language instructions). EVM programs, called "smart contracts," are written in high-level languages (e.g., Solidity or Vyper) and compiled to bytecode for execution on the EVM.
    4. **Data structures**: Ethereum's state is stored locally on each node as a database (usually Google's LevelDB), which contains the transactions and system state in a serialized hashed data structure called a Merkle Patricia Tree.

9. Ethereum's core components (continued):

    1. **Consensus algorithm**: Ethereum uses Bitcoin's consensus model, Nakamoto Consensus, which uses sequential single-signature blocks, weighted in importance by Proof-of-Work (PoW) to determine the longest chain and therefore the current state.
    2. However, this is being transitioned to a Proof-of-Stake (PoS) algorithm in Ethereum 2.0.

3. **Economic security**: Ethereum currently uses a PoW algorithm called Ethash, but this is being transitioned to a PoS algorithm in Ethereum 2.0.
4. **Clients**: Ethereum has several interoperable implementations of the client software, the most prominent of which are Go-Ethereum (Geth) and OpenEthereum. The others are Erigon, Nethermind and Turbo-geth. OpenEthereum is being deprecated to transition to Erigon, which is the former Turbo-geth.

10. Ethereum's ability to execute a stored program, in a state machine called the Ethereum Virtual Machine, while reading and writing data to memory makes it a Turing-complete system. Turing-complete systems face the challenge of the halting problem i.e. given an arbitrary program and its input, it is not solvable to determine whether the program will eventually stop running. So Ethereum cannot predict if a smart contract will terminate, or how long it will run. Therefore, to constrain the resources used by a smart contract, Ethereum introduces a metering mechanism called gas.

11. As the EVM executes a smart contract, it carefully accounts for every instruction (computation, data access, etc.). Each instruction has a predetermined cost in units of gas. When a transaction triggers the execution of a smart contract, it must include an amount of Gas that sets the upper limit of what can be consumed running the smart contract. The EVM will terminate execution if the amount of Gas consumed by computation exceeds the Gas available in the transaction. Gas is the mechanism Ethereum uses to allow Turing-complete computation while limiting the resources that any program can consume.

12. Ether needs to be sent along with a transaction and it needs to be explicitly earmarked for the purchase of gas, along with an acceptable Gas price. Just like at the pump, the price of Gas is not fixed. Gas is purchased for the transaction, the computation is executed, and any unused Gas is refunded back to the sender of the transaction.

13. A Decentralized Application, abbreviated as ĐApp, is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services and typically combines smart contracts with a web interface.

14. ĐApps represent a transition from "Web 2.0" where applications are centrally owned and managed to "Web 3.0" where applications are built on decentralised peer-to-peer protocols for compute (i.e. blockchain), storage and messaging.

15. Ethereum blockchain represents the decentralized compute part of Web 3.0. Swarm represents the decentralized storage and Whisper (now Waku) represents the decentralized messaging protocol.

16. Decentralization can be considered as three types
    1. Architectural decentralization
    2. Political decentralization
    3. Logical decentralization
17. Ethereum's currency unit is called ether or "ETH." Ether is subdivided into smaller units and the smallest unit is named wei. 10^3 wei is 1 Babbage, 10^6 wei is 1 Lovelace, 10^9 wei is 1 Shannon and 10^{18} wei is 1 Ether.
18. Ethereum uses public key cryptography to create public-private key pairs (considered a "pair" because the public key is derived from the private key) which are not used for encryption but for digital signatures.
19. Ethereum uses Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signatures (SECP-256k1 curve) which is based on Elliptic-curve cryptography (ECC), an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields.
20. An Ethereum private key is a 256-bit random number that uniquely determines a single Ethereum address also known as an account.
21. An Ethereum public key is a point on an elliptic curve calculated from the private key using elliptic curve multiplication. One cannot calculate the private key from the public key.
22. Ethereum state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts.
23. Ethereum account contains four fields
    1. **The nonce**, a counter used to make sure each transaction can only be processed once.
    2. The value, which is the account's current ether balance.
    3. The account's contract code, if present.
    4. The account's storage (empty by default).
24. Ethereum has two different types of accounts
    1. Externally Owned Accounts (**EOA**s) controlled by private keys.
    2. **Contract Accounts** controlled by their contract code.
25. Ownership of ether by EOAs is established through private keys, Ethereum addresses, and digital signatures. Anyone with a private key has control of the corresponding EOA account and any ether it holds.
26. An EOA has no code, and one can send messages from an EOA by creating and signing a transaction

27. A contract account has code and associated storage and every time it receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.

28. Smart contracts can be thought of as "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own ether balance and their own key/value store to keep track of persistent variables.

29. Ethereum uses Keccak-256 as its cryptographic hash function. Keccak-256 was the winning candidate for the SHA-3 competition held by NIST but is different from the finally adopted SHA-3 standard.

30. Ethereum address of an EOA account is the last 20 bytes (least significant bytes) of the Keccak-256 hash of the public key of the EOA's key pair.

31. Transactions are signed messages originated by an externally owned account (EOA), transmitted by the Ethereum network, and recorded on the Ethereum blockchain. Only transactions can trigger a change of state. Ethereum is a transaction-based state machine.

32. Transaction properties

    1. **Atomic**: it is all or nothing i.e. cannot be divided or interrupted by other transactions.
    2. **Serial**: Transactions are processed sequentially one after the other without any overlapping by other transactions.
    3. **Inclusion**: Transaction inclusion is not guaranteed and depends on network congestion and gasPrice among other things. Miners determine inclusion.
    4. **Order**: Transaction order is not guaranteed and depends on network congestion and gasPrice among other things. Miners determine order.

33. A transaction is a serialized binary message that contains the following components

    1. **nonce**: A sequence number, issued by the originating EOA, used to prevent message replay.
    2. **gasPrice**: The amount of ether (in wei) that the originator is willing to pay for each unit of gas.
    3. **gasLimit**: The maximum amount of Gas the originator is willing to pay for this transaction
    4. **recipient**: The destination Ethereum address.
    5. **value**: The amount of ether (in wei) to send to the destination.
    6. **data**: The variable-length binary data payload.

7. **v,r,s**: The three components of an ECDSA digital signature of the originating EOA.

34. **Nonce**: A scalar value equal to the number of transactions sent from the EOA account or, in the case of Contract accounts, it is the number of contract-creations made by the account.

35. **Gas price**: The price a transaction originator is willing to pay in exchange for gas. The price is measured in wei per Gas unit. The higher the Gas price, the faster the transaction is likely to be confirmed on the blockchain. The suggested Gas price depends on the demand for block space at the time of the transaction.

36. **Gas limit**: The maximum number of Gas units the transaction originator is willing to pay in order to complete the transaction.

37. **Recipient**: The 20-byte Ethereum address of the transaction's recipient which can be an EOA or a Contract account

    1. The Ethereum protocol does not validate recipient addresses in transactions. One can send a transaction to an address that has no corresponding private key or contract. Validation should be done at the user interface level.
    2. Note that there is no *from* address in the transaction because the EOA's public key can be derived from the v,r,s components of the ECDSA signature and the transaction originator's address can be derived from this public key

38. **Value**: The value of ether sent to the transaction recipient. If the recipient is an EOA then that account's balance will be increased by this value. If the recipient is a contract address then the result depends on any data that is sent as part of this transaction. If there is no data, the recipient contract's *receive* or *fallback* function is called if they are present. Depending on the implementation of those functions, the ether value is added to the contract account's balance or an exception occurs and this ether remains with the originator's account.

39. **Data**: The information (typically) sent to a contract account indicating the contract's function to be called and the arguments to that function.

40. **v,r,s**: r and s are the two parts of the ECDSA signature produced by the transaction originator using the private key. v is the recovery identifier which is calculated as either one of 27 or 28, or as the chain ID (Ethereum mainnet chainID is 1) doubled plus 35 or 36.

41. A**digital signature** serves three purposes in Ethereum

    1. Proves that the owner of the private key, who is by implication the owner of an Ethereum account, has authorized the spending of ether, or execution of a contract.

2. Guarantees **non-repudiation**: the proof of authorization is undeniable.
3. Proves that the transaction data has not been and cannot be modified by anyone after the transaction has been signed.

42. Contract creation transactions are sent to a special destination address called the zero address i.e. 0x0. A contract creation transaction contains a data payload with the compiled bytecode to create the contract. An optional ether amount in the value field will create the new contract with a starting balance.

43. Transactions vs Messages

    1. A transaction is produced by an EOA where an external actor sends a signed data package which either
        i. Triggers a message to another EOA where it leads to a transfer of value or
        ii. Triggers a message to a contract account where it leads to the recipient contract account running its code.
    2. A message is either
        i. Triggered by a transaction to another EOA or contract account or
        ii. Triggered internally within the EVM by a contract account when it executes the CALL family of opcodes and leads to the recipient contract account running its code or value transfer to the recipient EOA.

44. Transactions are grouped together into blocks. A blockchain contains a series of such blocks that are chained together.

45. **Blocks**: are batches of transactions with a hash of the previous block in the chain. This links blocks together (in a chain) because hashes are cryptographically derived from the block data. This prevents fraud, because one change in any block in history would invalidate all the following blocks as all subsequent hashes would change and everyone running the blockchain would notice. To preserve the transaction history, blocks are strictly ordered (every new block created contains a reference to its parent block), and transactions within blocks are strictly ordered as well.

46. **Ethereum node/client**: A node is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum nodes. A client is a specific implementation of Ethereum node. The two most common client implementations are Geth and OpenEthereum. Ethereum transactions are sent to Ethereum nodes to be broadcast across the peer-to-peer network.

47. **Miners**: are entities running Ethereum nodes that validate and execute these transactions and combine them into blocks. The process of validating each block by having a miner provide a mathematical proof is known as a "proof of work." Miners are rewarded for blocks accepted into the blockchain with a block reward in ether (currently 2 ETH). A miner also gets fees which is the ether spent on Gas by all the transactions included in the block.

48. **Block Gas limit** is set by miners and refers to the cap on the total amount of Gas expended by all transactions in the block, which ensures that blocks can't be arbitrarily large. Blocks therefore are not a fixed size in terms of the number of transactions because different transactions consume different amounts of gas. See **Appendix A** for historical block Gas limits.

49. Blocks take time to propagate through the network and multiple miners are simultaneously producing valid blocks. This leads to the blockchain considering multiple blocks at the same level but ultimately choosing only one block at any level that creates the canonical blockchain. This choice is dictated by Ethereum's Greedy Heaviest Observed Subtree (GHOST) protocol which includes stale blocks up to seven levels in the calculation of the longest chain. **Stale blocks** are called uncles or ommers.

50. **Consensus**: Decentralized consensus in the context of Ethereum refers to the process of determining which miner's block should be appended next to the blockchain. This involves two key components of Proof-of-Work (PoW) and the Longest-chain Rule. Miners apply these rules to build on the canonical blockchain. This is referred to as "Nakamoto Consensus" and is adapted from Bitcoin.

51. State is a mapping between addresses and account states implemented as a modified Merkle Patricia tree or trie. A Merkle tree or trie is a type of binary tree composed of a set of nodes with

    1. Leaf nodes at the bottom of the tree that contain the underlying data.
    2. Intermediate nodes, where each node is the hash of its two child nodes.
    3. A single root node formed from the hash of its two child nodes representing the top of the tree

52. Ethereum's proof-of-work algorithm is called "Ethash" (previously known as Dagger-Hashimoto)

    1. The algorithm is formally defined as

$$m = H_m \wedge n \leq \frac{2^{256}}{H_d}$$

with $(m, n) = \text{PoW}(H_{\not n}, H_n, d)$ where $H_{\not n}$ is the new block's header but without the nonce (n) and mix-hash (m) components; H_n is the nonce of the header; d is a large data set needed to compute the mixHash and H_d is the new block's difficulty value.

2. PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudorandom number cryptographically dependent on H and d.

53. Blocks contain block header, transactions and ommers' block headers. Block header contains

    1. **parentHash**: The Keccak 256-bit hash of the parent block's header, in its entirety.
    2. **ommersHash**: The Keccak 256-bit hash of the ommers list portion of this block.
    3. **beneficiary**: The 160-bit address to which all fees collected from the successful mining of this block be transferred.
    4. **stateRoot**: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied.
    5. **transactionsRoot**: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block.
    6. **receiptsRoot**: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block.
    7. **logsBloom**: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.
    8. **difficulty**: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp
    9. **number**: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero.
    10. **gasLimit**: A scalar value equal to the current limit of Gas expenditure per block.
    11. **gasUsed**: A scalar value equal to the total Gas used in transactions in this block.
    12. **timestamp**: A scalar value equal to the reasonable output of Unix's `time()` at this block's inception.
    13. **extraData**: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer.

14. **mixHash**: A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block.
15. **nonce**: A 64-bit value which, combined with the mixhash, proves that a sufficient amount of computation has been carried out on this block.

54. **stateRoot**, **transactionsRoot** and **receiptsRoot** are 256-bit hashes of the root nodes of modified Merkle-Patricia trees. The leaves of stateRoot are key-value pairs of all Ethereum address-account pairs, where each respective account consists of

    1. **nonce**: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account.
    2. **balance**: A scalar value equal to the number of Wei owned by this address.
    3. **storageRoot**: A 256-bit hash of the root node of a modified Merkle-Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values.
    4. **codeHash**: The hash of the EVM code of this account-this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction.

55. Transaction receipt is a tuple of four items comprising

    1. The **cumulative gas** used in the block containing the transaction receipt as of immediately after the transaction has happened.
    2. The **set of logs** created through execution of the transaction.
    3. The **Bloom filter** composed from information in those logs.
    4. The **status code** of the transaction.

56. **Gas refund and beneficiary**: Any unused Gas in a transaction (gasLimit minus Gas used by the transaction) is refunded to the sender's account at the same gasPrice. Ether used to purchase Gas used for the transaction is credited to the beneficiary address (specified in the block header), the address of an account typically under the control of the miner. This is the transaction "fees" paid to the miner.

57. EVM is a quasi Turing complete machine where the quasi qualification comes from the fact that the computation is intrinsically bounded through a parameter, gas, which limits the total amount of computation done. EVM is the runtime environment for smart contracts.

58. The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes (hence called bytecode), where each byte represents an operation.

59. The EVM is a simple stack-based architecture consisting of the stack, volatile memory, non-volatile storage with a word size of 256-bit (chosen to facilitate the Keccak256 hash scheme and elliptic-curve computations) and Calldata.

60. Stack is made up of 1024 256-bit elements. EVM instructions can operate with the top 16 stack elements. Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. `PUSH`, `POP`, \verbSWAP|, `DUP` etc.

61. Memory is a linear byte-array addressable at a byte-level and is volatile. All locations are well-defined initially as zero. This is accessed with `MLOAD`, `MSTORE` and `MSTORE8` instructions.

62. Storage is a 256-bit to 256-bit key-value store. Unlike memory, which is volatile, storage is non-volatile and is maintained as part of the system state. All locations are well-defined initially as zero. This is accessed with `SLOAD`/`SSTORE` instructions.

63. Calldata is a read-only byte-addressable space where the data parameter of a transaction or call is held. This is accessed with `CALLDATASIZE`/ `CALLDATALOAD`/`CALLDATACOPY` instructions.

64. EVM does not follow the standard von Neumann architecture. Rather than storing program code in generally accessible memory or storage, it is stored separately in a virtual ROM accessible only through a specialized instruction.

65. EVM uses big-endian ordering where the most significant byte of a word is stored at the smallest memory address and the least significant byte at the largest

66. EVM instruction set can be classified into 11 categories

    1. Stop and Arithmetic Operations.
    2. Comparison & Bitwise Logic Operations.
    3. SHA3.
    4. Environmental Information.
    5. Block Information.
    6. Stack, Memory, Storage and Flow Operations.
    7. Push Operations.
    8. Duplication Operations.

9. Exchange Operations.
10. Logging Operations.
11. System Operations.

67. Stop and Arithmetic Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x00 STOP 0 0` Halts execution.
    2. `0x01 ADD 2 1` Addition operation.
    3. `0x02 MUL 2 1` Multiplication operation.
    4. `0x03 SUB 2 1` Subtraction operation.
    5. `0x04 DIV 2 1` Integer division operation.
    6. `0x05 SDIV 2 1` Signed integer division operation (truncated).
    7. `0x06 MOD 2 1` Modulo remainder operation.
    8. `0x07 SMOD 2 1` Signed modulo remainder operation.
    9. `0x08 ADDMOD 3 1` Modulo addition operation.
    10. `0x09 MULMOD 3 1` Modulo multiplication operation.
    11. `0x0a EXP 2 1` Exponential operation.
    12. `0x0b SIGNEXTEND 2 1` Extend length of two's complement signed integer.

68. Comparison & Bitwise Logic Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x10 LT 2 1` Less-than comparison.
    2. `0x20 GT 2 1` Greater-than comparison.
    3. `0x12 SLT 2 1` Signed less-than comparison.
    4. `0x13 SGT 2 1` Signed greater-than comparison.
    5. `0x14 EQ 2 1` Equality comparison.
    6. `0x15 ISZERO 1 1` Simple not operator.
    7. `0x16 AND 2 1` Bitwise AND operation.
    8. `0x17 OR 2 1` Bitwise OR operation.
    9. `0x18 XOR 2 1` Bitwise XOR operation.
    10. `0x19 NOT 1 1` Bitwise NOT operation.
    11. `0x1a BYTE 2 1` Retrieve single byte from word.
    12. `0x1b SHL 2 1` Left shift operation.
    13. `0x1c SHR 2 1` Logical right shift operation.
    14. `0x1d SAR 2 1` Arithmetic (signed) right shift operation.

69. SHA3 (Opcode, Mnemonic, Stack items removed, Stack items placed, Description): `0x20 SHA3 2 1` Compute Keccak-256 hash.

70. Environmental Information (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x30 ADDRESS 0 1` Get address of currently executing account.
    2. `0x31 BALANCE 1 1` Get balance of the given account.
    3. `0x32 ORIGIN 0 1` Get execution origination address.
    4. `0x33 CALLER 0 1` Get caller address.
    5. `0x34 CALLVALUE 0 1` Get deposited value by the instruction/transaction responsible for this execution.
    6. `0x35 CALLDATALOAD 1 1` Get input data of current environment.
    7. `0x36 CALLDATASIZE 0 1` Get size of input data in current environment.
    8. `0x37 CALLDATACOPY 3 0` Copy input data in current environment to memory.
    9. `0x38 CODESIZE 0 1` Get size of code running in current environment.
    10. `0x39 CODECOPY 3 0` Copy code running in current environment to memory.
    11. `0x3a GASPRICE 0 1` Get price of Gas in current environment.
    12. `0x3b EXTCODESIZE 1 1` Get size of an account's code.
    13. `0x3c EXTCODECOPY 4 0` Copy an account's code to memory.
    14. `0x3d RETURNDATASIZE 0 1` Get size of output data from the previous call from the current environment.
    15. `0x3e RETURNDATACOPY 3 0` Copy output data from the previous call to memory.
    16. `0x3f EXTCODEHASH 1 1` Get hash of an account's code.

71. Block Information (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x40 BLOCKHASH 1 1` Get the hash of one of the 256 most recent complete blocks.
    2. `0x41 COINBASE 0 1` Get the block's beneficiary address.
    3. `0x42 TIMESTAMP 0 1` Get the block's timestamp.
    4. `0x43 NUMBER 0 1` Get the block's number.
    5. `0x44 DIFFICULTY 0 1` Get the block's difficulty.
    6. `0x45 GASLIMIT 0 1` Get the block's Gas limit.

72. Stack, Memory, Storage and Flow Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

1. `0x50 POP 1 0` Remove item from stack.
2. `0x51 MLOAD 1 1` Load word from memory.
3. `0x52 MSTORE 2 0` Save word to memory.
4. `0x53 MSTORE8 2 0` Save byte to memory.
5. `0x54 SLOAD 1 1` Load word from storage.
6. `0x55 SSTORE 2 0` Save word to storage.
7. `0x56 JUMP 1 0` Alter the program counter.
8. `0x57 JUMPI 2 0` Conditionally alter the program counter.
9. `0x58 PC 0 1` Get the value of the program counter prior to the increment corresponding to this instruction.
10. `0x59 MSIZE 0 1` Get the size of active memory in bytes.
11. `0x5a Gas 0 1` Get the amount of available gas, including the corresponding reduction for the cost of this instruction.
12. `0x5b JUMPDEST 0 0` Mark a valid destination for jumps. This operation has no effect on machine state during execution.

73. Push Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x60 PUSH1 0 1` Place 1 byte item on stack.
    2. `0x61 PUSH2 0 1` Place 2-byte item on stack.
    3. `PUSH3`, `PUSH4`, `PUSH5`,... `PUSH31` place 3, 4, 5, ..., 31 byte items on stack respectively
    4. `0x7f PUSH32 0 1` Place 32-byte (full word) item on stack.

74. Duplication Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x80 DUP1 1 2` Duplicate 1st stack item.
    2. `DUP2`, `DUP3`,... `DUP15` duplicate 2nd, 3rd, ..., 15th stack item respectively.
    3. `0x8f DUP16 16 17` Duplicate 16th stack item.

75. Exchange Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0x90 SWAP1 2 2` Exchange 1st and 2nd stack items.
    2. `0x91 SWAP2 3 3` Exchange 1st and 3rd stack items.
    3. `SWAP3`, `SWAP4`,... `SWAP15` exchange 1st and 3rd, 4th, ..., 15th stack items respectively.
    4. `0x9f SWAP16 17 17` Exchange 1st and 17th stack items.

76. Logging Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0xa0 LOG0 2 0` Append log record with no topics.
    2. `0xa1 LOG1 3 0` Append log record with one topic.
    3. `0xa2 LOG2 4 0` Append log record with two topics.
    4. `0xa3 LOG3 5 0` Append log record with three topics.
    5. `0xa4 LOG4 6 0` Append log record with four topics.

77. System Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

    1. `0xf0 CREATE 3 1` Create a new account with associated code.
    2. `0xf1 CALL 7 1` Message-call into an account.
    3. `0xf2 CALLCODE 7 1` Message-call into this account with an alternative account's code.
    4. `0xf3 RETURN 2 0` Halt execution returning output dat.
    5. `0xf4 DELEGATECALL 6 1` Message-call into this account with an alternative account's code, but persisting the current values for sender and value.
    6. `0xf5 CREATE2 4 1` Create a new account with associated code.
    7. `0xfa STATICCALL 6 1` Static message-call into an account.
    8. `0xfd REVERT 2 0` Halt execution reverting state changes but returning data and remaining gas.
    9. `0xfe INVALID` ∅ ∅ Designated invalid instruction.
    10. `0xff SELFDESTRUCT 1 0` Halt execution and register account for later deletion.

78. Gas costs for different instructions are different depending on their computational/storage load on the client. Examples are:

    1. `STOP`, `INVALID` and `REVERT` are 0 gas.
    2. Most arithmetic, logic and stack operations are 3-5 gas.
    3. `CALL*`, `BALANCE` and `EXT*` are 2600 gas.
    4. `MLOAD`/`MSTORE`/`MSTORE8` are 3 gas.
    5. `SLOAD` is 2100 Gas and `SSTORE` is 20000 Gas to set a storage slot from 0 to non-0 and 5000 Gas otherwise.
    6. `CREATE` is 32000 Gas and `SELFDESTRUCT` is 5000 gas.

79. A transaction reverts for different exceptional conditions such as running out of gas, invalid instructions etc. in which case all state changes made so far are discarded and the original state of the account is restored as it was before this transaction executed.

80. A transaction with a contract address destination has the contract's function target and the required arguments in the data field of the transaction. These are encoded according to the Application Binary Interface (**ABI**).

81. **Application Binary Interface** (ABI): The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction.

    1. **Interface functions** of a contract are **strongly typed**, known at compilation time and static.
    2. **Contracts** will have the interface definitions of any contracts they call available at **compile-time**.

82. **Function Selector**: The first four bytes of the call data for a function call specifies the function to be called.

    1. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function.
    2. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.
    3. Function Arguments: The encoded arguments follow the function selector from the fifth byte onwards.

83. **Block explorers**: are portals that allow anyone to see real-time data on blocks, transactions, accounts, contract interactions etc. A popular Ethereum block explorer is `etherscan.io`.

84. **Mainnet**: Short for "main network," this is the main public Ethereum blockchain. There are other Ethereum "testnets" where protocol or smart contract developers test their protocol upgrades or contracts. While mainnet uses real ETH, testnets use test ETH that can be obtained from faucets. The popular testnets are:

    1. **Görli**: A proof-of-authority (a small number of nodes are allowed to validate transactions and create blocks) testnet that works across clients.
    2. **Kovan**: A proof-of-authority testnet for those running OpenEthereum clients.
    3. **Rinkeby**: A proof-of-authority testnet for those running Geth client.
    4. **Ropsten**: A proof-of-work testnet. This means it's the best representation of mainnet Ethereum.

85. **Ethereum Improvement Proposals** (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards. Standards Track EIPs are separated into a number of types

    1. **Core**: Improvements requiring a consensus fork as well as changes that are not necessarily consensus critical but may be relevant to "core dev" discussions.
    2. **Networking**: Includes improvements around ÐΞVp2p and Light Ethereum Subprotocol, as well as proposed improvements to network protocol specifications of whisper and swarm.
    3. **Interface**: Includes improvements around client API/RPC specifications and standards, and also certain language-level standards like method names and contract ABIs. The label "interface" aligns with the interfaces repo and discussion should primarily occur in that repository before an EIP is submitted to the EIPs repository
    4. **ERC**: Application-level standards and conventions, including contract standards such as token standards (ERC-20), name registries, URI schemes, library/package formats, and wallet formats.
    5. **Meta**: Describes a process surrounding Ethereum or proposes a change to (or an event in) a process.
    6. **Informational**: Describes a Ethereum design issue, or provides general guidelines or information to the Ethereum community, but does not propose a new feature.

86. Eth2 or Ethereum 2.0: refers to a set of interconnected upgrades that will make Ethereum more scalable, more secure, and more sustainable.

87. **Immutable code**: Once a contract's code is deployed, it becomes immutable (with exceptions noted below). Standard software development practices that rely on being able to fix bugs and add new features to deployed code do not apply here. This represents a significant security challenge for smart contract development. There are three exceptions:

    1. The modified contract can be deployed at a new address (and old state carried over) but all interacting entities should be notified/enabled to interact with the updated contract at the new address. This is typically considered impractical.
    2. The modified contract can be deployed as a new implementation in a proxy pattern where the proxy points to the modified contract after the update. This is the most commonly used approach to update/add functionality.
    3. `CREATE2` opcode allows updating in place using `init_code`.

88. **Web3**: is a permissionless, trust-minimized and censorship-resistant network for transfer of value and information.

1. The popular approach to realise Web3 is to build it over a foundation of peer-to-peer network of nodes for compute, communication and storage.
2. In the Ethereum ecosystem, this is a combination of the Ethereum blockchain, Waku (previously Whisper) and Swarm respectively.
3. Privacy and anonymity are big motivating factors in Web3.
4. Most of the foundational security design principles and development practices from Web2 still apply to Web3. But Web3 security is indeed a paradigm shift along many frontiers.

89. **Languages**: Web2 programming languages such as JavaScript, Go, Rust and Nim are used extensively in Web3. But the entire domain of smart contracts is new and specific to Web3. Languages such as Solidity and Vyper were created exclusively for Web3.

90. **On-chain** vs **Off-chain**: Smart contracts are "on-chain" Web3 components and they interact with "off-chain" components that are very similar to Web2 software. So the major differences in security perspectives between Web3 and Web2 mostly narrow down to security considerations of smart contracts vis-a-vis Web2 software.

91. **Open-source** & **Transparent**: Given the emphasis on trust-minimization, Web3 software, especially smart contracts, are expected to be open-source by default.

    1. The deployed bytecode is also expected to be source code verified (on a service such as Etherscan). Security by obscurity with proprietary code is not part of Web3's ethos.
    2. All interactions with smart contracts are recorded on the blockchain as transactions. This includes the transactions' senders, data and outcome. Having complete visibility into the entire history of transactions and state transitions is akin to having a publicly accessible audit log of a system since inception.
    3. Furthermore, transactions that are still "in flight" and are yet to be confirmed on the blockchain are also publicly visible in pending transaction queues (i.e. mempools) and lend to front-running attacks.

92. **Unstoppable** & **Immutable**: Web3 applications, popularly known as Decentralized Applications (ĐApps), are expected to be unstoppable and immutable because they run on a decentralized blockchain network.

    1. There should not be any one entity that can unilaterally decide to stop a running ĐApp or make changes to it. Transactions and data on the blockchain are guaranteed to be immutable unless a majority of the network decides otherwise.

2. Smart contracts, in general, are expected (by users) to not have kill switches controlled by deployers. They are also expected to not be arbitrarily upgradeable. Both these stem from the Web3 goal of trust-minimization, i.e. lack of need to trust potentially malicious ĐApp developers. However, this makes fixing security vulnerabilities in deployed code and responding to exploits very challenging.

93. **Pseudonymous Teams** & **DAO**s: Perhaps inspired by Bitcoin's Satoshi Nakamoto, there is a trend among some project teams in Web3 to be pseudonymous and known only by their online handles.

    1. One reason for this could be to avoid any potential legal implications in future, given the regulatory uncertainty in this space. This makes it harder to associate any social reputation as it pertains to perceived security trustworthiness of the product or the processes behind its development. It also makes it tricky to hold anyone legally/socially liable or accountable.
    2. "Trust software not wetware" (i.e. people) is the mantra here. While this may be an extreme view, there are still social processes around rollout and governance of projects which affect security posture.
    3. To minimise the role and influence of a few privileged individuals in the lifecycle of projects, there is an increasing trend towards governance by token-holding community members - a Decentralized Autonomous Organization (DAO) of pseudonymous token-holding blockchain addresses making voting-based decisions on project treasury spending and protocol changes. While this reduces centralized points of wetware failure, it potentially slows down decision-making on security-critical aspects and may even lead to project forks.

94. **New Architecture**, **Language** & **Toolchains**: Ethereum has a new virtual machine (EVM) architecture which is a stack-based machine with 256-bit words and associated Gas semantics.

    1. Solidity language continues to dominate smart contracts without much real competition (except Vyper perhaps).
    2. The associated toolchains which include development environments (e.g. Truffle, Brownie, Hardhat), libraries (e.g. OpenZeppelin), security tools (e.g. Slither, MythX, Securify) and wallets (e.g. Metamask) are maturing but still playing catch up to the exponential growth of the space.

95. **Byzantine Threat Model**: The Web3 threat model is based on byzantine faults dealing with arbitrary malicious behavior and governed by mechanism design.

    1. Given the aspirational absence of trusted intermediaries, everyone and everything is meant to be untrusted by default. Participants

in this model include developers, miners/validators, infrastructure providers and users, all of whom could potentially be adversaries.

2. This is a fundamentally different threat model from that of Web2 where there are generalized notions of trusted insiders with authorized access to resources/assets that have to be protected against untrusted outsiders (and malicious insiders). Web3 is the ultimate zero-trust scenario.

96. **Keys** & **Tokens**: While "crypto" may indeed mean cryptocurrencies to some non-technical observers, it factually refers to cryptography which is a fundamental bedrock of Web3. As much as we unknowingly use cryptography in the Web2 world, Web3 is taking it to the masses. Cryptographic keys are first-class members of the Web3 world.

    1. Without the presence of Web2 trusted intermediaries who can otherwise reset passwords or restore accounts/assets from their centralized databases, Web3 ideologically pushes the onus of managing keys (and the assets they control) to end users in their wallets. Loss of private keys (or seed phrases) is irreversible and many assets have been lost to such incidents. This is a significant mindset shift from the Web2 world where passwords have become far too common, security pundits are tired of bemoaning the use of commonly reused simple passwords, password databases continue to be dumped and password-killing technologies continue to evade us. Web2 passwords here symbolize the role of trusted centralized intermediaries that Web3 is seeking to replace.
    2. Web2 security breaches targeting financial assets (i.e. excluding ransomware and botnets for DDoS) typically involve stealing of financial or personal data which is then sold on the dark web and used for monetary gain. This is getting much harder because of various checks and measures (both technical and regulatory) being put in place (at centralized intermediaries) to reduce such cybersecurity incidents and prevent anomalous asset transfers. When such unauthorised asset transfers do happen, the involved intermediaries may even cooperate to reverse such transactions and make good.
    3. The notion of assets in Web3 is fundamentally different. Cryptoassets are borderless digital tokens whose accounting ledger is managed by consensus on the blockchain and ownership is determined by access to corresponding cryptographic keys. If someone gets access to your private keys controlling cryptoassets, they can transfer those assets to blockchain addresses controlled by their keys. In a perfectly decentralized world, no intermediary (e.g. centralized exchange) should exist that can reverse such a loss - transactions are immutable. Because there are limited response options, preventive security measures become more critical in the Web3 space.

97. **Composability by Design**: Permissionless innovation and censorship-resistance are core aspirational goals of Web3.

    1. There are numerous stories of Web2 companies that initially enticed developers to build on their platforms only to shut them out later when they were perceived as a competitive threat.
    2. Web3 applications, especially smart contracts, are open by design and can be accessed permissionlessly by end users and other smart contracts alike.
    3. This composability lends itself to applications that can be layered on top of others like legos, which is great if everything holds up and new lego toys are reliably built on others. However, this unconstrained composability introduces unexpected cross-systemic dependencies that may trigger invalid assumptions across components (likely built by different teams with different constraints in mind) and expose attack surfaces or modes previously unconsidered.
    4. This makes characterizing Web3 vulnerabilities and exploit scenarios very challenging without deep knowledge of all interacting components, constraints and configurations.

98. **Compressed Timescales**: It feels like innovation in the Web3 space moves at warp speed. Aspects of transparent-development and composability-by-design are strong catalysts to accelerating permissionless and borderless participation which is further incentivized by Internet-native cryptoeconomic tokens - a perfect storm.

    1. This shrinks innovation timescales by orders of magnitude where new waves of experiments happen over weeks or months instead of the years it typically takes within the walled gardens of Web2. It may seem like the only moat here is the speed of execution.
    2. This compressed timescale has a tangible impact on security considerations during design, development and deployment. Corners are cut and shortcuts taken to ride new waves of hype. The end result is a poorly tested system that holds millions of dollars worth of tokens but is vulnerable to exploits.

99. **Test-in-Prod**: A combination of compressed timescale, unrestricted composability, byzantine threat model and challenges of replicating full state for predicting failure modes of interacting components built with rapidly evolving experimental software/tools in many ways forces realistic testing to happen only in production, i.e. on the "mainnet". This implies that complex technical and cryptoeconomic exploits may only be discoverable upon production deployment.

100. **Audit-as-a-Silver-Bullet**: **Secure Software Development Lifecycle** (SSDLC) processes for Web2 products have evolved over several decades

to a point where they are expected to meet some minimum requirements of a combination of internal validation, external assessments (e.g. product/process audits, penetration testing) and certifications depending on the value of managed assets, anticipated risk, threat model and the market domain of products (e.g. financial sector has stricter regulatory compliance requirements).

101. Web3 projects seem to increasingly rely on external audits as a stamp of security approval. This is typically justified by the lack of sufficient in-house security expertise. While the optics of this approach seems to falsely convince speculators, this approach is untenable for several reasons:

    1. Audits currently are very expensive because demand is much greater than supply for top-rated audit teams that have the experience and reputation to analyze complex projects.
    2. Audits are typically commissioned once at the end of project development just before production release.
    3. Upgrades to projects go unaudited for commercial or logistical reasons.
    4. The expectation (from the project team and users) is that audits are a panacea for all vulnerabilities and that the project is "bug-free" after a short audit (typically few weeks)

B.2 Solidity 101

(See https://secureum.substack.com/p/solidity-101 for the original article.)

1. `Solidity` is a high-level language for implementing smart contracts on Ethereum (and the blockchains) targeting the EVM. Solidity was proposed in 2014 by Gavin Wood and was later developed by Ethereum's Solidity team, led by Christian Reitwiessner, Alex Beregszaszi & others (see [10]).

2. It is influenced mainly by `C++`, a little from `Python` and early-on from `JavaScript`. The syntax and OOP concepts are from `C++`. `Solidity`'s modifiers, multiple inheritance, C3 linearization and the super keyword are influences from `Python`. Function-level scoping and var keyword were `JavaScript` influences early-on, but those have been reduced since `v0.4.0` (see [26] and [25]).

3. `Solidity` is statically typed, supports inheritance, libraries and complex user-defined types. It is a fully-featured high-level language.

4. The layout of a Solidity source file can contain an arbitrary number of `pragma` directives, `import` directives and `struct`/`enum`/`contract` definitions. The best-practices for layout within a contract is the following order: state variables, events, modifiers, constructor and functions.

5. **SPDX License Identifier**: `Solidity` source files are recommended to start with a comment indicating its license e.g.:"*// SPDX-License-Identifier: MIT*", where the compiler includes the supplied string in the bytecode metadata to make it machine readable. SPDX stands for Software Package Data Exchange.

6. **Pragmas**: The `pragma` keyword is used to enable certain compiler features or checks. A `pragma` directive is always local to a source file, so you have to add the `pragma` to all your files if you want to enable it in your whole project. If you import another file, the `pragma` from that file does not automatically apply to the importing file. There are two types

    1. Version
        i. Compiler version
        ii. ABI Coder version
    2. Experimental
        i. SMTChecker

7. **Version Pragma**: This indicates the specific `Solidity` compiler version to be used for that source file and is used as follows: `pragma solidity x.y.z;` where `x.y.z` indicates the version of the compiler.

1. Using the version pragma does not change the version of the compiler. It also does not enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.
2. The latest compiler versions are in the 0.8.z range
3. A different `y` in `x.y.z` indicates breaking changes e.g. 0.6.0 introduces breaking changes over 0.5.z. A different `z` in `x.y.z` indicates bug fixes.
4. A `^` symbol prefixed to `x.y.z` in the pragma indicates that the source file may be compiled only from versions starting with `x.y.z` until `x.(y+1).z`. For e.g., `pragma solidity ^0.8.3;` indicates that source file may be compiled with compiler version starting from 0.8.3 until any 0.8.z but not 0.9.z. This is known as a "floating pragma".
5. Complex pragmas are also possible using `>`, `>=`, `<` and `<=` symbols to combine multiple versions e.g. `pragma solidity >=0.8.0 <0.8.3;`.

8. **ABI Coder Pragma**: This indicates the choice between the two implementations of the ABI encoder and decoder: `pragma abicoder v1;` or `pragma abicoder v2;`.
    1. The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. It might produce less optimal code and has not received as much testing as the old encoder. This is activated by default.
    2. The set of types supported by the new encoder is a strict superset of the ones supported by the old one. Contracts that use it can interact with ones that do not without limitations. The reverse is possible only as long as the non-abicoder v2 contract does not try to make calls that would require decoding types only supported by the new encoder. The compiler can detect this and will issue an error. Simply enabling abicoder v2 for your contract is enough to make the error go away.
    3. This pragma applies to all the code defined in the file where it is activated, regardless of where that code ends up eventually. This means that a contract whose source file is selected to compile with ABI coder v1 can still contain code that uses the new encoder by inheriting it from another contract. This is allowed if the new types are only used internally and not in external function signatures.
9. **Experimental Pragma**: This can be used to enable features of the compiler or language that are not yet enabled by default
    1. **SMTChecker**: The use of `pragma experimental SMTChecker;` performs additional safety checks which are obtained by querying an SMT solver.

2. The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always `true`. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.
3. **Other checks**: Arithmetic underflow and overflow, Division by zero, Trivial conditions and unreachable code, Popping an empty array, Out of bounds index access, Insufficient funds for a transfer.

10. **Imports**: `Solidity` supports `import` statements to help modularise your code that are similar to those available in `JavaScript` (from ES6 on) e.g. `import "filename";`.

11. **Comments**: Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible. Comments are recommended as in-line documentation of what contracts, functions, variables, expressions, control and data flow are expected to do as per the implementation, and any assumptions/invariants made/needed. They help in readability and maintainability.

12. **NatSpec Comments**: NatSpec stands for "Ethereum Natural Language Specification Format." These are written with a triple slash (`///`) or a double asterisk block(`/** ... */`) directly above function declarations or statements to generate documentation in `JSON` format for developers and end-users. It is recommended that `Solidity` contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). These comments contain different types of tags

    1. `@title`: A title that should describe the contract/interface.
    2. `@author`: The name of the author (for contract, interface).
    3. `@notice`: Explain to an end user what this does (for contract, interface, function, public state variable, event).
    4. `@dev`: Explain to a developer any extra details (for contract, interface, function, state variable, event).
    5. `@param`: Documents a parameter (just like in `doxygen`) and must be followed by parameter name (for function, event).
    6. `@return`: Documents the return variables of a contract's function (function, public state variable).
    7. `@inheritdoc`: Copies all missing tags from the base function and must be followed by the contract name (for function, public state variable).

8. `@custom...`: Custom tag, semantics is application-defined (for everywhere).

13. **Contracts**: They are similar to classes in object-oriented languages in that they contain persistent data in state variables and functions that can modify these variables. Contracts can inherit from other contracts.

14. Contracts can contain declarations of State Variables, Functions, Function Modifiers, Events, Errors, Struct Types and Enum Types.

15. **State Variables**: They are variables that can be accessed by all functions of the contract and whose values are permanently stored in contract storage.

16. **State Visibility Specifiers**: State variables have to be specified as being public, internal or private.

    1. **public**: Public state variables are part of the contract interface and can be either accessed internally or via messages. An automatic getter function is generated.
    2. **internal**: Internal state variables can only be accessed internally from within the current contract or contracts deriving from it.
    3. **private**: Private state variables can only be accessed from the contract they are defined in and not even in derived contracts. Everything that is inside a contract is visible to all observers external to the blockchain. Making variables private only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

17. **State Variables**: Constant & Immutable

    1. State variables can be declared as *constant* or *immutable*. In both cases, the variables cannot be modified after the contract has been constructed. For *constant* variables, the value has to be fixed at compile-time, while for *immutable*, it can still be assigned at construction time i.e. in the constructor or point of declaration.
    2. For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. `block.timestamp`, `address(this).balance` or `block.number`) or execution data (`msg.value or gasleft()`) or makes calls to external contracts is disallowed.
    3. Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They cannot be read during construction time and can only be assigned once.

4. The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

18. Compared to regular state variables, the gas costs of constant and immutable variables are much lower

    1. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations.
    2. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, constant values can sometimes be cheaper than immutable values.
    3. The only supported types are strings (only for constants) and value types.

19. **Functions**: Functions are the executable units of code. Functions are usually defined inside a contract, but they can also be defined outside of contracts. They have different levels of visibility towards other contracts.

20. **Function parameters**: Function parameters are declared the same way as variables, and the name of unused parameters can be omitted. Function parameters can be used as any other local variable and they can also be assigned to.

21. **Function Return Variables**: Function return variables are declared with the same syntax after the returns keyword.

    1. The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their default value and have that value until they are (re-)assigned.
    2. You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or multiple ones) directly with the return statement.
    3. If you use an early return to leave a function that has return variables, you must provide return values together with the return statement.
    4. When a function has multiple return types, the statement return `(v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an implicit conversion.

22. **Function Modifiers**: They can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function. The function's control flow continues after the "_" in the preceding modifier. Multiple

modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented. The modifier can choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body. The `_` symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

23. **Function Visibility Specifiers**: Functions have to be specified as being public, external, internal or private
    1. **public**: Public functions are part of the contract interface and can be either called internally or via messages.
    2. **external**: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works).
    3. **internal**: Internal functions can only be accessed internally from within the current contract or contracts deriving from it.
    4. **private**: Private functions can only be accessed from the contract they are defined in and not even in derived contracts.
24. **Function Mutability Specifiers**: Functions can be specified as being pure or view
    1. **view** functions can read contract state but cannot modify it. This is enforced at runtime via `STATICCALL` opcode. The following are considered state modifying
        i. Writing to state variables.
        ii. Emitting events.
        iii. Creating other contracts.
        iv. Using selfdestruct.
        v. Sending Ether via calls.
        vi. Calling any function not marked view or pure.
        vii. Using low-level calls.
        viii. Using inline assembly that contains certain opcodes.
    2. **pure** functions can neither read contract state nor modify it. The following are considered reading from state
        i. Reading from state variables.
        ii. Accessing `address(this).balance` or `<address>.balance`.
        iii. Accessing any of the members of `block`, `tx`, `msg` (with the exception of `msg.sig` and `msg.data`).
        iv. Calling any function not marked pure.
        v. Using inline assembly that contains certain opcodes.

