Introduction to Ethereum, Smart Contracts and