3. It is not possible to prevent functions from reading the state at the level of the EVM. It is only possible to prevent them from writing to the state via `STATICCALL`. Therefore, only **view** can be enforced at the EVM level, but not **pure**.

25. **Function Overloading**: A contract can have multiple functions of the same name but with different parameter types. This process is called "overloading."

    1. Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call.
    2. Return parameters are not taken into account for overload resolution.

26. **Free Functions**: Functions that are defined outside of contracts are called "free functions" and always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.

27. **Events**: They are an abstraction on top of the EVM's logging functionality. Emitting events cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible. The Log and its event data is not accessible from within contracts (not even from the contract that created them). Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

28. **Indexed Event Parameters**: Adding the attribute indexed for up to three parameters adds them to a special data structure known as "topics" instead of the data part of the log. If you use arrays (including string and bytes) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes). All parameters without the indexed attribute are ABI-encoded into the data part of the log. Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

29. **Emit**: Events are emitted using 'emit', followed by the name of the event and the arguments e.g. `emit Deposit(msg.sender, _id, msg.value);`.

30. **Struct Types**: They are custom defined types that can group several variables of same/different types together to create a custom data structure. The struct members are accessed using '`.`' e.g.: `struct s {address user; uint256 amount}` where `s.user` and `s.amount` access the struct members.

31. **Enums**: They can be used to create custom types with a finite set of constant values to improve readability. They need a minimum of one member and can have a maximum of 256. They can be explicitly converted to/from integers. The options are represented by unsigned integer values starting from 0. The default value is the first member.

32. **Constructor**: Contracts can be created "from outside" via Ethereum transactions or from within `Solidity` contracts. When a contract is created, its constructor (a function declared with the `constructor` keyword) is executed once. A constructor is optional and only one constructor is allowed. After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

33. **Receive Function**: A contract can have at most one receive function, declared using `receive() external payable {...}` without the `function` keyword. This function cannot have arguments, cannot return anything and must have external visibility and payable state mutability.

    1. The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers via `.send()` or `.transfer()`.
    2. In the worst case, the receive function can only rely on 2300 gas being available (for example when send or transfer is used), leaving little room to perform other operations except basic logging.
    3. A contract without a receive Ether function can receive Ether as a recipient of a `coinbase transaction` (aka miner block reward) or as a destination of a `selfdestruct`. A contract cannot react to such Ether transfers and thus also cannot reject them. This means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

34. **Fallback Function**: A contract can have at most one fallback function, declared using either `fallback() external[payable]` or `fallback(bytes calldata _input) external[payable] returns(bytes memory _output)`, both without the `function` keyword. This function must have external visibility.

    1. The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function. The fallback function always receives data, but in order to also receive Ether it must be marked payable.

2. In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available

35. `Solidity` is a statically-typed language, which means that the type of each variable (state and local) needs to be specified in code at compile-time. This is unlike dynamically-typed languages where types are required only with runtime values. Statically-typed languages perform compile-time type-checking according to the language rules. Other examples are `C`, `C++`, `Java`, `Rust`, `Go` or `Scala`.

36. `Solidity` has two categories of types: Value Types and Reference Types. Value Types are called so because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments. In contrast, Reference Types can be modified through multiple different names i.e. references to the same underlying variable.

37. **Value Types**: Types that are passed by value, i.e. they are always copied when they are used as function arguments or in assignments — Booleans, Integers, Fixed Point Numbers, Address, Contract, Fixed-size Byte Arrays (bytes1, bytes2, ..., bytes32), Literals (Address, Rational, Integer, String, Unicode, Hexadecimal), Enums, Functions.

38. **Reference Types**: Types that can be modified through multiple different names. Arrays (including Dynamically-sized bytes array bytes and string), Structs, Mappings.

39. **Default Values**: A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the typical "zero-state" of whatever the type is. For example, the default value for a bool is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, bytes and string, the default value is an empty array or string. For the enum type, the default value is its first member.

40. **Scoping**: Scoping in `Solidity` follows the widespread scoping rules of C99

    1. Variables are visible from the point right after their declaration until the end of the smallest `{ }`-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.
    2. Variables that are parameter-like (function parameters, modifier parameters, catch parameters,...) are visible inside the code block that follows - the body of the function/modifier for a function and modifier parameter and the catch block for a catch parameter.

3. Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

41. **Boolean**: `bool` Keyword and the possible values are constants `true` and `false`.

    1. Operators are `!` (logical negation), `&&` (logical conjunction, "and"), `||` (logical disjunction, "or"), `==` (equality) and `!=` (inequality).
    2. The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

42. **Integers**: `int`/`uint`: Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of 8 (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively. Operators are

    1. **Comparisons**: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`).
    2. **Bit operators**: `&`, `|`, `^` (bitwise exclusive or) `~` (bitwise negation).
    3. **Shift operators**: `<<` (left shift), `>>` (right shift).
    4. **Arithmetic operators**: `+`, `-`, unary `-` (only for signed integers), `*`, `/`, `%` (modulo), `**` (exponentiation).

43. Integers in `Solidity` are restricted to a certain range. For example, with `uint32`, this is from 0 up to $2^{32} - 1$. There are two modes in which arithmetic is performed on these types: The "wrapping" or "unchecked" mode and the "checked" mode. By default, arithmetic is always "checked", which means that if the result of an operation falls outside the value range of the type, the call is reverted through a failing assertion. You can switch to "unchecked" mode using `unchecked {...}`. This was introduced in compiler version 0.8.0.

44. **Fixed Point Numbers**: Fixed point numbers using keywords `fixed`/`ufixed` are not fully supported by `Solidity` yet. They can be declared, but cannot be assigned to or from. There are fixed-point libraries that are widely used for this such as `DSMath`, `PRBMath`, `ABDKMath64x64` etc.

45. **Address Type**: The address type comes in two types

    1. **address**: Holds a 20 byte value (size of an Ethereum address).
    2. **address payable**: Same as address, but with the additional members `transfer` and `send`.

    Address payable is an address you can send Ether to, while a plain address cannot be sent Ether.

1. **Operators** are `<=`, `<`, `==`, `!=`, `>=` and `>`.
2. **Conversions**: implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(<address>)`. Explicit conversions to and from address are allowed for `uint160`, integer literals, `bytes20` and contract types.
3. Only expressions of type `address` and contract-type can be converted to the type `address payable` via the explicit conversion `payable(...)`. For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a receive or a payable fallback function.

46. Members of Address Type

    1. `<address>.balance (uint256)`: balance of the Address in Wei.
    2. `<address>.code (bytes memory)`: code at the Address (can be empty).
    3. `<address>.codehash (bytes32)`: the codehash of the Address.
    4. `<address payable>.transfer(uint256 amount)`: send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable.
    5. `<address payable>.send(uint256 amount) returns (bool)`: send given amount of Wei to Address, returns false on failure, forwards 2300 gas stipend, not adjustable.
    6. `<address>.call(bytes memory) returns (bool, bytes memory)`: issue low-level `CALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable.
    7. `<address>.delegatecall(bytes memory) returns (bool, bytes memory)`: issue low-level `DELEGATECALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable.
    8. `<address>.staticcall(bytes memory) returns (bool, bytes memory)`: issue low-level `STATICCALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable.

47. **Transfer**: The `transfer` function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The `transfer` function reverts on failure. The code in receive function or if not present then in `fallback` function is executed with the transfer call. If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

48. **Send**: The `send` function is the low-level counterpart of `transfer`. If the execution fails then `send` only returns `false` and does not revert unlike `transfer`. So the return value of `send` must be checked by the caller.

49. **Call/Delegatecall/Staticcall**: In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single `bytes memory` parameter and return the success condition (as a `bool`) and the returned data (`bytes memory`). The functions `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` and `abi.encodeWithSignature` can be used to encode structured data.

    1. `gas` and `value` modifiers can be used with these functions (`delegatecall` doesn't support `value`) to specify the amount of gas and Ether value passed to the callee.
    2. With `delegatecall`, only the code of the given address is used but all other aspects (storage, balance, msg.sender etc.) are taken from the current contract. The purpose of `delegatecall` is to use library/-logic code which is stored in callee contract but operate on the state of the caller contract.
    3. With `staticcall`, the execution will revert if the called function modifies the state in any way.

50. **Contract Type**: Every contract defines its own type. Contracts can be explicitly converted to and from the `address` type. Contract types do not support any operators. The members of contract types are the external functions of the contract including any state variables marked as public.

51. **Fixed-size Byte Arrays**: The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from º to up to 32. The type `byte[]` is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the `bytes` type instead.

52. **Literals**: They can be of 5 types

    1. **Address Literals**: Hexadecimal literals that pass the address checksum test are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. The mixed-case address checksum format is defined in EIP-55.
    2. **Rational and Integer Literals**: Integer literals are formed from a sequence of numbers in the range 0-9. Decimal fraction literals are formed by a `.` with at least one number on one side. Scientific notation is also supported, where the base can have fractions and the exponent cannot. Underscores can be used to separate the digits of a numeric literal to aid readability and are semantically ignored.
    3. **String Literals**: String literals are written with either double or single-quotes (`"foo"` or `'bar'`). They can only contain printable `ASCII` characters and a set of escape characters.

4. **Unicode Literals**: Unicode literals prefixed with the keyword `unicode` can contain any valid `UTF-8` sequence. They also support the very same escape sequences as regular string literals.
5. **Hexadecimal Literals**: Hexadecimal literals are hexadecimal digits prefixed with the keyword `hex` and are enclosed in double or single-quotes e.g. `hex"001122FF"`, `hex'0011_22_FF'`.

53. **Enums**: Enums are one way to create a user-defined type in `Solidity`. They require at least one member and its default value when declared is the first member. They cannot have more than 256 members.

54. **Function Types**: Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. They come in two flavours - internal and external functions. Internal functions can only be called inside the current contract. External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

55. **Reference Types & Data Location**: Every reference type has an additional annotation — the data location where it is stored. There are three data locations: `memory`, `storage` and `calldata`.

    1. `memory`: whose lifetime is limited to an external function call.
    2. `storage`: whose lifetime is limited to the lifetime of a contract and the location where the state variables are stored.
    3. `calldata`: which is a non-modifiable, non-persistent area where function arguments are stored and behaves mostly like memory. It is required for parameters of external functions but can also be used for other variables.

56. **Data Location & Assignment**: Data locations are not only relevant for persistence of data, but also for the semantics of assignments.

    1. Assignments between storage and memory (or from calldata) always create an independent copy.
    2. Assignments from memory to memory only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
    3. Assignments from storage to a local storage variable also only assign a reference.
    4. All other assignments to storage always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

57. **Arrays**: Arrays can have a compile-time fixed size, or they can have a dynamic size.

1. The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.
2. Indices are zero-based.
3. Array elements can be of any type, including `mapping` or `struct`.
4. Accessing an array past its end causes a failing assertion.

58. **Array members**

    1. `length`: returns number of elements in array.
    2. `push()`: appends a zero-initialised element at the end of the array and returns a reference to the element.
    3. `push(x)`: appends a given element at the end of the array and returns nothing.
    4. `pop`: removes an element from the end of the array and implicitly calls `delete` on the removed element.

59. Variables of type `bytes` and `string` are special arrays.

    1. `bytes` is similar to `byte[]`, but it is packed tightly in `calldata` and `memory`.
    2. `string` is equal to `bytes` but does not allow length or index access.
    3. `Solidity` does not have string manipulation functions, but there are third-party string libraries.
    4. Use`bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data.
    5. Use `bytes` over `byte[]` because it is cheaper, since `byte[]` adds 31 padding bytes between the elements.
    6. If you can limit the length to a certain number of bytes, always use one of the value types `bytes1` to `bytes32` because they are much cheaper.

60. **Memory Arrays**: Memory arrays with dynamic length can be created using the `new` operator.

    1. As opposed to storage arrays, it is not possible to resize memory arrays i.e. the `.push` member functions are not available.
    2. You either have to calculate the required size in advance or create a new memory array and copy every element.

61. **Array Literals**: An array literal is a comma-separated list of one or more expressions, enclosed in square brackets (`[...]`).

    1. It is always a statically-sized memory array whose length is the number of expressions.

2. The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a `type error` if this is not possible.
3. Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays.

62. Gas costs of `push()` and `pop()`: Increasing the length of a storage array by calling `push()` has constant gas costs because storage is zero-initialised, while decreasing the length by calling `pop()` has a cost that depends on the "size" of the element being removed. If that element is an `array`, it can be very costly, because it includes explicitly clearing the removed elements similar to calling `delete` on them.

63. **Array Slices**: Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a `uint256` type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

    1. If start is greater than end or if end is greater than the length of the array, an exception is thrown.
    2. Both start and end are optional: start defaults to 0 and end defaults to the length of the array.
    3. Array slices do not have any members.
    4. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.
    5. Array slices do not have a type name which means no variable can have an array slices as type and they only exist in intermediate expressions.
    6. Array slices are only implemented for `calldata` arrays.
    7. Array slices are useful to ABI-decode secondary data passed in function parameters.

64. **Struct Types**: Structs help define new aggregate types by combining other value/reference types into one unit. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays. It is not possible for a struct to contain a member of its own type.

65. **Mapping Types**: Mappings define key-value pairs and are declared using the syntax `mapping(_KeyType => _ValueType) _VariableName`.

    1. The `_KeyType` can be any built-in value type, `bytes`, `string`, or any contract or enum type. Other user-defined or complex types, such as `mappings`, `structs` or array types are not allowed. `_ValueType` can be any type, including `mappings`, `arrays` and `structs`.

2. Key data is not stored in a mapping, only its keccak256 hash is used to look up the value.
3. They do not have a length or a concept of a key or value being set.
4. They can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions.
5. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.
6. You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that.

66. Operators Involving LValues (i.e. a variable or something that can be assigned to)

    1. `a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=` and `^=` are defined accordingly.
    2. `a++` and `a--` are equivalent to `a += 1` and `a -= 1` respectively, but the expression itself still has the previous value of `a`.
    3. In contrast, `--a` and `++a` have the same effect on a but return the value after the change.

67. `delete`

    1. `delete a` assigns the initial value for the type to `a`.
    2. For integers it is equivalent to `a = 0`.
    3. For arrays, it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value.
    4. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched.
    5. For structs, it assigns a struct with all members reset.
    6. `delete` has no effect on mappings. So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.
    7. For mappings, individual keys and what they map to can be deleted: If a is a mapping, then `delete a[x]` will delete the value stored at `x`.

68. **Implicit Conversions**: An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators.

    1. implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

2. For example, `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256`, because `uint256` cannot hold values such as `-1`.

69. **Explicit Conversions**: If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler e.g. `int` to `uint`.

    1. If an integer is explicitly converted to a smaller type, higher-order bits are cut off.
    2. If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end).
    3. Fixed-size bytes types while explicitly converting to a smaller type and will cut off the bytes to the right.
    4. Fixed-size bytes types while explicitly converting to a larger type and will pad bytes to the right.

70. Conversions between Literals and Elementary Types

    1. Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation.
    2. Decimal number literals cannot be implicitly converted to fixed-size byte arrays.
    3. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the bytes type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type.
    4. String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters matches the size of the bytes type.

71. A literal number can take a suffix of wei, gwei (10^9) or ether (10^{18}) to specify a sub-denomination of Ether.

72. Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit where 1 == 1 seconds,1 minutes == 60 seconds, 1 hours == 60 minutes, 1 days == 24 hours and 1 weeks == 7 days.

    1. Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds.
    2. These suffixes cannot be applied directly to variables but can be applied by multiplication.

73. **Block** and **Transaction Properties**:

    1. `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent, excluding current, blocks.
    2. `block.chainid (uint)`: current chain id.
    3. `block.coinbase (address payable)`: current block miner's address.
    4. `block.difficulty (uint)`: current block difficulty.
    5. `block.gaslimit (uint)`: current block gaslimit.
    6. `block.number (uint)`: current block number.
    7. `block.timestamp (uint)`: current block timestamp as seconds since unix epoch.
    8. `msg.data (bytes calldata)`: complete calldata.
    9. `msg.sender (address)`: sender of the message (current call).
    10. `msg.sig (bytes4)`: first four bytes of the calldata (i.e. function identifier).
    11. `msg.value (uint)`: number of wei sent with the message.
    12. `tx.gasprice (uint)`: gas price of the transaction.
    13. `gasleft() returns (uint256)`: remaining gas.
    14. `tx.origin (address)`: sender of the transaction (full call chain).

74. The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every external function call. This includes calls to library functions.

75. Do not rely on `block.timestamp` or `blockhash` as a source of randomness. Both the timestamp and the block hash can be influenced by miners to some degree. The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

76. The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

77. ABI Encoding and Decoding Functions:

    1. `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI-decodes the given data, while the types are given in parentheses as second argument.
    2. `abi.encode(...) returns (bytes memory)`: ABI-encodes the given arguments.

3. `abi.encodePacked(...) returns (bytes memory)`: Performs packed encoding of the given arguments. Note that packed encoding can be ambiguous!
4. `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: ABI-encodes the given arguments starting from the second and prepends the given four-byte selector.
5. `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`.

78. Error Handling:

    1. `assert(bool condition)`: causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.
    2. `require(bool condition)`: reverts if the condition is not met - to be used for errors in inputs or external components.
    3. `require(bool condition, string memory message)`: reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.
    4. `revert()`: abort execution and revert state changes.
    5. `revert(string memory reason)`: abort execution and revert state changes, providing an explanatory string.

79. Mathematical and Cryptographic Functions:

    1. `addmod(uint x, uint y, uint k) returns (uint)`: compute $(x + y)\%k$ where the addition is performed with arbitrary precision and does not wrap around at 2^{256}. Assert that `k != 0` starting from version 0.5.0.
    2. `mulmod(uint x, uint y, uint k) returns (uint)`: *compute*$(x * y)\%k$ where the multiplication is performed with arbitrary precision and does not wrap around at 2^{256}. Assert that `k != 0` starting from version 0.5.0.
    3. `keccak256(bytes memory) returns (bytes32)`: compute the Keccak-256 hash of the input.
    4. `sha256(bytes memory) returns (bytes32)`: compute the SHA-256 hash of the input.
    5. `ripemd160(bytes memory) returns (bytes20)`: compute RIPEMD-160 hash of the input.
    6. `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature:

r = {first 32 bytes of signature}, s = {second 32 bytes of signature},
v = {final 1 byte of signature}. `ecrecover` returns an address, and not an `address payable`.

80. Contract Related:
    1. `this` (current contract's type): the current contract, explicitly convertible to `address`.
    2. `selfdestruct(address payable recipient)`: Destroy the current contract, sending its funds to the given Address and end execution.
81. `selfdestruct` has some peculiarities: the receiving contract's receive function is not executed and the contract is only really destroyed at the end of the transaction and revert's might "undo" the destruction.
82. **Type Information**: The expression `type(x)` can be used to retrieve information about the `type x`, where `x` can be either a contract or an integer type. For a `contract type C`, the following type information is available:
    1. `type(C).name`: The name of the contract.
    2. `type(C).creationCode`: Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the `CREATE2` opcode. This property cannot be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.
    3. `type(C).runtimeCode`: Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of `C`. If `C` has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with `.creationCode` also apply for this property.
    4. For an interface type I, the following type information is available: `type(I).interfaceId`: A bytes4 value containing the EIP-165 interface identifier of the given interface I. This identifier is defined as the XOR of all function selectors defined within the interface itself - excluding all inherited functions.
83. For an `integer type T`, the following type information is available:
    1. `type(T).min`: The smallest value representable by `type T`.
    2. `type(T).max`: The largest value representable by `type T`.

84. **Control Structures**: `Solidity` has `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from `C` or `JavaScript`.

    1. Parentheses can not be omitted for conditionals, but curly braces can be omitted around single-statement bodies.
    2. Note that there is no type conversion from non-boolean to boolean types as there is in `C` and `JavaScript`, so `if (1) {...}` is not valid `Solidity`.

85. **Exceptions**: `Solidity` uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller

    1. When exceptions happen in a sub-call, they "bubble up" (i.e., exceptions are rethrown) automatically. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `staticcall`: they return false as their first return value in case of an exception instead of "bubbling up".
    2. Exceptions in external calls can be caught with the `try/catch` statement.
    3. Exceptions can contain data that is passed back to the caller. This data consists of a 4-byte selector and subsequent ABI-encoded data. The selector is computed in the same way as a function selector, i.e., the first four bytes of the keccak256-hash of a function signature - in this case an error signature.
    4. `Solidity` supports two error signatures: `Error(string)` and `Panic(uint256)`. The first ("`error`") is used for "regular" error conditions while the second ("`panic`") is used for errors that should not be present in bug-free code.

86. The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

87. The `assert` function creates an error of type `Panic(uint256)`. `assert` should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a `Panic`, not even on invalid external input.

88. A `Panic` exception is generated in the following situations. The error code supplied with the error data indicates the kind of `panic`:

    1. `0x01`: If you call `assert` with an argument that evaluates to `false`.
    2. `0x11`: If an arithmetic operation results in underflow or overflow outside of an unchecked `{ ... }` block.

3. `0x12`: If you divide or modulo by zero (e.g. `5/0` or `23%0`).
4. `0x21`: If you convert a value that is too big or negative into an `enum` type.
5. `0x22`: If you access a `storage byte array` that is incorrectly encoded.
6. `0x31`: If you call `.pop()` on an empty array.
7. `0x32`: If you access an `array`, `bytesN` or an array slice at an out-of-bounds or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
8. `0x41`: If you allocate too much memory or create an array that is too large.
9. `0x51`: If you call a zero-initialized variable of internal function type.

89. The `require` function either creates an error of type `Error(string)` or an error without any error data and it should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts. You can optionally provide a message string for `require`, but not for `assert`.

90. A `Error(string)` exception (or an exception without data) is generated in the following situations:

    1. Calling `require` with an argument that evaluates to `false`.
    2. If you perform an external function call targeting a contract that contains no code.
    3. If your contract receives Ether via a public function without payable modifier (including the constructor and the fallback function).
    4. If your contract receives Ether via a public getter function.

91. `revert`: A direct revert can be triggered using the `revert` statement and the `revert` function. The `revert` statement takes a custom error as a direct argument without parentheses: `revert CustomError(arg1, arg2)`. The `revert()` function is another way to trigger exceptions from within other code blocks to flag an error and revert the current call. The function takes an optional string message containing details about the error that is passed back to the caller and it will create an `Error(string)` exception. Using a custom error instance will usually be much cheaper than a string description, because you can use the name of the error to describe it, which is encoded in only four bytes. A longer description can be supplied via `NatSpec` which does not incur any costs.

92. `try/catch`: The `try` keyword has to be followed by an expression representing an external function call or a contract creation (`new ContractName()`). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The returns

part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract's execution continues inside the first success block. If the end of the success block is reached, execution continues after the `catch` blocks.

93. `Solidity` supports different kinds of `catch` blocks depending on the type of error

    1. `catch Error(string memory reason) {...}`: This `catch` clause is executed if the error was caused by `revert("reasonString")` or `require(false, "reasonString")` (or an internal error that causes such an exception).
    2. `catch Panic(uint errorCode) {...}`: If the error was caused by a `panic`, i.e. by a failing assert, division by zero, invalid array access, arithmetic overflow and others, this catch clause will be run.
    3. `catch (bytes memory lowLevelData) {...}`: This clause is executed if the error signature does not match any other clause, if there was an error while decoding the error message, or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.
    4. `catch {...}`: If you are not interested in the error data, you can just use `catch {...}` (even as the only `catch` clause) instead of the previous clause.

94. If execution reaches a `catch`-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a `catch` block or the execution of the `try/catch` statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).

95. The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract: The error might have happened deeper down in the call chain and the called contract just forwarded it. Also, it could be due to an out-of-gas situation and not a deliberate error condition: the caller always retains 63/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left.

96. **Programming style**: coding conventions for writing `Solidity` code. Style is about consistency. Consistency with style is important. Consistency within a project is more important. Consistency within one module or function is most important. Two main categories

    1. Layout
    2. Naming Conventions

Programming style affects readability and maintainability, both of which affect security.

97. **Code Layout**

    1. **Indentation**: Use 4 spaces per indentation level.
    2. **Tabs or Spaces**: Spaces are the preferred indentation method. Mixing tabs and spaces should be avoided.
    3. **Blank Lines**: Surround top level declarations in solidity source with two blank lines.
    4. **Maximum Line Length**: Keeping lines to a maximum of 79 (or 99) characters helps readers easily parse the code.
    5. **Wrapped lines** should conform to the following guidelines: The first argument should not be attached to the opening parenthesis. One, and only one, indent should be used. Each argument should fall on its own line. The terminating element, `);`, should be placed on the final line by itself.
    6. **Source File Encoding**: `UTF-8` or `ASCII` encoding is preferred.
    7. **Imports**: `import` statements should always be placed at the top of the file.
    8. **Order of Functions**: Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. Functions should be grouped according to their visibility and ordered: `constructor`, `receive` function (if exists), `fallback` function (if exists), external, public, internal, private. Within a grouping, place the view and pure functions last.

98. More Code Layout

    1. **Whitespace in Expressions**: Avoid extraneous whitespace in the following situations — Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.
    2. **Control Structures**: The braces denoting the body of a contract, library, functions and structs should: open on the same line as the declaration, close on their own line at the same indentation level as the beginning of the declaration. The opening brace should be preceded by a single space.
    3. **Function Declaration**: For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration. The closing brace should be at the same indentation level as the function declaration. The opening brace should be preceded by a single space.
    4. **Mappings**: In variable declarations, do not separate the keyword `mapping` from its type by a space. Do not separate any nested `mapping` keyword from its type by whitespace.

5. **Variable Declarations**: Declarations of array variables should not have a space between the type and the brackets.
6. Strings should be quoted with double-quotes instead of single-quotes.
7. **Operators**: Surround operators with a single space on either side. Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence.This is meant to allow for improved readability for complex statements. You should always use the same amount of whitespace on either side of an operator.
8. Layout contract elements in the following order: Pragma statements, Import statements, Interfaces, Libraries, Contracts. Inside each contract, library or interface, use the following order: Type declarations, State variables, Events, Functions.

99. **Naming Conventions**:

    1. **Types**: `lowercase`, `lower_case_with_underscores`, `UPPERCASE`, `UPPER_CASE_WITH_UNDERSCORES`, `CapitalizedWords`, `mixedCase`, `Capitalized_Words_With_Underscores`.
    2. **Names to Avoid**: `l` - Lowercase letter L, `O` - Uppercase letter o, `I` - Uppercase letter i. Never use any of these for single letter variable names. They are often indistinguishable from the numerals 1 and 0.
    3. Contracts and libraries should be named using the CapWords style. Contract and library names should also match their filenames. If a contract file includes multiple contracts and/or libraries, then the filename should match the core contract. This is not recommended however if it can be avoided. Examples: `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
    4. Structs should be named using the CapWords style. Examples: `MyCoin`, `Position`, `PositionXY`.
    5. Events should be named using the CapWords style. Examples: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.
    6. Functions should use mixedCase. Examples: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

100. **More Naming Conventions**:

    1. Function arguments should use mixedCase. Examples: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.
    2. Local and state variable names should use mixedCase. Examples: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.
    3. Constants should be named with all capital letters with underscores separating words. Examples: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

4. Modifier names should use mixedCase. Examples: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.
5. Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.
6. **Avoiding Naming Collisions**: `single_trailing_underscore_`. This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

B.3 Solidity 201

(See https://secureum.substack.com/p/solidity-201 for the original article.)

1. `Solidity` supports multiple inheritance including polymorphism:
    1. Polymorphism means that a function call (internal and external) always executes the function of the same name (and parameter types) in the most derived contract in the inheritance hierarchy
    2. When a contract inherits from other contracts, only a single contract is created on the blockchain, and the code from all the base contracts is compiled into the created contract.
    3. **Function Overriding**: Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header.
    4. Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. `Solidity` is similar to `Python` in that it uses "C3 Linearization" to force a specific order in the directed acyclic graph (DAG) of base classes. So when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in `Python`) in a depth-first manner, stopping at the first match.
2. Contract Types:
    1. **Abstract Contracts**: Contracts need to be marked as abstract when at least one of their functions is not implemented. They use the `abstract` keyword.
    2. **Interfaces**: They cannot have any functions implemented. There are further restrictions:
        i. They cannot inherit from other contracts, but they can inherit from other interfaces.
        ii. All declared functions must be external.
        iii. They cannot declare a constructor.
        iv. They cannot declare state variables.

        They use the `interface` keyword.
    3. **Libraries**: They are deployed only once at a specific address and their code is reused using the `DELEGATECALL` opcode. This means that if library functions are called, their code is executed in the context of the calling contract. They use the `library` keyword.

3. `using for`: The directive `using A for B;` can be used to attach library functions (from the library (`A`) to any type (`B`) in the context of a contract. These functions will receive the object they are called on as their first parameter.

    1. The `using A for B` directive is active only within the current contract, including within all of its functions, and has no effect outside of the contract in which it is used.
    2. The directive may only be used inside a contract, not inside any of its functions.

4. **Base Class Functions**: It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using `ContractName.functionName()` or using `super.functionName()` if you want to call the function one level higher up in the flattened inheritance hierarchy.

5. **State Variable Shadowing**: This is considered as an error. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

6. **Function Overriding Changes**: The overriding function may only change the visibility of the overridden function from external to public. The mutability may be changed to a more strict one following the order: `nonpayable` can be overridden by `view` and `pure`. `view` can be overridden by `pure`. `payable` is an exception and cannot be changed to any other mutability.

7. **Virtual Functions**: Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered virtual. Functions with `private` visibility cannot be virtual.

8. **Public State Variable Override**: Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable. While public state variables can override external functions, they themselves cannot be overridden.

9. **Modifier Overriding**: Function modifiers can override each other. This works in the same way as function overriding (except that there is no overloading for modifiers). The `virtual` keyword must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier

10. **Base Constructors**: The constructors of all the base contracts will be called following the linearization rules. If the base constructors have arguments, derived contracts need to specify all of them either in the inheritance list or in the derived constructor.

11. **Name Collision Error**: It is an error when any of the following pairs in a contract have the same name due to inheritance:
    1. a function and a modifier
    2. a function and an event
    3. an event and a modifier
12. **Library Restrictions**: In comparison to contracts, libraries are restricted in the following ways:
    1. they cannot have state variables
    2. they cannot inherit nor be inherited
    3. they cannot receive Ether
    4. they cannot be destroyed
    5. it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise)
    6. Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are view or pure functions), because libraries are assumed to be stateless
13. **EVM Storage**: Storage is a `(key, value)` store that maps 256-bit words to 256-bit words and is accessed with EVM's `SSTORE`/`SLOAD` instructions. All locations in storage are initialized as zero.
14. **Storage Layout**: State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0
15. **Storage Layout Packing**: For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:
    1. The first item in a storage slot is stored lower-order aligned
    2. Value types use only as many bytes as are necessary to store them
    3. If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot
16. **Storage Layout & Structs/Arrays**:
    1. Structs and array data always start a new slot and their items are packed tightly according to these rules
    2. Items following struct or array data always start a new storage slot

3. The elements of structs and arrays are stored after each other, just as if they were given as individual values.

17. **Storage Layout & Inheritance**: For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

18. **Storage Layout & Types**: It might be beneficial to use reduced-size types if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation.
    If you are not reading or writing all the values in a slot at the same time, this can have the opposite effect, though: when one value is written to a multi-value storage slot, the storage slot has to be read first and then combined with the new value such that other data in the same slot is not destroyed.

19. **Storage Layout & Ordering**: Ordering of storage variables and struct members affects how they can be packed tightly. For example, declaring your storage variables in the order of `uint128`, `uint128`, `uint256` instead of `uint128`, `uint256`, `uint128`, as the former will only take up 2 slots of storage whereas the latter will take up 3.

20. **Storage Layout for Mappings & Dynamically-sized Arrays**: Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored "in between" the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the rules above and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

21. **Storage Layout for Dynamic Arrays**: If the storage location of the array ends up being a slot `p` after applying the storage layout rules, this slot stores the number of elements in the array (byte arrays and strings are an exception). Array data is located starting at `keccak256(p)` and it is laid out in the same way as statically-sized array data would: one element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively.

22. **Storage Layout for Mappings**: For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations. The value corresponding to a mapping key `k` is located at `keccak256(h(k) . p)` where `.` is concatenation and `h` is a function that is applied to the key depending on its type:

1. for value types, `h` pads the value to 32 bytes in the same way as when storing the value in memory.
2. for strings and byte arrays, `h` computes the `keccak256` hash of the unpadded data. If the mapping value is a non-value type, the computed slot marks the start of the data.

If the value is of struct type, for example, you have to add an offset corresponding to the struct member to reach the member.

23. **Storage Layout for bytes and string**: bytes and string are encoded identically. In general, the encoding is similar to `byte1[]`, in the sense that there is a slot for the array itself and a data area that is computed using a `keccak256` hash of that slot's position. However, for short values (shorter than 32 bytes) the array elements are stored together with the length in the same slot.

    If the data is at most 31 bytes long, the elements are stored in the higher-order bytes (left aligned) and the lowest-order byte stores the value `length * 2`. For byte arrays that store data which is 32 or more bytes long, the main slot `p` stores `length * 2 + 1` and the data is stored as usual in `keccak256(p)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set: short (not set) and long (set).

24. **EVM Memory**: EVM memory is linear and can be addressed at byte level and accessed with `MSTORE`/`MSTORE8`/`MLOAD` instructions. All locations in memory are initialized as zero.

25. **Memory Layout**: `Solidity` places new memory objects at the free memory pointer, and memory is never freed. The free memory pointer points to `0x80` initially.

26. Reserved Memory: `Solidity` reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

    1. `0x00` - `0x3f` (64 bytes): scratch space for hashing methods.
    2. `0x40` - `0x5f` (32 bytes): currently allocated memory size (aka. free memory pointer).
    3. `0x60` - `0x7f` (32 bytes): zero slot (The zero slot is used as initial value for dynamic memory arrays and should never be written to).

27. **Memory Layout & Arrays**: Elements in memory arrays in `Solidity` always occupy multiples of 32 bytes (this is even true for `byte[]`, but not for bytes and string).

    1. Multi-dimensional memory arrays are pointers to memory arrays.

    2. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

28. **Free Memory Pointer**: There is a "free memory pointer" at position `0x40` in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. Considering the reserved memory, allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

29. **Zeroed Memory**: There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory.

30. Reserved Keywords: These keywords are reserved in `Solidity`. They might become part of the syntax in the future: `after`, `alias`, `apply`, `auto`, `case`, `copyof`, `default`, `define`, `final`, `immutable`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof` and `unchecked`.

31. **Inline Assembly**: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of `Solidity`. You should only use it for tasks that need it, and only if you are confident with using it.

    1. The language used for inline assembly in `Solidity` is called Yul
    2. An inline assembly block is marked by `assembly { ... }`, where the code inside the curly braces is code in the Yul language

32. Inline Assembly Access to External Variables, Functions and Libraries:

    1. You can access `Solidity` variables and other identifiers by using their name.
    2. Local variables of value type are directly usable in inline assembly
    3. Local variables that refer to memory/`calldata` evaluate to the address of the variable in memory/`calldata` and not the value itself
    4. For local storage variables or state variables, a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, their "address" is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x.slot`, and to retrieve the byte-offset you use `x.offset`. Using `x` itself will result in an error.
    5. Local `Solidity` variables are available for assignments
    6. Assignments are possible to assembly-local variables and to function-local variables. Take care that when you assign to variables that point to memory or storage, you will only change the pointer and not the data.

7. You can assign to the `.slot` part of a local storage variable pointer. For these (structs, arrays or mappings), the `.offset` part is always zero. It is not possible to assign to the `.slot` or `.offset` part of a state variable, though.

33. **Yul Syntax**: Yul parses comments, literals and identifiers in the same way as `Solidity`. Inside a code block, the following elements can be used:

    1. literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters).
    2. calls to builtin functions, e.g. `add(1, mload(0))`.
    3. variable declarations, e.g. `let x := 7`, `let x := add(y, 3)` or `let x` (initial value of 0 is assigned).
    4. identifiers (variables), e.g. `add(3, x)`.
    5. assignments, e.g. `x := add(y, 3)`.
    6. blocks where local variables are scoped inside, e.g. `{ let x := 3 {let y := add(x, 1)}}`.
    7. `if` statements,
       e.g. `if lt(a, b) {sstore(0, 1)}`.
    8. `switch` statements,
       e.g. `switch mload(0) case 0 {revert()} default {mstore(0, 1)}`.
    9. `for` loops,
       e.g. `for {let i := 0} lt(i, 10) {i := add(i, 1)} {mstore(i, 7)}`.
    10. function definitions, e.g. `function f(a, b) -> c {c := add(a, b)}`.

34. `Solidity v0.6.0` Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it: the resulting type of an exponentiation is the type of the base. It used to be the smallest type that can hold both the type of the base and the type of the exponent, as with symmetric operations. Additionally, signed types are allowed for the base of the exponentiation.

35. `Solidity v0.6.0` **Explicitness Requirements**:

    1. Functions can now only be overridden when they are either marked with the `virtual` keyword or defined in an interface. Functions without implementation outside an interface have to be marked `virtual`. When overriding a function or modifier, the new keyword override must be used. When overriding a function or modifier defined in multiple parallel bases, all bases must be listed in parentheses after the keyword like so: `override(Base1, Base2)`.
    2. Member-access to length of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays by assigning a new value to their length. Use `push()`, `push(value)` or `pop()` instead, or assign a full array, which will of course overwrite the existing content. The reason behind this is to prevent storage collisions of gigantic storage arrays.

3. The new keyword `abstract` can be used to mark contracts as `abstract`. It has to be used if a contract does not implement all its functions. Abstract contracts cannot be created using the `new` operator, and it is not possible to generate bytecode for them during compilation.
4. Libraries have to implement all their functions, not only the internal ones.
5. The names of variables declared in inline assembly may no longer end in `_slot` or `_offset`.
6. Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.
7. State variable shadowing is now disallowed. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

36. `Solidity v0.6.0` **Semantic and Syntactic Changes**:

    1. Conversions from external function types to address are now disallowed. Instead external function types have a member called `address`, similar to the existing selector member.
    2. The function `push(value)` for dynamic storage arrays does not return the new length anymore (it returns nothing).
    3. The `unnamed` function commonly referred to as "fallback function" was split up into a new fallback function that is defined using the `fallback` keyword and a receive ether function defined using the `receive` keyword.
    4. If present, the receive ether function is called whenever the `calldata` is empty (whether or not ether is received). This function is implicitly `payable`.
    5. The new fallback function is called when no other function matches (if the receive ether function does not exist then this includes calls with empty `calldata`). You can make this function `payable` or not. If it is not payable then transactions not matching any other function which `send` value will revert. You should only need to implement the new fallback function if you are following an upgrade or proxy pattern.

37. `Solidity v0.6.0` **New Features**:

    1. The `try`/`catch` statements allows you to react on failed external calls.
    2. struct and enum types can be declared at file level.

3. Array slices can be used for calldata arrays, for example `abi.decode(msg.data[4:], (uint, uint))` is a low-level way to decode the function call payload.
4. Natspec supports multiple return parameters in developer documentation, enforcing the same naming check as `@param`.
5. Yul and Inline Assembly have a new statement called `leave` that exits the current function.
6. Conversions from `address` to address `payable` are now possible via `payable(x)`, where `x` must be of type `address`.

38. `Solidity v0.7.0` Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it exponentiation and shifts of literals by non-literals (e.g. `1 << x` or `2 ** x`) will always use either the type `uint256` (for non-negative literals) or `int256` (for negative literals) to perform the operation. Previously, the operation was performed in the type of the shift amount / the exponent which can be misleading.

39. `Solidity v0.7.0` Changes to the Syntax - changes that might cause existing contracts to not compile anymore:

    1. In external function and contract creation calls, Ether and gas is now specified using a new syntax:
    `x.f{gas: 10000, value: 2 ether}(arg1, arg2)`. The old syntax – `x.f.gas(10000).value(2 ether)(arg1, arg2)` – will cause an error.
    2. The global variable `now` is deprecated, `block.timestamp` should be used instead. The single identifier `now` is too generic for a global variable and could give the impression that it changes during transaction processing, whereas `block.timestamp` correctly reflects the fact that it is just a property of the block.
    3. NatSpec comments on variables are only allowed for public state variables and not for local or internal variables.
    4. The token gwei is a keyword now (used to specify, e.g. `2 gwei` as a number) and cannot be used as an identifier.
    5. String literals now can only contain printable `ASCII` characters and this also includes a variety of escape sequences, such as hexadecimal (`\xff`) and unicode escapes (`\u20ac`).
    6. Unicode string literals are supported now to accommodate valid `UTF-8` sequences. They are identified with the unicode prefix: unicode"Hello ☺" (whose unicode notation is `U+1F603`).
    7. State Mutability: The state mutability of functions can now be restricted during inheritance. Functions with default state mutability can be overridden by pure and view functions while view functions

can be overridden by pure functions. At the same time, public state variables are considered view and even pure if they are constants.

8. Disallow `.` in user-defined function and variable names in inline assembly. It is still valid if you use `Solidity` in Yul-only mode.
9. Slot and offset of storage pointer variable `x` are accessed via `x.slot` and `x.offset` instead of `x_slot` and `x_offset`.

40. `Solidity v0.7.0` **Removal of Unused or Unsafe Features**

    1. If a struct or array contains a mapping, it can only be used in storage. Previously, mapping members were silently skipped in memory, which is confusing and error-prone.
    2. Assignments to structs or arrays in storage do not work if they contain mappings. Previously, mappings were silently skipped during the copy operation, which is misleading and error-prone.
    3. Visibility (`public`/`external`) is not needed for constructors anymore: To prevent a contract from being created, it can be marked `abstract`. This makes the visibility concept for constructors obsolete.
    4. Type Checker: Disallow virtual for library functions: Since libraries cannot be inherited from, library functions should not be virtual.
    5. Multiple events with the same name and parameter types in the same inheritance hierarchy are disallowed.
    6. `using A for B` only affects the contract it is mentioned in. Previously, the effect was inherited. Now, you have to repeat the using statement in all derived contracts that make use of the feature.
    7. Shifts by signed types are disallowed. Previously, shifts by negative amounts were allowed, but reverted at runtime.
    8. The `finney` and `szabo` denominations are removed. They are rarely used and do not make the actual amount readily visible. Instead, explicit values like `1e20` or the very common `gwei` can be used.
    9. The keyword `var` cannot be used anymore. Previously, this keyword would parse, but result in a type error and a suggestion about which type to use. Now, it results in a parser error.

41. `Solidity v0.8.0` Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

    1. Arithmetic operations revert on underflow and overflow. You can use `unchecked {...}` to use the previous wrapping behaviour. Checks for overflow are very common, so they are the default to increase readability of code, even if it comes at a slight increase of gas costs.

2. ABI coder v2 is activated by default. You can choose to use the old behaviour using `pragma abicoder v1;`. The pragma `pragma experimental ABIEncoderV2;` is still valid, but it is deprecated and has no effect. If you want to be explicit, please use `pragma abicoder v2;` instead.
3. Exponentiation is right associative, i.e., the expression `a**b**c` is parsed as `a**(b**c)`. Before `0.8.0`, it was parsed as `(a**b)**c`. This is the common way to parse the exponentiation operator.
4. Failing assertions and other internal checks like division by zero or arithmetic overflow do not use the `invalid` opcode but instead the `revert` opcode. More specifically, they will use error data equal to a function call to Panic(`uint256`) with an error code specific to the circumstances. This will save gas on errors while it still allows static analysis tools to distinguish these situations from a revert on invalid input, like a failing `require`.
5. If a byte array in storage is accessed whose length is encoded incorrectly, a panic is caused. A contract cannot get into this situation unless inline assembly is used to modify the raw representation of storage byte arrays.
6. If constants are used in array length expressions, previous versions of `Solidity` would use arbitrary precision in all branches of the evaluation tree. Now, if constant variables are used as intermediate expressions, their values will be properly rounded in the same way as when they are used in run-time expressions.
7. The type `byte` has been removed. It was an alias of `bytes1`.

42. `Solidity v0.8.0` New Restrictions - changes that might cause existing contracts to not compile anymore:

    1. Explicit conversions from negative literals and literals larger than `type(uint160).max` to address are disallowed.
    2. Explicit conversions between literals and an integer `type T` are only allowed if the literal lies between `type(T).min` and `type(T).max`. In particular, replace usages of `uint(-1)` with `type(uint).max`.
    3. Explicit conversions between literals and enums are only allowed if the literal can represent a value in the enum.
    4. Explicit conversions between literals and address type (e.g. `address(literal)`) have the type address instead of address payable. One can get a payable address type by using an explicit conversion, i.e., `payable(literal)`.
    5. Address literals have the type `address` instead of `address payable`. They can be converted to `address payable` by using an explicit conversion.

6. Function call options can only be given once, i.e. `c.f{gas: 10000}{value: 1}()` is invalid and has to be changed to `c.f{gas: 10000, value: 1}()`.
7. The global functions `log0`, `log1`, `log2`, `log3` and `log4` have been removed. These are low-level functions that were largely unused. Their behaviour can be accessed from inline assembly.
8. `enum` definitions cannot contain more than 256 members. This will make it safe to assume that the underlying type in the ABI is always `uint8`.
9. Declarations with the name `this`, `super` and `_` are disallowed, with the exception of public functions and events.
10. The global variables `tx.origin` and `msg.sender` have the type `address` instead of address payable. One can convert them into `address payable` by using an explicit conversion.
11. Explicit conversion into `address` type always returns a non-payable address type.
12. The `chainid` builtin in inline assembly is now considered `view` instead of `pure`.

43. **Zero Address Check**: `address(0)` which is 20-bytes of `0`'s is treated specially in `Solidity` contracts because the private key corresponding to this address is unknown. Ether and tokens sent to this address cannot be retrieved and setting access control roles to this address also won't work (no private key to sign transactions). Therefore zero addresses should be used with care and checks should be implemented for user-supplied address parameters.

44. `tx.origin`**Check**: Recall that Ethereum has two types of accounts: Externally Owned Account (EOA) and Contract Account. Transactions can originate only from EOAs. In situations where contracts would like to determine if the `msg.sender` was a contract or not, checking if `msg.sender` is equal to `tx.origin` is an effective check.

45. **Overflow/Underflow Check**: Until `Solidity` version `0.8.0` which introduced checked arithmetic by default, arithmetic was unchecked and therefore susceptible to overflows and underflows which could lead to critical vulnerabilities. The recommended best-practice for such contracts is to use OpenZeppelin's SafeMath library for arithmetic.

46. **OpenZeppelin Libraries**: OpenZeppelin's smart contract libraries are perhaps the most commonly used libraries in smart contract projects. These include contracts for popular token standards, access control, security, safe math, proxies and other utilities.

47. **OpenZeppelin ERC20**: Implements the popular ERC20 token standard. The functions are:

1. `constructor(string name_, string symbol_)`: Sets the values for name and symbol. The default value of decimals is 18. To select a different value for decimals you should overload it. All three of these values are immutable: they can only be set once during construction.
2. `name()`→`string`: Returns the name of the token.
3. `symbol()`→`string`: Returns the symbol of the token, usually a shorter version of the name.
4. `decimalse()`→`uint8`: Returns the number of decimals used to get its user representation. For example, if `decimals` equals 2, a balance of 505 tokens should be displayed to a user as `5.05 (505 / 10 ** 2)`. Tokens usually opt for a value of 18, imitating the relationship between Ether and Wei. This is the value ERC20 uses, unless this function is overridden.
5. `totalSupply()`: Returns the amount of tokens in existence.
6. `balanceOf(address account)`→`uint256`: Returns the amount of tokens owned by account.
7. `transfer(address recipient, uint256 amount)`→`bool`: Moves amount tokens from the caller's account to recipient. Returns a boolean value indicating whether the operation succeeded. Emits a Transfer event.
8. `allowance(address owner, address spender)`→`uint256`: Returns the remaining number of tokens that spender will be allowed to spend on behalf of owner through transferFrom. This is zero by default. This value changes when approve or transferFrom are called.
9. `approve(addres spender, uint256 amount)`→`bool`: Sets amount as the allowance of spender over the caller's tokens. Returns a boolean value indicating whether the operation succeeded. Emits an Approval event. Warning: changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.
10. `transferFrom(address sender, address recipient, uint256 amount)`→`bool`: Moves amount tokens from sender to recipient using the allowance mechanism. amount is then deducted from the caller's allowance. Returns a boolean value indicating whether the operation succeeded. Emits a Transfer event.
11. `increaseAllowance(address spender, uint256 addedValue)`→`bool`: Atomically increases the allowance granted to spender by the caller. This is an alternative to approve that can be used as a mitigation for the warning above. Emits an Approval event indicating the updated allowance. Requirement is that spender cannot be the zero address.

12. `decreaseAllowance(address spender, uint256 subtractedValue)→bool`: Atomically decreases the allowance granted to spender by the caller. This is an alternative to approve that can be used as a mitigation for the warning described above. Emits an Approval event indicating the updated allowance. Requirements are: 1) spender cannot be the zero address. 2) spender must have allowance for the caller of at least `subtractedValue`.

The different extensions are:

1. **OpenZeppelin ERC20Burnable**: Extension of ERC20 that allows token holders to destroy both their own tokens and those that they have an allowance for, in a way that can be recognized off-chain (via event analysis).
2. **OpenZeppelin ERC20Capped**: Extension of ERC20 that adds a cap to the supply of tokens and enforces it in the mint function.
3. **OpenZeppelin ERC20Pausable**: ERC20 token with pausable token transfers, minting and burning. Useful for scenarios such as preventing trades until the end of an evaluation period, or having an emergency switch for freezing all token transfers in the event of a large bug. The `_beforeTokenTransfer()` internal function enforces the not paused condition.
4. **OpenZeppelin ERC20Snapshot**: This contract extends an ERC20 token with a snapshot mechanism. When a snapshot is created, the balances and total supply at the time are recorded for later access. This can be used to safely create mechanisms based on token balances such as trustless dividends or weighted voting. Snapshots are created by the internal `_snapshot` function, which will emit the Snapshot event and return a snapshot id. To get the total supply at the time of a snapshot, call the function `totalSupplyAt` with the snapshot id. To get the balance of an account at the time of a snapshot, call the `balanceOfAt` function with the snapshot id and the account address.
5. **OpenZeppelin ERC20PresetFixedSupply**: ERC20 token, including:
   i. Preminted initial supply.
   ii. Ability for holders to burn (destroy) their tokens.
   iii. No access control mechanism (for minting/pausing) and hence no governance.

   This contract uses `ERC20Burnable` contract to include burn capabilities.
6. **OpenZeppelin ERC20PresetMinterPauser**: ERC20 token, including:
   i. Ability for holders to burn (destroy) their tokens.

ii. A minter role that allows for token minting (creation).

iii. A pauser role that allows to stop all token transfers.

This contract uses `AccessControl` contract to lock permissioned functions using the different roles. The account that deploys the contract will be granted the minter and pauser roles, as well as the default admin role, which will let it grant both minter and pauser roles to other accounts.

48. **OpenZeppelin SafeERC20**: Wrappers around ERC20 operations that throw on failure when the token contract implementation returns false. Tokens that return no value and instead revert or throw on failure are also supported with non-reverting calls assumed to be successful. Adds `safeTransfer`, `safeTransferFrom`, `safeApprove`, `safeDecreaseAllowance` and `safeIncreaseAllowance`.

49. **OpenZeppelin TokenTimelock**: A token holder contract that will allow a beneficiary to extract the tokens after a given release time. Useful for simple vesting schedules like "advisors get all of their tokens after 1 year".

50. **OpenZeppelin ERC721**: Implements the popular ERC721 Non-Fungible Token Standard. The functions are:

    1. `balanceOf(address owner)→uint256 balance`: Returns the number of tokens in owner's account.
    2. `ownerOf(uint256 tokenId)→address owner`: Returns the owner of the tokenId token. Requirements:
    `balanceOf(address owner)→uint256 balance`tokenId| must exist.
    3. `transferFrom(address from, address to, uint256 tokenId)`: Transfers tokenId token from from to to. Requirements: `from` cannot be the zero address. `to` cannot be the zero address. `tokenId` token must be owned by `from`. If the caller is not `from`, it must be approved to move this token by either `approve` or `setApprovalForAll`. Emits a Transfer event.
    4. `safeTransferFrom(address from, address to, uint256 token Id)`: Safely transfers `tokenId` token from `from` to `to`, checking first that contract recipients are aware of the ERC721 protocol to prevent tokens from being forever locked. Requirements: 1) `from` cannot be the zero address 2) `to` cannot be the zero address. 3) `tokenId` token must exist and be owned by `from` 4) If the caller is not `from`, it must be have been allowed to move this token by either `approve` or `setApprovalForAll` 5) If `to` refers to a smart contract, it must implement `IERC721Receiver.onERC721Received`, which is called upon a safe transfer. Emits a Transfer event. (The use of this

function is encouraged over the related but unsafe transferFrom function.)

5. `approve(address to, uint256 tokenId)`: Gives permission to `to` to transfer `tokenId` token to another account. The approval is cleared when the token is transferred. Only a single account can be approved at a time, so approving the zero address clears previous approvals. Requirements: 1) The caller must own the token or be an approved operator 2) `tokenId` must exist. Emits an Approval event.
6. `getApproved(uint256 tokenId)→address operator`: Returns the account approved for tokenId token. Requirements: tokenId must exist.
7. `setApprovalForAll(address operator, bool _approved)`: Approve or remove operator as an operator for the caller. Operators can call `transferFrom` or `safeTransferFrom` for any token owned by the caller. Requirements: The operator cannot be the caller. Emits an ApprovalForAll event.
8. `isApprovedForAll(address owner, address operator)→bool`: Returns if the operator is allowed to manage all of the assets of owner.

The different extensions/presets/utilities are:

1. **OpenZeppelin ERC721Burnable**: ERC721 Token that can be irreversibly burned (destroyed).
2. **OpenZeppelin ERC721Enumerable**: This implements an optional extension of ERC721 defined in the EIP that adds enumerability of all the token ids in the contract as well as all token ids owned by each account.
3. **OpenZeppelin ERC721Pausable**: ERC721 token with pausable token transfers, minting and burning. Useful for scenarios such as preventing trades until the end of an evaluation period, or having an emergency switch for freezing all token transfers in the event of a large bug.
4. **OpenZeppelin ERC721URIStorage**: ERC721 token with storage based token URI management.
5. **OpenZeppelin ERC721PresetMinterPauserAutoId**: ERC721 token, including:
   i. ability for holders to burn (destroy) their tokens.
   ii. a minter role that allows for token minting (creation).
   iii. a pauser role that allows to stop all token transfers.
   iv. token ID and URI autogeneration.

   This contract uses `AccessControl` to lock permissioned functions using the different roles. The account that deploys the contract will

be granted the minter and pauser roles, as well as the default admin role, which will let it grant both minter and pauser roles to other accounts.

6. **OpenZeppelin ERC721Holder**: Implementation of the `IERC721Receiver` interface. Accepts all token transfers.

51. **OpenZeppelin ERC777**: Like ERC20, ERC777 is a standard for fungible tokens with improvements such as getting rid of the confusion around decimals, minting and burning with proper events, among others, but its killer feature is receive hooks. ERC777 is backwards compatible with ERC20 (see [17]).

    1. A hook is simply a function in a contract that is called when tokens are sent to it, meaning accounts and contracts can react to receiving tokens. This enables a lot of interesting use cases, including atomic purchases using tokens (no need to do `approve` and `transferFrom` in two separate transactions), rejecting reception of tokens (by reverting on the hook call), redirecting the received tokens to other addresses, among many others.
    2. Both contracts and regular addresses can control and reject which token they send by registering a `tokensToSend` hook. (Rejection is done by reverting in the hook function.)
    3. Both contracts and regular addresses can control and reject which token they receive by registering a `tokensReceived` hook. (Rejection is done by reverting in the hook function.)
    4. The `tokensReceived` hook allows to send tokens to a contract and notify it in a single transaction, unlike ERC-20 which requires a double call (`approve`/`transferFrom`) to achieve this.
    5. Furthermore, since contracts are required to implement these hooks in order to receive tokens, no tokens can get stuck in a contract that is unaware of the ERC777 protocol, as has happened countless times when using ERC20s.
    6. It mandates that decimals always returns a fixed value of 18, so there's no need to set it ourselves
    7. Has a concept of `defaultOperators` which are special accounts (usually other smart contracts) that will be able to transfer tokens on behalf of their holders
    8. Implements `send` (besides `transfer`) where if the recipient contract has not registered itself as aware of the ERC777 protocol then transfers to it are disabled to prevent tokens from being locked forever. Accounts can be notified of tokens being sent to them by having a contract implement this `IERC777Recipient` interface and registering it on the ERC1820 global registry.

52. **OpenZeppelin ERC1155**: is a novel token standard that aims to take the best from previous standards to create a fungibility-agnostic and gas-efficient token contract.

    1. The distinctive feature of ERC1155 is that it uses a single smart contract to represent multiple tokens at once
    2. Accounts have a distinct balance for each token id, and non-fungible tokens are implemented by simply minting a single one of them.
    3. This approach leads to massive gas savings for projects that require multiple tokens. Instead of deploying a new contract for each token type, a single ERC1155 token contract can hold the entire system state, reducing deployment costs and complexity.
    4. Because all state is held in a single contract, it is possible to operate over multiple tokens in a single transaction very efficiently. The standard provides two functions, `balanceOfBatch` and `safeBatchTransferFrom`, that make querying multiple balances and transferring multiple tokens simpler and less gas-intensive.

53. **OpenZeppelin Ownable**: provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions. By default, the owner account will be the one that deploys the contract. This can later be changed with `transferOwnership`. This module is used through inheritance. It will make available the modifier `onlyOwner`, which can be applied to your functions to restrict their use to the owner.

54. **OpenZeppelin AccessControl**: provides a general role based access control mechanism. Multiple hierarchical roles can be created and assigned each to multiple accounts. Roles can be used to represent a set of permissions. `hasRole` is used to restrict access to a function call. Roles can be granted and revoked dynamically via the `grantRole` and `revokeRole` functions which can only be called by the role's associated admin accounts.

    While the simplicity of `Ownable` can be useful for simple systems or quick prototyping, different levels of authorization are often needed. You may want for an account to have permission to ban users from a system, but not create new tokens. Role-Based Access Control (RBAC) offers flexibility in this regard. We will effectively be defining multiple roles, each allowed to perform different sets of actions. An account may have, for example, `moderator`, `minter` or `admin` roles, which you will then check for instead of simply using `onlyOwner`. Separately, you will be able to define rules for how accounts can be granted a role, have it revoked, and more.

OpenZeppelin AccessControlEnumerable: Extension of AccessControl that allows enumerating the members of each role.

55. **OpenZeppelin Pausable**: provides an emergency stop mechanism using functions pause and unpause that can be triggered by an authorized account. This module is used through inheritance. It will make available the modifiers `whenNotPaused` and `whenPaused`, which can be applied to the functions of your contract. Only the functions using the modifiers will be affected when the contract is paused or unpaused.

56. **OpenZeppelin ReentrancyGuard**: prevents reentrant calls to a function. Inheriting from `ReentrancyGuard` will make the `nonReentrant` modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

57. **OpenZeppelin PullPayment**: provides a pull-payment strategy, where the paying contract doesn't invoke any functions on the receiver account which must withdraw its payments itself. Pull-payments are often considered the best practice when it comes to sending Ether, security-wise. It prevents recipients from blocking execution and eliminates reentrancy concerns.

58. **OpenZeppelin Address**: Collection of functions related to the address type:

    1. `isContract(address account)→bool`: Returns true if account is a contract. It is unsafe to assume that an address for which this function returns false is an externally-owned account (EOA) and not a contract. Among others, `isContract` will return `false` for the following types of addresses: 1) an externally-owned account 2) a contract in construction 3) an address where a contract will be created 4) an address where a contract lived, but was destroyed
    2. `sendValue(address payable recipient, uint256 amount)`: Replacement for `Solidity`'s `transfer`: sends amount wei to recipient, forwarding all available gas and reverting on errors. EIP1884 increases the gas cost of certain opcodes, possibly making contracts go over the 2300 gas limit imposed by `transfer`, making them unable to receive funds via `transfer`. `sendValue` removes this limitation.
    3. `functionCall(address target, bytes data)→bytes`: Performs a `Solidity` function call using a low level call. A plain `call` is an unsafe replacement for a `functionCall`: use this function instead. If target reverts with a revert reason, it is bubbled up by this function (like regular `Solidity` function calls). Returns the raw returned data. Requirements: target must be a contract. calling target with data must not revert.
    4. `functionCallWithValue(address target, bytes data, uint256 value)→bytes`: Same as functionCall, but also transferring value wei to target.

Requirements: 1) the calling contract must have an ETH balance of at least `value`. 2) the called `Solidity` function must be payable.

5. `functionStaticCall(address target, bytes data)`→`bytes`: Same as `functionCall`, but performing a `staticcall`.
6. `functionDelegateCall(address target, bytes data)`→`bytes`: Same as `functionCall`, but performing a `delegatecall`.

The above `functionCall*` functions have variants which pass an `errorMessage` parameter that specifies the fallback revert reason when target reverts.

59. **OpenZeppelin Arrays**: Collection of functions related to array types:

    1. `findUpperBound(uint256[] array, uint256 element)`→`uint256`: Searches a sorted array and returns the first index that contains a value greater or equal to `element`. If no such index exists (i.e. all values in the array are strictly less than element), the array length is returned. Time complexity $\mathcal{O}(\log(n))$. array is expected to be sorted in ascending order, and to contain no repeated elements.

60. **OpenZeppelin Context**: Provides information about the current execution context, including the sender of the transaction and its data. While these are generally available via `msg.sender` and `msg.data`, they should not be accessed in such a direct manner, since when dealing with meta-transactions the account sending and paying for execution may not be the actual sender (as far as an application is concerned). This contract is only required for intermediate, library-like contracts.

61. **OpenZeppelin Counters**: Provides counters that can only be incremented or decremented by one. This can be used e.g. to track the number of elements in a mapping, issuing ERC721 ids, or counting request ids. Functions are:

    1. `current(struct Counters.Counter counter)`→`uint256`
    2. `increment(struct Counters.Counter counter)`
    3. `decrement(struct Counters.Counter counter)`

62. **OpenZeppelin Create2**: makes usage of the `CREATE2` EVM opcode easier and safer. `CREATE2` can be used to compute in advance the address where a smart contract will be deployed, which allows for interesting new mechanisms known as 'counterfactual interactions'.

    1. `deploy(uint256 amount, bytes32 salt, bytes bytecode)`→`address`: Deploys a contract using `CREATE2`. The address where the contract will be deployed can be known in advance via `computeAddress`. The bytecode for a contract can be obtained from `Solidity` with `type(contractName).creationCode`. Requirements: 1) bytecode

must not be empty. 2) `salt` must have not been used for bytecode already. 3) the factory must have a balance of at least amount. 4) if amount is non-zero, bytecode must have a payable constructor.

2. `computeAddress(bytes32 salt, bytes32 bytecodeHash)→address`: Returns the address where a contract will be stored if deployed via deploy. Any change in the `bytecodeHash` or `salt` will result in a new destination address.
3. `computeAddress(bytes32 salt, bytes32 bytecodeHash, address deployer)→address`: Returns the address where a contract will be stored if deployed via `deploy` from a contract located at deployer. If the deployer is this contract's address, it returns the same value as `computeAddress`.

63. **OpenZeppelin Multicall**: Provides a function to batch together multiple calls in a single external call.
`multicall(bytes[] calldata data) external→bytes[]`: Receives and executes a batch of function calls on this contract.

64. **OpenZeppelin Strings**: String operations:

    1. `toString(uint256 value)→string`: Converts a `uint256` to its `ASCII` string decimal representation.
    2. `toHexString(uint256 value)→string`: Converts a `uint256` to its `ASCII` string hexadecimal representation.
    3. `toHexString(uint256 value, uint256 length)→string`: Converts a `uint256` to its `ASCII` string hexadecimal representation with fixed length.

65. **OpenZeppelin ECDSA**: provides functions for recovering and managing Ethereum account ECDSA signatures. These are often generated via `web3.eth.sign`, and are a 65 byte array (of type bytes in `Solidity`) arranged the following way: `[[v (1)], [r (32)], [s (32)]]`. The data signer can be recovered with `ECDSA.recover`, and its address compared to verify the signature. Most wallets will hash the data to sign and add the prefix `'\x19Ethereum Signed Message:\n'`, so when attempting to recover the signer of an Ethereum signed message hash, you'll want to use `toEthSignedMessageHash`. The `ecrecover` EVM opcode allows for malleable (non-unique) signature. This library prevents that by requiring the s value to be in the lower half order, and the v value to be either 27 or 28.

66. **OpenZeppelin MerkleProof**: This deals with verification of Merkle Trees proofs. `verify` can prove that some value is part of a Merkle tree. Returns `true` if a `leaf` can be proved to be a part of a Merkle tree defined by `root`. For this, a `proof` must be provided, containing sibling hashes on the branch from the leaf to the root of the tree. Each pair of leaves and each pair of pre-images are assumed to be sorted.

67. **OpenZeppelin SignatureChecker**: Provide a single mechanism to verify both private-key (EOA) ECDSA signature and ERC1271 contract signatures. Using this instead of `ECDSA.recover` in your contract will make them compatible with smart contract wallets such as Argent and Gnosis.

    1. Externally Owned Accounts (EOA) can sign messages with their associated private keys, but currently contracts cannot. This is a problem for many applications that implement signature based off-chain methods, since contracts can't easily interact with them as they do not possess a private key. ERC 1271 proposes a standard way for any contracts to verify whether a signature on behalf of a given contract is valid.
    2. Note: unlike ECDSA signatures, contract signatures are revocable, and the outcome of this function can thus change through time. It could return `true` at block N and `false` at block $N + 1$ (or the opposite).

68. **OpenZeppelin EIP712**: EIP 712 is a standard for hashing and signing of typed structured data. This contract implements the EIP 712 domain separator (`_domainSeparatorV4`) that is used as part of the encoding scheme, and the final step of the encoding to obtain the message digest that is then signed via ECDSA (`_hashTypedDataV4`). Protocols need to implement the type-specific encoding they need in their contracts using a combination of `abi.encode` and `keccak256`.

    1. `constructor(string name, string version)`: Initializes the domain separator and parameter caches. The meaning of name and version is specified in EIP 712: 1) `name` is the user readable name of the signing domain, i.e. the name of the ĐApp or the protocol 2) `version`: the current major version of the signing domain.
    2. `_domainSeparatorV4()`→`bytes32`: Returns the domain separator for the current chain.

69. **OpenZeppelin Escrow**: holds funds designated for a payee until they withdraw them. The contract that uses this escrow as its payment method should be its owner, and provide public methods redirecting to the escrow's deposit and withdraw if the escrow rules are satisfied.

    1. `depositsOf(address payee)`→`uint256`
    2. `deposit(address payee)`: Stores the sent amount as credit to be withdrawn.
    3. `withdraw(address payable payee)`: Withdraw accumulated balance for a payee, forwarding all gas to the recipient.

70. **OpenZeppelin ConditionalEscrow**: Derived from Escrow and only allows withdrawal if a condition is met by providing the

`withdrawalAllowed()` function which returns whether an address is allowed to withdraw their funds and is to be implemented by derived contracts.

71. **OpenZeppelin RefundEscrow**: Derived from `ConditionalEscrow` and holds funds for a beneficiary, deposited from multiple parties. The owner account (that is, the contract that instantiates this contract) may deposit, close the deposit period, and allow for either withdrawal by the beneficiary, or refunds to the depositors.

72. **OpenZeppelin ERC165**: In `Solidity`, it's frequently helpful to know whether or not a contract supports an interface you'd like to use. ERC165 is a standard that helps do runtime interface detection using a lookup table. You can register interfaces using `_registerInterface(bytes4)` and `supportsInterface(bytes4 interfaceId)` returns a `bool` indicating if that interface is supported or not.

73. **OpenZeppelin Math**: Standard math utilities missing in the `Solidity` language:

    1. `max(uint256 a, uint256 b)`: Returns the larger of two numbers.
    2. `min(uint256 a, uint256 b)`: Returns the smaller of two numbers.
    3. `average(uint256 a, uint256 b)`: Returns the average of two numbers. The result is rounded towards zero.

74. **OpenZeppelin SafeMath**: provides mathematical functions that protect your contract from overflows and underflows. Include the contract with `using SafeMath for uint256;` and then call the functions:

    1. `myNumber.add(otherNumber)`: Returns the addition of two unsigned integers, reverting on overflow. Counterpart to `Solidity`'s `+` operator.
    2. `myNumber.sub(otherNumber)`: Returns the subtraction of two unsigned integers, reverting on overflow (when the result is negative). Counterpart to `Solidity`'s `-` operator.
    3. `myNumber.div(otherNumber)`: Returns the division of two unsigned integers, reverting on overflow. The result is rounded towards zero. Counterpart to `Solidity`'s `/` operator.
    4. `myNumber.mul(otherNumber)`: Returns the multiplication of two unsigned integers, reverting on overflow. Counterpart to `Solidity`'s `*` operator.
    5. `myNumber.mod(otherNumber)`: Returns the modulus of two unsigned integers, reverting when dividing by zero. Counterpart to Solidity's `%` operator.

    The corresponding `try*` fucntions return results with an overflow flag instead of reverting.

75. **OpenZeppelin SignedSafeMath**: provides the same mathematical functions as `SafeMath` but for signed integers

    1. `myNumber.add(otherNumber)`: Returns the addition of two signed integers, reverting on overflow. Counterpart to `Solidity`'s `+` operator.
    2. `myNumber.sub(otherNumber)`: Returns the subtraction of two signed integers, reverting on overflow (when the result is negative). Counterpart to `Solidity`'s `-` operator.
    3. `myNumber.div(otherNumber)`: Returns the division of two signed integers, reverting on overflow. The result is rounded towards zero. Counterpart to `Solidity`'s `/` operator.
    4. `myNumber.mul(otherNumber)`: Returns the multiplication of two signed integers, reverting on overflow. Counterpart to `Solidity`'s `*` operator.

76. **OpenZeppelin SafeCast**: Wrappers over `Solidity`'s `uintXX`/`intXX` casting operators with added overflow checks. Downcasting from `uint256`/`int256` in `Solidity` does not revert on overflow. This can easily result in undesired exploitation or bugs, since developers usually assume that overflows raise errors. `SafeCast` restores this intuition by reverting the transaction when such an operation overflows.

    1. `toUint128(uint256 value) returns (uint128)`: Returns the downcasted `uint128` from `uint256`, reverting on overflow (when the input is greater than largest `uint128`). Similar functions are available for `toUint64(uint256 value)`, `toUint32(uint256 value)`, `toUint16(uint256 value)` and `toUint8(uint256 value)`.
    2. `toInt128(int256 value) internal pure returns (uint256)`: Returns the downcasted `int128` from `int256`, reverting on overflow (when the input is less than smallest `int128` or greater than largest `int128`). Similar functions are available for `toInt64(int256 value)`, `toInt32(int256 value)`, `toInt16(int256 value)` and `toInt8(int256 value)`.
    3. `function toInt256(uint256 value) returns (int256)`: Converts an unsigned `uint256` into a signed `int256`.
    4. `function toUint256(int256 value) returns (uint256)`: Converts a signed `int256` into an unsigned `uint256`.
    5. Similar functions downcasting to `224`/`96`/`64`/`32`/`16`/`8` bits for both unsigned and signed.

77. **OpenZeppelin EnumerableMap**: Library for managing an enumerable variant of `Solidity`'s mapping type. Maps havethe following properties: 1) Entries are added, removed, and checked for existence in constant

time ($\mathcal{O}(1)$) 2) Entries are enumerated in $\mathcal{O}(n)$. No guarantees are made on the ordering. As of `v3.0.0`, only maps of type `uint256→ address` (`UintToAddressMap`) are supported.

1. `set(struct EnumerableMap.UintToAddressMap map, uint256 key, address value)→bool`: Adds a `(key, value)` pair to a map, or updates the value for an existing key. Returns `true` if the key was added to the map, that is if it was not already present.
2. `remove(struct EnumerableMap.UintToAddressMap map, uint256 key)→bool`: Removes a value from a set. Returns `true` if the key was removed from the map, that is if it was present.
3. `contains(struct EnumerableMap.UintToAddressMap map, uint256 key)→bool`: Returns `true` if the key is in the map.
4. `length(struct EnumerableMap.UintToAddressMap map)→uint256`: Returns the number of elements in the map.
5. `at(struct EnumerableMap.UintToAddressMap map, uint256 index)→uint256, address`: Returns the element stored at position `index` in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: `index` must be strictly less than `length`.
6. `tryGet(struct EnumerableMap.UintToAddressMap map, uint256 key)→bool, address`: Tries to return the `value` associated with `key`. Does not revert if key is not in the map.
7. `get(struct EnumerableMap.UintToAddressMap map, uint256 key)→address`: Returns the value associated with `key`. Requirements: `key` must be in the map.

78. **OpenZeppelin EnumerableSet**: Library for managing sets of primitive types. Sets have the following properties: 1) Elements are added, removed, and checked for existence in constant time ($\mathcal{O}(1)$) 2) Elements are enumerated in $\mathcal{O}(n)$. No guarantees are made on the ordering. As of `v3.3.0`, sets of type `bytes32` (`Bytes32Set`), `address` (`AddressSet`) and `uint256` (`UintSet`) are supported.

    1. `add(struct EnumerableSet.Bytes32Set set, bytes32 value)→bool`: Add a value to a set. Returns `true` if the value was added to the set, that is if it was not already present.
    2. `remove(struct EnumerableSet.Bytes32Set set, bytes32 value)→bool`: Removes a value from a set. Returns `true` if the value was removed from the set, that is if it was present.
    3. `contains(struct EnumerableSet.Bytes32Set set, bytes32 value)→bool`: Returns `true` if the value is in the set.
    4. `length(struct EnumerableSet.Bytes32Set set)→uint256`: Returns the number of values in the set.

5. `at(struct EnumerableSet.Bytes32Set set, uint256 index)→bytes32`: Returns the value stored at position `index` in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: `index` must be strictly less than `length`.
6. `add(struct EnumerableSet.AddressSet set, address value)→bool`: Add a value to a set. Returns `true` if the value was added to the set, that is if it was not already present.
7. `remove(struct EnumerableSet.AddressSet set, address value)→bool`: Removes a value from a set. Returns `true` if the value was removed from the set, that is if it was present.
8. `contains(struct EnumerableSet.AddressSet set, address value)→bool`: Returns true if the value is in the set. length(struct EnumerableSet.AddressSet set) → uint256: Returns the number of values in the set.
9. `at(struct EnumerableSet.AddressSet set, uint256 index)→address`: Returns the value stored at position `index` in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: `index` must be strictly less than `length`.
10. `add(struct EnumerableSet.UintSet set, uint256 value)→bool`: Add a value to a set. Returns `true` if the value was added to the set, that is if it was not already present.
11. `remove(struct EnumerableSet.UintSet set, uint256 value)→bool`: Removes a value from a set. Returns `true` if the value was removed from the set, that is if it was present.
12. `contains(struct EnumerableSet.UintSet set, uint256 value)→bool`: Returns `true` if the value is in the set. $\mathcal{O}(1)$.
13. `length(struct EnumerableSet.UintSet set)→uint256`: Returns the number of values on the set.
14. `at(struct EnumerableSet.UintSet set, uint256 index)→uint256`: Returns the value stored at position `index` in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: `index` must be strictly less than `length`.

79. **OpenZeppelin BitMaps**: Library for managing `uint256` to `bool` mapping in a compact and efficient way, providing the keys are sequential.
    1. `struct BitMap: mapping(uint256 => uint256) _data;`
    2. `get(BitMap storage bitmap, uint256 index)→bool`: Returns whether the bit at `index` is set.
    3. `setTo(BitMap storage bitmap, uint256 index, bool value)`: Sets the bit at `index` to the boolean `value`.

4. `function set(BitMap storage bitmap, uint256 index)`: Sets the bit at `index`.
5. `function unset(BitMap storage bitmap, uint256 index)`: Unsets the bit at `index`.

80. **OpenZeppelin PaymentSplitter**: allows to split Ether payments among a group of accounts. The sender does not need to be aware that the Ether will be split in this way, since it is handled transparently by the contract. The split can be in equal parts or in any other arbitrary proportion. The way this is specified is by assigning each account to a number of shares. Of all the Ether that this contract receives, each account will then be able to claim an amount proportional to the percentage of total shares they were assigned.

    PaymentSplitter follows a pull payment model. This means that payments are not automatically forwarded to the accounts but kept in this contract, and the actual transfer is triggered as a separate step by calling the release function.

81. **OpenZeppelin TimelockController**: acts as a timelocked controller. When set as the owner of an `Ownable` smart contract, it enforces a timelock on all `onlyOwner` maintenance operations. This gives time for users of the controlled contract to exit before a potentially dangerous maintenance operation is applied. By default, this contract is self administered, meaning administration tasks have to go through the timelock process. The proposer (resp executor) role is in charge of proposing (resp executing) operations. A common use case is to position this TimelockController as the owner of a smart contract, with a multisig or a DAO as the sole proposer.

    1. `constructor(uint256 minDelay, address[] proposers, address[] executors)`: Initializes the contract with a given `minDelay`.
    2. `receive()`: Contract might receive/hold ETH as part of the maintenance process.
    3. `isOperation(bytes32 id)→bool pending`: Returns whether an id corresponds to a registered operation. This includes both Pending, Ready and Done operations.
    4. `isOperationPending(bytes32 id)→bool pending`: Returns whether an operation is pending or not.
    5. `isOperationReady(bytes32 id)→bool ready`: Returns whether an operation is ready or not.
    6. `isOperationDone(bytes32 id)→bool done`: Returns whether an operation is done or not.

7. `getTimestamp(bytes32 id)→uint256 timestamp`: Returns the timestamp at which an operation becomes ready (0 for unset operations, 1 for done operations).
8. `getMinDelay()→uint256 duration`: Returns the minimum delay for an operation to become valid. This value can be changed by executing an operation that calls `updateDelay`.
9. `hashOperation(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt)→bytes32 hash`: Returns the identifier of an operation containing a single transaction.
10. `hashOperationBatch(address[] targets, uint256[] values, bytes[] datas, bytes32 predecessor, bytes32 salt)→bytes32 hash`: Returns the identifier of an operation containing a batch of transactions.
11. `schedule(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt, uint256 delay)`: Schedule an operation containing a single transaction. Emits a `CallScheduled` event. Requirements: the caller must have the 'proposer' role.
12. `scheduleBatch(address[] targets, uint256[] values, bytes[] datas, bytes32 predecessor, bytes32 salt, uint256 delay)`: Schedule an operation containing a batch of transactions. Emits one `CallScheduled` event per transaction in the batch. Requirements: the caller must have the 'proposer' role.
13. `cancel(bytes32 id)`: Cancel an operation. Requirements: the caller must have the 'proposer' role.
14. `execute(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt)`: Execute an (ready) operation containing a single transaction. Emits a CallExecuted event. Requirements: the caller must have the 'executor' role.
15. `executeBatch(address[] targets, uint256[] values, bytes[] datas, bytes32 predecessor, bytes32 salt)`: Execute an (ready) operation containing a batch of transactions. Emits one `CallExecuted` event per transaction in the batch. Requirements: the caller must have the 'executor' role.
16. `updateDelay(uint256 newDelay)`: Changes the minimum timelock duration for future operations. Emits a `MinDelayChange` event. Requirements: the caller must be the timelock itself. This can only be achieved by scheduling and later executing an operation where the timelock is the target and the data is the ABI-encoded call to this function.

82. **OpenZeppelin ERC2771Context**: A Context variant for ERC2771. ERC2771 provides support for meta transactions, which are transactions that have been:

1. Authorized by the Transaction Signer. For example, signed by an externally owned account
2. Relayed by an untrusted third party that pays for the gas (the Gas Relay)

The problem is that for a contract that is not natively aware of meta transactions, the `msg.sender` of the transaction will make it appear to be coming from the Gas Relay and not the Transaction Signer. A secure protocol for a contract to accept meta transactions needs to prevent the Gas Relay from forging, modifying or duplicating requests by the Transaction Signer. The entities are:

1. Transaction Signer - entity that signs & sends to request to Gas Relay.
2. Gas Relay - receives a signed request off-chain from Transaction Signer and pays gas to turn it into a valid transaction that goes through Trusted Forwarder.
3. Trusted Forwarder - a contract that is trusted by the Recipient to correctly verify the signature and nonce before forwarding the request from Transaction Signer.
4. Recipient - a contract that can securely accept meta-transactions through a Trusted Forwarder by being compliant with this standard.

83. **OpenZeppelin MinimalForwarder**: provides a simple minimal forwarder (as described above) to be used together with an ERC2771 compatible contract. It verifies the nonce and signature of the forwarded request before calling the destination contract.

    1. `struct ForwardRequest {address from; address to; uint256 value; uint256 gas; uint256 nonce; bytes data;}`
    2. `verify(ForwardRequest calldata req, bytes calldata signature) public view→(bool)`
    3. `execute(ForwardRequest calldata req, bytes calldata signature) → (success, returndata)`

84. **OpenZeppelin Proxy**: This abstract contract provides a fallback function that delegates all calls to another contract using the EVM instruction `delegateCall`. We refer to the second contract as the implementation behind the proxy, and it has to be specified by overriding the virtual `_implementation` function. Additionally, delegation to the implementation can be triggered manually through the `_fallback` function, or to a different contract through the `_delegate` function. The success and return data of the delegated call will be returned back to the caller of the proxy. Upgradeability is only provided internally through `_upgradeTo`.

1. `constructor(address _logic, bytes _data)`: Initializes the upgradeable proxy with an initial implementation specified by `_logic`. If `_data` is nonempty, it's used as data in a delegate call to `_logic`. This will typically be an encoded function call, and allows initializing the storage of the proxy like a `Solidity` constructor.
2. `_implementation()→address impl`: Returns the current implementation address.
3. `_upgradeTo(address newImplementation)`: Upgrades the proxy to a new implementation. Emits an Upgraded event.

85. **OpenZeppelin TransparentUpgradeableProxy**: implements a proxy that is upgradeable by an admin. To avoid proxy selector clashing, which can potentially be used in an attack, this contract uses the transparent proxy pattern. This pattern implies two things that go hand in hand: 1) If any account other than the admin calls the proxy, the call will be forwarded to the implementation, even if that call matches one of the admin functions exposed by the proxy itself 2) If the admin calls the proxy, it can access the admin functions, but its calls will never be forwarded to the implementation. If the admin tries to call a function on the implementation it will fail with an error that says "admin cannot fallback to proxy target".

    These properties mean that the admin account can only be used for admin actions like upgrading the proxy or changing the admin, so it's best if it's a dedicated account that is not used for anything else. This will avoid headaches due to sudden errors when trying to call a function from the proxy implementation.

    1. `constructor(address _logic, address admin_, bytes _data)`: Initializes an upgradeable proxy managed by `_admin`, backed by the implementation at `_logic`, and optionally initialized with `_data`.
    2. `admin()→address admin_`: Returns the current admin.
    3. `implementation()→address implementation_`: Returns the current implementation.
    4. `changeAdmin(address newAdmin)`: Changes the admin of the proxy. Emits an `AdminChanged` event.
    5. `upgradeTo(address newImplementation)`: Upgrade the implementation of the proxy.
    6. `upgradeToAndCall(address newImplementation, bytes data)`: Upgrade the implementation of the proxy, and then call a function from the new implementation as specified by data, which should be an encoded function call. This is useful to initialize new storage variables in the proxied contract.
    7. `_admin()→address adm`: Returns the current admin.

8. `_beforeFallback()`: Makes sure the admin cannot access the fallback function.

86. **OpenZeppelin ProxyAdmin**: This is an auxiliary contract meant to be assigned as the admin of a `TransparentUpgradeableProxy`.

    1. `getProxyImplementation(contract TransparentUpgradeableProxy proxy)→address`: Returns the current implementation of proxy. Requirements: This contract must be the admin of proxy.
    2. `getProxyAdmin(contract TransparentUpgradeableProxy proxy)→address`: Returns the current admin of proxy. Requirements: This contract must be the admin of proxy.
    3. `changeProxyAdmin(contract TransparentUpgradeableProxy proxy, address newAdmin)`: Changes the admin of proxy to `newAdmin`. Requirements: This contract must be the current admin of proxy.
    4. `upgrade(contract TransparentUpgradeableProxy proxy, address implementation)`: Upgrades proxy to implementation. Requirements: This contract must be the admin of proxy.
    5. `upgradeAndCall(contract TransparentUpgradeableProxy proxy, address implementation, bytes data)`: Upgrades proxy to implementation and calls a function on the new implementation. Requirements: This contract must be the admin of proxy.

87. **OpenZeppelin BeaconProxy**: implements a proxy that gets the implementation address for each call from a `UpgradeableBeacon`. The beacon address is stored in storage slot `uint256(keccak256('eip1967.proxy.beacon')) - 1`, so that it doesn't conflict with the storage layout of the implementation behind the proxy.

    1. `constructor(address beacon, bytes data)`: Initializes the proxy with beacon. If data is nonempty, it's used as data in a delegate call to the implementation returned by the beacon. This will typically be an encoded function call, and allows initializing the storage of the proxy like a `Solidity` constructor. Requirements: beacon must be a contract with the interface IBeacon.
    2. `_beacon()→address beacon`: Returns the current beacon address.
    3. `_implementation()→address`: Returns the current implementation address of the associated beacon.
    4. `_setBeacon(address beacon, bytes data)`: Changes the proxy to use a new beacon. If data is nonempty, it's used as data in a delegate call to the implementation returned by the beacon. Requirements: 1) beacon must be a contract 2) The implementation returned by beacon must be a contract.

88. **OpenZeppelin UpgradeableBeacon**: is used in conjunction with one or more instances of `BeaconProxy` to determine their implementation contract, which is where they will delegate all function calls. An owner is able to change the implementation the beacon points to, thus upgrading the proxies that use this beacon.

    1. `constructor(address implementation_)`: Sets the address of the initial implementation, and the deployer account as the owner who can upgrade the beacon.
    2. `implementation()→address`: Returns the current implementation address.
    3. `upgradeTo(address newImplementation)`: Upgrades the beacon to a new implementation. Emits an Upgraded event. Requirements: 1) `msg.sender` must be the owner of the contract 2) `newImplementation` must be a contract.

89. **OpenZeppelin Clones**: EIP 1167 is a standard for deploying minimal proxy contracts, also known as "clones". To simply and cheaply clone contract functionality in an immutable way, this standard specifies a minimal bytecode implementation that delegates all calls to a known, fixed address. The library includes functions to deploy a proxy using either `create` (traditional deployment) or `create2` (salted deterministic deployment). It also includes functions to predict the addresses of clones deployed using the deterministic method.

    1. `clone(address implementation)→address instance`: Deploys and returns the address of a clone that mimics the behaviour of implementation. This function uses the `create` opcode, which should never revert.
    2. `cloneDeterministic(address implementation, bytes32 salt)→address instance`: Deploys and returns the address of a clone that mimics the behaviour of implementation. This function uses the `create2` opcode and a `salt` to deterministically deploy the clone. Using the same implementation and `salt` multiple times will revert, since the clones cannot be deployed twice at the same address.
    3. `predictDeterministicAddress(address implementation, bytes32 salt, address deployer)$\rightarrow$\verb`address predicted|: Computes the address of a clone deployed using `Clones.cloneDeterministic`.
    4. `predictDeterministicAddress(address implementation, bytes32 salt)→address predicted`: Computes the address of a clone deployed using `Clones.cloneDeterministic`.

90. **OpenZeppelin Initializable**: aids in writing upgradeable contracts, or any kind of contract that will be deployed behind a proxy. Since a proxied

contract cannot have a constructor, it is common to move constructor logic to an external initializer function, usually called `initialize`. It then becomes necessary to protect this initializer function so it can only be called once. The initializer modifier provided by this contract will have this effect.

To avoid leaving the proxy in an uninitialized state, the initializer function should be called as early as possible by providing the encoded function call as the `_data` argument. When used with inheritance, manual care must be taken to not invoke a parent initializer twice, or to ensure that all initializers are idempotent. This is not verified automatically as constructors are by `Solidity`.

91. **Dappsys DSProxy**: implements a proxy deployed as a standalone contract which can then be used by the owner to execute code. A user would pass in the bytecode for the contract as well as the `calldata` for the function they want to execute. The proxy will create a contract using the bytecode. It will then `delegatecall` the function and arguments specified in the calldata.

92. **Dappsys DSMath**: provides arithmetic functions for the common numerical primitive types of `Solidity`. You can safely add, subtract, multiply, and divide `uint` numbers without fear of integer overflow. You can also find the minimum and maximum of two numbers. Additionally, this package provides arithmetic functions for two new higher level numerical concepts called `wad` (18 decimals) and `ray` (27 decimals). These are used to represent fixed-point decimal numbers. A wad is a decimal number with 18 digits of precision and a ray is a decimal number with 27 digits of precision. These functions are necessary to account for the difference between how integer arithmetic behaves normally, and how decimal arithmetic should actually work.

    The standard functions are the `uint` set, so their function names are not prefixed: `add`, `sub`, `mul`, `min` and `max`. There is no `div` function, as divide-by-zero checking is built into the `Solidity` compiler. The `int` functions have an `i` prefix: `imin` and `imax`. Wad functions have a `w` prefix: `wmul`, `wdiv`. Ray functions have a `r` prefix: `rmul`, `rdiv` and `rpow`.

93. **Dappsys DSAuth**: Provides a flexible and updatable auth pattern which is completely separate from application logic. By default, the auth modifier will restrict function-call access to the including contract owner and the including contract itself. The auth modifier provided by DSAuth triggers the internal `isAuthorized` function to require that the `msg.sender` is authorized ie. the sender is either: 1) the contract owner 2) the contract itself or 3) has been granted permission via a specified authority.

94. **Dappsys DSGuard**: Manages an Access Control List which maps source and destination addresses to function signatures. Intended to be used as an authority for DSAuth where it acts as a lookup table for the `canCall` function to provide boolean answers as to whether a particular address is authorized to call a given function at another address. The ACL is a mapping of `[src][dst][sig] =>; boolean` where an address `src` can be either permitted or forbidden access to a function `sig` at address `dst` according to the boolean value. When used as an authority by `DSAuth` the `src` is considered to be the `msg.sender`, the `dst` is the including contract and `sig` is the function which invoked the auth modifier.

95. **Dappsys DSRoles**: A role-driven authority for ds-auth which facilitates access to lists of user roles and capabilities. Works as a set of lookup tables for the `canCall` function to provide boolean answers as to whether a user is authorized to call a given function at given address. `DSRoles` provides 3 different ways of permitting/forbidding function call access to users

    1. **Root Users**: any users added to the `_root_users` whitelist will be authorized to call any function regardless of what roles or capabilities might be defined.
    2. **Public Capabilities**: public capabilities are global capabilities which apply to all users and take precedence over any user specific role-capabilities which might be defined.
    3. **Role Capabilities**: capabilities which are associated with a particular role.

    Role capabilities are only checked if the user does not have root access and the capability is not public.

96. **WETH**: WETH stands for Wrapped Ether. For protocols that work with ERC-20 tokens but also need to handle Ether, WETH contracts allow converting Ether to its ERC-20 equivalent WETH (called wrapping) and vice-versa (called unwrapping). WETH can be created by sending ether to a WETH smart contract where the Ether is stored and in turn receiving the WETH ERC-20 token at a 1:1 ratio. This WETH can be sent back to the same smart contract to be "unwrapped" i.e. redeemed back for the original Ether at a 1:1 ratio. The most widely used WETH contract is WETH9 which holds more than 7 million Ether for now.

97. **Uniswap V2**: Uniswap is an automated liquidity protocol powered by a constant product formula and implemented in a system of non-upgradeable smart contracts on the Ethereum blockchain. The automated market making algorithm used by Uniswap is `x*y=k`, where `x` and `y` represent a token pair that allow one token to be exchanged for the other as long as the "constant product" formula is preserved i.e. trades must not change the product (`k`) of a pair's reserve balances (`x` and `y`). Core concepts:

1. **Pools**: Each Uniswap liquidity pool is a trading venue for a pair of ERC20 tokens. When a pool contract is created, its balances of each token are 0; in order for the pool to begin facilitating trades, someone must seed it with an initial deposit of each token. This first liquidity provider is the one who sets the initial price of the pool. They are incentivized to deposit an equal value of both tokens into the pool. Whenever liquidity is deposited into a pool, unique tokens known as liquidity tokens are minted and sent to the provider's address. These tokens represent a given liquidity provider's contribution to a pool.
2. **Swaps**: allows one to trade one ERC-20 token for another, where one token is withdrawn (purchased) and a proportional amount of the other deposited (sold), in order to maintain the constant `x*y=k`.
3. **Flash Swaps**: allows one to withdraw up to the full reserves of any ERC20 token on Uniswap and execute arbitrary logic at no upfront cost, provided that by the end of the transaction they either: 1) pay for the withdrawn ERC20 tokens with the corresponding pair tokens 2) return the withdrawn ERC20 tokens along with a small fee.
4. **Oracles**: enables developers to build highly decentralized and manipulation-resistant on-chain price oracles. A price oracle is any tool used to view price information about a given asset. Every pair measures (but does not store) the market price at the beginning of each block, before any trades take place i.e. price at the end of the previous block which is added to a single cumulative-price variable weighted by the amount of time this price existed. This variable can be used by external contracts to track accurate time-weighted average prices (TWAPs) across any time interval.

For more information, see [12].

98. **Uniswap V3** introduces:

    1. **Concentrated liquidity**: giving individual LPs granular control over what price ranges their capital is allocated to. Individual positions are aggregated together into a single pool, forming one combined curve for users to trade against.
    2. **Multiple fee tiers**: allowing LPs to be appropriately compensated for taking on varying degrees of risk.
    3. V3 oracles are capable of providing time-weighted average prices (TWAPs) on demand for any period within the last $\tilde{9}$ days. This removes the need for integrators to checkpoint historical values.

    For more information, see [13].

99. **Chainlink Oracles & Price Feeds**: Chainlink Price Feeds provide aggregated data (via its `AggregatorV3Interface` contract interface) from various high quality data providers, fed on-chain by decentralized oracles

on the Chainlink Network. To get price data into smart contracts for an asset that isn't covered by an existing price feed, such as the price of a particular stock, one can customize Chainlink oracles to call any external API.

B.4 Security Pitfalls & Best Practices 101

(See https://secureum.substack.com/p/security-pitfalls-and-best-practices-101 for the original article.)

1. **Solidity versions**: Using very old versions of Solidity prevents benefits of bug fixes and newer security checks. Using the latest versions might make contracts susceptible to undiscovered compiler bugs. Consider using one of these versions: `0.7.5`, `0.7.6` or `0.8.4`. (see here)

2. **Unlocked pragma**: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using `^` in `pragma solidity 0.5.10`) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. (see here)

3. **Multiple Solidity pragma**: It is better to use one Solidity compiler version across all contracts instead of different versions with different bugs and security checks. (see here)

4. **Incorrect access control**: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic. (see here and here)

5. **Unprotected withdraw function**: Unprotected (`external`/`public`) function calls sending Ether/tokens to user-controlled addresses may allow users to withdraw unauthorized funds. (see here)

6. **Unprotected call to selfdestruct**: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions. (see here)

7. **Modifier side-effects**: Modifiers should only implement checks and not make state changes and external calls which violates the checks-effects-interactions pattern. These side-effects may go unnoticed by developers/auditors because the modifier code is typically far from the function implementation. (see here)

8. **Incorrect modifier**: If a modifier does not execute `"_"` or `revert`, the function using that modifier will return the default value causing unexpected behavior. (see here)

9. **Constructor names:** Before `solc 0.4.22`, constructor names had to be the same name as the contract class containing it. Misnaming it wouldn't make it a constructor which has security implications. `Solc 0.4.22` introduced the `constructor` keyword. Until `solc 0.5.0`, contracts could have both old-style and new-style constructor names with the first defined one taking precedence over the second if both existed, which also led to

security issues. `Solc 0.5.0` forced the use of the `constructor` keyword. (see here and here)

10. **Void constructor:** Calls to base contract constructors that are unimplemented leads to misplaced assumptions. Check if the constructor is implemented or remove call if not. (see here)

11. **Implicit constructor callValue check**: The creation code of a contract that does not define a constructor but has a base that does, did not revert for calls with non-zero callValue when such a constructor was not explicitly payable. This is due to a compiler bug introduced in `v0.4.5` and fixed in `v0.6.8`. Starting from Solidity 0.4.5 the creation code of contracts without explicit payable constructor is supposed to contain a callvalue check that results in contract creation reverting, if non-zero value is passed. However, this check was missing in case no explicit constructor was defined in a contract at all, but the contract has a base that does define a constructor. In these cases it is possible to send value in a contract creation transaction or using inline assembly without revert, even though the creation code is supposed to be non-payable. (see here)

12. **Controlled delegatecall:** `delegatecall()` or `callcode()` to an address controlled by the user allows execution of malicious contracts in the context of the caller's state. Ensure trusted destination addresses for such calls. (see here)

13. **Reentrancy vulnerabilities**: Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards. (see here)

14. **ERC777 callbacks and reentrancy:** ERC777 tokens allow arbitrary callbacks via hooks that are called during token transfers. Malicious contract addresses may cause reentrancy on such callbacks if reentrancy guards are not used. (see here)

15. **Avoid transfer()/send()as reentrancy mitigations**: Although `transfer()` and `send()` have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. Use `call()` instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection. (see here and here)

16. **Private on-chain data**: Marking variables `private` does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain. (see here)

17. **Weak PRNG**: PRNG relying on `block.timestamp`, `now` or `blockhash` can be influenced by miners to some extent and should be avoided. (see here)

18. **Block values as time proxies:** `block.timestamp` and `block.number` are not good proxies (i.e. representations, not to be confused with smart contract proxy/implementation pattern) for time because of issues with synchronization, miner manipulation and changing block times. (see here)

19. **Integer overflow/underflow**: Not using OpenZeppelin's SafeMath (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. `Solc v0.8.0` introduced default overflow/underflow checks for all arithmetic operations. (see here and here)

20. **Divide before multiply:** Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate. (see here)

21. **Transaction order dependence**: Race conditions can be forced on specific Ethereum transactions by monitoring the mempool. For example, the classic ERC20 `approve()` change can be front-run using this method. Do not make assumptions about transaction order dependence. (see here)

22. **ERC20 approve() race condition**: Use `safeIncreaseAllowance()` and `safeDecreaseAllowance()` from OpenZeppelin's `SafeERC20` implementation to prevent race conditions from manipulating the allowance amounts. (see here)

23. **Signature malleability**: The `ecrecover` function is susceptible to signature malleability which could lead to replay attacks. Consider using OpenZeppelin'sECDSA library. (see here, here and here)

24. **ERC20 transfer() does not return boolean:** Contracts compiled with `solc > 0.4.22` interacting with such functions will revert. Use OpenZeppelin's SafeERC20 wrappers. (see here and here)

25. **Incorrect return values for ERC721 ownerOf()**: Contracts compiled with `solc > 0.4.22` interacting with ERC721 `ownerOf()` that returns a `bool` instead of `address` type will revert. Use OpenZeppelin's ERC721 contracts. (see here)

26. **Unexpected Ether and this.balance**: A contract can receive Ether via `payable` functions, `selfdestruct(), coinbase` transaction or pre-sent before creation. Contract logic depending on `this.balance` can therefore be manipulated. (see here and here)

27. **fallback vs receive()**: Check that all precautions and subtleties of `fallback`/`receive` functions related to visibility, state mutability and Ether transfers have been considered. (see here and here)

28. **Dangerous strict equalities**: Use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using `>` or `<=` instead of `==` for such variables depending on the contract logic. (see here)

29. **Locked Ether**: Contracts that accept Ether via `payable` functions but without withdrawal mechanisms will lock up that Ether. Remove `payable` attribute or add withdraw function. (see here)

30. **Dangerous usage of tx.origin**: Use of `tx.origin` for authorization may be abused by a MITM malicious contract forwarding calls from the legitimate user who interacts with it. Use `msg.sender` instead. (see here)

31. **Contract check**: Checking if a call was made from an Externally Owned Account (EOA) or a contract account is typically done using `extcodesize` check which may be circumvented by a contract during construction when it does not have source code available. Checking if `tx.origin == msg.sender` is another option. Both have implications that need to be considered. (see here)

32. **Deleting a mapping within a struct**: Deleting a `struct` that contains a `mapping` will not delete the `mapping` contents which may lead to unintended consequences. (see here)

33. **Tautology or contradiction**: Tautologies (always true) or contradictions (always false) indicate potential flawed logic or redundant checks. e.g. `x > 0` which is always true if `x` is `uint`. (see here)

34. **Boolean constant**: Use of Boolean constants (`true`/`false`) in code (e.g. conditionals) is indicative of flawed logic. (see here)

35. **Boolean equality**: Boolean variables can be checked within conditionals directly without the use of equality operators to `true`/`false`. (see here)

36. **State-modifying functions**: Functions that modify state (in assembly or otherwise) but are labelled `constant`/`pure`/`view` revert in `solc >0.5.0` (work in prior versions) because of the use of `STATICCALL` opcode. (see here)

37. **Return values of low-level calls**: Ensure that return values of low-level calls (`call`/`callcode`/`delegatecall`/`send`/etc.) are checked to avoid unexpected failures. (see here)

38. **Account existence check for low-level calls**: Low-level calls `call`/`delegatecall`/`staticcall` return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed. (see here)

39. **Dangerous shadowing:** Local variables, state variables, functions, modifiers, or events with names that shadow (i.e. override) builtin Solidity symbols e.g. `now` or other declarations from the current scope are misleading and may lead to unexpected usages and behavior. (see here)

40. **Dangerous state variable shadowing**: Shadowing state variables in derived contracts may be dangerous for critical variables such as contract owner (for e.g. where modifiers in base contracts check on base variables but shadowed variables are set mistakenly) and contracts incorrectly use base/shadowed variables. Do not shadow state variables. (see here)

41. **Pre-declaration usage of local variables**: Usage of a variable before its declaration (either declared later or in another scope) leads to unexpected behavior in `solc < 0.5.0` but `solc > 0.5.0` implements C99-style scoping rules where variables can only be used after they have been declared and only in the same or nested scopes. (see here)

42. **Costly operations inside a loop**: Operations such as state variable updates (use SSTOREs) inside a loop cost a lot of gas, are expensive and may lead to out-of-gas errors. Optimizations using local variables are preferred. (see here)

43. **Calls inside a loop:** Calls to external contracts inside a loop are dangerous (especially if the loop index can be user-controlled) because it could lead to DoS if one of the calls reverts or execution runs out of gas. Avoid calls within loops, check that loop index cannot be user-controlled or is bounded. (see here)

44. **DoS with block gas limit**: Programming patterns such as looping over arrays of unknown size may lead to DoS when the gas cost of execution exceeds the block gas limit. (see here)

45. **Missing events**: Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain. (see here)

46. **Unindexed event parameters**: Parameters of certain events are expected to be indexed (e.g. ERC20 Transfer/Approval events) so that they're included in the block's bloom filter for faster access. Failure to do so might confuse off-chain tooling looking for such indexed events. (see here)

47. **Incorrect event signature in libraries**: Contract types used in events in libraries cause an incorrect event signature hash. Instead of using the type 'address' in the hashed signature, the actual contract name was used, leading to a wrong hash in the logs. This is due to a compiler bug introduced in `v0.5.0` and fixed in `v0.5.8`. (see here)

48. **Dangerous unary expressions**: Unary expressions such as `x =+ 1` are likely errors where the programmer really meant to use `x += 1`. Unary `+` operator was deprecated in `solc v0.5.0`. (see here)

49. **Missing zero address validation**: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever. (see here)

50. **Critical address change**: Changing critical addresses in contracts should be a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) replaces the old address with the new one (i.e. claims ownership). This gives an opportunity to recover from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible. (see here and here)

51. **assert()/require() state change**: Invariants in `assert()` and `require()` statements should not modify the state per best practices. (see here)

52. **require() vs assert()**: `require()` should be used for checking error conditions on inputs and return values while `assert()` should be used for invariant checking. Between `solc 0.4.10` and `0.8.0`, `require()` used `REVERT` (`0xfd`) opcode which refunded remaining gas on failure while `assert()` used INVALID (`0xfe`) opcode which consumed all the supplied gas. (see here)

53. **Deprecated keywords**: Use of deprecated functions/operators such as `block.blockhash()` for `blockhash()`, `msg.gas` for `gasleft()`, `throw` for `revert()`, `sha3()` for `keccak256()`, `callcode()` for `delegatecall()`, `suicide()` for `selfdestruct()`, `constant` for `view` or `var` for `actual type name` should be avoided to prevent unintended errors with newer compiler versions. (see here)

54. **Function default visibility**: Functions without a visibility type specifier are `public` by default in `solc < 0.5.0`. This can lead to a vulnerability where a malicious user may make unauthorized state changes. `solc > 0.5.0` requires explicit function visibility specifiers. (see here)

55. **Incorrect inheritance order**: Contracts inheriting from multiple contracts with identical functions should specify the correct inheritance order i.e. more general to more specific to avoid inheriting the incorrect function implementation. (see here)

56. **Missing inheritance**: A contract might appear (based on name or functions implemented) to inherit from another interface or abstract contract without actually doing so. (see here)

57. **Insufficient gas griefing**: Transaction relayers need to be trusted to provide enough gas for the transaction to succeed. (see here)

58. **Modifying reference type parameters**: Structs/Arrays/Mappings passed as arguments to a function may be by value (memory) or reference (storage) as specified by the data location (optional before `solc 0.5.0`). Ensure correct usage of memory and storage in function parameters and make all data locations explicit. (see here)

59. **Arbitrary jump with function type variable:** Function type variables should be carefully handled and avoided in assembly manipulations to prevent jumps to arbitrary code locations. (see here)

60. **Hash collisions with multiple variable length arguments**: Using `abi.encodePacked()` with multiple variable length arguments can, in certain situations, lead to a hash collision. Do not allow users access to parameters used in `abi.encodePacked()`, use fixed length arrays or use `abi.encode()`. (see here and here)

61. **Malleability risk from dirty high order bits**: Types that do not occupy the full 32 bytes might contain "dirty higher order bits" which does not affect operation on types but gives different results with `msg.data`. (see here)

62. **Incorrect shift in assembly**: Shift operators (`shl(x, y)`, `shr(x, y)`, `sar(x, y)`) in Solidity assembly apply the shift operation of `x` bits on `y` and not the other way around, which may be confusing. Check if the values in a shift operation are reversed. (see here)

63. **Assembly usage**: Use of EVM assembly is error-prone and should be avoided or double-checked for correctness. (see here)

64. **Right-To-Left-Override control character (U+202E)**: Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract. `U+202E` character should not appear in the source code of a smart contract. (see here)

65. **Constant state variables**: Constant state variables should be declared constant to save gas. (see here)

66. **Similar variable names**: Variables with similar names could be confused for each other and therefore should be avoided. (see here)

67. **Uninitialized state/local variables**: Uninitialized state/local variables are assigned zero values by the compiler and may cause unintended results e.g. transferring tokens to zero address. Explicitly initialize all state/local variables. (see here and here)

68. **Uninitialized storage pointers:** Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to vulnerabilities. `Solc 0.5.0` and above disallow such pointers. (see here)

69. **Uninitialized function pointers in constructors:** Calling uninitialized function pointers in constructors of contracts compiled with `solc` versions `0.4.5-0.4.25` and `0.5.0-0.5.7` lead to unexpected behavior because of a compiler bug. (see here)

70. **Long number literals**: Number literals with many digits should be carefully checked as they are prone to error. (see here)

71. **Out-of-range enum:** `Solc < 0.4.5` produced unexpected behavior with out-of-range enums. Check enum conversion or use a newer compiler. (see here)

72. **Uncalled public functions**: `Public` functions that are never called from within the contracts should be declared `external` to save gas. (see here)

73. **Dead/Unreachable code**: Dead code may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

74. **Unused return values**: Unused return values of function calls are indicative of programmer errors which may have unexpected behavior. (see here)

75. **Unused variables**: Unused state/local variables may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

76. **Redundant statements**: Statements with no effects that do not produce code may be indicative of programmer error or missing logic, which needs to be flagged for removal or addressed appropriately. (see here)

77. **Storage array with signed Integers with ABIEncoderV2**: Assigning an array of signed integers to a storage array of different type can lead to data corruption in that array. This is due to a compiler bug introduced in `v0.4.7` and fixed in `v0.5.10`. (see here)

78. **Dynamic constructor arguments clipped with ABIEncoderV2**: A contract's constructor which takes structs or arrays that contain dynamically sized arrays reverts or decodes to invalid data when ABIEncoderV2 is used. This is due to a compiler bug introduced in `v0.4.16` and fixed in `v0.5.9`. (see here)

79. **Storage array with multiSlot element with ABIEncoderV2**: Storage arrays containing structs or other statically sized arrays are not read properly when directly encoded in external function calls or in `abi.encode()`. This is due to a compiler bug introduced in `v0.4.16` and fixed in `v0.5.10`. (see here)

80. **Calldata structs with statically sized and dynamically encoded members with ABIEncoderV2**: Reading from calldata structs that contain dynamically encoded, but statically sized members can result in incorrect values. This is due to a compiler bug introduced in `v0.5.6` and fixed in `v0.5.11`. (see here)

81. **Packed storage with ABIEncoderV2:** Storage structs and arrays with types shorter than 32 bytes can cause data corruption if encoded directly from storage using ABIEncoderV2. This is due to a compiler bug introduced in `v0.5.0` and fixed in `v0.5.7`. (see here)

82. **Incorrect loads with Yul optimizer and ABIEncoderV2**: The Yul optimizer incorrectly replaces `MLOAD` and `SLOAD` calls with values that have been previously written to the load location. This can only happen if ABIEncoderV2 is activated and the experimental Yul optimizer has been activated manually in addition to the regular optimizer in the compiler settings. This is due to a compiler bug introduced in `v0.5.14` and fixed in `v0.5.15`. (see here)

83. **Array slice dynamically encoded base type with ABIEncoderV2**: Accessing array slices of arrays with dynamically encoded base types (e.g. multi-dimensional arrays) can result in invalid data being read. This is due to a compiler bug introduced in `v0.6.0` and fixed in `v0.6.8`. (see here)

84. **Missing escaping in formatting with ABIEncoderV2:** String literals containing double backslash characters passed directly to external or encoding function calls can lead to a different string being used when ABIEncoderV2 is enabled. This is due to a compiler bug introduced in `v0.5.14` and fixed in `v0.6.8`. (see here)

85. **Double shift size overflow**: Double bitwise shifts by large constants whose sum overflows 256 bits can result in unexpected values. Nested logical shift operations whose total shift size is `2**256` or more are incorrectly optimized. This only applies to shifts by numbers of bits that are compile-time constant expressions. This happens when the optimizer is used and `evmVersion > Constantinople`. This is due to a compiler bug introduced in `v0.5.5` and fixed in `v0.5.6`. (see here)

86. **Incorrect byte instruction optimization:** The optimizer incorrectly handles byte opcodes whose second argument is 31 or a constant expression that evaluates to 31. This can result in unexpected values. This can

happen when performing index access on `bytesNN` types with a compile time constant value (not index) of 31 or when using the byte opcode in inline assembly. This is due to a compiler bug introduced in `v0.5.5` and fixed in `v0.5.7`. (see here)

87. **Essential assignments removed with Yul Optimizer** : The Yul optimizer can remove essential assignments to variables declared inside `for` loops when Yul's `continue` or `break` statement is used mostly while using inline assembly with `for` loops and `continue` and `break` statements. This is due to a compiler bug introduced in `v0.5.8`/`v0.6.0` and fixed in `v0.5.16`/`v0.6.1`. (see here)

88. **Private methods overridden**: While private methods of base contracts are not visible and cannot be called directly from the derived contract, it is still possible to declare a function of the same name and type and thus change the behaviour of the base contract's function. This is due to a compiler bug introduced in `v0.3.0` and fixed in `v0.5.17`. (see here)

89. **Tuple assignment multi stack slot components**: Tuple assignments with components that occupy several stack slots, i.e. nested tuples, pointers to external functions or references to dynamically sized calldata arrays, can result in invalid values. This is due to a compiler bug introduced in `v0.1.6` and fixed in `v0.6.6`. (see here)

90. **Dynamic array cleanup**: When assigning a dynamically sized array with types of size at most 16 bytes in storage causing the assigned array to shrink, some parts of deleted slots were not zeroed out. This is due to a compiler bug fixed in `v0.7.3`. (see here)

91. **Empty byte array copy**: Copying an empty byte array (or string) from memory or calldata to storage can result in data corruption if the target array's length is increased subsequently without storing new data. This is due to a compiler bug fixed in `v0.7.4`. (see here)

92. **Memory array creation overflow**: The creation of very large memory arrays can result in overlapping memory regions and thus memory corruption. This is due to a compiler bug introduced in `v0.2.0` and fixed in `v0.6.5`. (see here)

93. **Calldata using for**: Function calls to internal library functions with calldata parameters called via "`using for`" can result in invalid data being read. This is due to a compiler bug introduced in `v0.6.9` and fixed in `v0.6.10`. (see here)

94. **Free function redefinition**: The compiler does not flag an error when two or more free functions (functions outside of a contract) with the same name and parameter types are defined in a source unit or when an imported free function alias shadows another free function with a different

name but identical parameter types. This is due to a compiler bug introduced in `v0.7.1` and fixed in `v0.7.2`. (see here)

95. **Unprotected initializers in proxy-based upgradeable contracts**: Proxy-based upgradeable contracts need to use `public` initializer functions instead of constructors that need to be explicitly called only once. Preventing multiple invocations of such initializer functions (e.g. via `initializer` modifier from OpenZeppelin's `Initializable` library) is a must. (see here and here)

96. **Initializing state-variables in proxy-based upgradeable contracts**: This should be done in initializer functions and not as part of the state variable declarations in which case they won't be set. (see here)

97. **Import upgradeable contracts in proxy-based upgradeable contracts**: Contracts imported from proxy-based upgradeable contracts should also be upgradeable where such contracts have been modified to use initializers instead of constructors. (see here)

98. **Avoid selfdestruct or delegatecall in proxy-based upgradeable contracts**: This will cause the logic contract to be destroyed and all contract instances will end up delegating calls to an address without any code. (see here)

99. **State variables in proxy-based upgradeable contracts**: The declaration order/layout and type/mutability of state variables in such contracts should be preserved exactly while upgrading to prevent critical storage layout mismatch errors. (see here)

100. **Function ID collision between proxy/implementation in proxy-based upgradeable contracts**: Malicious proxy contracts may exploit function ID collision to invoke unintended proxy functions instead of delegating to implementation functions. Check for function ID collisions. (see here and here)

101. **Function shadowing between proxy/contract in proxy-based upgradeable contracts**: Shadow functions in proxy contract prevent functions in logic contract from being invoked. (see here)

B.5 Security Pitfalls & Best Practices 201

(See https://secureum.substack.com/p/security-pitfalls-and-best-practices-201 for the original article.)

102. **ERC20 transfer and transferFrom**: Should return a boolean. Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail. (See here.)

103. **ERC20 name, decimals, and symbol functions**: Are present if used. These functions are optional in the `ERC20` standard and might not be present. (See here.)

104. **ERC20 decimals returns a uint8**: Several tokens incorrectly return a `uint256`. If this is the case, ensure the value returned is below 255. (See here.)

105. **ERC20 approve race-condition**: The `ERC20` standard has a known `ERC20` race condition that must be mitigated to prevent attackers from stealing tokens. (See here.)

106. **ERC777 hooks**: `ERC777` tokens have the concept of a hook function that is called before any calls to `send`, `transfer`, `operatorSend`, minting and burning. While these hooks enable a lot of interesting use cases, care should be taken to make sure they do not make external calls because that can lead to reentrancies. (See here.)

107. **Token Deflation via fees**: `transfer` and `transferFrom` should not take a fee. Deflationary tokens can lead to unexpected behavior. (See here.)

108. **Token Inflation via interest**: Potential interest earned from the token should be taken into account. Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account. (See here.)

109. **Token contract avoids unneeded complexity**: The token should be a simple contract; a token with complex code requires a higher standard of review. (See here.)

110. **Token contract has only a few non–token-related functions**: Non–token-related functions increase the likelihood of an issue in the contract. (See here.)

111. **Token only has one address**: Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g. `balances[token_address][msg.sender]` might not reflect the actual balance). (See here.)

112. **Token is not upgradeable**: Upgradeable contracts might change their rules over time. (See here.)

113. **Token owner has limited minting capabilities**: Malicious or compromised owners can abuse minting capabilities. (See here.)

114. **Token is not pausable**: Malicious or compromised owners can trap contracts relying on pausable tokens. (See here.)

115. **Token owner cannot blacklist the contract**: Malicious or compromised owners can trap contracts relying on tokens with a blacklist. (See here.)

116. **Token development team is known and can be held responsible for abuse**: Contracts with anonymous development teams, or that reside in legal shelters should require a higher standard of review. (See here.)

117. **No token user owns most of the supply**: If a few users own most of the tokens, they can influence operations based on the token's repartition. (See here.)

118. **Token total supply is sufficient**: Tokens with a low total supply can be easily manipulated. (See here.)

119. **Tokens are located in more than a few exchanges**: If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token. (See here.)

120. **Token balance and Flash loans**: Users understand the associated risks of large funds or flash loans. Contracts relying on the token balance must carefully take in consideration attackers with large funds or attacks through flash loans. (See here.)

121. **Token does not allow flash minting**: Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token. (See here.)

122. **ERC1400 permissioned addresses**: Can block transfers from/to specific addresses. (See here.)

123. **ERC1400 forced transfers**: Trusted actors have the ability to transfer funds however they choose. (See here.)

124. **ERC1644 forced transfers**: Controller has the ability to steal funds. (See here.)

125. **ERC621 control of totalSupply**: `totalsupply` can be changed by trusted actors (See here.)

126. **ERC884 cancel and reissue**: Token implementers have the ability to cancel an address and move its tokens to a new address (See here.)

127. **ERC884 whitelisting**: Tokens can only be sent to whitelisted addresses (See here.)

128. **Guarded launch via asset limits**: Limiting the total asset value managed by a system initially upon launch and gradually increasing it over time may reduce impact due to initial vulnerabilities or exploits. (See here.)

129. **Guarded launch via asset types:** Limiting types of assets that can be used in the protocol initially upon launch and gradually expanding to other assets over time may reduce impact due to initial vulnerabilities or exploits. (See here.)

130. **Guarded launch via user limits:** Limiting the total number of users that can interact with a system initially upon launch and gradually increasing it over time may reduce impact due to initial vulnerabilities or exploits. Initial users may also be whitelisted to limit to trusted actors before opening the system to everyone. (See here.)

131. **Guarded launch via usage limits**: Enforcing transaction size limits, daily volume limits, per-account limits, or rate-limiting transactions may reduce impact due to initial vulnerabilities or exploits. (See here.)

132. **Guarded launch via composability limits:** Restricting the composability of the system to interface only with whitelisted trusted contracts before expanding to arbitrary external contracts may reduce impact due to initial vulnerabilities or exploits. (See here.)

133. **Guarded launch via escrows**: Escrowing high value transactions/operations with time locks and a governance capability to nullify or revert transactions may reduce impact due to initial vulnerabilities or exploits. (See here.)

134. **Guarded launch via circuit breakers**: Implementing capabilities to pause/unpause a system in extreme scenarios may reduce impact due to initial vulnerabilities or exploits. (See here.)

135. **Guarded launch via emergency shutdown:** Implement capabilities that allow governance to shutdown new activity in the system and allow users to reclaim assets may reduce impact due to initial vulnerabilities or exploits. (See here.)

136. **System specification**: Ensure that the specification of the entire system is considered, written and evaluated to the greatest detail possible. Specification describes how (and why) the different components of the system behave to achieve the design requirements. Without specification, a

system implementation cannot be evaluated against the requirements for correctness.

137. **System documentation**: Ensure that roles, functionalities and interactions of the entire system are well documented to the greatest detail possible. Documentation describes what (and how) the implementation of different components of the system does to achieve the specification goals. Without documentation, a system implementation cannot be evaluated against the specification for correctness and one will have to rely on analyzing the implementation itself.

138. **Function parameters**: Ensure input validation for all function parameters especially if the visibility is `external`/`public` where (untrusted) users can control values. This is especially required for address parameters where maliciously/accidentally used incorrect/zero addresses can cause vulnerabilities or unexpected behavior.

139. **Function arguments**: Ensure that the arguments to function calls at the caller sites are the correct ones and in the right order as expected by the function definition.

140. **Function visibility**: Ensure that the strictest visibility is used for the required functionality. An accidental `external`/`public` visibility will allow (untrusted) users to invoke functionality that is supposed to be restricted internally.

141. **Function modifiers**: Ensure that the right set of function modifiers are used (in the correct order) for the specific functions so that the expected access control or validation is correctly enforced.

142. **Function return values**: Ensure that the appropriate return value(s) are returned from functions and checked without ignoring at function call sites, so that error conditions are caught and handled appropriately.

143. **Function invocation timeliness**: Externally accessible functions (`external`/`public` visibility) may be called at any time (or never). It is not safe to assume they will only be called at specific system phases (e.g. after initialization, when unpaused, during liquidation) that is meaningful to the system design. The reason for this can be accidental or malicious. Function implementation should be robust enough to track system state transitions, determine meaningful states for invocations and withstand arbitrary calls. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called atomically along with contract deployment to prevent anyone else from initializing with arbitrary values.

144. **Function invocation repetitiveness**: Externally accessible functions (`external`/`public` visibility) may be called any number of times. It is not safe to assume they will only be called only once or a specific number

of times that is meaningful to the system design. Function implementation should be robust enough to track, prevent, ignore or account for arbitrarily repetitive invocations. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called only once.

145. **Function invocation order:** Externally accessible functions (`external`/`public` visibility) may be called in any order (with respect to other defined functions). It is not safe to assume they will only be called in the specific order that makes sense to the system design or is implicitly assumed in the code. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called before other system functions can be called.

146. **Function invocation arguments**: Externally accessible functions (`external`/`public` visibility) may be called with any possible arguments. Without complete and proper validation (e.g. zero address checks, bound checks, threshold checks etc.), they cannot be assumed to comply with any assumptions made about them in the code.

147. **Conditionals**: Ensure that in conditional expressions (e.g. if statements), the correct variables are being checked and the correct operators, if any, are being used to combine them. For e.g. using `||` instead of `&&` is a common error.

148. **Access control specification**: Ensure that the various system actors, their access control privileges and trust assumptions are accurately specified in great detail so that they are correctly implemented and enforced across different contracts, functions and system transitions/flows.

149. **Access control implementation**: Ensure that the specified access control is implemented uniformly across all the subjects (actors) seeking access and objects (variables, functions) being accessed so that there are no paths/flows where the access control is missing or may be side-stepped.

150. **Missing modifiers**: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. Ensure that such modifiers are present on all relevant functions which require that specific access control.

151. **Incorrectly implemented modifiers**: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. A system can have multiple roles with different privileges. Ensure that modifiers are implementing the expected checks on the correct roles/addresses with the right composition e.g. incorrect use of `||` instead of `&&` is a common vulnerability while composing access checks.

152. **Incorrectly used modifiers**: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. A system can have multiple roles with different privileges. Ensure that correct modifiers are used on functions requiring specific access control enforced by that modifier.

153. **Access control changes**: Ensure that changes to access control (e.g. change of ownership to new addresses) are handled with extra security so that such transitions happen smoothly without contracts getting locked out or compromised due to use of incorrect credentials.

154. **Comments**: Ensure that the code is well commented both with NatSpec and inline comments for better readability and maintainability. The comments should accurately reflect what the corresponding code does. Stale comments should be removed. Discrepancies between code and comments should be addressed. Any "*TODO*"'s indicated by comments should be addressed. Commented code should be removed.

155. **Tests**: Tests indicate that the system implementation has been validated against the specification. Unit tests, functional tests and integration tests should have been performed to achieve good test coverage across the entire codebase. Any code or parameterisation used specifically for testing should be removed from production code.

156. **Unused constructs**: Any unused imports, inherited contracts, functions, parameters, variables, modifiers, events or return values should be removed (or used appropriately) after careful evaluation. This will not only reduce gas costs but improve readability and maintainability of the code.

157. **Redundant constructs**: Redundant code and comments can be confusing and should be removed (or changed appropriately) after careful evaluation. This will not only reduce gas costs but improve readability and maintainability of the code.

158. **ETH Handling**: Contracts that accept/manage/transfer ETH should ensure that functions handling ETH are using `msg.value` appropriately, logic that depends on ETH value accounts for less/more ETH sent, logic that depends on contract ETH balance accounts for the different direct/indirect (e.g. `coinbase` transaction, `selfdestruc` recipient) ways of receiving ETH and transfers are reentrancy safe. Functions handling ETH should be checked extra carefully for access control, input validation and error handling.

159. **Token Handling**: Contracts that accept/manage/transfer ERC tokens should ensure that functions handling tokens account for different types of ERC tokens (e.g. `ERC20` vs `ERC777`), deflationary/inflationary tokens, rebasing tokens and trusted/external tokens. Functions handling tokens should be checked extra carefully for access control, input validation and error handling.

160. **Trusted actors**: Ideally there should be no trusted actors while interacting with smart contracts. However, in guarded launch scenarios, the goal is to start with trusted actors and then progressively decentralise towards automated governance by community/DAO. For the trusted phase, all the trusted actors, their roles and capabilities should be clearly specified, implemented accordingly and documented for user information and examination.

161. **Privileged roles and EOAs**: Trusted actors who have privileged roles with capabilities to deploy contracts, change critical parameters, pause/unpause system, trigger emergency shutdown, withdraw/transfer/drain funds and allow/deny other actors should be addresses controlled by multiple, independent, mutually distrusting entities. They should not be controlled by private keys of EOAs but with Multisigs with a high threshold (e.g. 5-of-7, 9-of-11) and eventually by a DAO of token holders. EOA has a single point of failure.

162. **Two-step change of privileged roles**: When privileged roles are being changed, it is recommended to follow a two-step approach:

    1. The current privileged role proposes a new address for the change.
    2. The newly proposed address then claims the privileged role in a separate transaction.

    This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/malicious privileged role. For e.g., in a single-step change, if the current admin accidentally changes the new admin to a zero-address or an incorrect address (where the private keys are not available), the system is left without an operational admin and will have to be redeployed.

163. **Time-delayed change of critical parameters**: When critical parameters of systems need to be changed, it is required to broadcast the change via event emission and recommended to enforce the changes after a time-delay. This is to allow system users to be aware of such critical changes and give them an opportunity to exit or adjust their engagement with the system accordingly. For e.g. reducing the rewards or increasing the fees in a system might not be acceptable to some users who may wish to withdraw their funds and exit.

164. **Explicit over Implicit**: While Solidity has progressively adopted explicit declarations of intent for e.g. with function visibility and variable storage, it is recommended to do the same at the application level where all requirements should be explicitly validated (e.g. input parameters) and assumptions should be documented and checked. Implicit requirements and assumptions should be flagged as dangerous.

165. **Configuration issues**: Misconfiguration of system components such contracts, parameters, addresses and permissions may lead to security issues.

166. **Initialization issues**: Lack of initialization, initializing with incorrect values or allowing untrusted actors to initialize system parameters may lead to security issues.

167. **Cleanup issues**: Missing to clean up old state or cleaning up incorrectly/insufficiently will lead to reuse of stale state which may lead to security issues.

168. **Data processing issues**: Processing data incorrectly will cause unexpected behavior which may lead to security issues.

169. **Data validation issues**: Missing validation of data or incorrectly/insufficiently validating data, especially tainted data from untrusted users, will cause untrustworthy system behavior which may lead to security issues.

170. **Numerical issues**: Incorrect numerical computation will cause unexpected behavior which may lead to security issues.

171. **Accounting issues**: Incorrect or insufficient tracking or accounting of business logic related aspects such as states, phases, permissions, roles, funds (deposits/withdrawals) and tokens (mints/burns/transfers) may lead to security issues.

172. **Access control issues**: Incorrect or insufficient access control or authorization related to system actors, roles, assets and permissions may lead to security issues.

173. **Auditing/logging issues**: Incorrect or insufficient emission of events will impact off-chain monitoring and incident response capabilities which may lead to security issues.

174. **Cryptography issues**: Incorrect or insufficient cryptography especially related to on-chain signature verification or off-chain key management will impact access control and may lead to security issues.

175. **Error-reporting issues**: Incorrect or insufficient detecting, reporting and handling of error conditions will cause exceptional behavior to go unnoticed which may lead to security issues.

176. **Denial-of-Service (DoS) issues**: Preventing other users from successfully accessing system services by modifying system parameters or state causes denial-of-service issues which affects the availability of the system. This may also manifest as security issues if users are not able to access their funds locked in the system.

177. **Timing issues**: Incorrect assumptions on timing of user actions, system state transitions or blockchain state/blocks/transactions may lead to security issues.

178. **Ordering issues**: Incorrect assumptions on ordering of user actions or system state transitions may lead to security issues. For e.g., a user may accidentally/maliciously call a finalization function even before the initialization function if the system allows.

179. **Undefined behavior issues**: Any behavior that is undefined in the specification but is allowed in the implementation will result in unexpected outcomes which may lead to security issues.

180. **External interaction issues**: Interacting with external components (e.g. tokens, contracts, oracles) forces system to trust or make assumptions on their correctness/availability requiring validation of their existence and outputs without which may lead to security issues.

181. **Trust issues**: Incorrect or Insufficient trust assumption about/among system actors and external entities will lead to privilege escalation/abuse which may lead to security issues.

182. **Gas issues**: Incorrect assumptions about gas requirements especially for loops or external calls will lead to out-of-gas exceptions which may lead to security issues such as failed transfers or locked funds.

183. **Dependency issues**: Dependencies on external actors or software (imports, contracts, libraries, tokens etc.) will lead to trust/availability/correctness assumptions which if/when broken may lead to security issues.

184. **Constant issues**: Incorrect assumptions about system actors, entities or parameters being constant may lead to security issues if/when such factors change unexpectedly.

185. **Freshness issues**: Incorrect assumptions about the status of or data from system actors or entities being fresh (i.e. not stale), because of lack of updation or availability, may lead to security issues if/when such factors have been updated. For e.g., getting a stale price from an Oracle.

186. **Scarcity issues**: Incorrect assumptions about the scarcity of tokens/funds available to any system actor will lead to unexpected outcomes which may lead to security issues. For e.g., flash loans.

187. **Incentive issues**: Incorrect assumptions about the incentives of system/external actors to perform or not perform certain actions will lead to unexpected behavior being triggered or expected behavior not being triggered, both of which may lead to security issues. For e.g., incentive to liquidate positions, lack of incentive to DoS or grief system.

188. **Clarity issues**: Lack of clarity in system specification, documentation, implementation or UI/UX will lead to incorrect expectations/outcome which may lead to security issues.

189. **Privacy issues**: Data and transactions on the Ethereum blockchain are not private. Anyone can observe contract state and track transactions (both included in a block and pending in the mempool). Incorrect assumptions about privacy aspects of data or transactions can be abused which may lead to security issues.

190. **Cloning issues**: Copy-pasting code from other libraries, contracts or even different parts of the same contract may result in incorrect code semantics for the context being copied to, copy over any vulnerabilities or miss any security fixes applied to the original code. All these may lead to security issues.

191. **Business logic issues**: Incorrect or insufficient assumptions about the higher-order business logic being implemented in the application will lead to differences in expected and actual behavior, which may result in security issues.

192. **Principle of Least Privilege**: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job" — Ensure that various system actors have the least amount of privilege granted as required by their roles to execute their specified tasks. Granting excess privilege is prone to misuse/abuse when trusted actors misbehave or their access is hijacked by malicious entities. (See Saltzer and Schroeder's Secure Design Principles.)

193. **Principle of Separation of Privilege**: "Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key" — Ensure that critical privileges are separated across multiple actors so that there are no single points of failure/abuse. A good example of this is to require a multisig address (not EOA) for privileged actors (e.g. owner, admin, governor, deployer) who control key contract functionality such as pause/unpause/shutdown, emergency fund drain, upgradeability, allow/deny list and critical parameters. The multisig address should be composed of entities that are different and mutually distrusting/verifying. (See Saltzer and Schroeder's Secure Design Principles.)

194. **Principle of Least Common Mechanism**: "Minimize the amount of mechanism common to more than one user and depended on by all users" — Ensure that only the least number of security-critical modules/paths as required are shared amongst the different actors/code so that impact from any vulnerability/compromise in shared components is limited and contained to the smallest possible subset. (See Saltzer and Schroeder's Secure Design Principles.)

195. **Principle of Fail-safe Defaults**: "Base access decisions on permission rather than exclusion". Ensure that variables or permissions are initialized to fail-safe default values which can be made more inclusive later instead

of opening up the system to everyone including untrusted actors. (See Saltzer and Schroeder's Secure Design Principles.)

196. **Principle of Complete Mediation**: "Every access to every object must be checked for authority". Ensure that any required access control is enforced along all access paths to the object or function being protected. (See Saltzer and Schroeder's Secure Design Principles.)

197. **Principle of Economy of Mechanism**: "Keep the design as simple and small as possible". Ensure that contracts and functions are not overly complex or large so as to reduce readability or maintainability. Complexity typically leads to insecurity. (See Saltzer and Schroeder's Secure Design Principles.)

198. **Principle of Open Design**: "The design should not be secret". Smart contracts are expected to be open-sourced and accessible to everyone. Security by obscurity of code or underlying algorithms is not an option. Security should be derived from the strength of the design and implementation under the assumption that (byzantine) attackers will study their details and try to exploit them in arbitrary ways. (See Saltzer and Schroeder's Secure Design Principles.)

199. **Principle of Psychological Acceptability**: "It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly". Ensure that security aspects of smart contract interfaces and system designs/flows are user-friendly and intuitive so that users can interact with minimal risk. (See Saltzer and Schroeder's Secure Design Principles.)

200. **Principle of Work Factor**: "Compare the cost of circumventing the mechanism with the resources of a potential attacker". Given the magnitude of value managed by smart contracts, it is safe to assume that byzantine attackers will risk the greatest amounts of intellectual/financial/social capital possible to subvert such systems. Therefore, the mitigation mechanisms must factor in the highest levels of risk. (See Saltzer and Schroeder's Secure Design Principles.)

201. **Principle of Compromise Recording**: "Mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss". Ensure that smart contracts and their accompanying operational infrastructure can be monitored/analyzed at all times (development/deployment/runtime) for minimizing loss from any compromise due to vulnerabilities/exploits. For e.g., critical operations in contracts should necessarily emit events to facilitate monitoring at runtime. (See Saltzer and Schroeder's Secure Design Principles.)

B.6 Audit Techniques & Tools 101

(See https://secureum.substack.com/p/audit-techniques-and-tools-101 for the original article.)

1. **Audit**: is an external security assessment of a project codebase, typically requested and paid-for by the project team
    1. It detects and describes (in a report) security issues with underlying vulnerabilities, severity/difficulty, potential exploit scenarios and recommended fixes.
    2. It also provides subjective insights into code quality, documentation and testing.
    3. The scope/depth/format of audit reports varies across auditing teams but they generally cover similar aspects.
2. **Audit Scope**: For Ethereum-based smart-contract projects, the scope is typically the on-chain smart contract code and sometimes includes the off-chain components that interact with the smart contracts.
3. **Audit Goal**: The goal of audits is to assess project code (with any associated specification, documentation) and alert project team, typically before launch, of potential security-related issues that need to be addressed to improve security posture, decrease attack surface and mitigate risk.
4. **Audit Non-goal**: Audit is **not** a security guarantee of "bug-free" code by any stretch of imagination but a best-effort endeavour by trained security experts operating within reasonable constraints of time, understanding, expertise and of course, decidability.
5. **Audit Target**: Security companies execute audits for clients who pay for their services. Engagements are therefore geared towards priorities of project owners and **not** project users/investors. Audits are **not** intended to alert potential project users of any inherent risk. That is not their business/technical goal.
6. **Audit Need**: Smart contract based projects do not have sufficient in-house Ethereum smart contract security expertise and/or time to perform internal security assessments and therefore rely on external experts who have domain expertise in those areas. Even if projects have some expertise in-house, they would still benefit from an unbiased external team with supplementary/complementary skill sets that can review the assumptions, design, specification and implementation of the project codebase.
7. **Audit Types**: depend on the scope/nature/status of projects but generally fall into the following categories:
    1. **New audit**: for a new project that is being launched

2. **Repeat audit**: for a new version of an existing project being revised with new/fixed features
3. **Fix audit**: for reviewing the fixes made to the findings from a current/prior audit
4. **Retainer audit**: for constantly reviewing project updates
5. **Incident audit**: for reviewing an exploit incident, root causing the incident, identifying the underlying vulnerabilities and proposing fixes.

8. **Audit Timeline**: depends on the scope/nature/status of the project to be assessed and the type of audit. This may vary from a few days for a fix/retainer audit to several weeks for a new/repeat/incident audit.

9. **Audit Effort**: typically involves more than one auditor simultaneously for getting independent, redundant or supplementary/complementary assessment expertise on the project.

10. **Audit Costs**: depends on the type/scope of audit but typically costs upwards of USD $10K/week depending on the complexity of the project, market demand/supply for audits and the strength/reputation of the auditing firm.

11. **Audit Prerequisites** should include:

    1. Clear definition of the scope of the project to be assessed typically in the form of a specific commit hash of project files/folders on a github repository
    2. Public/private repository
    3. Public/anonymous team
    4. Specification of the project's design and architecture
    5. Documentation of the project's implementation and business logic
    6. Threat models and specific areas of concern
    7. Prior testing, tools used, other audits
    8. Timeline, effort and costs/payments
    9. Engagement dynamics/channels for questions/clarifications, findings communication and reports
    10. Points of contact on both sides

12. **Audit Limitations**: Audits are necessary (for now at least) but not sufficient:

    1. There is risk reduction but residual risk exists because of several factors such as limited amount of audit time/effort, limited insights into project specification/implementation, limited security expertise

in the new and fast evolving technologies, limited audit scope, significant project complexity and limitations of automated/manual analysis.

2. Not all audits are equal — it greatly depends on the expertise/experience of auditors, effort invested vis-a-vis project complexity/quality and tools/processes used.
3. Audits provide a project's security snapshot over a brief (typically few weeks) period. However, smart contracts need to evolve over time to add new features, fix bugs or optimize. Relying on external audits after every change is impractical.

13. **Audit Reports**: include details of the scope, goals, effort, timeline, approach, tools/techniques used, findings summary, vulnerability details, vulnerability classification, vulnerability severity/difficulty/likelihood, vulnerability exploit scenarios, vulnerability fixes and informational recommendations/suggestions on programming best-practices.

14. **Audit Findings Classification**: The vulnerabilities found during the audit are typically classified into different categories which helps to understand the nature of the vulnerability, potential impact/severity, impacted project components/functionality and exploit scenarios. Trail of Bits, for example, uses the below classification:

    1. Access Controls: Related to authorization of users and assessment of rights
    2. Auditing and Logging: Related to auditing of actions or logging of problems
    3. Authentication: Related to the identification of users
    4. Configuration: Related to security configurations of servers, devices or software
    5. Cryptography: Related to protecting the privacy or integrity of data
    6. Data Exposure: Related to unintended exposure of sensitive information
    7. Data Validation: Related to improper reliance on the structure or values of data
    8. Denial of Service: Related to causing system failure
    9. Error Reporting: Related to the reporting of error conditions in a secure fashion
    10. Patching: Related to keeping software up to date
    11. Session Management: Related to the identification of authenticated users
    12. Timing: Related to race conditions, locking or order of operations

13. Undefined Behavior: Related to undefined behavior triggered by the program

15. **Audit Findings Likelihood/Difficulty**: Per OWASP, likelihood or difficulty is a rough measure of how likely or difficult this particular vulnerability is to be uncovered and exploited by an attacker. OWASP proposes three Likelihood levels of Low, Medium and High. Trail of Bits, for example, classifies every finding into four difficulty levels:

    1. **Undetermined**: The difficulty of exploit was not determined during this engagement
    2. **Low**: Commonly exploited, public tools exist or can be scripted that exploit this flaw
    3. **Medium**: Attackers must write an exploit, or need an in-depth knowledge of a complex system
    4. **High**: The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

16. **Audit Findings Impact**: Per OWASP, this estimates the magnitude of the technical and business impact on the system if the vulnerability were to be exploited. OWASP proposes three Impact levels of Low, Medium and High.

17. **Audit Findings Severity**: Per OWASP, the Likelihood estimate and the Impact estimate are put together to calculate an overall Severity for this risk. This is done by figuring out whether the Likelihood is Low, Medium, or High and then do the same for impact.

    1. OWASP proposes a 3x3 Severity Matrix which combines the three Likelihood levels with the three Impact levels
    2. Severity Matrix (Likelihood-Impact = Severity): Low-Low = Note; Low-Medium = Low; Low-High = Medium; Medium-Low = Low; Medium-Medium = Medium; Medium-High = High; High-Low = Medium; High-Medium = High; High-High = Critical;
    3. Trail of Bits uses:
        i. **Informational**: The issue does not pose an immediate risk, but is relevant to security best practices or Defence in Depth.
        ii. **Undetermined**: The extent of the risk was not determined during this engagement.
        iii. **Low**: The risk is relatively small or is not a risk the customer has indicated is important.
        iv. **Medium**: Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client.

v. **High**: Large numbers of users, very bad for client's reputation, or serious legal or financial implications.

4. ConsenSys uses:
    i. **Minor**: issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
    ii. **Medium**: issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
    iii. **Major**: issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
    iv. **Critical**: issues are directly exploitable security vulnerabilities that need to be fixed.

18. **Audit Checklist For Projects** (See here for Trail of Bits recommendations):

    1. Resolve the easy issues: 1) Enable and address every last compiler warning 2) Increase unit and feature test coverage 3) Remove dead code, stale branches, unused libraries, and other extraneous weight.
    2. Document: 1) Describe what your product does, who uses it, why, and how it delivers. 2) Add comments about intended behavior inline with the code. 3) Label and describe your tests and results, both positive and negative. 4) Include past reviews and bugs.
    3. Deliver the code batteries included: 1) Document the steps to create a build environment from scratch on a computer that is fully disconnected from your internal network 2) Include external dependencies 3) Document the build process, including debugging and the test environment 4) Document the deployment process and environment, including all the specific versions of external tools and libraries for this process.

19. **Audit Techniques**: involve a combination of different methods that are applied to the project codebase with accompanying specification and documentation. Many are automated analyses performed with tools and some require manual assistance.

    1. Specification analysis (manual)
    2. Documentation analysis (manual)
    3. Testing (automated)
    4. Static analysis (automated)
    5. Fuzzing (automated)

6. Symbolic checking (automated)
7. Formal verification (automated)
8. Manual analysis (manual)

One may also think of these as manual/semi-automated/fully-automated, where the distinction between semi-automated and fully-automated is the difference between a tool that requires a user to define properties versus a tool that requires (almost) no user configuration except to triage results. Fully-automated tools tend to be straightforward to use, while semi-automated tools require some human assistance and are therefore more resource-expensive.

20. **Specification analysis**: Specification describes in detail what (and sometimes why) the project and its various components are supposed to do functionally as part of their design and architecture.

    1. From a security perspective, it specifies what the assets are, where they are held, who are the actors, privileges of actors, who is allowed to access what and when, trust relationships, threat model, potential attack vectors, scenarios and mitigations.
    2. Analysing the specification of a project provides auditors with the above details and lets them evaluate the assumptions made and indicate any shortcomings
    3. Very few smart contract projects have detailed specifications at their first audit stage. At best, they have some documentation about what is implemented. Auditors spend a lot of time inferring specification from documentation/implementation which leaves them with less time for vulnerability assessment.

21. **Documentation analysis**: Documentation is a description of what has been implemented based on the design and architectural requirements.

    1. Documentation answers 'how' something has been designed/architected/implemented without necessarily addressing the 'why' and the design/requirement goals
    2. Documentation is typically in the form of Readme files in the Github repository describing individual contract functionality combined with functional NatSpec and individual code comments.
    3. Documentation in many cases serves as a substitute for specification and provides critical insights into the assumptions, requirements and goals of the project team
    4. Understanding the documentation before looking at the code helps auditors save time in inferring the architecture of the project, contract interactions, program constraints, asset flow, actors, threat model and risk mitigation measures

5. Mismatches between the documentation and the code could indicate stale/poor documentation, software defects or security vulnerabilities
6. Auditors are expected to encourage the project team to document thoroughly so that they do not need to waste their time inferring this by reading code

22. **Testing**: Software testing or validation is a well-known fundamental software engineering primitive to determine if software produces expected outputs when executed with different chosen inputs.

    1. Smart contract testing has a similar motivation but is arguably more complicated despite their relatively smaller sizes (in lines of code) compared to Web2 software
    2. Smart contract development platforms (Truffle, Embark, Brownie, Waffle, Hardhat etc.) are relatively new with different levels of support for testing
    3. Projects, in general, have very little testing done at the audit stage. Testing integrations and composability with mainnet contracts and state is non-trivial
    4. Test coverage and test cases give a good indication of project maturity and also provide valuable insights to auditors into assumptions/edge-cases for vulnerability assessments
    5. Auditors should expect a high-level of testing and test coverage because this is a must-have software-engineering discipline, especially when smart contracts that are by-design exposed to everyone on the blockchain end up holding assets worth tens of millions of dollars
    6. "Program testing can be used to show the presence of bugs, but never to show their absence!" - E.W. Dijkstra

23. **Static analysis**: is a technique of analyzing program properties without actually executing the program.

    1. This is in contrast to software testing where programs are actually executed/run with different inputs
    2. For smart contracts, static analysis can be performed on the `Solidity` code or on the EVM bytecode. Slither performs static analysis at the `Solidity` level while Mythril analyzes EVM bytecode.
    3. Static analysis typically is a combination of control flow and data flow analyses

24. Fuzzing: or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks

1. Fuzzing is especially relevant to smart contracts because anyone can interact with them on the blockchain with random inputs without necessarily having a valid reason or expectation (arbitrary byzantine behaviour)
2. Echidna and Harvey are two popular tools for smart contract fuzzing

25. Symbolic checking: is a technique of checking for program correctness, i.e. proving/verifying, by using symbolic inputs to represent set of states and transitions instead of enumerating individual states/transitions separately

    1. Model checking or property checking is a method for checking whether a finite-state model of a system meets a given specification (also known as correctness)
    2. In order to solve such a problem algorithmically, both the model of the system and its specification are formulated in some precise mathematical language. To this end, the problem is formulated as a task in logic, namely to check whether a structure satisfies a given logical formula.
    3. A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure
    4. Instead of enumerating reachable states one at a time, the state space can sometimes be traversed more efficiently by considering large numbers of states at a single step. When such state space traversal is based on representations of a set of states and transition relations as logical formulas, binary decision diagrams (BDD) or other related data structures, the model-checking method is symbolic.
    5. Model-checking tools face a combinatorial blow up of the state-space, commonly known as the state explosion problem, that must be addressed to solve most real-world problems
    6. Symbolic algorithms avoid explicitly constructing the graph for the finite state machines (FSM); instead, they represent the graph implicitly using a formula in quantified propositional logic

26. Formal verification: is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics

    1. Formal verification is effective at detecting complex bugs which are hard to detect manually or using simpler automated tools
    2. Formal verification needs a specification of the program being verified and techniques to translate/compare the specification with the actual implementation
    3. Certora's Prover and ChainSecurity's VerX are examples of formal verification tools for smart contracts. KEVM from Runtime Verification Inc is a formal verification framework that models EVM semantics.

27. **Manual analysis**: is complimentary to automated analysis using tools and serves a critical need in smart contract audits

    1. Automated analysis using tools is cheap (typically open-source free software), fast, deterministic and scalable (varies depending on the tool being semi-/fully-automated) but however is only as good as the properties it is made aware of, which is typically limited to `Solidity` and EVM related constraints
    2. Manual analysis with humans, in contrast, is expensive, slow, non-deterministic and not scalable because human expertise in smart contact security is a rare/expensive skill set today and we are slower, prone to error and inconsistent.
    3. Manual analysis is however the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found

28. **False Positives**: are findings which indicate the presence of vulnerabilities but which in fact are not vulnerabilities. Such false positives could be due to incorrect assumptions or simplifications in analysis which do not correctly consider all the factors required for the actual presence of vulnerabilities.

    1. False positives require further manual analysis on findings to investigate if they are indeed false or true positives
    2. High number of false positives increases manual effort in verification and lowers the confidence in the accuracy of the earlier automated/-manual analysis
    3. True positives might sometimes be classified as false positives which leads to vulnerabilities being exploited instead of being fixed

29. **False Negatives**: are missed findings that should have indicated the presence of vulnerabilities but which are in fact are not reported at all. Such false negatives could be due to incorrect assumptions or inaccuracies in analysis which do not correctly consider the minimum factors required for the actual presence of vulnerabilities.

    1. False negatives, per definition, are not reported or even realised unless a different analysis reveals their presence or the vulnerabilities are exploited
    2. High number of false negatives lowers the confidence in the effectiveness of the earlier manual/automated analysis.

30. Audit Firms (representative; not exhaustive): ABDK, Arcadia, Beosin, Blockchain Consilium, BlockSec, CertiK, ChainSafe, ChainSecurity, Chainsulting, CoinFabrik, ConsenSys Diligence, Dedaub, G0, Hacken, Haechi, Halborn, HashEx, Iosiro, Least Authority, MixBytes, NCC,

NewAlchemy, OpenZeppelin, PeckShield, Pessimistic, PepperSec, Pickle, Quantstamp, QuillHash, Runtime Verification, Sigma Prime, SlowMist, SmartDec, Solidified, Somish, Trail of Bits and Zokyo.

31. **Smart contract security tools**: are critical in assisting smart contract developers and auditors with showcasing (potentially) exploitable vulnerabilities, highlighting dangerous programming styles or surfacing common patterns of misuse. None of these however replace the need for manual review/validation to evaluate contract-specific business logic and other complex control-flow, data-flow & value-flow aspects.

32. **Categories of security tools**: tools for testing, test coverage, linting, disassembling, visualization, static analysis, dynamic analysis and formal verification of smart contracts.

33. Slither is a `Solidity` static analysis framework written in Python 3. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. It implements 74 detectors in the publicly available free version (with trophies that showcase Slither findings in real-world contracts).

34. **Slither features**:

    1. Detects vulnerable `Solidity` code with low false positives
    2. Identifies where the error condition occurs in the source code
    3. Easily integrates into continuous integration and Truffle builds
    4. Built-in 'printers' quickly report crucial contract information
    5. Detector API to write custom analyses in Python
    6. Ability to analyze contracts written with `Solidity` >= `0.4`
    7. Intermediate representation (SlithIR) enables simple, high-precision analyses
    8. Correctly parses 99.9% of all public `Solidity` code
    9. Average execution time of less than 1 second per contract

35. Slither bugs and optimizations detection can run on a Truffle/Embark/-Dapp/Etherlime/Hardhat application or on a single `Solidity` file:

    1. Slither runs all its detectors by default. To run only selected detectors, use `--detect detector1,detector2`. To exclude detectors, use `--exclude detector1,detector2`.
    2. To exclude detectors with an informational or low severity, use `--exclude-informational` or `--exclude-low`
    3. `--list-detectors` lists available detectors

36. Slither printers allow printing contract information with `--print` and following options (with contract-summary, human-summary, and inheritance-graph for quick review, and others such as call-graph, cfg, function-summary and vars-and-auth for in-depth review):
    1. `call-graph`: Export the call-graph of the contracts to a dot file
    2. `cfg`: Export the CFG of each functions
    3. `constructor-calls`: Print the constructors executed
    4. `contract-summary`: Print a summary of the contracts
    5. `data-dependency`: Print the data dependencies of the variables
    6. `echidna`: Export Echidna guiding information
    7. `evm`: Print the evm instructions of nodes in functions
    8. `function-id`: Print the keccack256 signature of the functions
    9. `function-summary`: Print a summary of the functions
    10. `human-summary`: Print a human-readable summary of the contracts
    11. `inheritance`: Print the inheritance relations between contracts
    12. `inheritance-graph`: Export the inheritance graph of each contract to a dot file
    13. `modifiers`: Print the modifiers called by each function
    14. `require`: Print the require and assert calls of each function
    15. `slithir`: Print the slithIR representation of the functions
    16. `slithir-ssa`: Print the slithIR representation of the functions
    17. `variable-order`: Print the storage order of the state variables
    18. `vars-and-auth`: Print the state variables written and the authorization of the functions
37. Slither upgradeability checks helps review contracts that use the delegatecall proxy pattern using `slither-check-upgradeability` tool with following options:
    1. `became-constant`: Variables that should not be constant
    2. `function-id-collision`: Functions ids collision
    3. `function-shadowing`: Functions shadowing
    4. `missing-calls`: Missing calls to init functions
    5. `missing-init-modifier`: `initializer()` is not called
    6. `multiple-calls`: Init functions called multiple times
    7. `order-vars-contracts`: Incorrect vars order with the v2
    8. `order-vars-proxy`: Incorrect vars order with the proxy
    9. `variables-initialized`: State variables with an initial value

10. `were-constant`: Variables that should be constant
11. `extra-vars-proxy`: Extra vars in the proxy
12. `missing-variables`: Variable missing in the v2
13. `extra-vars-v2`: Extra vars in the v2
14. `init-inherited`: Initializable is not inherited
15. `init-missing`: Initializable is missing
16. `initialize-target`: Initialize function that must be called
17. `initializer-missing`: `initializer()` is missing

38. Slither code similarity detector (a research-oriented tool) uses state-of-the-art machine learning to detect similar (vulnerable) `Solidity` functions

    1. It uses a pre-trained model from etherscan_verified_contracts with 60,000 contracts and more than 850,000 functions
    2. It uses FastText, a vector embedding technique, to generate compact numerical representations of every function
    3. It has four modes: (1) `test` - finds similar functions to your own in a dataset of contracts (2) `plot` - provide a visual representation of similarity of multiple sampled functions (3) `train` - builds new models of large datasets of contracts (4) `info` - inspects the internal information of the pre-trained model or the assessed code

39. Slither contract flattening tool `slither-flat` produces a flattened version of the codebase with the following features:

    1. Supports three strategies: 1) MostDerived: Export all the most derived contracts (every file is standalone) 2) OneFile: Export all the contracts in one standalone file 3) LocalImport: Export every contract in one separate file, and include `import ".."` in their preludes
    2. Supports circular dependency
    3. Supports all the compilation platforms (Truffle, embark, buidler, etherlime, ...).

40. Slither format tool `slither-format` generates automatically patches. The patches are compatible with `git`. Patches should be carefully reviewed before applying. Detectors supported with this tool are:

    1. `unused-state`
    2. `solc-version`
    3. `ragma`
    4. `naming-convention`
    5. `external-function`
    6. `constable-states`

7. `constant-function`

41. Slither ERC conformance tool `slither-check-erc` checks the following for `ERC`'s conformance for `ERC20`, `ERC721`, `ERC777`, `ERC165`, `ERC223` and `ERC1820`:

    1. All the functions are present
    2. All the events are present
    3. Functions return the correct type
    4. Functions that must be `view` are `view`
    5. Events' parameters are correctly indexed
    6. The functions emit the events
    7. Derived contracts do not break the conformance

42. Slither property generation tool `slither-prop` generates code properties (e.g., invariants) that can be tested with unit tests or Echidna, entirely automatically. The `ERC20` scenarios that can be tested are:

    1. `Transferable` - Test the correct tokens transfer
    2. `Pausable` - Test the pausable functionality
    3. `NotMintable` - Test that no one can mint tokens
    4. `NotMintableNotBurnable` - Test that no one can mint or burn tokens
    5. `NotBurnable` - Test that no one can burn tokens
    6. `Burnable` - Test the burn of tokens. Require the `burn(address) returns()` function

43. Slither new detectors: Slither's plugin architecture lets you integrate new detectors that run from the command line. The skeleton for a detector has:

    1. `ARGUMENT`: lets you run the detector from the command line
    2. `HELP`: is the information printed from the command line
    3. `IMPACT`: indicates the impact of the issue. Allowed values are `INFORMATIONAL|LOW|MEDIUM|HIGH`
    4. `CONFIDENCE`: indicates your confidence in the analysis. Allowed values are `LOW|MEDIUM|HIGH`
    5. `WIKI`: constants are used to generate automatically the documentation.
    6. `_detect()` is the function that implements the detector logic and needs to return a list of findings.

44. Manticore is a symbolic execution tool for analysis of Ethereum smart contracts (besides Linux binaries & WASM modules). See tutorial for details.

1. Program Exploration: Manticore can execute a program with symbolic inputs and explore all the possible states it can reach
2. Input Generation: Manticore can automatically produce concrete inputs that result in a given program state
3. Error Discovery: Manticore can detect crashes and other failure cases in binaries and smart contracts
4. Instrumentation: Manticore provides fine-grained control of state exploration via event callbacks and instruction hooks
5. Programmatic Interface: Manticore exposes programmatic access to its analysis engine via a Python API

45. Echidna is a `Haskell` program designed for fuzzing/property-based testing of Ethereum smart contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or `Solidity` assertions.

46. Echidna Features:

    1. Generates inputs tailored to your actual code
    2. Optional corpus collection, mutation and coverage guidance to find deeper bugs
    3. Powered by Slither to extract useful information before the fuzzing campaign
    4. Source code integration to identify which lines are covered after the fuzzing campaign
    5. Curses-based retro UI, text-only or JSON output
    6. Automatic test case minimization for quick triage
    7. Seamless integration into the development workflow
    8. Maximum gas usage reporting of the fuzzing campaign
    9. Support for a complex contract initialization with Etheno and Truffle

47. Echidna Usage (see tutorial for details):

    1. Executing the test runner: The core Echidna functionality is an executable called echidna-test. echidna-test takes a contract and a list of invariants (properties that should always remain true) as input. For each invariant, it generates random sequences of calls to the contract and checks if the invariant holds. If it can find some way to falsify the invariant, it prints the call sequence that does so. If it can't, you have some assurance the contract is safe.
    2. Writing invariants: Invariants are expressed as `Solidity` functions with names that begin with `echidna_`, have no arguments, and return a boolean.

3. Collecting and visualizing coverage: After finishing a campaign, Echidna can save a coverage maximizing corpus in a special directory specified with the corpusDir config option. This directory will contain two entries: (1) a directory named coverage with JSON files that can be replayed by Echidna and (2) a plain-text file named covered.txt, a copy of the source code with coverage annotations.

48. Eth-security-toolbox is a `Docker` container preinstalled and preconfigured with all of Trail of Bits' Ethereum security tools. This includes:

    1. Echidna property-based fuzz tester
    2. Etheno integration tool and differential tester
    3. Manticore symbolic analyzer and formal contract verifier
    4. Slither static analysis tool
    5. Rattle EVM lifter
    6. Not So Smart Contracts repository

49. Ethersplay is Binary Ninja plugin which enables an EVM disassembler and related analysis tools.

    1. Takes as input the evm bytecode in raw binary format
    2. Renders control flow graph of all functions
    3. Shows Manticore coverage

50. Pyevmasm is an assembler and disassembler library for the Ethereum Virtual Machine (EVM). It includes a command line utility and a `Python` API.

51. Rattle is an EVM binary static analysis framework designed to work on deployed smart contracts (not actively developed anymore).

    1. Takes EVM byte strings and uses a flow-sensitive analysis to recover the original control flow graph
    2. Lifts the control flow graph into an SSA/infinite register form, and optimizes the SSA – removing DUPs, SWAPs, PUSHs, and POPs
    3. The conversion from a stack machine to SSA form removes 60%+ of all EVM instructions and presents a much friendlier interface to those who wish to read the smart contracts they're interacting with

52. Evm_cfg_builder is a tool used to extract a control flow graph (CFG) from EVM bytecode and used by Ethersplay, Manticore, and other tools from Trail of Bits.

    1. Reliably recovers a Control Flow Graph (CFG) from EVM bytecode using a dedicated Value Set Analysis
    2. Recovers functions names

3. Recovers attributes (e.g., payable, view, pure)
4. Outputs the CFG to a dot file
5. Library API

53. Crytic-compile is a smart contract compilation library which is used in Trail of Bits' security tools and supports Truffle, Embark, Etherscan, Brownie, Waffle, Hardhat and other development environments. The plugin is used in Crytic tools, including:

    1. Slither
    2. Echidna
    3. Manticore
    4. evm-cfg-builder

54. Solc-select is a script to quickly switch between `Solidity` compiler versions.

    1. `solc-select`: manages installing and setting different `solc` compiler versions
    2. `solc`: wrapper around `solc` which picks the right version according to what was set via `solc-select`
    3. `solc` binaries are downloaded from this link which contains official artifacts for many historical and modern `solc` versions for Linux and macOS

55. Etheno is the Ethereum testing Swiss Army knife. It's a `JSON` RPC multiplexer, analysis tool wrapper, and test integration tool.

    1. `JSON` RPC Multiplexing: Etheno runs a JSON RPC server that can multiplex calls to one or more clients: 1) API for filtering and modifying `JSON` RPC calls 2) Enables differential testing by sending JSON RPC sequences to multiple Ethereum clients 3) Deploy to and interact with multiple networks at the same time
    2. Analysis Tool Wrapper: Etheno provides a `JSON` RPC client for advanced analysis tools like Manticore 1) Lowers barrier to entry for using advanced analysis tools 2) No need for custom scripts to set up account and contract state 3) Analyze arbitrary transactions without `Solidity` source code
    3. Integration with Test Frameworks like Ganache and Truffle: 1) Run a local test network with a single command 2) Use Truffle migrations to bootstrap Manticore analyses 3) Symbolic semantic annotations within unit tests

56. MythX is a powerful security analysis service that finds `Solidity` vulnerabilities in your Ethereum smart contract code during your development life

cycle. It is a paid API-based service which uses several tools on the backend including a static analyzer (Maru), symbolic analyzer (Mythril) and a greybox fuzzer (Harvey) to implement a total of 46 detectors. Mythril is the open-source component of MythX.

57. **MythX process**:
    1. Submit your code: The analysis requests are encrypted with TLS and the code you submit is accessed only by you. Submit both the source code and the compiled bytecode of your smart contracts for best results.
    2. Activate a full suite of analysis techniques: The longer MythX runs, the more it can detect more security weaknesses.
    3. Receive a detailed analysis report: MythX detects a majority of vulnerabilities listed in the SWC Registry. The report will return a listing of all the weaknesses found in your code, including the exact position of the issue and its SWC ID. Reports generated can be only accessed by you. MythX offers 3 scan modes, quick, standard and deep. You can see the differences here.
58. **MythX tools**: When you submit your code to the API it gets analyzed by multiple microservices in parallel where these tools cooperate to return the more comprehensive results in the execution time provided.
    1. A static analyzer that parses the `Solidity` AST
    2. a symbolic analyzer that detects possible vulnerable states, and
    3. a greybox fuzzer that detects vulnerable execution paths
59. **MythX coverage**: extends to most SWCs found in the SWC Registry with the 46 detectors listed here along with the type of analysis used.
60. MythX is based on a SaaS (Security as a Service) platform based on the premise that:
    1. Higher performance compared to running security tools locally
    2. Higher vulnerability coverage than any standalone tool
    3. Benefit from continuous improvements to our security analysis technology with new and improved security tests as the smart contract security landscape evolves.
61. MythX privacy guarantee for the smart contract code submitted using their SaaS APIs:
    1. Code analysis requests are encrypted with TLS
    2. To provide comprehensive reports and improve performance, it stores some of the contract data in our database, including parts of the source code and bytecode but that data never leaves their secure server and is not shared with any outside parties.

3. It keeps the results of your analysis so you can retrieve them later, but the report can be accessed by you only.

62. **MythX running time**: Quick scan runs for 5 minutes, Standard scan runs for 30 minutes, and Deep scan runs for 90 minutes.

63. MythX official integrations, tools and libraries include:

    1. **MythX CLI**: Unified tool to use MythX as a Command Line Interface (CLI) now with full Truffle Projects support.
    2. **MythX-JS**: Typescript library to integrate MythX in your JS or TS projects.
    3. **PythX**: Python library to integrate MythX in your Python projects.
    4. **MythX VSCode**: VSCode extension which allows you to scan smart-contracts and view results directly from your code editor.

64. **MythX pricing**:

    1. On Demand (US$9.99/3 scans): All scan modes and Prepaid scan packs
    2. Developer (US$49/mo): Quick and Standard scan modes; 500 scans/month
    3. Professional (US$249/mo): All scan modes; 10000 scans/month
    4. Enterprise (Custom pricing): Custom plans for your team's specific needs; Custom Verification Service; Retainer for Custom Support

65. Scribble is a verification language and runtime verification tool that translates high-level specifications into `Solidity` code. It allows you to annotate a `Solidity` smart contract with properties (see here).

    1. **Principles/Goals**:
        i. Specifications are easy to understand by developers and auditors.
        ii. Specifications are simple to reason about.
        iii. Specifications can be efficiently checked using off-the-shelf analysis tools.
        iv. A small number of core specification constructs are sufficient to express and reason about more advanced constructs
    2. Transforms annotations in the Scribble specification language into concrete assertions
    3. With these instrumented but equivalent contracts, one can then use Mythril, Harvey, MythX

66. Fuzzing-as-a-Service: is a service recently launched by ConsenSys Diligence where projects can submit their smart contracts along with embedded inlined specifications or properties written using the Scribble language.

These contracts are run through the Harvey fuzzer which uses the specified properties to optimize fuzzing campaigns. Any violations from fuzzing are reported back from the service for the project to fix.

67. Karl is a monitor for smart contracts that checks for security vulnerabilities using the Mythril detection engine. It can be used to monitor the Ethereum blockchain for newly deployed vulnerable smart contracts in real-time.

68. Theo is an exploitation tool with a Metasploit-like interface, drops you into a Python REPL console, where you can use the available features to do smart contract reconnaissance, check the storage, run exploits or frontrun or backrun transactions targeting a specific smart contract. Features:

    1. Automatic smart contract scanning which generates a list of possible exploits
    2. Sending transactions to exploit a smart contract
    3. Transaction pool monitor
    4. Web3 console
    5. Frontrunning and backrunning transactions
    6. Waiting for a list of transactions and sending out others
    7. Estimating gas for transactions means only successful transactions are sent
    8. Disabling gas estimation will send transactions with a fixed gas quantity.

69. Visual Auditor is a Visual Studio Code extension that provides security-aware syntax and semantic highlighting for Solidity and Vyper.

    1. **Syntax Highlighting**: access modifiers (`external`, `public`, `payable`, ...), security relevant built-ins, globals, methods and user/miner-tainted information, (`address.call()`, `tx.origin`, `msg.data`, `block.*`, `now`), storage access modifiers (`memory`, `storage`), developer notes in comments (`TODO`, `FIXME`, `HACK`, ...), custom function modifiers, contract creation/event invocations, easily differentiate between arithmetics vs. logical operations, make Constructor and Fallback function more prominent
    2. **Semantic Highlighting**: highlights StateVars (`constant`, `inherited`), detects and alerts about StateVar shadowing, highlights function arguments in the function body
    3. **Review Features**: audit annotations/bookmarks `@audit - <msg>`, `@audit - ok - <msg>` (see below), generic interface for importing external scanner results - cdili `JSON` format (see below), codelens inline action: graph, report, dependencies, inheritance, parse, ftrace, flatten, generate unit test stub, function signature hashes, uml.

4. **Graph and Reporting Features**: access your favorite Sūrya features from within vscode, interactive call graphs with call flow highlighting and more, auto-generate UML diagrams from code to support your threat modelling exercises or documentation
5. **Code Augmentation**: Hover over Ethereum Account addresses to download the byte-code, source-code or open it in the browser, Hover over ASM instructions to show their signatures, Hover over keywords to show basic Security Notes, Hover over StateVar's to show declaration information
6. **Views**: Cockpit vs Outline

70. Surya aids auditors in understanding and visualizing `Solidity` smart contracts by providing information about the contracts' structure and generates call graphs and inheritance graphs. It also supports querying the function call graph in multiple ways to aid in the manual inspection of contracts.

    1. Integrated with Visual Auditor
    2. Commands: `graph`, `ftrace`, `flatten`, `describe`, `inheritance`, `dependencies`, `parse` and `mdreport`.

71. SWC Registry: The Smart Contract Weakness Classification Registry (SWC Registry) is an implementation of the weakness classification scheme proposed in `EIP-1470`.

    1. It is loosely aligned to the terminologies and structure used in the Common Weakness Enumeration (CWE) while overlaying a wide range of weakness variants that are specific to smart contracts
    2. The goals of this project are as follows:
        i. Provide a straightforward way to classify security issues in smart contract systems.
        ii. Define a common language for describing security issues in smart contract systems' architecture, design, or code.
        iii. Serve as a way to train and increase performance for smart contract security analysis tools.
    3. This repository is maintained by the team behind MythX and currently contains 37 entries

72. Securify: is a security scanner for Ethereum smart contracts which Implements static analysis written in Datalog and supports 38 vulnerabilities

73. VerX: is a verifier that can automatically prove temporal safety properties of Ethereum smart contracts. The verifier is based on a careful combination of three ideas:

    1. Reduction of temporal safety verification to reachability checking.

2. An efficient symbolic execution engine used to compute precise symbolic states within a transaction.
3. Delayed abstraction which approximates symbolic states at the end of transactions into abstract states.

74. SmartCheck: is an extensible static analysis tool for discovering vulnerabilities and other code issues in Ethereum smart contracts written in the `Solidity` programming language. It translates `Solidity` source code into an XML-based intermediate representation and checks it against XPath patterns.

75. K-Framework based analysis, modelling and verification tools from Runtime Verification (RV): provides KEVM which is a model of EVM in the K-Framework. It is the first executable specification of the EVM that completely passes official test-suites and serves as a platform for building a wide range of analysis tools and other semantic extensions for EVM.

76. Certora Prover: checks that a smart contract satisfies a set of rules written in a language called Specify. Each rule is checked on all possible transactions, though of course this is not done by explicitly enumerating transactions, but rather through symbolic techniques.

    1. The Certora Prover provides complete path coverage for a set of safety rules provided by the user. For example, a rule might check that only a bounded number of tokens can be minted in an `ERC20` contract. The prover either guarantees that a rule holds on all paths and all inputs or produces a test input that demonstrates a violation of the rule.
    2. The problem addressed by the Certora Prover is known to be undecidable which means that there will always be pathological programs and rules for which the Certora prover will time out without a definitive answer.
    3. The Certora Prover takes as input the smart contract (either as EVM bytecode or `Solidity` source code) and a set of rules, written in Certora's specification language. The Prover then automatically determines whether or not the contract satisfies all the rules using a combination of two computer science techniques: abstract interpretation and constraint solving.

77. DappHub's Hevm: is an implementation of the EVM made specifically for unit testing and debugging smart contracts. It can run unit tests, property tests, interactively debug contracts while showing the `Solidity` source, or run arbitrary EVM code.

78. Capture the Flag (CTF): are fun and educational challenges where participants have to hack different (dummy) smart contracts that have vulnerabilities in them. They help understand the complexities around how vulnerabilities may be exploited in the wild. Popular ones include:

1. Capture The Ether: is a set of twenty challenges created by Steve Marx which test knowledge of Ethereum concepts of contracts, accounts and math among other things.
2. Ethernaut: is a Web3/`Solidity` based war game from OpenZeppelin that is played in the Ethereum Virtual Machine. Each level is a smart contract that needs to be 'hacked'. The game is 100% open source and all levels are contributions made by other players.
3. Damn Vulnerable DeFi v2: is a set of 12 DeFi related challenges created by tinchoabbate. Depending on the challenge, you should either stop the system from working, steal as much funds as you can, or do some other unexpected things.
4. Paradigm CFT: is a set of seventeen challenges created by samczsun at Paradigm.

79. Smart contract security tools are useful in assisting auditors while reviewing smart contracts. They automate many of the tasks that can be codified into rules with different levels of coverage, correctness and precision. They are fast, cheap, scalable and deterministic compared to manual analysis. But they are also susceptible to false positives. They are especially well-suited currently to detect common security pitfalls and best-practices at the `Solidity` and EVM level. With varying degrees of manual assistance, they can also be programmed to check for application-level, business-logic constraints.

80. Audit Process could be thought of as a ten-step process as follows:

    1. Read specification/documentation of the project to understand the requirements, design and architecture.
    2. Run fast automated tools such as linters or static analyzers to investigate common `Solidity` pitfalls or missing smart contract best-practices.
    3. Manual code analysis to understand business logic and detect vulnerabilities in it
    4. Run slower but more deeper automated tools such as symbolic checkers, fuzzers or formal verification analyzers which typically require formulation of properties/constraints beforehand, hand holding during the analyses and some post-run evaluation of their results.
    5. Discuss (with other auditors) the findings from above to identify any false positives or missing analyses.
    6. Convey status to project team for clarifying questions on business logic or threat model.
    7. Iterate the above for the duration of the audit leaving some time for report writing.

8. Write report summarizing the above with details on findings and recommendations.
9. Deliver the report to the project team and discuss findings, severity and potential fixes.
10. Evaluate fixes from the project team and verify that they indeed remove the vulnerabilities identified in findings.

81. **Reading specification/documentation**: For projects that have a specification of the design and architecture of their smart contracts, this is the recommended starting point. Very few new projects have a specification at the audit stage. Some of them have documentation in parts. Some key points:

    1. Specification starts with the project's technical and business goals and requirements. It describes how the project's design and architecture help achieve those goals.
    2. The actual implementation of smart contracts is a functional manifestation of the goals, requirements, specification, design and architecture, understanding of which is critical in evaluating if the implementation indeed meets the goals and requirements.
    3. Documentation is a description of what has been implemented based on the design and architectural requirements.
    4. Specification answers 'why' something needs to be designed/architected/implemented the way it has been done. Documentation answers 'how' something has been designed/architected/implemented without necessarily addressing the 'why' and leaves it up to the auditors to speculate on the reasons.
    5. Documentation is typically in the form of Readme files describing individual contract functionality combined with functional NatSpec and individual code comments. Encouraging projects to provide a detailed specification and documentation saves a lot of time and effort for the auditors in understanding the project goals/structure and prevents them from making the same assumptions as the implementation which is a leading cause of vulnerabilities.
    6. In the absence of both specification and documentation, auditors are forced to infer goals, requirements, design and architecture from reading code and using tools such as Surya and Slither printers. This takes up a lot of time leaving less time for deeper/complex security analyses.

82. **Running static analyzers**: Automated tools such as linters or static analyzers help investigate common `Solidity` pitfalls or missing smart contract best-practices

1. Tools such as Slither and MythX perform control-flow and data-flow analyses on the smart contracts in the context of their detectors which encode common security pitfalls and best-practices.
2. Evaluating their findings, which are usually available in seconds/minutes, is a good starting point to detect common vulnerabilities based on well-known constraints/properties of `Solidity` language, EVM or Ethereum blockchain.
3. False positives are possible among some of the detector findings and need to be verified manually if they are true/false positives.

83. **Manual code review**: is required to understand business logic and detect vulnerabilities in it.

    1. Automated analyzers do not understand application-level logic and their constraints. They are limited to constraints/properties of `Solidity` language, EVM or Ethereum blockchain.
    2. Manual analysis of the code is required to detect security-relevant deviations in implementation vis-a-vis the specification or documentation.
    3. Auditors may need to infer business logic and their implied constraints directly from the code or from discussions with the project team and thereafter evaluate if those constraints/properties hold in all parts of the codebase.

84. **Running deeper automated tools**: such as fuzzers e.g. Echidna, symbolic checkers such as Manticore, tool suite such as MythX and formally verifying custom properties with Scribble or Certora Prover takes more setup and preparation time but helps run deeper analyses to discover edge-cases in application-level properties and mathematical errors, among other things.

    1. Given these require understanding of the project's application logic, they are recommended to be used at least after an initial manual code review or sometimes after deeper discussion about the specification/implementation with the project team
    2. Analyzing the output of these tools requires significant expertise with the tools themselves, their domain-specific language and sometimes even their inner workings
    3. Evaluating false-positives is sometimes challenging with these tools but the true positives they discover are significant and extreme corner cases missed even by the best manual analyses

85. **Brainstorming with other auditors**: Linus's law: "*Given enough eyeballs, all bugs are shallow*" might apply with auditors too if they brainstorm on the smart contract implementation, assumptions, findings and vulnerabilities.

1. While some audit firms encourage active/passive discussion, there are others whose approach is to let auditors separately perform the assessment to encourage independent thinking instead of group thinking. The premise is that group thinking might bias the audit team to focus on certain aspects while missing some vulnerabilities.
2. A hybrid approach might be interesting where the audit team initially brainstorms to discuss the project's goals, specification/documentation and implementation but later firewall themselves to independently pursue the assessments and finally come together to compile their findings.

86. **Discussion with project team**: Having an open communication channel with the project team is useful to clarify any assumptions in specification, documentation, implementation, or discuss interim findings.

    1. Findings may also be shared with the project team immediately on a private repository to discuss impact, fixes and other implications.
    2. If the audit spans multiple weeks, it may help to have a weekly sync up call. A counterpoint to this is to independently perform the entire assessment so as to not get biased by the project team's inputs and opinions.

87. **Report writing**: The audit report is a final compilation of the entire assessment and presents all aspects of the audit including the audit scope/-coverage, timeline, team/effort, summaries, tools/techniques, findings, exploit scenarios, suggested fixes, short-/long-term recommendations and any appendices with further details on tools and rationale.

    1. An executive summary typically gives an overview of the audit report with highlights/lowlights illustrating the number/type/severity of vulnerabilities found and an overall assessment of risk. It may also include a description of the smart contracts, (inferred) actors, assets, roles, permissions, access control, interactions, threat model and existing risk mitigation measures.
    2. The bulk of the report focuses on the findings from the audit, their type/category, likelihood/impact, severity, justifications for these ratings, potential exploit scenarios, affected parts of smart contracts and potential remediations.
    3. It may also address subjective aspects of code quality, readability/auditability and other software-engineering best practices related to documentation, code structure, function/variable naming conventions, test coverage etc. that do not pose an imminent security risk but are indicators of anti-patterns and processes influencing the introduction and persistence of security vulnerabilities.

88. **Report delivery**: The delivery of the report to the project team is a critical deliverable and milestone. Unless interim findings/status is shared,

this will be the first time the project team will have access to the assessment details.

1. The delivery typically happens via a shared online document and is accompanied with a readout where the auditors present the report highlights to the project team for discussion and any debate on findings and their severity ratings.
2. The project team typically takes some time to review the audit report and respond back with any counterpoints on findings, severities or suggested fixes.
3. Depending on the prior agreement, the project team and audit firm might release the audit report publicly (after all required fixes have been made) or the project may decide to keep it private for some reason.

89. **Evaluating fixes**: Post audit, the project team may work on any required fixes for reported findings and request the audit firm for reviewing their responses.

    1. Fixes may be applied for a majority of the findings and the review may need to confirm that applied fixes (could be different from audit's recommended fixes) indeed mitigate the risk reported by the findings.
    2. Findings may be contested as not being relevant, outside the project's threat model or simply acknowledged as being within the project's acceptable risk model.
    3. Audit firms may evaluate the specific fixes applied and confirm/deny their risk mitigation. Unless it is a fix/retainer type audit, this phase typically takes not more than a day because it would usually be outside the agreed upon duration of the audit.

90. **Manual review approaches**: Auditors have different approaches to manual reviewing smart contract code for vulnerabilities.

    1. Starting with access control
    2. Starting with asset flow
    3. Starting with control flow
    4. Starting with data flow
    5. Inferring constraints
    6. Understanding dependencies
    7. Evaluating assumptions
    8. Evaluating security checklists

91. **Starting with access control**: Access control is the most fundamental security primitive which addresses 'who' has authorised access to 'what.'

(In a formal access control model, the 'who' refers to subjects, 'what' refers to objects and an access control matrix indicates the permissions between subjects and objects.)

1. While the overall philosophy might be that smart contracts are permissionless, in reality, they do indeed have different permissions/roles for different actors who interact/use them.
2. The general classification is that of users and admin(s). For purposes of guarded launch or otherwise, many smart contracts have an admin role that is typically the address that deployed the contract. Admins typically have control over critical configuration and application parameters including (emergency) transfers/withdrawals of contract funds.
3. Starting with understanding the access control implemented by the smart contracts and checking if they have applied correctly, completely and consistently is a good approach to understanding access flow and detecting violations.

92. **Starting with asset flow**: Assets are Ether or `ERC20`/`ERC721`/other tokens managed by smart contracts. Given that exploits target assets of value, it makes sense to start evaluating the flow of assets into/outside/within/across smart contracts and their dependencies.

    1. **Who**: Assets should be withdrawn/deposited only by authorised/specified addresses as per application logic.
    2. **When**: Assets should be withdrawn/deposited only in authorised/specified time windows or under authorised/specified conditions as per application logic (when)
    3. **Which**: Assets, only those authorised/specified types, should be withdrawn/deposited as per application logic
    4. **Why**: Assets should be withdrawn/deposited only for authorised/specified reasons as per application logic
    5. **Where**: Assets should be withdrawn/deposited only to authorised/specified addresses as per application logic
    6. **What type**: Assets, only of authorised/specified types, should be withdrawn/deposited as per application logic
    7. **How much**: Assets, only in authorised/specified amounts, should be withdrawn/deposited as per application logic

93. **Evaluating control flow**: Control flow analyzes the transfer of control, i.e. execution order, across and within smart contracts.

    1. Interprocedural (procedure is just another name for a function) control flow is typically indicated by a call graph which shows which functions (callers) call which other functions (callees), across or within smart contracts.

2. Intraprocedural (i.e. within a function) control flow is dictated by conditionals (`if`/`else`), loops (`for`/`while`/`do`/`continue`/`break`) and return statements.
3. Both intra and interprocedural control flow analysis help track the flow of execution and data in smart contracts

94. **Evaluating data flow**: Data flow analyzes the transfer of data across and within smart contracts

    1. Interprocedural data flow is evaluated by analyzing the data (variables/constants) used as argument values for function parameters at call sites
    2. Intraprocedural data flow is evaluated by analyzing the assignment and use of (`state`/`memory`/`calldata`) variables/constants along the control flow paths within functions.
    3. Both intra and interprocedural data flow analysis help track the flow of global/local storage/memory changes in smart contracts

95. **Inferring constraints**: Program constraints are basically rules that should be followed by the program. Language-level and EVM-level security constraints are well-known because they are part of the language and EVM specification. However, application-level constraints are rules that are implicit to the business logic implemented and may not be explicitly described in the specification e.g. mint an `ERC721` token to the address when it makes a certain deposit of `ERC20` tokens to the smart contract and burn it when it withdraws the earlier deposit. Such constraints may have to be inferred by the auditors while manually analyzing the smart contract code.

    1. One approach to inferring program constraints is to evaluate what is being done on most program paths related to a particular logic and treat it as a constraint. If such a constraint is missing on one or very few program paths then it could be an indicator of a vulnerability (assuming the constraint is security-related) or those program paths are exceptional conditions where the constraints do not need to hold.
    2. Program constraints can also be verified using a symbolic checker which generates counter-examples or witnesses along execution paths where such constraints do not hold.

96. **Understanding dependencies**: Dependencies exist when the correct compilation or functioning of program code relies on code/data from other smart contracts that were not necessarily developed by the project team.

    1. Explicit program dependencies are captured in the import statements and the inheritance hierarchy. For e.g., many projects use the community-developed, audited and time-tested smart contracts from OpenZeppelin for tokens, access control, proxy, security etc.

2. Composability is expected and encouraged via smart contracts interfacing with other protocols and vice-versa, which results in emergent or implicit dependencies on the state/logic of external smart contracts via oracles for example.
3. This is especially of interest/concern in DeFi protocols that rely on other related protocols for stablecoins, yield generation, borrowing/lending, derivatives, oracles etc.

97. **Evaluating assumptions**: Many security vulnerabilities result from faulty assumptions e.g. who can access what and when, under what conditions, for what reasons etc. Identifying the assumptions made by the program code and evaluating if they are indeed correct can be the source of many audit findings. Some common examples of faulty assumptions are:

    1. Only admins can call these functions.
    2. Initialization functions will only be called once by the contract deployer (e.g. for upgradeable contracts).
    3. Functions will always be called in a certain order (as expected by the specification).
    4. Parameters can only have non-zero values or values within a certain threshold e.g. addresses will never be zero valued.
    5. Certain addresses or data values can never be attacker controlled. They can never reach program locations where they can be misused. (In program analysis literature, this is known as taint analysis).
    6. Function calls will always be successful and so checking for return values is not required.

98. **Evaluating security checklists**: Checklists are lists of itemized points that can be quickly and methodically followed (and referenced later by their list number) to make sure all listed items have been processed according to the domain of relevance.

    1. This checklist-based approach was made popular in the book "*The Checklist Manifesto. How to Get Things Right*" by Atul Gawande who is a noted surgeon, writer and public health leader. In his review of this book, Malcolm Gladwell writes that: "*Gawande begins by making a distinction between errors of ignorance (mistakes we make because we don't know enough), and errors of ineptitude (mistakes we made because we don't make proper use of what we know). Failure in the modern world, he writes, is really about the second of these errors, and he walks us through a series of examples from medicine showing how the routine tasks of surgeons have now become so incredibly complicated that mistakes of one kind or another are virtually inevitable: it's just too easy for an otherwise competent doctor to miss a step,*

or forget to ask a key question or, in the stress and pressure of the moment, to fail to plan properly for every eventuality. Gawande then visits with pilots and the people who build skyscrapers and comes back with a solution. Experts need checklists–literally–written guides that walk them through the key steps in any complex procedure. In the last section of the book, Gawande shows how his research team has taken this idea, developed a safe surgery checklist, and applied it around the world, with staggering success".

2. Given the mind-boggling complexities of the fast-evolving Ethereum infrastructure (new platforms, new languages, new tools and new protocols) and the risks associated with deploying smart contracts managing millions of dollars, there are so many things to get right with smart contracts that it is easy to miss a few checks, make incorrect assumptions or fail to consider potential situations. Smart contract experts therefore need checklists too.
3. Smart contract security checklists (such as the articles in this series) help in navigating the vast number of key aspects to be remembered and applied. They help in going over the itemized features, concepts, pitfalls, best-practices and examples in a methodical manner without missing any items. Checklists are known to increase retention and have a faster recall. They also help in referencing specific items of interest e.g. #42 in Security Pitfalls & Best Practices 101 or #98 in Audit Techniques & Tools 101.

99. **Presenting proof-of-concept exploits**: Exploits are incidents where vulnerabilities are triggered by malicious actors to misuse smart contracts resulting, for example, in stolen/frozen assets
    1. Presenting proof-of-concepts of such exploits either in code or written descriptions of hypothetical scenarios make audit findings more realistic and relatable by illustrating specific exploit paths and justifying severity of findings.
    2. Codified exploits should always be on a testnet, kept private and responsibly disclosed to project teams without any risk of being actually executed on live systems resulting in real loss of funds or access.
    3. Descriptive exploit scenarios should make realistic assumptions on roles/powers of actors, practical reasons for their actions and sequencing of events that trigger vulnerabilities and illustrate the paths to exploitation.

100. **Estimating the likelihood and impact**: Likelihood indicates the probability of a vulnerability being discovered by malicious actors and triggered to successfully exploit the underlying weakness. Impact indicates the magnitude of implications on the technical and business aspects of the system if the vulnerability were to be exploited. Estimating if likelihood/impact are low/medium/high is non-trivial in many cases.

1. If the exploit can be triggered by a few transactions manually without requiring much resources/access (e.g. not admin) and without assuming many conditions to hold true then the likelihood is evaluated as High. Exploits that require deep knowledge of the system workings, privileged roles, large resources or multiple edge conditions to hold true are evaluated as Medium likelihood. Others that require even harder assumptions to hold true, miner collusion, chain forks or insider collusion for e.g., are considered as Low likelihood.
2. If there is any loss or locking up of funds then the impact is evaluated as High. Exploits that do not affect funds but disrupt the normal functioning of the system are typically evaluated as Medium. Anything else is of Low impact.
3. Many likelihood and impact evaluations are contentious and debatable between the audit and project teams, typically with security-conscious audit teams pressing for higher likelihood and impact and project teams downplaying the risks.

Estimating the severity: Severity, per OWASP, is a combination of likelihood and impact. With reasonable evaluations of those two, severity estimates from the OWASP matrix should be straightforward.

101. **Summary**: Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities but not their absence.

B.7 Audit Findings 101

(See https://secureum.substack.com/p/audit-findings-101 for the original article.)

1. **Unhandled return values of transfer and transferFrom**: `ERC20` implementations are not always consistent. Some implementations of `transfer` and `transferFrom` could return `false` on failure instead of reverting. It is safer to wrap such calls into `require()` statements to these failures.
    - **Recommendation**: Check the return value and revert on 0/false or use OpenZeppelin's `SafeERC20` wrapper functions
    - Medium severity finding from Consensys Diligence Audit of Aave Protocol V2.
2. **Random task execution**: In a scenario where a user takes a flash loan, `_parseFLAndExecute()` gives the flash loan wrapper contract (`FLAaveV2`, `FLDyDx`) the permission to execute functions on behalf of the user's `DSProxy` . This execution permission is revoked only after the entire recipe execution is finished, which means that in case that any of the external calls along the recipe execution is malicious, it might call `executeAction()` back, i.e. Reentrancy Attack, and inject any task it wishes (e.g. take user's funds out, drain approved tokens, etc)
    - **Recommendation**: A reentrancy guard (mutex) should be used to prevent such attack
    - Critical severity finding from Consensys Diligence Audit of Defi Saver.
3. **Tokens with more than 18 decimal points will cause issues**: It is assumed that the maximum number of decimals for each token is 18. However uncommon, it is possible to have tokens with more than 18 decimals, as an example YAMv2 has 24 decimals. This can result in broken code flow and unpredictable outcomes
    - **Recommendation**: Make sure the code won't fail in case the token's decimals is more than 18
    - Major severity finding from Consensys Diligence Audit of Defi Saver.
4. **Error codes of Compound's Comptroller.enterMarket, Comptroller.exitMarket are not checked**: Compound's `enterMarket` /`exitMarket` functions return an error code instead of reverting in case of failure. DeFi Saver smart contracts never check for the error codes returned from Compound smart contracts.

- **Recommendation**: Caller contract should revert in case the error code is not 0
- Major severity finding from Consensys Diligence Audit of Defi Saver.

5. **Reversed order of parameters in allowance function call**: the parameters that are used for the allowance function call are not in the same order that is used later in the call to `safeTransferFrom` .

   - **Recommendation**: Reverse the order of parameters in allowance function call to fit the order that is in the safeTransferFrom function call.
   - Medium severity finding from Consensys Diligence Audit of Defi Saver.

6. **Token approvals can be stolen in DAOfiV1Router01.addLiquidity()**: `DAOfiV1Router01.addLiquidity()` creates the desired pair contract if it does not already exist, then transfers tokens into the pair and calls `DAOfiV1Pair.deposit()` . There is no validation of the address to transfer tokens from, so an attacker could pass in any address with nonzero token approvals to `DAOfiV1Router` . This could be used to add liquidity to a pair contract for which the attacker is the `pairOwner` , allowing the stolen funds to be retrieved using DAOfiV1Pair.withdraw().

   - **Recommendation**: Transfer tokens from msg.sender instead of lp.sender
   - Critical severity finding from Consensys Diligence Audit of DAOfi.

7. **swapExactTokensForETH checks the wrong return value**: Instead of checking that the amount of tokens received from a swap is greater than the minimum amount expected from this swap, it calculates the difference between the initial receiver's balance and the balance of the router

   - **Recommendation**: Check the intended values
   - Major severity finding from Consensys Diligence Audit of DAOfi.

8. **DAOfiV1Pair.deposit() accepts deposits of zero, blocking the pool**: `DAOfiV1Pair.deposit()` is used to deposit liquidity into the pool. Only a single deposit can be made, so no liquidity can ever be added to a pool where deposited == true. The deposit() function does not check for a nonzero deposit amount in either token, so a malicious user that does not hold any of the `baseToken` or `quoteToken` can lock the pool by calling `deposit()` without first transferring any funds to the pool.

   - **Recommendation**: Require a minimum deposit amount with nonzero checks
   - Medium severity finding from Consensys Diligence Audit of DAOfi.

9. **GenesisGroup.commit overwrites previously-committed values**: The amount stored in the recipient's `committedFGEN` balance overwrites any previously-committed value. Additionally, this also allows anyone to commit an amount of "0" to any account, deleting their commitment entirely.

    - **Recommendation**: Ensure the committed amount is added to the existing commitment.
    - Critical severity finding from Consensys Diligence Audit of Fei Protocol.

10. **Purchasing and committing still possible after launch**: Even after `GenesisGroup.launch` has successfully been executed, it is still possible to invoke `GenesisGroup.purchase` and `GenesisGroup.commit` .

    - **Recommendation**: Consider adding validation in `GenesisGroup.purchase` and `GenesisGroup.commit` to make sure that these functions cannot be called after the launch.
    - Critical severity finding from Consensys Diligence Audit of Fei Protocol.

11. **UniswapIncentive overflow on pre-transfer hooks**: Before a token transfer is performed, Fei performs some combination of mint/burn operations via `UniswapIncentive.incentivize` . Both `incentivizeBuy` and `incentivizeSell` calculate buy/sell incentives using overflow-prone math, then mint / burn from the target according to the results. This may have unintended consequences, like allowing a caller to mint tokens before transferring them, or burn tokens from their recipient.

    - **Recommendation**: Ensure casts in `getBuyIncentive` and `getSellPenalty` do not overflow
    - Major severity finding from Consensys Diligence Audit of Fei Protocol.

12. **BondingCurve allows users to acquire FEI before launch**: allocate can be called before genesis launch, as long as the contract holds some nonzero PCV. By force-sending the contract 1 wei, anyone can bypass the majority of checks and actions in allocate, and mint themselves FEI each time the timer expires.

    - **Recommendation**: Prevent allocate from being called before genesis launch
    - Medium severity finding from Consensys Diligence Audit of Fei Protocol.

13. **Timed.isTimeEnded returns true if the timer has not been initialized**: Timed initialization is a 2-step process:

1. `Timed.duration` is set in the constructor
2. `Timed.startTime` is set when the method `_initTimed` is called. Before this second method is called, `isTimeEnded` () calculates remaining time using a `startTime` of 0, resulting in the method returning true for most values, even though the timer has not technically been started.

- **Recommendation**: If Timed has not been initialized, `isTimeEnded` () should return `false`, or revert
- Medium severity finding from Consensys Diligence Audit of Fei Protocol.

14. **Overflow/underflow protection**: Having overflow/underflow vulnerabilities is very common for smart contracts. It is usually mitigated by using SafeMath or using `Solidity` version `^0.8.z` (after `Solidity 0.8.z` arithmetical operations already have default overflow/underflow protection). In this code, many arithmetical operations are used without the 'safe' version. The reasoning behind it is that all the values are derived from the actual ETH values, so they can't overflow.

- **Recommendation**: In our opinion, it is still safer to have these operations in a safe mode. So we recommend using SafeMath or `Solidity` version `^0.8` compiler.
- Medium severity finding from Consensys Diligence Audit of Fei Protocol.

15. **Unchecked return value for IWETH.transfer call**: In `EthUniswapPCVController`, there is a call to `IWETH.transfer` that does not check the return value. It is usually good to add a require-statement that checks the return value or to use something like `safeTransfer`; unless one is sure the given token reverts in case of a failure.

- **Recommendation**: Consider adding a require-statement or using `safeTransfer`.
- Medium severity finding from Consensys Diligence Audit of Fei Protocol.

16. **GenesisGroup.emergencyExit remains functional after launch**: `emergencyExit` is intended as an escape mechanism for users in the event the genesis launch method fails or is frozen. `emergencyExit` becomes callable 3 days after launch is callable. These two methods are intended to be mutually-exclusive, but are not: either method remains callable after a successful call to the other. This may result in accounting edge cases.

- **Recommendation**:
    1. Ensure launch cannot be called if `emergencyExit` has been called

2. Ensure `emergencyExit` cannot be called if launch has been called

- Medium severity finding from Consensys Diligence Audit of Fei Protocol.

17. **ERC20 tokens with no return value will fail to transfer**: Although the `ERC20` standard suggests that a transfer should return true on success, many tokens are non-compliant in this regard. In that case, the .transfer() call here will revert even if the transfer is successful, because solidity will check that the `RETURNDATASIZE` matches the `ERC20` interface.

    - **Recommendation**: Consider using OpenZeppelin's Safe`ERC20`
    - Major severity finding from Consensys Diligence Audit of bitbank.

18. **Reentrancy vulnerability in MetaSwap.swap()**: If an attacker is able to reenter `swap()`, they can execute their own trade using the same tokens and get all the tokens for themselves.

    - **Recommendation**: Use a simple reentrancy guard, such as OpenZeppelin's `ReentrancyGuard` to prevent reentrancy in `MetaSwap.swap()`
    - Major severity finding from Consensys Diligence Audit of MetaSwap.

19. **A new malicious adapter can access users' tokens**: The purpose of the `MetaSwap` contract is to save users gas costs when dealing with a number of different aggregators. They can just approve() their tokens to be spent by `MetaSwap` (or in a later architecture, the Spender contract). They can then perform trades with all supported aggregators without having to reapprove anything. A downside to this design is that a malicious (or buggy) adapter has access to a large collection of valuable assets. Even a user who has diligently checked all existing adapter code before interacting with `MetaSwap` runs the risk of having their funds intercepted by a new malicious adapter that's added later.

    - **Recommendation**: Make `MetaSwap` contract the only contract that receives token approval. It then moves tokens to the Spender contract before that contract `DELEGATECALLs` to the appropriate adapter. In this model, newly added adapters shouldn't be able to access users' funds.
    - Medium severity finding from Consensys Diligence Audit of MetaSwap.

20. **Owner can front-run traders by updating adapters**: `MetaSwap` owners can front-run users to swap an adapter implementation. This could be used by a malicious or compromised owner to steal from users. Because adapters are `DELEGATECALL`'ed, they can modify storage. This means any adapter can overwrite the logic of another adapter, regardless of what policies are put in place at the contract level. Users must fully trust every

adapter because just one malicious adapter could change the logic of all other adapters.

- **Recommendation**: At a minimum, disallow modification of existing adapters. Instead, simply add new adapters and disable the old ones.
- Medium severity finding from Consensys Diligence Audit of MetaSwap.

21. **Users can collect interest from SavingsContract by only staking mTokens momentarily**: The `SAVE` contract allows users to deposit `mAssets` in return for lending yield and swap fees. When depositing `mAsset`, users receive a "credit" tokens at the momentary credit/`mAsset` exchange rate which is updated at every deposit. However, the smart contract enforces a minimum timeframe of 30 minutes in which the interest rate will not be updated. A user who deposits shortly before the end of the timeframe will receive credits at the stale interest rate and can immediately trigger an update of the rate and withdraw at the updated (more favorable) rate after the 30 minutes window. As a result, it would be possible for users to benefit from interest payouts by only staking mAssets momentarily and using them for other purposes the rest of the time.

    - **Recommendation**: Remove the 30 minutes window such that every deposit also updates the exchange rate between credits and tokens.
    - Medium severity finding from Consensys Diligence Audit of mstable-1.1.

22. **Oracle updates can be manipulated to perform atomic front-running attack**: It is possible to atomically arbitrage rate changes in a risk-free way by "sandwiching" the Oracle update between two transactions. The attacker would send the following 2 transactions at the moment the Oracle update appears in the mempool:

    The first transaction, which is sent with a higher gas price than the Oracle update transaction, converts a very small amount. This "locks in" the conversion weights for the block since `handleExternalRateChange()` only updates weights once per block. By doing this, the arbitrageur ensures that the stale Oracle price is initially used when doing the first conversion in the following transaction. The second transaction, which is sent at a slightly lower gas price than the transaction that updates the Oracle, performs a large conversion at the old weight, adds a small amount of Liquidity to trigger rebalancing and converts back at the new rate. The attacker can obtain liquidity for step 2 using a flash loan. The attack will deplete the reserves of the pool.

    - **Recommendation**: Do not allow users to trade at a stale Oracle rate and trigger an Oracle price update in the same transaction.

- Critical severity finding from Consensys Diligence Audit of Bancor v2 AMM.

23. **Certain functions lack input validation routines**: The functions should first check if the passed arguments are valid first. These checks should include, but not be limited to:

    1. `uint` should be larger than 0 when 0 is considered invalid
    2. `uint` should be within constraints
    3. `int` should be positive in some cases
    4. length of arrays should match if more arrays are sent as arguments
    5. addresses should not be `0x0`

    - **Recommendation**: Add tests that check if all of the arguments have been validated. Consider checking arguments as an important part of writing code and developing the system.
    - Major severity finding from Consensys Diligence Audit of Shell Protocol.

24. **Remove Loihi methods that can be used as backdoors by the administrator**: There are several functions in `Loihi` that give extreme powers to the shell administrator. The most dangerous set of those is the ones granting the capability to add assimilators. Since assimilators are essentially a proxy architecture to delegate code to several different implementations of the same interface, the administrator could, intentionally or unintentionally, deploy malicious or faulty code in the implementation of an assimilator. This means that the administrator is essentially totally trusted to not run code that, for example, drains the whole pool or locks up the users' and LPs' tokens. In addition to these, the function `safeApprove` allows the administrator to move any of the tokens the contract holds to any address regardless of the balances any of the users have. This can also be used by the owner as a backdoor to completely drain the contract.

    - **Recommendation**: Remove the `safeApprove` function and, instead, use a trustless escape-hatch mechanism. For the assimilator addition functions, our recommendation is that they are made completely internal, only callable in the constructor, at deploy time. Even though this is not a big structural change (in fact, it reduces the attack surface), it is, indeed, a feature loss. However, this is the only way to make each shell a time-invariant system. This would not only increase Shell's security but also would greatly improve the trust the users have in the protocol since, after deployment, the code is now static and auditable.
    - Major severity finding from Consensys Diligence Audit of Shell Protocol.

25. **A reverting fallback function will lock up all payouts**: In `BoxExchange.sol`, the internal function `_transferEth()` reverts if the transfer does not succeed. The `_payment()` function processes a list of transfers to settle the transactions in an `ExchangeBox`. If any of the recipients of an ETH transfer is a smart contract that reverts, then the entire payout will fail and will be unrecoverable.

    - **Recommendation**:
        1. Implement a queuing mechanism to allow buyers/sellers to initiate the withdrawal on their own using a pull-over-push pattern.
        2. Ignore a failed transfer and leave the responsibility up to users to receive them properly.
    - Critical severity finding from Consensys Diligence Audit of Lien Protocol.

26. **Saferagequit makes you lose funds**: `safeRagequit` and `ragequit` functions are used for withdrawing funds from the LAO. The difference between them is that ragequit function tries to withdraw all the allowed tokens and `safeRagequit` function withdraws only some subset of these tokens, defined by the user. It's needed in case the user or `GuildBank` is blacklisted in some of the tokens and the transfer reverts. The problem is that even though you can quit in that case, you'll lose the tokens that you exclude from the list. To be precise, the tokens are not completely lost, they will belong to the LAO and can still potentially be transferred to the user who quit. But that requires a lot of trust, coordination, time and anyone can steal some part of these tokens.

    - **Recommendation**: Implementing pull pattern for token withdrawals should solve the issue. Users will be able to quit the LAO and burn their shares but still keep their tokens in the LAO's contract for some time if they can't withdraw them right now.
    - Critical severity finding from Consensys Diligence Audit of The Lao.

27. **Creating proposal is not trustless**: Usually, if someone submits a proposal and transfers some amount of tribute tokens, these tokens are transferred back if the proposal is rejected. But if the proposal is not processed before the emergency processing, these tokens will not be transferred back to the proposer. This might happen if a tribute token or a deposit token transfers are blocked. Tokens are not completely lost in that case, they now belong to the LAO shareholders and they might try to return that money back. But that requires a lot of coordination and time and everyone who ragequits during that time will take a part of that tokens with them.

    - **Recommendation**: Pull pattern for token transfers would solve the issue

- Critical severity finding from Consensys Diligence Audit of The Lao.

28. **Emergency processing can be blocked**: The main reason for the emergency processing mechanism is that there is a chance that some token transfers might be blocked. For example, a sender or a receiver is in the USDC blacklist. Emergency processing saves from this problem by not transferring tribute token back to the user (if there is some) and rejecting the proposal. The problem is that there is still a deposit transfer back to the sponsor and it could be potentially blocked too. If that happens, proposal can't be processed and the LAO is blocked.

    - **Recommendation**: Pull pattern for token transfers would solve the issue
    - Critical severity finding from Consensys Diligence Audit of The Lao.

29. **Token Overflow might result in system halt or loss of funds**: If a token overflows, some functionality such as `processProposal`, `cancelProposal` will break due to `SafeMath` reverts. The overflow could happen because the supply of the token was artificially inflated to oblivion.

    - **Recommendation**: We recommend to allow overflow for broken or malicious tokens. This is to prevent system halt or loss of funds. It should be noted that in case an overflow occurs, the balance of the token will be incorrect for all token holders in the system
    - Major severity finding from Consensys Diligence Audit of The Lao.

30. **Whitelisted tokens limit**: `_ragequit` function is iterating over all whitelisted tokens. If the number of tokens is too big, a transaction can run out of gas and all funds will be blocked forever.

    - **Recommendation**: A simple solution would be just limiting the number of whitelisted tokens. If the intention is to invest in many new tokens over time, and it's not an option to limit the number of whitelisted tokens, it's possible to add a function that removes tokens from the whitelist. For example, it's possible to add a new type of proposal that is used to vote on token removal if the balance of this token is zero. Before voting for that, shareholders should sell all the balance of that token.
    - Major severity finding from Consensys Diligence Audit of The Lao.

31. **Summoner can steal funds using bailout**: The `bailout` function allows anyone to transfer kicked user's funds to the summoner if the user does not call safeRagequit (which forces the user to lose some funds). The intention is for the summoner to transfer these funds to the kicked member afterwards. The issue here is that it requires a lot of trust to the summoner on the one hand, and requires more time to kick the member out of the LAO.

- **Recommendation**: By implementing pull pattern for token transfers, kicked member won't be able to block the ragekick and the LAO members would be able to kick anyone much quicker. There is no need to keep the `bailout` function.
- Major severity finding from Consensys Diligence Audit of The Lao.

32. **Sponsorship front-running**: If proposal submission and sponsorship are done in 2 different transactions, it's possible to front-run the `sponsorProposal` function by any member. The incentive to do that is to be able to block the proposal afterwards.

    - **Recommendation**: Pull pattern for token transfers will solve the issue. Front-running will still be possible but it doesn't affect anything.
    - Major severity finding from Consensys Diligence Audit of The Lao.

33. **Delegate assignment front-running**: Any member can front-run another member's `delegateKey` assignment. If you try to submit an address as your `delegateKey`, someone else can try to assign your delegate address to themselves. While incentive of this action is unclear, it's possible to block some address from being a delegate forever.

    - **Recommendation**: Make it possible for a `delegateKey` to approve `delegateKey` assignment or cancel the current delegation. Commit-reveal methods can also be used to mitigate this attack.
    - Medium severity finding from Consensys Diligence Audit of The Lao.

34. **Queued transactions cannot be canceled**: The Governor contract contains special functions to set it as the admin of the `Timelock`. Only the admin can call `Timelock.cancelTransaction`. There are no functions in Governor that call `Timelock.cancelTransaction`. This makes it impossible for `Timelock.cancelTransaction` to ever be called.

    - **Recommendation**: Short term, add a function to the Governor that calls `Timelock.cancelTransaction`. It is unclear who should be able to call it, and what other restrictions there should be around cancelling a transaction. Long term, consider letting Governor inherit from `Timelock`. This would allow a lot of functions and code to be removed and significantly lower the complexity of these two contracts.
    - High Risk severity finding from ToB's Audit of Origin Dollar.

35. **Proposal transactions can be executed separately and block Proposal.execute call**: Missing access controls in the `Timelock.executeTransaction` function allow Proposal transactions to be executed separately, circumventing the `Governor.execute` function.

    - **Recommendation**: Short term, only allow the admin to call `Timelock.executeTransaction`

- High Risk severity finding from ToB's Audit of Origin Dollar.

36. **Proposals could allow Timelock.admin takeover**: The Governor contract contains special functions to let the guardian queue a transaction to change the `Timelock.admin`. However, a regular Proposal is also allowed to contain a transaction to change the `Timelock.admin`. This poses an unnecessary risk in that an attacker could create a Proposal to change the `Timelock.admin`.

    - **Recommendation**: Short term, add a check that prevents `setPendingAdmin` to be included in a Proposal
    - High Risk severity finding from ToB's Audit of Origin Dollar.

37. **Reentrancy and untrusted contract call in mintMultiple**: Missing checks and no reentrancy prevention allow untrusted contracts to be called from `mintMultiple`. This could be used by an attacker to drain the contracts.

    - **Recommendation**: Short term, add checks that cause `mintMultiple` to revert if the amount is zero or the asset is not supported. Add a reentrancy guard to the `mint`, `mintMultiple`, `redeem`, and `redeemAll` functions. Long term, make use of Slither which will flag the reentrancy. Or even better, use Crytic and incorporate static analysis checks into your CI/CD pipeline. Add reentrancy guards to all non-view functions callable by anyone. Make sure to always revert a transaction if an input is incorrect. Disallow calling untrusted contracts.
    - High Risk severity finding from ToB's Audit of Origin Dollar.

38. **Lack of return value checks can lead to unexpected results**: Several function calls do not check the return value. Without a return value check, the code is error-prone, which may lead to unexpected results.

    - **Recommendation**: Short term, check the return value of all calls mentioned above. Long term, subscribe to Crytic.io to catch missing return checks. Crytic identifies this bug type automatically.
    - High Risk severity finding from ToB's Audit of Origin Dollar.

39. **External calls in loop can lead to denial of service**: Several function calls are made in unbounded loops. This pattern is error-prone as it can trap the contracts due to the gas limitations or failed transactions.

    - **Recommendation**: Short term, review all the loops mentioned above and either:
        1. allow iteration over part of the loop, or
        2. remove elements.

Long term, subscribe to Crytic.io to review external calls in loops. Crytic catches bugs of this type.

- High Risk severity finding from ToB's Audit of Origin Dollar.

40. **OUSD allows users to transfer more tokens than expected**: Under certain circumstances, the OUSD contract allows users to transfer more tokens than the ones they have in their balance. This issue seems to be caused by a rounding issue when the `creditsDeducted` is calculated and subtracted.

    - **Recommendation**: Short term, make sure the balance is correctly checked before performing all the arithmetic operations. This will make sure it does not allow to transfer more than expected. Long term, use Echidna to write properties that ensure `ERC20` transfers are transferring the expected amount.
    - High Risk severity finding from ToB's Audit of Origin Dollar.

41. **OUSD total supply can be arbitrary, even smaller than user balances**: The OUSD token contract allows users to opt out of rebasing effects. At that point, their exchange rate is "fixed", and further rebases will not have an impact on token balances (until the user opts in).

    - **Recommendation**: Short term, we would advise making clear all common invariant violations for users and other stakeholders. Long term, we would recommend designing the system in such a way to preserve as many commonplace invariants as possible.
    - High Risk severity finding from ToB's Audit of Origin Dollar.

42. **Flash minting can be used to redeem fyDAI**: The flash-minting feature from the `fyDAI` token can be used to redeem an arbitrary amount of funds from a mature token.

    - **Recommendation**: Short term, disallow calls to redeem in the `YDai` and Unwind contracts during flash minting. Long term, do not include operations that allow any user to manipulate an arbitrary amount of funds, even if it is in a single transaction. This will prevent attackers from gaining leverage to manipulate the market and break internal invariants.
    - Medium Risk severity finding from ToB's Audit of Yield Protocol.

43. **Lack of chainID validation allows signatures to be re-used across forks**: `YDai` implements the draft ERC 2612 via the `ERC20Permit` contract it inherits from. This allows a third party to transmit a signature from a token holder that modifies the `ERC20` allowance for a particular user. These signatures used in calls to permit in `ERC20Permit` do not account for chain splits. The `chainID` is included in the domain separator. However, it is not updatable and not included in the signed data as part of the permit

call. As a result, if the chain forks after deployment, the signed message may be considered valid on both forks.

- **Recommendation**: Short term, include the `chainID` opcode in the permit schema. This will make replay attacks impossible in the event of a post-deployment hard fork. Long term, document and carefully review any signature schemas, including their robustness to replay on different wallets, contracts, and blockchains. Make sure users are aware of signing best practices and the danger of signing messages from untrusted sources.
- High Risk severity finding from ToB's Audit of Yield Protocol.

44. **Lack of a contract existence check allows token theft**: Since there's no existence check for contracts that interact with external tokens, an attacker can steal funds by registering a token that's not yet deployed. `_safeTransferFrom` will return success even if the token is not yet deployed, or was self-destructed. An attacker that knows the address of a future token can register the token in Hermez, and deposit any amount prior to the token deployment. Once the contract is deployed and tokens have been deposited in Hermez, the attacker can steal the funds. The address of a contract to be deployed can be determined by knowing the address of its deployer.

    - **Recommendation**: Short term, check for contract existence in `_safeTransferFrom`. Add a similar check for any low-level calls, including in `WithdrawalDelayer`. This will prevent an attacker from listing and depositing tokens in a contract that is not yet deployed. Long term, carefully review the Solidity documentation, especially the Warnings section. The Solidity documentation warns: The low-level call, `delegatecall` and `callcode` will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.
    - High Risk severity finding from ToB's Audit of Hermez.

45. **No incentive for bidders to vote earlier**: Hermez relies on a voting system that allows anyone to vote with any weight at the last minute. As a result, anyone with a large fund can manipulate the vote. Hermez's voting mechanism relies on bidding. There is no incentive for users to bid tokens well before the voting ends. Users can bid a large amount of tokens just before voting ends, and anyone with a large fund can decide the outcome of the vote. As all the votes are public, users bidding earlier will be penalized, because their bids will be known by the other participants. An attacker can know exactly how much currency will be necessary to change the outcome of the voting just before it ends.

    - **Recommendation**: Short term, explore ways to incentivize users to vote earlier. Consider a weighted bid, with a weight decreasing over

time. While it won't prevent users with unlimited resources from manipulating the vote at the last minute, it will make the attack more expensive and reduce the chance of vote manipulation. Long term, stay up to date with the latest research on blockchain-based online voting and bidding. Blockchain-based online voting is a known challenge. No perfect solution has been found yet.

- Medium Risk severity finding from ToB's Audit of Hermez.

46. **Lack of access control separation is risky**: The system uses the same account to change both frequently updated parameters and those that require less frequent updates. This architecture is error-prone and increases the severity of any privileged account compromises.

    - **Recommendation**: Short term, use a separate account to handle updating the tokens/USD ratio. Using the same account for the critical operations and update the tokens/USD ratio increases underlying risks. Long term, document the access controls and set up a proper authorization architecture. Consider the risks associated with each access point and their frequency of usage to evaluate the proper design.
    - High Risk severity finding from ToB's Audit of Hermez.

47. **Lack of two-step procedure for critical operations leaves them error-prone**: Several critical operations are done in one function call. This schema is error-prone and can lead to irrevocable mistakes. For example, the setter for the whitehack group address sets the address to the provided argument. If the address is incorrect, the new address will take on the functionality of the new role immediately. However, a two-step process is similar to the approve-transferFrom functionality: The contract approves the new address for a new role, and the new address acquires the role by calling the contract.

    - **Recommendation**: Short term, use a two-step procedure for all non-recoverable critical operations to prevent irrecoverable mistakes. Long term, identify and document all possible actions and their associated risks for privileged accounts. Identifying the risks will assist codebase review and prevent future mistakes.
    - High Risk severity finding from ToB's Audit of Hermez.

48. **Initialization functions can be front-run**: `Hermez`, `HermezAuctionProtocol` and `WithdrawalDelayer` have initialization functions that can be front-run, allowing an attacker to incorrectly initialize the contracts. Due to the use of the `delegatecall` proxy pattern, `Hermez`, `HermezAuctionProtocol`, and `WithdrawalDelayer` cannot be initialized with a constructor, and have initializer functions. All these functions can be front-run by an attacker, allowing them to initialize the contracts with malicious values.

- **Recommendation**: Short term, either:
  1. Use a factory pattern that will prevent front-running of the initialization, or
  2. ensure the deployment scripts are robust in case of a front-running attack.

  Carefully review the `Solidity` documentation, especially the Warnings section. Carefully review the pitfalls of using `delegatecall` proxy pattern.
- High Risk severity finding from ToB's Audit of Hermez.

49. **Missing validation of _owner argument could indefinitely lock owner role**: A lack of input validation of the `_owner` argument in both the constructor and `setOwner` functions could permanently lock the owner role, requiring a costly redeploy. To resolve an incorrect owner issue, Uniswap would need to redeploy the factory contract and re-add pairs and liquidity. Users might not be happy to learn of these actions, which could lead to reputational damage. Certain users could also decide to continue using the original factory and pair contracts, in which owner functions cannot be called. This could lead to the concurrent use of two versions of Uniswap, one with the original factory contract and no valid owner and another in which the owner was set correctly. Trail of Bits identified four distinct cases in which an incorrect owner is set:

    1. Passing `address(0)` to the constructor
    2. Passing `address(0)` to the `setOwner` function
    3. Passing an incorrect address to the constructor
    4. Passing an incorrect address to the `setOwner` function.

    - **Recommendation**: Several improvements could prevent the four above mentioned cases:
      1. Designate `msg.sender` as the initial owner, and transfer ownership to the chosen owner after deployment.
      2. Implement a two-step ownership-change process through which the new owner needs to accept ownershop.
      3. If it needs to be possible to set the owner to `address(0)`, implement a `renounceOwnership` function.
    - Medium Risk severity finding from ToB's Audit of Uniswap V3.

50. **Incorrect comparison enables swapping and token draining at no cost**: An incorrect comparison in the swap function allows the swap to succeed even if no tokens are paid. This issue could be used to drain any pool of all of its tokens at no cost. The swap function calculates how many tokens the initiator (`msg.sender`) needs to pay (`amountIn`) to receive the requested amount of tokens (`amountOut`). It then calls

the `uniswapV3SwapCallback` function on the initiator's account, passing in the amount of tokens to be paid. The callback function should then transfer at least the requested amount of tokens to the pool contract. Afterward, a require inside the swap function verifies that the correct amount of tokens (amountIn) has been transferred to the pool. However, the check inside the require is incorrect. The operand used is `>` instead of `<=`.

- **Recommendation**: Replace `>=` with `<=` in the require statement.
- High Risk severity finding from ToB's Audit of Uniswap V3.

51. **Unbound loop enables denial of service**: The swap function relies on an unbounded loop. An attacker could disrupt swap operations by forcing the loop to go through too many operations, potentially trapping the swap due to a lack of gas.

    - **Recommendation**: Bound the loops and document the bounds.
    - Medium Risk severity finding from ToB's Audit of Uniswap V3.

52. **Front-running pool's initialization can lead to draining of liquidity provider's initial deposits**: A front-run on `UniswapV3Pool.initialize` allows an attacker to set an unfair price and to drain assets from the first deposits. There are no access controls on the initialize function, so anyone could call it on a deployed pool. Initializing a pool with an incorrect price allows an attacker to generate profits from the initial liquidity provider's deposits.

    - **Recommendation**:
        1. moving the price operations from initialize to the constructor,
        2. adding access controls to initialize, or
        3. ensuring that the documentation clearly warns users about incorrect initialization
    - Medium Risk severity finding from ToB's Audit of Uniswap V3.

53. **Swapping on zero liquidity allows for control of the pool's price**: Swapping on a tick with zero liquidity enables a user to adjust the price of 1 wei of tokens in any direction. As a result, an attacker could set an arbitrary price at the pool's initialization or if the liquidity providers withdraw all of the liquidity for a short time.

    - **Recommendation**: No straightforward way to prevent the issue. Ensure pools don't end up in unexpected states. Warn users of potential risks.
    - Medium Risk severity finding from ToB's Audit of Uniswap V3.

54. **Failed transfer may be overlooked due to lack of contract existence check**: Because the pool fails to check that a contract exists, the pool may assume that failed transactions involving destructed tokens are successful. `TransferHelper.safeTransfer` performs a transfer with a low-level call without confirming the contract's existence. As a result, if the tokens have not yet been deployed or have been destroyed, safeTransfer will return success even though no transfer was executed.

    - **Recommendation**: Short term, check the contract's existence prior to the low-level call in `TransferHelper.safeTransfer`. Long term, avoid low-level calls.
    - High Risk severity finding from ToB's Audit of Uniswap V3.

55. **Use of undefined behavior in equality check**: On the left-hand side of the equality check, there is an assignment of the variable `outputAmt_`. The right-hand side uses the same variable. The Solidity 0.7.3. documentation states that "The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done" which means that this check constitutes an instance of undefined behavior. As such, the behavior of this code is not specified and could change in a future release of Solidity.

    - **Recommendation**: Short term, rewrite the if statement such that it does not use and assign the same variable in an equality check. Long term, ensure that the codebase does not contain undefined Solidity or EVM behavior.
    - High Risk severity finding from ToB's Audit of DFX Finance.

56. **Assimilators' balance functions return raw values**: The system converts raw values to numeraire values for its internal arithmetic. However, in one instance it uses raw values alongside numeraire values. Interchanging raw and numeraire values will produce unwanted results and may result in loss of funds for liquidity provider.

    - **Recommendation**: Short term, change the semantics of the three functions listed above in the CADC, XSGD, and EURS assimilators to return the numeraire balance. Long term, use unit tests and fuzzing to ensure that all calculations return the expected values. Additionally, ensure that changes to the Shell Protocol do not introduce bugs such as this one.
    - High Risk severity finding from ToB's Audit of DFX Finance.

57. **System always assumes USDC is equivalent to USD**: Throughout the system, assimilators are used to facilitate the processing of various

stablecoins. However, the `UsdcToUsdAssimilator`'s implementation of the `getRate` method does not use the USDC-USD oracle provided by Chainlink; instead, it assumes 1 USDC is always worth 1 USD. A deviation in the exchange rate of 1 USDC = 1 USD could result in exchange errors.

- **Recommendation**: Short term, replace the hard-coded integer literal in the `UsdcToUsdAssimilator`'s `getRate` method with a call to the relevant Chainlink oracle, as is done in other assimilator contracts. Long term, ensure that the system is robust against a decrease in the price of any stablecoin.
- Medium Risk severity finding from ToB's Audit of DFX Finance.

58. **Assimilators use a deprecated Chainlink API**: The old version of the Chainlink price feed API (`AggregatorInterface`) is used throughout the contracts and tests. For example, the deprecated function `latestAnswer` is used. This function is not present in the latest API reference (`AggregatorInterfaceV3`). However, it is present in the deprecated API reference. In the worst-case scenario, the deprecated contract could cease to report the latest values, which would very likely cause liquidity providers to incur losses.

    - **Recommendation**: Use the latest stable versions of any external libraries or contracts leveraged by the codebase
    - Undetermined Risk severity finding from ToB's Audit of DFX Finance.

59. **cancelOrdersUpTo can be used to permanently block future orders**: Users can cancel an arbitrary number of future orders, and this operation is not reversible. The `cancelOrdersUpTo` function can cancel an arbitrary number of orders in a single, fixed-size transaction. This function uses a parameter to discard any order with salt less than the input value. However, `cancelOrdersUpTo` can cancel future orders if it is called with a very large value (e.g., `MAX_UINT256 - 1`). This operation will cancel future orders, except for the one with salt equal to `MAX_UINT256`.

    - **Recommendation**: Properly document this behavior to warn users about the permanent effects of `cancelOrderUpTo` on future orders. Alternatively, disallow the cancelation of future orders.
    - High Risk severity finding from ToB's Audit of 0x Protocol.

60. **Specification-Code mismatch for AssetProxyOwner timelock period**: The specification for `AssetProxyOwner` says: "The `AssetProxyOwner` is a time-locked multi-signature wallet that has permission to perform administrative functions within the protocol. Submitted transactions must pass a 2 week timelock before they are executed."

The `MultiSigWalletWithTimeLock.sol` and `AssetProxyOwner.sol` contracts' timelock-period implementation/usage does not enforce the two-week period, but is instead configurable by the wallet owner without any range checks. Either the specification is outdated (most likely), or this is a serious flaw.

- **Recommendation**: Short term, implement the necessary range checks to enforce the timelock described in the specification. Otherwise correct the specification to match the intended behavior. Long term, make sure implementation and specification are in sync. Use Echidna or Manticore to test that your code properly implements the specification.
- High Risk severity finding from ToB's Audit of 0x Protocol.

61. **Unclear documentation on how order filling can fail**: The 0x documentation is unclear about how to determine whether orders are fillable or not. Even some fillable orders cannot be completely filled. The 0x specification does not state clearly enough how fillable orders are determined.

    - **Recommendation**: Define a proper procedure to determine if an order is fillable and document it in the protocol specification. If necessary, warn the user about potential constraints on the orders.
    - High Risk severity finding from ToB's Audit of 0x Protocol.

62. **Market makers have a reduced cost for performing front-running attacks**: Market makers receive a portion of the protocol fee for each order filled, and the protocol fee is based on the transaction gas price. Therefore market makers are able to specify a higher gas price for a reduced overall transaction rate, using the refund they will receive upon disbursement of protocol fee pools.

    - **Recommendation**: Short term, properly document this issue to make sure users are aware of this risk. Establish a reasonable cap for the protocolFeeMultiplier to mitigate this issue. Long term, consider using an alternative fee that does not depend on the tx.gasprice to avoid reducing the cost of performing front-running attacks.
    - Medium Risk severity finding from ToB's Audit of 0x Protocol.

63. **setSignatureValidatorApproval race condition may be exploitable**: If a validator is compromised, a race condition in the signature validator approval logic becomes exploitable. The `setSignatureValidatorApproval` function allows users to delegate the signature validation to a contract. However, if the validator is compromised, a race condition in this function could allow an attacker to validate any amount of malicious transactions.

- **Recommendation**: Short term, document this behavior to make sure users are aware of the inherent risks of using validators in case of a compromise. Long term, consider monitoring the blockchain using the `SignatureValidatorApproval` events to catch front-running attacks.
- Medium Risk severity finding from ToB's Audit of 0x Protocol.

64. **Batch processing of transaction execution and order matching may lead to exchange griefing**: Batch processing of transaction execution and order matching will iteratively process every transaction and order, which all involve filling. If the asset being filled does not have enough allowance, the asset's transferFrom will fail, causing `AssetProxyDispatcher` to revert. `NoThrow` variants of batch processing, which are available for filling orders, are not available for transaction execution and order matching. So if one transaction or order fails this way, the entire batch will revert and will have to be re-submitted after the reverting transaction is removed.

    - **Recommendation**: Short term, implement `NoThrow` variants for batch processing of transaction execution and order matching. Long term, take into consideration the effect of malicious inputs when implementing functions that perform a batch of operations.
    - Medium Risk severity finding from ToB's Audit of 0x Protocol.

65. **Zero fee orders are possible if a user performs transactions with a zero gas price**: Users can submit valid orders and avoid paying fees if they use a zero gas price. The computation of fees for each transaction is performed in the `calculateFillResults` function. It uses the gas price selected by the user and the `protocolFeeMultiplier` coefficient. Since the user completely controls the gas price of their transaction and the price could even be zero, the user could feasibly avoid paying fees.

    - **Recommendation**: Short term, select a reasonable minimum value for the protocol fee for each order or transaction. Long term, consider not depending on the gas price for the computation of protocol fees. This will avoid giving miners an economic advantage in the system.
    - Medium Risk severity finding from ToB's Audit of 0x Protocol.

66. **Calls to setParams may set invalid values and produce unexpected behavior in the staking contracts**: Certain parameters of the contracts can be configured to invalid values, causing a variety of issues and breaking expected interactions between contracts. `setParams` allows the owner of the staking contracts to reparameterize critical parameters. However, reparameterization lacks sanity/threshold/limit checks on all parameters.

- **Recommendation**: Add proper validation checks on all parameters in `setParams`. If the validation procedure is unclear or too complex to implement on-chain, document the potential issues that could produce invalid values.
- Medium Risk severity finding from ToB's Audit of 0x Protocol.

67. **Improper Supply Cap Limitation Enforcement**: The `openLoan()` function does not check if the loan to be issued will result in the supply cap being exceeded. It only enforces that the supply cap is not reached before the loan is opened. As a result, any account can create a loan that exceeds the maximum amount of sETH that can be issued by the `EtherCollateral` contract.

    - **Recommendation**: Introduce a require statement in the `openLoan()` function to prevent the total cap from being exceeded by the loan to be opened.
    - High Risk severity finding from Sigma Prime's Audit of Synthetix EtherCollateral.

68. **Improper Storage Management of Open Loan Accounts**: When loans are open, the associated account address gets added to the `accountsWithOpenLoans` array regardless of whether the account already has a loan/is already included in the array. Additionally, it is possible for a malicious actor to create a denial of service condition exploiting the unbound storage array in `accountsSynthLoans`.

    - **Recommendation**:
        1. Consider changing the `storeLoan` function to only push the account to the `accountsWithOpenLoans` array if the loan to be stored is the first one for that particular account
        2. Introduce a limit to the number of loans each account can have.
    - High Risk severity finding from Sigma Prime's Audit of Synthetix EtherCollateral.

69. **Contract Owner Can Arbitrarily Change Minting Fees and Interest Rates**: The `issueFeeRate` and `interestRate` variables can both be changed by the `EtherCollateral` contract owner after loans have been opened. As a result, the owner can control fees such as they equal/exceed the collateral for any given loan.

    - **Recommendation**: While "dynamic" interest rates are common, we recommend considering the minting fee (`issueFeeRate`) to be a constant that cannot be changed by the owner.
    - Medium Risk severity finding from Sigma Prime's Audit of Synthetix EtherCollateral.

70. **Inadequate Proxy Implementation Preventing Contract Upgrades**: The `TokenImpl` smart contract requires Owner , name , symbol and decimals of `TokenImpl` to be set by the `TokenImpl` constructor. Consider two smart contracts, contract A and contract B . If contract A performs a delegatecall on contract B , the state/storage variables of contract B are not accessible by contract A . Therefore, when `TokenProxy` targets an implementation of `TokenImpl` and interacts with it via a `DELEGATECALL` , it will not be able to access any of the state variables of the `TokenImpl` contract. Instead, the `TokenProxy` will access its local storage, which does not contain the variables set in the constructor of the `TokenImpl` implementation. When the `TokenProxy` contract is constructed it will only initialize and set two storage slots: • The proxy admin address (`_setAdmin` internal function) • The token implementation address (`_setImplementation` private function) Hence when a proxy call to the implementation is made, variables such as `Owner` will be uninitialised (effectively set to their default value). This is equivalent to the owner being the `0x0` address. Without access to the implementation state variables, the proxy contract is rendered unusable.

    - **Recommendation**:
        1. Set fixed constant parameters as Solidity constants. The solidity compiler replaces all occurrences of a constant in the code and thus does not reserve state for them. Thus if the correct getters exist for the `ERC20` interface, the proxy contract doesn't need to initialise anything.
        2. Create a constructor-like function that can only be called once within `TokenImpl` . This can be used to set the state variables as is currently done in the constructor, however if called by the proxy after deployment, the proxy will set its state variables.
        3. Create getter and setter functions that can only be called by the owner . Note that this strategy allows the owner to change various parameters of the contract after deployment.
        4. Predetermine the slots used by the required variables and set them in the constructor of the proxy. The storage slots used by a contract are deterministic and can be computed. Hence the variables Owner , name , symbol and decimals can be set directly by their slot in the proxy constructor.
    - Critical Risk severity finding from Sigma Prime's Audit of InfiniGold.

71. **Blacklisting Bypass via transferFrom() Function**: The `transferFrom()` function in the `TokenImpl` contract does not verify that the sender (i.e. the `from` address) is not blacklisted. As such, it is possible for a user to allow an account to spend a certain allowance regardless of their blacklisting status.

- **Recommendation**: At present the function `transferFrom()` uses the `notBlacklisted(address)` modifier twice, on the msg.sender and to addresses. The `notBlacklisted(address)` modifier should be used a third time against the from address.
- High Risk severity finding from Sigma Prime's Audit of InfiniGold.

72. **Wrong Order of Operations Leads to Exponentiation of rewardPerTokenStored**: `rewardPerTokenStored` is mistakenly used in the numerator of a fraction instead of being added to the fraction. The result is that `rewardPerTokenStored` will grow exponentially thereby severely overstating each individual's rewards earned. Individuals will therefore either be able to withdraw more funds than should be allocated to them or they will not be able to withdraw their funds at all as the contract has insufficient SNX balance. This vulnerability makes the Unipool contract unusable.

    - **Recommendation**: Adjust the function `rewardPerToken()` to represent the original functionality.
    - Critical Risk severity finding from Sigma Prime's Audit of Synthetix Unipool.

73. **Staking Before Initial notifyRewardAmount Can Lead to Disproportionate Rewards**: If a user successfully stakes an amount of UNI tokens before the function `notifyRewardAmount()` is called for the first time, their initial `userRewardPerTokenPaid` will be set to zero. The staker would be paid out funds greater than their share of the SNX rewards.

    - **Recommendation**: We recommend preventing `stake()` from being called before `notifyRewardAmount()` is called for the first time.
    - High Risk severity finding from Sigma Prime's Audit of Synthetix Unipool.

74. **External Call Reverts if Period Has Not Elapsed**: The function `notifyRewardAmount()` will revert if `block.timestamp >= periodFinish`. However this function is called indirectly via the `Synthetix.mint()` function. A revert here would cause the external call to fail and thereby halt the mint process. `Synthetix.mint()` cannot be successfully called until enough time has elapsed for the period to finish.

    - **Recommendation**: Consider handling the case where the reward period has not elapsed without reverting the call.
    - High Risk severity finding from Sigma Prime's Audit of Synthetix Unipool.

75. **Gap Between Periods Can Lead to Erroneous Rewards**: The SNX rewards are earned each period based on reward and duration as specified in the `notifyRewardAmount()` function. The contract will output more rewards than it receives. Therefore if all stakers call `getReward()` the contract will not have enough SNX balance to transfer out all the rewards and some stakers may not receive any rewards.

    - **Recommendation**: We recommend enforcing each period start exactly at the end of the previous period.
    - Medium Risk severity finding from Sigma Prime's Audit of Synthetix Unipool.

76. **Malicious Users Can DOS/Hijack Requests From Chainlinked Contracts**: Malicious users can hijack or perform Denial of Service (DOS) attacks on requests of Chainlinked contracts by replicating or front-running legitimate requests. The Chainlinked (`Chainlinked.sol`) contract contains the `checkChainlinkFulfillment`() modifier.

    This modifier is demonstrated in the examples that come with the repository. In these examples this modifier is used within the functions which contracts implement that will be called by the Oracle when fulfilling requests. It requires that the caller of the function be the Oracle that corresponds to the request that is being fulfilled. Thus, requests from Chainlinked contracts are expected to only be fulfilled by the Oracle that they have requested. However, because a request can specify an arbitrary callback address, a malicious user can also place a request where the callback address is a target Chainlinked contract.

    If this malicious request gets fulfilled first (which can ask for incorrect or malicious results), the Oracle will call the legitimate contract and fulfil it with incorrect or malicious results. Because the known requests of a Chainlinked contract gets deleted, the legitimate request will fail. It could be such that the Oracle fulfils requests in the order in which they are received. In such cases, the malicious user could simply front-run the requests to be higher in the queue.

    - **Recommendation**: This issue arises due to the fact that any request can specify its own arbitrary callback address. A restrictive solution would be where callback addresses are localised to the requester themselves.
    - High Risk severity finding from Sigma Prime's Audit of Chainlink.

77. **Lack of event emission after sensitive actions**: The `_getLatestFundingRate` function of the `FundingRateApplier` contract does not emit relevant events after executing the sensitive actions of setting the `fundingRate`, `updateTime` and `proposalTime`, and transferring the rewards.

- **Recommendation**: Consider emitting events after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contract's activity.
- Medium Risk severity finding from OpenZeppelin's Audit of UMA Phase 4.

78. **Functions with unexpected side-effects**: Some functions have side-effects. For example, the `_getLatestFundingRate` function of the `FundingRateApplier` contract might also update the funding rate and send rewards. The `getPrice` function of the OptimisticOracle contract might also settle a price request. These side-effect actions are not clear in the name of the functions and are thus unexpected, which could lead to mistakes when the code is modified by new developers not experienced in all the implementation details of the project.

    - **Recommendation**: Consider splitting these functions in separate getters and setters. Alternatively, consider renaming the functions to describe all the actions that they perform.
    - Medium Risk severity finding from OpenZeppelin's Audit of UMA Phase 4.

79. **Mooniswap pairs cannot be unpaused**: The `MooniswapFactoryGovernance` contract has a shutdown function that can be used to pause the contract and prevent any future swaps. However there is no function to unpause the contract. There is also no way for the factory contract to redeploy a Mooniswap instance for a given pair of tokens. Therefore, if a Mooniswap contract is ever shutdown/paused, it will not be possible for that pair of tokens to ever be traded on the Mooniswap platform again, unless a new factory contract is deployed.

    - **Recommendation**: Consider providing a way for Mooniswap contracts to be unpaused.
    - Medium Risk severity finding from OpenZeppelin's Audit of 1inch Liquidity Protocol Audit.

80. **Attackers can prevent honest users from performing an instant withdraw from the Wallet contract**: An attacker who sees an honest user's call to `MessageProcessor.instantWithdraw` in the mempool can grab the `oracleMessage` and `oracleSignature` parameters from the user's transaction, then submit their own transaction to `instantWithdraw` using the same parameters, a higher gas price (so as to frontrun the honest user's transaction), and carefully choosing the gas limit for their transactions such that the internal call to the `callInstantWithdraw` will fail on line 785 with an out-of-gas error, but will successfully execute the `if(!success)` block. The result is that the attacker's instant withdraw will fail (so the user will not receive their funds), but the

`userInteractionNumber` will be successfully reserved by the `ReplayTracker`. As a result, the honest user's transaction will revert because it will be attempting to use a `userInteractionNumber` that is no longer valid.

- **Recommendation**: Consider adding an access control mechanism to restrict who can submit `oracleMessages` on behalf of the user.
- High Risk severity finding from OpenZeppelin's Audit of Futureswap V2.

81. **Not using upgrade safe contracts in FsToken inheritance**: The `FsToken` contract is intended to be an upgradeable contract, used behind a proxy (namely, the `FsTokenProxy` contract). However, the contracts `ERC20Snapshot`, `ERC20Mintable` and `ERC20Burnable` in the inheritance chain of FsToken are not imported from the upgrade safe library `@openzeppelin/contracts-ethereum-package` but instead from `@openzeppelin/contracts`.

    - **Recommendation**: Use the upgrades safe library in this case will ensure the inheritance from Initializable and the other contracts is always linearized as expected by the compiler.
    - Medium Risk severity finding from OpenZeppelin's Audit of Futureswap V2.

82. **Unchecked output of the ECDSA recover function**: The `ECDSA.recover` function (in version `2.5.1`) returns `address(0)` if the signature provided is invalid. This function is used twice in the Futureswap code: Once to recover an `oracleAddress` from an `oracleSignature`, and again to recover the user's address from their signature. If the oracle signature was invalid, the `oracleAddress` is set to `address(0)`. Similarly, if the user's signature is invalid, then the `userMessage.signer` or the withDrawer is set to `address(0)`. This can result in unintended behavior.

    - **Recommendation**: Consider reverting if the output of the `ECDSA.recover` is ever `address(0)`
    - Medium Risk severity finding from OpenZeppelin's Audit of Futureswap V2.

83. **Adding new variables to multi-level inherited upgradeable contracts may break storage layout**: The Notional protocol uses the OpenZeppelin/SDK contracts to manage upgradeability in the system, which follows the unstructured storage pattern. When using this upgradeability approach, and when working with multi-level inheritance, if a new variable is introduced in a parent contract, that addition can potentially overwrite the beginning of the storage layout of the child contract, causing critical misbehaviors in the system.

- **Recommendation**: consider preventing these scenarios by defining a storage gap in each upgradeable parent contract at the end of all the storage variable definitions as follows:

  ```
  uint256[50] __gap; // gap to reserve storage in the contract for future
  ```

  In such an implementation, the size of the gap would be intentionally decreased each time a new variable was introduced, thereby avoiding overwriting preexisting storage values.
- Medium Risk severity finding from OpenZeppelin's Audit of Notional Protocol.

84. **Unsafe division in rdivide and wdivide functions**: The function `rdivide` on line 227 and the function `wdivide` on line 230 of the `GlobalSettlement` contract, accept the divisor y as an input parameter. However, these functions do not check if the value of y is 0. If that is the case, the call will revert due to the division by zero error.

    - **Recommendation**: consider adding a `require` statement in the functions to ensure `y > 0`, or consider using the `div` functions provided in OpenZeppelin's SafeMath libraries
    - Medium Risk severity finding from OpenZeppelin's Audit of GEB Protocol.

85. **Incorrect safeApprove usage**: The `safeApprove` function of the OpenZeppelin SafeERC20 library prevents changing an allowance between non-zero values to mitigate a possible front-running attack. Instead, the `safeIncreaseAllowance` and `safeDecreaseAllowance` functions should be used. However, the `UniERC20` library simply bypasses this restriction by first setting the allowance to zero. This reintroduces the front-running attack and undermines the value of the `safeApprove` function. Consider introducing an `increaseAllowance` function to handle this case.

    - **Recommendation**: `safeIncreaseAllowance` and `safeDecreaseAllowance` functions should be used
    - Medium Risk severity finding from OpenZeppelin's Audit of 1inch Exchange Audit.

86. **ETH could get trapped in the protocol**: The Controller contract allows users to send arbitrary actions such as possible flash loans through the `_call` internal function. Among other features, it allows sending ETH with the action to then perform a call to a `CalleeInterface` type of contract. To do so, it saves the original `msg.value` sent with the operate function call in the `ethLeft` variable and it updates the remaining ETH left after each one of those calls to revert in case that it is not enough. Nevertheless, if the user sends more than the necessary ETH for the batch of actions, the remaining ETH (stored in the ethLeft variable after the

last iteration) will not be returned to the user and will be locked in the contract due to the lack of a `withdrawEth` function.

- **Recommendation**: Consider either returning all the remaining ETH to the user or creating a function that allows the user to collect the remaining ETH after performing a Call action type, taking into account that sending ETH with a push method may trigger the fallback function on the caller's address.
- High Risk severity finding from OpenZeppelin's Audit of Opyn Gamma Protocol.

87. **Use of transfer might render ETH impossible to withdraw**: When withdrawing ETH deposits, the `PayableProxyController` contract uses Solidity's `transfer` function. This has some notable shortcomings when the withdrawer is a smart contract, which can render ETH deposits impossible to withdraw. Specifically, the withdrawal will inevitably fail when:

    1. The withdrawer smart contract does not implement a payable fallback function.
    2. The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units.
    3. The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

    - **Recommendation**: `sendValue` function available in OpenZeppelin Contract's Address library can be used to transfer the withdrawn Ether without being limited to 2300 gas units. Risks of reentrancy stemming from the use of this function can be mitigated by tightly following the "Check-effects-interactions" pattern and using OpenZeppelin Contract's `ReentrancyGuard` contract.
    - Medium Risk severity finding from OpenZeppelin's Audit of Opyn Gamma Protocol.

88. **Not following the Checks-Effects-Interactions pattern**: The `finalizeGrant` function of the Fund contract is setting the `grant.complete` storage variable after a token transfer. Solidity recommends the usage of the Check-Effects-Interaction Pattern to avoid potential security issues, such as reentrancy. The `finalizeGrant` function can be used to conduct a reentrancy attack, where the token transfer in line 129 can call back again the same function, sending to the admin multiple times an amount of fee, before setting the grant as completed. In this way the grant.recipient can receive less than expected and the contract funds can be drained unexpectedly leading to an unwanted loss of funds.

- **Recommendation**: Consider always following the "Check-Effects-Interactions" pattern, thus modifying the contract's state before making any external call to other contracts.
- High Risk severity finding from OpenZeppelin's Audit of Endaoment.

89. **Updating the Governance registry and Guardian addresses emits no events**: In the Governance contract the `registryAddress` and the `guardianAddress` are highly sensitive accounts. The first one holds the contracts that can be proposal targets, and the second one is a superuser account that can execute proposals without voting. These variables can be updated by calling `setRegistryAddress` and `transferGuardianship`, respectively. Note that these two functions update these sensitive addresses without logging any events. Stakers who monitor the Audius system would have to inspect all transactions to notice that one address they trust is replaced with an untrusted one.

    - **Recommendation**: Consider emitting events when these addresses are updated. This will be more transparent, and it will make it easier for clients to subscribe to the events when they want to keep track of the status of the system.
    - High Risk severity finding from OpenZeppelin's Audit of Audius.

90. **The quorum requirement can be trivially bypassed with sybil accounts**: While the final vote on a proposal is determined via a token-weighted vote, the quorum check in the `evaluateProposalOutcome` function can be trivially bypassed by splitting one's tokens over multiple accounts and voting with each of the accounts. Each of these sybil votes increases the `proposals[_proposalId].numVotes` variable. This means anyone can make the quorum check pass.

    - **Recommendation**: Consider measuring quorum size by the percentage of existing tokens that have voted, rather than the number of unique accounts that have voted.
    - High Risk severity finding from OpenZeppelin's Audit of Audius.

91. **Inconsistently checking initialization**: When a contract is initialized, its `isInitialized` state variable is set to true. Since interacting with uninitialized contracts would cause problems, the `_requireIsInitialized` function is available to make this check. However, this check is not used consistently. For example, it is used in the `getVotingQuorum` function of the Governance contract, but it is not used in the `getRegistryAddress` function of the same contract. There is no obvious difference between the functions to explain this difference, and it could be misleading and cause uninitialized contracts to be called.

    - **Recommendation**: Consider calling `_requireIsInitialized` consistently in all the functions of the `InitializableV2` contracts. If

there is a reason to not call it in some functions, consider documenting it. Alternatively, consider removing this check altogether and preparing a good deployment script that will ensure that all contracts are initialized in the same transaction that they are deployed. In this alternative, it would be required to check that contracts resulting from new proposals are also initialized before they are put in production.

- Medium Risk severity finding from OpenZeppelin's Audit of Audius.

92. **Voting period and quorum can be set to zero**: When the Governance contract is initialized, the values of `votingPeriod` and `votingQuorum` are checked to make sure that they are greater than 0. However, the corresponding setter functions `setVotingPeriod` and setVotingQuorum allow these variables to be reset to 0. Setting the `votingPeriod` to zero would cause spurious proposals that cannot be voted. Setting the quorum to zero is worse because it would allow proposals with 0 votes to be executed.

    - **Recommendation**: Consider adding the validation to the setter functions
    - Medium Risk severity finding from OpenZeppelin's Audit of Audius.

93. **Some state variables are not set during initialize**: The Audius contracts can be upgraded using the unstructured storage proxy pattern. This pattern requires the use of an initializer instead of the constructor to set the initial values of the state variables. In some of the contracts, the initializer is not initializing all of the state variables.

    - **Recommendation**: Consider setting all the required variables in the initializer. If there is a reason for leaving them uninitialized, consider documenting it, and adding checks on the functions that use those variables to ensure that they are not called before initialization.
    - Medium Risk severity finding from OpenZeppelin's Audit of Audius.

94. **Expired and/or paused options can still be traded**: Option tokens can still be freely transferred when the Option contract is either paused or expired (or both). This would allow malicious option holders to sell paused / expired options that cannot be exercised in the open market to exchanges and users who do not take the necessary precautions before buying an option minted by the Primitive protocol.

    - **Recommendation**: Should this be the system's expected behavior, consider clearly documenting it in user-friendly documentation so as to raise awareness in option sellers and buyers. Alternatively, if the described behavior is not intended, consider implementing the necessary logic in the Option contract to prevent transfers of tokens during pause and after expiration.

- Medium Risk severity finding from OpenZeppelin's Audit of Primitive.

95. **ERC20 transfers can misbehave**: The `_transferFromERC20` function is used throughout `ACOToken.sol` to handle transferring funds into the contract from a user. It is called within mint, within `mintTo`, and within `_validateAndBurn`. In each case, the destination is the ACOToken contract. Such transfers may behave unexpectedly if the token contract charges fees. As an example, the popular USDT token does not presently charge any fees upon transfer, but it has the potential to do so. In this case the amount received would be less than the amount sent. Such tokens have the potential to lead to protocol insolvency when they are used to mint new ACOTokens. In the case of `_transferERC20`, similar issues can occur, and could cause users to receive less than expected when collateral is transferred or when exercise assets are transferred.

    - **Recommendation**: Consider thoroughly vetting each token used within an ACO options pair, ensuring that failing `transferFrom` and `transfer` calls will cause reverts within `ACOToken.sol`. Additionally, consider implementing some sort of sanity check which enforces that the balance of the ACOToken contract increases by the desired amount when calling `_transferFromERC20`.
    - Medium Risk severity finding from OpenZeppelin's Audit of ACO Protocol.

96. **Incorrect event emission**: The `UniswapWindowUpdate` event of the `UniswapAnchoredView` contract is currently being emitted in the `pokeWindowValues` function using incorrect values. In particular, as it is being emitted before relevant state changes are applied to the `oldObservation` and `newObservation` variables, the data logged by the event will be outdated.

    - **Recommendation**: Consider emitting the `UniswapWindowUpdate` event after changes are applied so that all logged data is up-to-date.
    - Medium Risk severity finding from OpenZeppelin's Audit of Compound Open Price Feed – Uniswap Integration.

97. **Anyone can liquidate on behalf of another account**: The Perpetual contract has a public liquidateFrom function that bypasses the checks in the liquidate function. This means that it can be called to liquidate a position when the contract is in the `SETTLED` state. Additionally, any user can set an arbitrary from address, causing a third-party user to confiscate the under-collateralized trader's position. This means that any trader can unilaterally rearrange another account's position. They could also liquidate on behalf of the Perpetual Proxy, which could break some of the Automated Market Maker invariants, such as the condition that it only holds LONG positions.

- **Recommendation**: Consider restricting `liquidateFrom` to internal visibility
- Critical Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol.

98. **Orders cannot be cancelled**: When a user or broker calls `cancelOrder`, the cancelled mapping is updated, but this has no subsequent effects. In particular, `validateOrderParam` does not check if the order has been cancelled.

    - **Recommendation**: Consider adding this check to the order validation to ensure cancelled orders cannot be filled.
    - Critical Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol.

99. **Re-entrancy possibilities**: There are several examples of interactions preceding effects:

    1. In the deposit function of the Collateral contract, collateral is retrieved before the user balance is updated and an event is emitted.
    2. In the `_withdraw` function of the Collateral contract, collateral is sent before the event is emitted
    3. The same pattern occurs in the `depositToInsuranceFund`, `depositEtherToInsuranceFund` and `withdrawFromInsuranceFund` functions of the Perpetual contract.

    It should be noted that even when a correctly implemented `ERC20` contract is used for collateral, incoming and outgoing transfers could execute arbitrary code if the contract is also ERC777 compliant. These re-entrancy opportunities are unlikely to corrupt the internal state of the system, but they would affect the order and contents of emitted events, which could confuse external clients about the state of the system.

    - **Recommendation**: Consider always following the "Check-Effects-Interactions" pattern or use `ReentrancyGuard` contract is now used to protect those functions
    - Medium Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol.

100. **Governance parameter changes should not be instant**: Many sensitive changes can be made by any account with the `WhitelistAdmin` role via the functions `setGovernanceParameter` within the `AMMGovernance` and `PerpetualGovernance` contracts. For example, the `WhitelistAdmin` can change the fee schedule, the initial and maintenance margin rates, or the lot size parameters, and these new parameters instantly take effect in the protocol with important effects. For example, raising the maintenance margin rate could cause `isSafe` to return False when it would

have previously returned True. This would allow the user's position to be liquidated. By changing `tradingLotSize`, trades may revert when being matched, where they would not have before the change. These are only examples; the complexity of the protocol, combined with unpredictable market conditions and user actions means that many other negative effects likely exist as well.

- **Recommendation**: Since these changes are occasionally needed, but can create risk for the users of the protocol, consider implementing a time-lock mechanism for such changes to take place. By having a delay between the signal of intent and the actual change, users will have time to remove their funds or close trades that would otherwise be at risk if the change happened instantly.
- Medium Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol.

101. **Votes can be duplicated**: The Data Verification Mechanism uses a commit-reveal scheme to hide votes during the voting period. The intention is to prevent voters from simply voting with the majority. However, the current design allows voters to blindly copy each other's submissions, which undermines this goal. In particular, each commitment is a masked hash of the claimed price, but is not cryptographically tied to the voter. This means that anyone can copy the commitment of a target voter (for instance, someone with a large balance) and submit it as their own. When the target voter reveals their salt and price, the copycat can "reveal" the same values. Moreover, if another voter recognizes this has occurred during the commitment phase, they can also change their commitment to the same value, which may become an alternate Schelling point.

    - **Recommendation**: Consider including the voter address within the commitment to prevent votes from being duplicated. Additionally, as a matter of good practice, consider including the relevant timestamp, price identifier and round ID as well to limit the applicability (and reusability) of a commitment.
    - High Risk severity finding from OpenZeppelin's Audit of UMA Phase 1.

B.8 Audit Findings 201: Key aspects

(See https://secureum.substack.com/p/audit-findings-201 for the original article.)

102. **Document potential edge cases for hook receiver contracts**: The functions `withdrawTokenAndCall()` and `withdrawTokenAndCallOnBehalf()` make a call to a hook contract designated by the owner of the withdrawing stealth address. There are very few constraints on the parameters to these calls in the Umbra contract itself. Anyone can force a call to a hook contract by transferring a small amount of tokens to an address that they control and withdrawing these tokens, passing the target address as the hook receiver.

    1. Recommendation: Developers of these `UmbraHookReceiver` contracts should be sure to validate both the caller of the `tokensWithdrawn()` function and the function parameters.
    2. ConsenSys's Audit of Umbra

103. **Document token behavior restrictions**: As with any protocol that interacts with arbitrary ERC20 tokens, it is important to clearly document which tokens are supported. Often this is best done by providing a specification for the behavior of the expected ERC20 tokens and only relaxing this specification after careful review of a particular class of tokens and their interactions with the protocol.

    1. Recommendation: Known deviations from "normal" ERC20 behavior should be explicitly noted as NOT supported by the Umbra Protocol: 1) Deflationary or fee-on-transfer tokens: These are tokens in which the balance of the recipient of a transfer may not be increased by the amount of the transfer. There may also be some alternative mechanism by which balances are unexpectedly decreased. While these tokens can be successfully sent via the `sendToken()` function, the internal accounting of the Umbra contract will be out of sync with the balance as recorded in the token contract, resulting in loss of funds. 2) Inflationary tokens: The opposite of deflationary tokens. The Umbra contract provides no mechanism for claiming positive balance adjustments. 3) Rebasing tokens: A combination of the above cases, these are tokens in which an account's balance increases or decreases along with expansions or contractions in supply. The contract provides no mechanism to update its internal accounting in response to these unexpected balance adjustments, and funds may be lost as a result.
    2. ConsenSys's Audit of Umbra

104. **Full test suite is recommended**: The test suite at this stage is not complete and many of the tests fail to execute. For complicated systems such as DeFi Saver, which uses many different modules and interacts with different DeFi protocols, it is crucial to have a full test coverage that includes the edge cases and failed scenarios. Especially this helps with safer future development and upgrading each module. As we've seen in some smart contract incidents, a complete test suite can prevent issues that might be hard to find with manual reviews.

    1. Recommendation: Add a full coverage test suite.
    2. ConsenSys's Audit of DeFi Saver

105. **Kyber getRates code is unclear**: Function names don't reflect their true functionalities, and the code uses some undocumented assumptions.

    1. Recommendation: Refactor the code to separate getting rate functionality with getSellRate and getBuyRate. Explicitly document any assumptions in the code (slippage, etc).
    2. ConsenSys's Audit of DeFi Saver

106. **Return value is not used for TokenUtils.withdrawTokens**: The return value of `TokenUtils.withdrawTokens` which represents the actual amount of tokens that were transferred is never used throughout the repository. This might cause discrepancy in the case where the original value of `_amount` was `type(uint256).max`.

    1. Recommendation: The return value can be used to validate the withdrawal or used in the event emitted
    2. ConsenSys's Audit of DeFi Saver

107. **Missing access control for DefiSaverLogger.Log**: `DefiSaverLogger` is used as a logging aggregator within the entire dapp, but anyone can create logs.

    1. Recommendation: Add access control to all functions appropriately
    2. Consensys Audit of DeFi Saver

108. **Remove stale comments**: Remove inline comments that suggest the two uint256 values `DAOfiV1Pair.reserveBase` and `DAOfiV1Pair.reserveQuote` are stored in the same storage slot. This is likely a carryover from the `UniswapV2Pair` contract, in which `reserve0`, `reserve1`, and `blockTimestampLast` are packed into a single storage slot.

    1. Recommendation: Remove stale comments
    2. ConsenSys's Audit of DAOfi

109. **Discrepancy between code and comments**: There is a mismatch between what the code implements and what the corresponding comment describes that code implements.

    1. Recommendation: Update the code or the comment to be consistent
    2. ConsenSys's Audit of mstable-1.1

110. **Remove unnecessary call to DAOfiV1Factory.formula()**: The `DAOfiV1Pair` functions `initialize()`, `getBaseOut()`, and `getQuoteOut()` all use the private function `_getFormula()`, which makes a call to the factory to retrieve the address of the `BancorFormula` contract. The formula address in the factory is set in the constructor and cannot be changed, so these calls can be replaced with an immutable value in the pair contract that is set in its constructor.

    1. Recommendation: Remove unnecessary calls
    2. ConsenSys's Audit of DAOfi

111. **Deeper validation of curve math**: Increased testing of edge cases in complex mathematical operations could have identified at least one issue raised in this report. Additional unit tests are recommended, as well as fuzzing or property-based testing of curve-related operations. Improperly validated interactions with the `BancorFormula` contract are seen to fail in unanticipated and potentially dangerous ways, so care should be taken to validate inputs and prevent pathological curve parameters.

    1. Recommendation: More validation of mathematical operations
    2. ConsenSys's Audit of DAOfi

112. **GovernorAlpha proposals may be canceled by the proposer, even after they have been accepted and queued**: `GovernorAlpha` allows proposals to be canceled via cancel. A proposer may cancel proposals in any of these states: Pending, Active, Canceled, Defeated, Succeeded, Queued, Expired.

    1. Recommendation: Prevent proposals from being canceled unless they are in the Pending or Active states.
    2. ConsenSys's Audit of Fei Protocol

113. **Require a delay period before granting KYC_ADMIN_ROLE Acknowledged**: The KYC Admin has the ability to freeze the funds of any user at any time by revoking the `KYC_MEMBER_ROLE`. The trust requirements from users can be decreased slightly by implementing a delay on granting this ability to new addresses. While the management of private keys and admin access is outside the scope of this review, the addition of a time delay can also help protect the development team and the system itself in the event of private key compromise.

    1. Recommendation: Use a `TimelockController` as the `KYC_DEFAULT_ADMIN` of the eRLC contract
    2. ConsenSys's Audit of eRLC

114. **Improve inline documentation and test coverage**: The source-units hardly contain any inline documentation which makes it hard to reason about methods and how they are supposed to be used. Additionally, test-coverage seems to be limited. Especially for a public-facing exchange contract system test-coverage should be extensive, covering all methods and functions that can directly be accessed including potential security-relevant and edge-cases. This would have helped in detecting some of the findings raised with this report.

    1. Recommendation: Consider adding natspec-format compliant inline code documentation, describe functions, what they are used for, and who is supposed to interact with them. Document function or source-unit specific assumptions. Increase test coverage.
    2. ConsenSys's Audit of 1inch Liquidity Protocol

115. **Unspecific compiler version pragma**: For most source-units the compiler version pragma is very unspecific `^0.6.0`. While this often makes sense for libraries to allow them to be included with multiple different versions of an application, it may be a security risk for the actual application implementation itself. A known vulnerable compiler version may accidentally be selected or security tools might fall-back to an older compiler version ending up actually checking a different evm compilation that is ultimately deployed on the blockchain.

    1. Recommendation: Avoid floating pragmas. We highly recommend pinning a concrete compiler version (latest without security issues) in at least the top-level "deployed" contracts to make it unambiguous which compiler version is being used. Rule of thumb: a flattened source-unit should have at least one non-floating concrete solidity compiler version pragma.
    2. ConsenSys's Audit of 1inch Liquidity Protocol

116. **Use of hardcoded gas limits can be problematic**: Hardcoded gas limits can be problematic as the past has shown that gas economics in ethereum have changed, and may change again potentially rendering the contract system unusable in the future.

    1. Recommendation: Be conscious about this potential limitation and prepare for the case where gas prices might change in a way that negatively affects the contract system.
    2. ConsenSys's Audit of 1inch Liquidity Protocol

117. **Anyone can steal all the funds that belong to ReferralFeeReceiver**: The `ReferralFeeReceiver` receives pool shares when users `swap()` tokens in the pool. A `ReferralFeeReceiver` may be used with multiple pools and, therefore, be a lucrative target as it is holding pool shares. Any token or ETH that belongs to the `ReferralFeeReceiver` is at risk and can be drained by any user by providing a custom mooniswap pool contract that references existing token holdings. It should be noted that none of the functions in `ReferralFeeReceiver` verify that the user-provided mooniswap pool address was actually deployed by the linked MooniswapFactory.

    1. Recommendation: Enforce that the user-provided mooniswap contract was actually deployed by the linked factory. Other contracts cannot be trusted. Consider implementing token sorting and deduplication (`tokenA!=tokenB`) in the pool contract constructor as well. Consider employing a reentrancy guard to safeguard the contract from reentrancy attacks. Improve testing. The methods mentioned here are not covered at all. Improve documentation and provide a specification that outlines how this contract is supposed to be used.
    2. Critical finding in ConsenSys's Audit of 1inch Liquidity Protocol

118. **Unpredictable behavior for users due to admin front running or general bad timing**: In a number of cases, administrators of contracts can update or upgrade things in the system without warning. This has the potential to violate a security goal of the system. Specifically, privileged roles could use front running to make malicious changes just ahead of incoming transactions, or purely accidental negative effects could occur due to the unfortunate timing of changes. In general users of the system should have assurances about the behavior of the action they're about to take.

    1. Recommendation: We recommend giving the user advance notice of changes with a time lock. For example, make all system-parameter and upgrades require two steps with a mandatory time window between them. The first step merely broadcasts to users that a particular change is coming, and the second step commits that change after a suitable waiting period. This allows users that do not accept the change to withdraw immediately.
    2. ConsenSys's Audit of 1inch Liquidity Protocol

119. **Improve system documentation and create a complete technical specification**: A system's design specification and supporting documentation should be almost as important as the system's implementation itself. Users rely on high-level documentation to understand the big picture of how a system works. Without spending time and effort to create palatable documentation, a user's only resource is the code itself, something the vast

majority of users cannot understand. Security assessments depend on a complete technical specification to understand the specifics of how a system works. When a behavior is not specified (or is specified incorrectly), security assessments must base their knowledge in assumptions, leading to less effective review. Maintaining and updating code relies on supporting documentation to know why the system is implemented in a specific way. If code maintainers cannot reference documentation, they must rely on memory or assistance to make high-quality changes. Currently, the only documentation for Growth DeFi is a single README file, as well as code comments.

1. Recommendation: Improve system documentation and create a complete technical specification
2. ConsenSys's Audit of Growth DeFi

120. **Ensure system states, roles, and permissions are sufficiently restrictive**: Smart contract code should strive to be strict. Strict code behaves predictably, is easier to maintain, and increases a system's ability to handle nonideal conditions. Our assessment of Growth DeFi found that many of its states, roles, and permissions are loosely defined.

    1. Recommendation: Document the use of administrator permissions. Monitor the usage of administrator permissions. Specify strict operation requirements for each contract.
    2. ConsenSys's Audit of Growth DeFi

121. **Evaluate all tokens prior to inclusion in the system**: Review current and future tokens in the system for non-standard behavior. Particularly dangerous functionality to look for includes a callback (ie. ERC777) which would enable an attacker to execute potentially arbitrary code during the transaction, fees on transfers, or inflationary/deflationary tokens.

    1. Recommendation: Evaluate all tokens prior to inclusion in the system
    2. ConsenSys's Audit of Growth DeFi

122. **Use descriptive names for contracts and libraries**: The code base makes use of many different contracts, abstract contracts, interfaces, and libraries for inheritance and code reuse. In principle, this can be a good practice to avoid repeated use of similar code. However, with no descriptive naming conventions to signal which files would contain meaningful logic, codebase becomes difficult to navigate.

    1. Recommendation: Use descriptive names for contracts and libraries
    2. ConsenSys's Audit of Growth DeFi

123. **Prevent contracts from being used before they are entirely initialized**: Many contracts allow users to deposit / withdraw assets before

the contracts are entirely initialized, or while they are in a semi-configured state. Because these contracts allow interaction on semi-configured states, the number of configurations possible when interacting with the system makes it incredibly difficult to determine whether the contracts behave as expected in every scenario, or even what behavior is expected in the first place.

1. Recommendation: Prevent contracts from being used before they are entirely initialized
2. ConsenSys's Audit of Growth DeFi

124. **Potential resource exhaustion by external calls performed within an unbounded loop**: `DydxFlashLoanAbstraction._requestFlashLoan` performs external calls in a potentially-unbounded loop. Depending on changes made to DyDx's `SoloMargin`, this may render this flash loan provider prohibitively expensive. In the worst case, changes to `SoloMargin` could make it impossible to execute this code due to the block gas limit.

    1. Recommendation: Reconsider or bound the loo
    2. ConsenSys's Audit of Growth DeFi

125. **Owners can never be removed**: The intention of `setOwners()` is to replace the current set of owners with a new set of owners. However, the `isOwner` mapping is never updated, which means any address that was ever considered an owner is permanently considered an owner for purposes of signing transactions.

    1. Recommendation: In `setOwners_()`, before adding new owners, loop through the current set of owners and clear their `isOwner` booleans
    2. Critical finding in ConsenSys's Audit of Paxos

126. **Potential manipulation of stable interest rates using flash loans**: Flash loans allow users to borrow large amounts of liquidity from the protocol. It is possible to adjust the stable rate up or down by momentarily removing or adding large amounts of liquidity to reserves.

    1. Recommendation: This type of manipulation is difficult to prevent especially when flash loans are available. Aave should monitor the protocol at all times to make sure that interest rates are being rebalanced to sane values.
    2. ConsenSys's Audit of Aave Protocol V2

127. **Only whitelist validated assets**: Because some of the functionality relies on correct token behavior, any whitelisted token should be audited in the context of this system. Problems can arise if a malicious token is whitelisted because it can block people from voting with that specific token or gain unfair advantage if the balance can be manipulated.

    1. Recommendation: Make sure to audit any new whitelisted asset.
    2. ConsenSys's Audit of Aave Governance DAO

128. **Underflow if TOKEN_DECIMALS are greater than 18**: In `latestAnswer()`, the assumption is made that `TOKEN_DECIMALS` is less than 18.

    1. Recommendation: Add a simple check to the constructor to ensure the added token has 18 decimals or less
    2. ConsenSys's Audit of Aave CPM Price Provider

129. **Chainlink's performance at times of price volatility**: In order to understand the risk of the Chainlink oracle deviating significantly, it would be helpful to compare historical prices on Chainlink when prices are moving rapidly, and see what the largest historical delta is compared to the live price on a large exchange.

    1. Recommendation: Review Chainlink's performance at times of price volatility
    2. ConsenSys's Audit of Aave CPM Price Provider

130. **Consider an iterative approach to launching. Be aware of and prepare for worst-case scenarios**: The system has many components with complex functionality and no apparent upgrade path.

    1. Recommendation: We recommend identifying which components are crucial for a minimum viable system, then focusing efforts on ensuring the security of those components first, and then moving on to the others. During the early life of the system, have a method for pausing and upgrading the system.
    2. ConsenSys's Audit of Lien Protocol

131. **Use of modifiers for repeated checks**: It is recommended to use modifiers for common checks within different functions. This will result in less code duplication in the given smart contract and adds significant readability into the code base.

    1. Recommendation: Use of modifiers for repeated checks
    2. ConsenSys's Audit of Balancer Finance

132. **Switch modifier order**: `BPool` functions often use modifiers in the following order: `_logs_`, `_lock_`. Because `_lock_` is a reentrancy guard, it should take precedence over `_logs_`.

    1. Recommendation: Place `_lock_` before other modifiers; ensuring it is the very first and very last thing to run when a function is called.
    2. ConsenSys's Audit of Balancer Finance

133. **Address codebase fragility**: Software is considered "fragile" when issues or changes in one part of the system can have side-effects in conceptually unrelated parts of the codebase. Fragile software tends to break easily and may be challenging to maintain.

    1. Recommendation: Building an anti-fragile system requires careful thought and consideration outside of the scope of this review. In general, prioritize the following concepts: 1) Follow the single-responsibility principle of functions 2) Reduce reliance on external systems
    2. ConsenSys's Audit of MCDEX Mai Protocol V2

134. **Reentrancy could lead to incorrect order of emitted events**: The order of operations in the `_moveTokensAndETHfromAdjustment` function in the `BorrowOperations` contract may allow an attacker to cause events to be emitted out of order. In the event that the borrower is a contract, this could trigger a callback into `BorrowerOperations`, executing the `_adjustTrove` flow above again. As the `_moveTokensAndETHfromAdjustment` call is the final operation in the function the state of the system on-chain cannot be manipulated. However, there are events that are emitted after this call. In the event of a reentrant call, these events would be emitted in the incorrect order. The event for the second operation i s emitted first, followed by the event for the first operation. Any off-chain monitoring tools may now have an inconsistent view of on-chain state.

    1. Recommendation: Apply the checks-effects-interactions pattern and move the event emissions above the call to `_moveTokensAndETHfromAdjustment` to avoid the potential reentrancy.
    2. ToB's Audit of Liquidity

135. **Variable shadowing from OUSD to ERC20**: OUSD inherits from ERC20, but redefines the `_allowances` and `_totalSupply` state variables. As a result, access to these variables can lead to returning different values.

    1. Recommendation: Remove the shadowed variables (`_allowances` and `_totalSupply`) in OUSD.
    2. ToB's Audit of Origin Dollar

136. **VaultCore.rebase functions have no return statements**: `VaultCore.rebase()` and `VaultCore.rebase(bool)` return a `uint` but lack a return statement. As a result these functions will always return the default value, and are likely to cause issues for their callers. Both `VaultCore.rebase()` and `VaultCore.rebase(bool)` are expected to return a `uint256`. `rebase()` does not have a return statement. `rebase(bool)` has one return statement in one branch (return 0), but

lacks a return statement for the other paths. So both functions will always return zero. As a result, a third-party code relying on the return value might not work as intended.

1. Recommendation: Add the missing return statement(s) or remove the return type in `VaultCore.rebase()` and `VaultCore.rebase(bool)`. Properly adjust the documentation as necessary.
2. ToB's Audit of Origin Dollar

137. **Multiple contracts are missing inheritances**: Multiple contracts are the implementation of their interfaces, but do not inherit from them. This behavior is error-prone and might lead the implementation to not follow the interface if the code is updated.

1. Recommendation: Ensure contracts inherit from their interfaces
2. ToB's Audit of Origin Dollar

138. **Solidity compiler optimizations can be dangerous**: Yield Protocol has enabled optional compiler optimizations in Solidity. There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed . Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past . A high-severity bug in the emscripten -generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6 .

1. Recommendation: Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug. Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.
2. ToB's Audit of Yield Protocol

139. **Permission-granting is too simplistic and not flexible enough**: The Yield Protocol contracts implement an oversimplified permission system that can be abused by the administrator. The Yield Protocol implements several contracts that need to call privileged functions from each other. However, there is no way to specify which operation can be called for every privileged user. All the authorized addresses can call any restricted function, and the owner can add any number of them. Also, the privileged addresses are supposed to be smart contracts; however, there is no check for that. Moreover, once an address is added, it cannot be deleted.

1. Recommendation: Rewrite the authorization system to allow only certain addresses to access certain functions
2. ToB's Audit of Yield Protocol

140. **Lack of validation when setting the maturity value**: When a `fyDAI` contract is deployed, one of the deployment parameters is a maturity date, passed as a Unix timestamp. This is the date at which point `fyDAI` tokens can be redeemed for the underlying Dai. Currently, the contract constructor performs no validation on this timestamp to ensure it is within an acceptable range. As a result, it is possible to mistakenly deploy a `YDai` contract that has a maturity date in the past or many years in the future, which may not be immediately noticed.

    1. Recommendation: Short term, add checks to the `YDai` contract constructor to ensure maturity timestamps fall within an acceptable range. This will prevent maturity dates from being mistakenly set in the past or too far in the future. Long term, always perform validation of parameters passed to contract constructors. This will help detect and prevent errors during deployment.
    2. ToB's Audit of Yield Protocol

141. **Delegates can be added or removed repeatedly to bloat logs**: Several contracts in the Yield Protocol system inherit the Delegable contract. This contract allows users to delegate the ability to perform certain operations on their behalf to other addresses. When a user adds or removes a delegate, a corresponding event is emitted to log this operation. However, there is no check to prevent a user from repeatedly adding or removing a delegation that is already enabled or revoked, which could allow redundant events to be emitted repeatedly.

    1. Recommendation: Short term, add a `require` statement to check that the delegate address is not already enabled or disabled for the user. This will ensure log messages are only emitted when a delegate is activated or deactivated. Long term, review all operations and avoid emitting events in repeated calls to idempotent operations. This will help prevent bloated logs.
    2. ToB's Audit of Yield Protocol

142. **Lack of events for critical operations**: Several critical operations do not trigger events, which will make it difficult to review the correct behavior of the contracts once deployed. Users and blockchain monitoring systems will not be able to easily detect suspicious behaviors without events.

    1. Recommendation: Short term, add events where appropriate for all critical operations. Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts.

2. ToB's Audit of 0x Protocol

143. **_assertStakingPoolExists never returns true**: The `_assertStakingPoolExists` should return a bool to determine if the staking pool exists or not; however, it only returns false or reverts. The `_assertStakingPoolExists` function checks that a staking pool exists given a pool id parameter. However, this function does not use a return statement and therefore, it will always return false or revert.

    1. Recommendation: Add a return statement or remove the return type. Properly adjust the documentation, if needed.
    2. ToB's Audit of 0x Protocol

144. **_min* and _max* have unorthodox semantics**: Throughout the Curve contract, `_minTargetAmount` and `_maxOriginAmount` are used as open ranges (i.e., ranges that exclude the value itself). This contravenes the standard meanings of the terms "minimum" and "maximum," which are generally used to describe closed ranges.

    1. Recommendation: Short term, unless they are intended to be strict, make the inequalities in the require statements non-strict. Alternatively, consider refactoring the variables or providing additional documentation to convey that they are meant to be exclusive bounds. Long term, ensure that mathematical terms such as "minimum," "at least," and "at most" are used in the typical way—that is, to describe values inclusive of minimums or maximums (as relevant).
    2. ToB's Audit of DFX Finance

145. **CurveFactory.newCurve returns existing curves without provided arguments**: `CurveFactory.newCurve` takes values and creates a Curve contract instance for each `_baseCurrency` and `_quoteCurrency` pair, populating the Curve with provided weights and assimilator contract references. However, if the pair already exists, the existing Curve will be returned without any indication that it is not a newly created Curve with the provided weights. If an operator attempts to create a new Curve for a base-and-quote-currency pair that already exists, `CurveFactory` will return the existing Curve instance regardless of whether other creation parameters differ. A naive operator may overlook this issue.

    1. Recommendation: Consider rewriting `newCurve` such that it reverts in the event that a base-and-quote-currency pair already exists. A view function can be used to check for and retrieve existing Curves without any gas payment prior to an attempt at Curve creation.
    2. ToB's Audit of DFX Finance

146. **Missing zero-address checks in Curve.transferOwnership and Router.constructor**: Like other similar functions, `Curve._transfer`

and `Orchestrator.includeAsset` perform zero-address checks. However, `Curve.transferOwnership` and the Router constructor do not. This may make sense for `Curve.transferOwnership`, because without zero-address checks, the function may serve as a means of burning ownership. However, popular contracts that define similar functions often consider this case, such as OpenZeppelin's `Ownable` contracts. Conversely, a zero-address check should be added to the Router constructor to prevent the deployment of an invalid Router, which would revert upon a call to the zero address.

1. Recommendation: Short term, consider adding zero-address checks to the Router's constructor and Curve's `transferOwnership` function to prevent operator errors. Long term, review state variables which referencing contracts to ensure that the code that sets the state variables performs zero-address checks where necessary
2. ToB's Audit of DFX Finance

147. **safeApprove does not check return values for approve call**: Although the Router contract uses OpenZeppelin's `SafeERC20` library to perform safe calls to ERC20's approve function, the Orchestrator library defines its own `safeApprove` function. This function checks that a call to approve was successful but does not check returndata to verify whether the call returned true. In contrast, OpenZeppelin's `safeApprove` function checks return values appropriately. This issue may result in uncaught approve errors in successful Curve deployments, causing undefined behavior.

    1. Recommendation: Short term, leverage OpenZeppelin's `safeApprove` function wherever possible. Long term, ensure that all low-level calls have accompanying contract existence checks and return value checks where appropriate.
    2. ToB's Audit of DFX Finance

148. **ERC20 token Curve does not implement symbol, name, or decimals**: `Curve.sol` is an ERC20 token and implements all six required ERC20 methods: `balanceOf`, `totalSupply`, `allowance`, `transfer`, `approve`, and `transferFrom`. However, it does not implement the optional but extremely common view methods `symbol`, `name`, and `decimals`.

    1. Recommendation: Short term, implement `symbol`, `name`, and `decimals` on Curve contracts. Long term, ensure that contracts conform to all required and recommended industry standards.
    2. ToB's Audit of DFX Finance

149. **Insufficient use of SafeMath**: `CurveMath.calculateTrade` is used to compute the output amount for a trade. However, although `SafeMath` is used throughout the codebase to prevent underflows/overflows, it is not used in this calculation. Although we could not prove that the lack of

`SafeMath` would cause an arithmetic issue in practice, all such calculations would benefit from the use of `SafeMath`.

1. Recommendation: Review all critical arithmetic to ensure that it accounts for underflows, overflows, and the loss of precision. Consider using `SafeMath` and the safe functions of `ABDKMath64x64` where possible to prevent underflows and overflows.
2. ToB's Audit of DFX Finance

150. **setFrozen can be front-run to deny deposits/swaps**: Currently, a Curve contract owner can use the `setFrozen` function to set the contract into a state that will block swaps and deposits. A contract owner could leverage this process to front-run transactions and freeze contracts before certain deposits or swaps are made; the contract owner could then unfreeze them at a later time.

    1. Recommendation: Short term, consider rewriting `setFrozen` such that any contract freeze will not last long enough for a malicious user to easily execute an attack. Alternatively, depending on the intended use of this function, consider implementing permanent freezes.
    2. ToB's Audit of DFX Finance

151. **Account creation spam**: Hermez has a limit of `2**MAX_NLEVELS` accounts. There is no fee on account creation, so an attacker can spam the network with account creation to fill the tree. If `MAX_NLEVELS` is below 32, an attacker can quickly reach the account limit. If `MAX_NLEVELS` is above or equal to 32, the time required to fill the tree will depend on the number of transactions accepted per second, but will take at least a couple of months. Ethereum miners do not have to pay for account creation. Therefore, an Ethereum miner can spam the network with account creation by sending L1 user transactions.

    1. Recommendation: Short term, add a fee for account creation or ensure `MAX_NLEVELS` is at least 32. Also, monitor account creation and alert the community if a malicious coordinator spams the system. This will prevent an attacker from spamming the system to prevent new accounts from being created. Long term, when designing spam mitigation, consider that L1 gas cost can be avoided by Ethereum miners.
    2. ToB's Audit of Hermez Network

152. **Using empty functions instead of interfaces leaves contract error-prone**: `WithdrawalDelayerInterface` is a contract meant to be an interface. It contains functions with empty bodies instead of function signatures, which might lead to unexpected behavior. A contract inheriting from `WithdrawalDelayerInterface` will not require an override of these functions and will not benefit from the compiler checks on its correct interface.

1. Recommendation: Short term, use an interface instead of a contract in `WithdrawalDelayerInterface`. This will make derived contracts follow the interface properly. Long term, properly document the inheritance schema of the contracts. Use Slither's inheritance-graph printer to review the inheritance.
2. ToB's Audit of Hermez Network

153. **cancelTransaction can be called on non-queued transaction**: Without a transaction existence check in `cancelTransaction`, an attacker can confuse monitoring systems. `cancelTransaction` emits an event without checking that the transaction to be canceled exists. This allows a malicious admin to confuse monitoring systems by generating malicious events.

    1. Recommendation: Short term, check that the transaction to be canceled exists in `cancelTransaction`. This will ensure that monitoring tools can rely on emitted events. Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.
    2. ToB's Audit of Hermez Network

154. **Contracts used as dependencies do not track upstream changes**: Third-party contracts like `_concatStorage` are pasted into the Hermez repository. Moreover, the code documentation does not specify the exact revision used, or if it is modified. This makes updates and security fixes on these dependencies unreliable since they must be updated manually. `_concatStorage` is borrowed from the solidity-bytes-utils library, which provides helper functions for byte-related operations. Recently, a critical vulnerability was discovered in the library's slice function which allows arbitrary writes for user-supplied inputs.

    1. Recommendation: Short term, review the codebase and document each dependency's source and version. Include the third-party sources as submodules in your Git repository so internal path consistency can be maintained and dependencies are updated periodically. Long term, identify the areas in the code that are relying on external libraries and use an Ethereum development environment and NPM to manage packages as part of your project.
    2. ToB's Audit of Hermez Network

155. **Expected behavior regarding authorization for adding tokens is unclear**: `addToken` allows anyone to list a new token on Hermez. This contradicts the online documentation, which implies that only the governance should have this authorization. It is unclear whether the implementation or the documentation is correct.

1. Recommendation: Short term, update either the implementation or the documentation to standardize the authorization specification for adding tokens. Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.
2. ToB's Audit of Hermez Network

156. **Contract name duplication leaves codebase error-prone**: The codebase has multiple contracts that share the same name. This allows buidler-waffle to generate incorrect json artifacts, preventing third parties from using their tools. Buidler-waffle does not correctly support a codebase with duplicate contract names. The compilation overwrites compilation artifacts and prevents the use of third-party tools, such as Slither.

    1. Recommendation: Short term, prevent the re-use of duplicate contract names or change the compilation framework. Long term, use Slither, which will help detect duplicate contract names.
    2. ToB's Audit of Hermez Network

157. **Use of hard-coded addresses may cause errors**: Each contract needs contract addresses in order to be integrated into other protocols and systems. These addresses are currently hard-coded, which may cause errors and result in the codebase's deployment with an incorrect asset. Using hard-coded values instead of deployer-provided values makes these contracts incredibly difficult to test.

    1. Recommendation: Short term, set addresses when contracts are created rather than using hard-coded values. This practice will facilitate testing. Long term, to ensure that contracts can be tested and reused across networks, avoid using hard-coded parameters.
    2. ToB's Audit of Advanced Blockchains

158. **Borrow rate depends on approximation of blocks per year**: The borrow rate formula uses an approximation of the number of blocks mined annually. This number can change across different blockchains and years. The current value assumes that a new block is mined every 15 seconds, but on Ethereum mainnet, a new block is mined every 13 seconds. To calculate the base rate, the formula determines the approximate borrow rate over the past year and divides that number by the estimated number of blocks mined per year. However, `blocksPerYear` is an estimated value and may change depending on transaction throughput. Additionally, different blockchains may have different block-settling times, which could also alter this number.

    1. Recommendation: Short term, analyze the effects of a deviation from the actual number of blocks mined annually in borrow rate calculations and document the associated risks. Long term, identify all vari-

ables that are affected by external factors, and document the risks associated with deviations from their true values.

2. ToB's Audit of Advanced Blockchains

159. **Flash loan rate lacks bounds and can be set arbitrarily**: There are no lower or upper bounds on the flash loan rate implemented in the contract. The Blacksmith team could therefore set an arbitrarily high flash loan rate to secure higher fees. The Blacksmith team sets the `_flashLoanRate` when the Vault is first initialized. The `blackSmithTeam` address can then update this value by calling `updateFlashloanRate`. However, because there is no check on either setter function, the flash loan rate can be set arbitrarily. A very high rate could enable the Blacksmith team to steal vault deposits.

    1. Recommendation: Short term, introduce lower and upper bounds for all configurable parameters in the system to limit privileged users' abilities. Long term, identify all incoming parameters in the system as well as the financial implications of large and small corner-case values. Additionally, use Echidna or Manticore to ensure that system invariants hold.
    2. ToB's Audit of Advanced Blockchains

160. **Logic duplicated across code**: The logic in the repositories provided to Trail of Bits contains a significant amount of duplicated code. This development practice increases the risk that new bugs will be introduced into the system, as bug fixes must be copied and pasted into files across the system.

    1. Recommendation: Short term, use inheritance to allow code to be reused across contracts. Changes to one inherited contract will be applied to all files without requiring developers to copy and paste them. Long term, minimize the amount of manual copying and pasting required to apply changes made to one file to other files.
    2. ToB's Audit of Advanced Blockchains

161. **Insufficient testing**: The repositories under review lack appropriate testing, which increases the likelihood of errors in the development process and makes the code more difficult to review.

    1. Recommendation: Short term, ensure that the unit tests cover all public functions at least once, as well as all known corner cases. Long term, integrate coverage analysis tools into the development process and regularly review the coverage.
    2. ToB's Audit of Advanced Blockchains

162. **Project dependencies contain vulnerabilities**: Although dependency scans did not yield a direct threat to the projects under review, yarn audit

identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review.

1. Recommendation: Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation. Long term, consider integrating automated dependency auditing into the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency.
2. ToB's Audit of Advanced Blockchains

163. **Lack of contract documentation makes codebase difficult to understand**: The codebase lacks code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes. The documentation would benefit from more detail.

    1. Recommendation: Short term, review and properly document the above mentioned aspects of the codebase. Long term, consider writing a formal specification of the protocol.
    2. ToB's Audit of Advanced Blockchains

164. **ABIEncoderV2 is not production-ready**: The contracts use the new Solidity ABI encoder, `ABIEncoderV2`. This experimental encoder is not ready for production. More than 3% of all GitHub issues for the Solidity compiler are related to experimental features, primarily `ABIEncoderV2`. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with more than 20 high-severity bugs, some of which are so recent that they have not yet been included in a Solidity release. For example, in March 2019 a severe bug introduced in Solidity 0.5.5 was found in the encoder.

    1. Recommendation: Short term, use neither `ABIEncoderV2` nor any other experimental Solidity feature. Refactor the code such that structs do not need to be passed to or returned from functions. Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.
    2. ToB's Audit of Advanced Blockchains

165. **Contract owner has too many privileges**: The owner of the contracts has too many privileges relative to standard users. Users can lose all of their assets if a contract owner private key is compromised. The contract

owner can do the following: 1) Upgrade the system's implementation to steal funds 2) Upgrade the token's implementation to act maliciously 3) Increase the amount of `iTokens` for reward distribution to such an extent that rewards cannot be disbursed 4) Arbitrarily update the interest model contracts The concentration of these privileges creates a single point of failure. It increases the likelihood that the owner will be targeted by an attacker, especially given the insufficient protection on sensitive owner private keys. Additionally, it incentivizes the owner to act maliciously.

1. Recommendation: Short term: 1) Clearly document the functions and implementations the owner can change. 2) Split privileges to ensure that no one address has excessive ownership of the system. Long term, document the risks associated with privileged users and single points of failure. Ensure that users are aware of all the risks associated with the system.
2. ToB's Audit of dForce Lending

166. **Poor error-handling practices in test suite**: The test suite does not properly test expected behavior, as the contracts run in production. Additionally, certain components lack error-handling methods. These deficiencies can cause failed tests to be overlooked. In particular, the tests fail to properly check error messages. For example, errors are silenced with a try-catch statement. If this error is silenced, there will be no guarantee that a smart contract call has reverted for the right reason. As a result, if the test suite passes, it will provide no guarantee that the transaction call reverted correctly.

    1. Recommendation: Short term, test these operations against a specific error message. Testing will ensure that errors are never silenced, and the test suite will check that a contract call has reverted for the right reason. Long term, follow standard testing practices for smart contracts to minimize the number of issues during development.
    2. ToB's Audit of dForce Lending

167. **Redundant and Unused Code**: The `_recordLoanClosure()` function returns a boolean (`loanClosed`) which is never used by the calling function (see `_closeLoan()` , line [312]). Furthermore, since the `_recordLoanClosure()` function is only called via the `_closeLoan()` function, this means that `synthLoan.timeClosed` is always equal to zero (see `require` statement on line [305]). Therefore, the if statement on line [357] is redundant and unnecessary.

    1. Recommendation: 1) Using the return value of the `_recordLoanClosure()` function or changing the function definition to stop returning `loanClosed` 2) Removing the if statement in line [357
    2. Sigma Prime's Audit of Synthetix EtherCollateral

168. **Single Account Can Capture All Supply**: The `EtherCollateral` smart contract does not rely on a `maxLoanSize` to limit the amount of ETH that can be locked for a loan. As a result, a single account can issue a loan that will reach the total minting supply.

    1. Recommendation: Make sure this behaviour is understood and consider introducing and enforcing a cap (`maxLoanSize`) on the size of the loans allowed to be opened.
    2. Sigma Prime's Audit of Synthetix EtherCollateral

169. **Insufficient Input Validation**: The constructor of the `EtherCollateral` smart contract does not check the validity of the addresses provided as input parameters. It is possible to deploy an instance of the `EtherCollateral` contract with the `synthProxy` , `sUSDProxy` and depot addresses set to zero. Similarly, the effective interest rate can be equal to zero if `interestRate` is set to any value lesser than 31536000 (`SECONDS_IN_A_YEAR`), as `interestPerSecond` will be null.

    1. Recommendation: Consider introducing require statements to perform adequate input validation.
    2. Sigma Prime's Audit of Synthetix EtherCollateral

170. **Unused Event Logs**: log events are declared but never emitted.

    1. Recommendation: Remove these events from the EtherCollateral contract.
    2. Sigma Prime's Audit of Synthetix EtherCollateral

171. **Possible Unintended Token Burning in transferFrom() Function**: `InfiniGold` allows users to convert/exchange their PMGT tokens to "gold certificates", which are digital artefacts effectively redeemable for actual gold. To do so, users are supposed to send their PMGT tokens to a specific burn address. The `transferFrom()` function does not check the to address against this burn address. Users may send tokens to the burn address, using the `transferFrom()` function, without triggering the emission of the `Burn(address indexed burner, uint256 value)` event, which dictates how the gold certificates are created and distributed.

    1. Recommendation: Prevent sending tokens to the burn address in the `transferFrom()` function. This can be achieved by adding a `require` within `transferFrom()` which disallows the to address to be the `burnAddress` .
    2. Sigma Prime's Audit of InfiniGold

172. **Denial of Service Vector from Unbound List**: The `reset()` internal function (called by the `replaceAll()` function) resets the role linked list

by deleting all the elements (i.e. nodes) part of the bearer mapping. The caller is bound by the number of elements that are being removed for a particular role . Calling the `reset()` function will exceed the current block gas limit (i.e. 8,000,0000) for more than 371 total elements in a role linked list. Similarly, the `size()` and `toArray()` functions also loop through the linked list. This essentially means that listers, unlisters, minters, pausers, unpausers and owners can perform denial of service attacks on the lists they administer. In a scenario where the Roles library is leveraged by other smart contracts, calling these two functions will also result in a potential denial of service after a certain number of elements have been included in the linked list (this number would depend on the gas cost of the Opcodes implemented by the calling functions).

1. Recommendation: One way to ensure that the current block gas limit is not exceeded would be to introduce a condition in the `add()` function to check that the linked list size is strictly lesser than 371 elements before adding a new element. This additional condition would significantly increase the gas cost associated with calling the `add()` function, as a call to the `size()` function would be required to fetch the exact number of nodes in the linked list. Alternatively, the `gasleft()` Solidity special function could be used to make sure that going through the linked list does not exceed the block gas limit. Finally, the `reset()` could be changed to allow for removing an arbitrary number of nodes (by taking this number as a function parameter).
2. Sigma Prime's Audit of InfiniGold

173. **ERC20 Implementation Vulnerable to Front-Running**: Front-running attacks involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction. The ERC20 implementation is known to be affected by a front-running vulnerability, in its `approve()` function.

    1. Recommendation: Be aware of the front-running issues in `approve()` , potentially add extended approve functions which are not vulnerable to the front-running vulnerability for future third-party-applications. See the Open-Zeppelin [8] solution for an example. We note that modifying the ERC20 standard to address this issue may lead to backward incompatibilities with external third-party software.
    2. Sigma Prime's Audit of InfiniGold

174. **Unnecessary require Statement**: The following `require` statement in `Blacklistable.sol` can be removed: `require(to != address(0));` Indeed, this check is implemented in the `_transfer()` function in the `ERC20.sol` smart contract.

1. Recommendation: Consider removing the require statement for gas saving purposes.
2. Sigma Prime's Audit of InfiniGold

175. **Rounding to Zero if Duration is Greater Than Reward**: The `rewardRate` value is calculated as follows: `rewardRate = reward/duration`. Due to the integer representation of these variables, if duration is larger than reward the value of `rewardRate` will round to zero. Thus, stakers will not receive any of the reward for their stakes. Furthermore, due to the integer rounding, the total rewards distributed may be rounded down by up to one less than duration . As a result, the Unipool contract may slowly accumulate SNX.

    1. Recommendation: Beware of the rounding issues when calling the `notifyRewardAmount()` function. We also recommend some way of allowing the excess SNX reward from rounding to be claimed or withdrawn from the Unipool contract.
    2. Sigma Prime's Audit of Synthetix Unipool

176. **Withdrawn Event Log Poisoning**: Calling the `withdraw()` function will emit the Withdrawn event. No UNI tokens are required as this function can be called with `amount = 0`. As a result a user could continually call this function, creating a potentially infinite amount of events. This can lead to an event log poisoning situation where malicious external users spam the Unipool contract to generate arbitrary Withdrawn events.

    1. Recommendation: Consider adding a `require` or `if` statement preventing the `withdraw()` function from emitting the Withdrawn event when the amount variable is zero.
    2. Sigma Prime's Audit of Synthetix Unipool

177. **Insufficient incentives to liquidator**: The liquidation process is a very important part of every DeFi project because it allows to extinguish the problem of having the whole system under-collateralized under critical conditions of the market, and it needs a design that incentivizes its speed of execution. The Holdefi contract implements the liquidation process for those accounts that may have an under-collateralized balance or that may have been inactive for a whole year without interacting with the project. The liquidator would end up paying for the expensive liquidation process, without receiving any benefit. Buying discounted collateral assets could be considered as an incentive to the liquidators

    1. Recommendation: Consider improving the incentive design to give the liquidators higher incentives to execute the liquidation process
    2. OpenZeppelin's Audit of HoldeFi

178. **Markets can become insolvent**: When the value of all collateral is worth less than the value of all borrowed assets, we say a market is insolvent. The Holdefi codebase can do many things to reduce the risk of market insolvency, including: prudent selection of collateral-ratios, incentivizing third-party collateral liquidation, careful selection of which tokens are listed on the platform, etc. However, the risk of insolvency cannot be entirely eliminated, and there are numerous ways a market can become insolvent.

    1. Recommendation: This risk is not unique to the Holdefi project. All collateralized loans (even non-blockchain loans) have a risk of insolvency. However, it is important to know that this risk does exist, and that it can be difficult to recover from even a small dip into insolvency. Consider adding more targeted tests for these scenarios to better understand the behavior of the protocol, and designing relevant mechanics to make sure the platform operates properly. Also consider communicating the potential risks to the users if needed.
    2. OpenZeppelin's Audit of HoldeFi

179. **Not using OpenZeppelin contracts**: OpenZeppelin maintains a library of standard, audited, community-reviewed, and battle-tested smart contracts. Instead of always importing these contracts, the Holdefi project reimplements them in some cases, while in other cases it just copies them. This increases the amount of code that the Holdefi team will have to maintain and misses all the improvements and bug fixes that the OpenZeppelin team is constantly implementing with the help of the community.

    1. Recommendation: Consider importing the OpenZeppelin contracts instead of reimplementing or copying them. These contracts can be extended to add the extra functionalities required by Holdefi.
    2. OpenZeppelin's Audit of HoldeFi

180. **Lack of indexed parameters in events**: Throughout the Holdefi's codebase, none of the parameters in the events defined in the contracts are indexed.

    1. Recommendation: Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.
    2. OpenZeppelin's Audit of HoldeFi

181. **Named return variables**: There is an inconsistent use of named return variables across the entire codebase.

    1. Recommendation: Consider removing all named return variables, explicitly declaring them as local variables in the body of the function,

and adding the necessary explicit return statements where appropriate. This should favor both explicitness and readability of the project.

2. OpenZeppelin's Audit of HoldeFi

182. **block.timestamp Unreliable**: Code uses the `block.timestamp` as part of the calculations and time checks. Nevertheless, timestamps can be slightly altered by miners to favor them in contracts that have logics that depend strongly on them.

    1. Recommendation: Consider taking into account this issue and warning the users that such a scenario could happen. If the alteration of timestamps cannot affect the protocol in any way, consider documenting the reasoning and writing tests enforcing that these guarantees will be preserved even if the code changes in the future.
    2. OpenZeppelin's Audit of HoldeFi

183. **Assignment in require statement**: In the `YieldOracle` contract, there is a `require` statement that makes an assignment. This deviates from the standard usage and intention of `require` statements and can easily lead to confusion.

    1. Recommendation: Consider moving the assignment to its own line before the `require` statement and then using the `require` statement solely for condition checking.
    2. OpenZeppelin's Audit of BarnBrige Smart Yield Bonds

184. **Commented code**: Throughout the codebase there are lines of code that have been commented out with //. This can lead to confusion and is detrimental to overall code readability.

    1. Recommendation: Consider removing commented out lines of code that are no longer needed.
    2. OpenZeppelin's Audit of BarnBrige Smart Yield Bonds

185. **Misleading revert messages**: Error messages are intended to notify users about failing conditions, and should provide enough information so that the appropriate corrections needed to interact with the system can be applied. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality.

    1. Recommendation: Consider not only fixing the specific issues mentioned, but also reviewing the entire codebase to make sure every error message is informative and user-friendly enough. Furthermore, for consistency, consider reusing error messages when extremely similar conditions are checked.
    2. OpenZeppelin's Audit of Compound Governor Bravo

186. **Multiple outdated Solidity versions in use**: Outdated versions of Solidity are being used in all contracts. The compiler options in the truffle-config file specifies version 0.6.6, which was released on April 6, 2020. Throughout the codebase there are also different versions of Solidity being used.

    1. Recommendation: As Solidity is now under a fast release cycle, consider using a more recent version of the compiler, such as version 0.7.6. In addition, to avoid unexpected behavior, consider specifying explicit Solidity versions in pragma statements.
    2. OpenZeppelin's Audit of Fei Protocol

187. **Test and production constants in the same codebase**: The `CoreOrchestrator` contract defines the `TEST_MODE` boolean variable which is used to define several constants in the system. This decreases legibility of production code, and makes the system's integral values more error-prone.

    1. Recommendation: Consider having different environments for production and testing, with different contracts.
    2. OpenZeppelin's Audit of Fei Protocol

188. **Unnecessarily small integer sizes**: In Solidity, using integers smaller than 256 bits tends to increase gas costs because the Ethereum Virtual Machine must perform additional operations to zero out the unused bits. This can be justified by savings in storage costs in some scenarios, however, that is not generally the case in this codebase.

    1. Recommendation: Consider using integers of size 256 bits to improve gas efficiency and mitigate function reverts.
    2. OpenZeppelin's Audit of Fei Protocol

189. **Use of uint instead of uint256**: Across the codebase, there are hundreds of instances of `uint`, as opposed to `uint256`.

    1. Recommendation: In favor of explicitness, consider replacing all instances of `uint` with `uint256`.
    2. OpenZeppelin's Audit of Fei Protocol

190. **Functions with unexpected side-effects**: Some functions have side-effects. For example, the `_getLatestFundingRate` function of the `FundingRateApplier` contract might also update the funding rate and send rewards. The getPrice function of the `OptimisticOracle` contract might also settle a price request. These side-effect actions are not clear in the name of the functions and are thus unexpected, which could lead to mistakes when the code is modified by new developers not experienced in all the implementation details of the project.

1. Recommendation: Consider splitting these functions in separate getters and setters. Alternatively, consider renaming the functions to describe all the actions that they perform.
2. OpenZeppelin's Audit of Uma Phase 4

191. **Unsafe casting**: In line 554 of the `TaxCollector` contract, the value of `coinBalance(receiver)` is an `uint`. This is cast to an `int` and then negated. However, since `uint` can store higher values than `int`, it is possible that casting from `uint` to `int` may create an overflow.

    1. Recommendation: Consider verifying that the value of `coinBalance(receiver)` is within the acceptable range for negative `int` values before casting and negating. Consider using OpenZeppelin's `SafeCast` contract, which provides functions for safely casting between types.
    2. OpenZeppelin's Audit of GEB Protocol

192. **Unsafe division in rdivide and wdivide functions**: The function `rdivide` on line 227 and the function `wdivide` on line 230 of the `GlobalSettlement` contract, accept the divisor `y` as an input parameter. However, these functions do not check if the value of `y` is 0. If that is the case, the call will revert due to the division by zero error.

    1. Recommendation: To prevent such unsafe calculations, consider adding a `require` statement in the functions to ensure `y > 0`, or consider using the `div` functions provided in OpenZeppelin's `SafeMath` libraries.
    2. OpenZeppelin's Audit of GEB Protocol

193. **Uncommented assembly block**: The `OracleRelayer` contract includes an assembly block in the `rpower()` function. The same assembly block is repeated in the `TaxCollector` and `CoinSavingsAccount` contracts. While this does not pose a security risk per se, it is at the same time a complicated and critical part of the system. Moreover, as this is a low-level language that is harder to parse by readers, consider including extensive documentation regarding the rationale behind its use, clearly explaining what every single assembly instruction does. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it. Note that the use of assembly discards several important safety features of Solidity, which may render the code unsafer and more error-prone.

    1. Recommendation: Consider implementing thorough tests to cover all potential use cases of these functions to ensure they behave as expected.
    2. OpenZeppelin's Audit of GEB Protocol

194. **Unnecessary require statements**: There are several instances in the code base where the `require` statements or conditional checks are unnecessary. For instance: In the `OracleRelayer` contract, the `require` statement in the `modifyParameters` function at line 189 checks if the input parameter data > 0. This is unnecessary since the same condition is already checked in the `require` statement at line 187.

    1. Recommendation: To simplify the code and prevent wastage of gas, consider removing the unnecessary checks.
    2. OpenZeppelin's Audit of GEB Protocol

195. **Unnecessary event emission**: The `popDebtFromQueue` function of the `AccountingEngine` contract is emitting a useless event whenever someone tries to call it with a `debtBlockTimestamp` that has not been saved before.

    1. Recommendation: To simplify the code and prevent wastage of gas, avoid emitting unnecessary events.
    2. OpenZeppelin's Audit of GEB Protocol

196. **oToken can be created with a non-whitelisted collateral asset**: A product consists of a set of assets and an option type. Each product has to be whitelisted by the admin using the `whitelistProduct` function from the Whitelist contract.

    1. Recommendation: Consider validating if the assets involved in a product have been already whitelisted before allowing the creation of `oTokens`.
    2. OpenZeppelin's Audit of Opyn Gamma Protocol

197. **Mismatches between contracts and interfaces**: Interfaces define the exposed functionality of the implemented contracts. However, in several interfaces there are functions from the counterpart contracts that are not defined.

    1. Recommendation: Consider applying the necessary changes in the mentioned interfaces and contracts so that definitions and implementations fully match.
    2. OpenZeppelin's Audit of Opyn Gamma Protocol

198. **Actions not executed atomically might lead to inconsistent state**: The `setAssetPricer`, `setLockingPeriod`, and `setDisputePeriod` functions of the Oracle contract execute actions that are always expected to be performed atomically. Failing to do so can lead to inconsistent states in the system.

    1. Recommendation: Consider implementing an additional function that calls the `setAssetPricer`, `setLockingPeriod`, and `setDisputePeriod` functions, so that these actions can be executed atomically in a single transaction.

2. OpenZeppelin's Audit of Opyn Gamma Protocol

199. **Chainlink pricer is using a deprecated API**: The Chainlink Pricer is currently using multiple functions from a deprecated Chainlink API such as `latestAnswer()` in L61, `getTimestamp()` in L74. These functions might suddenly stop working if Chainlink stopped supporting deprecated APIs.

    1. Recommendation: Consider refactoring these to use the latest Chainlink API.
    2. OpenZeppelin's Audit of Opyn Gamma Protocol

200. **Funds can be lost**: The `sweepTimelockBalances` function accepts a list of users with unlocked balances to distribute. However, if there are duplicate users in the list, their balances will be counted multiple times when calculating the total amount to withdraw from the yield service.

    1. Recommendation: Consider checking for duplicate users when calculating the amount to withdraw.
    2. OpenZeppelin's Audit of PoolTogether V3

201. **Use delete to clear variables**: The Controller contract sets a variable to the zero address in order to clear it. Similarly, the `SetToken` clears the locker by assigning the zero address.

    1. Recommendation: The `delete` key better conveys the intention and is also more idiomatic. Consider replacing assignments of zero with `delete` statements.
    2. OpenZeppelin's Audit of Set Protocol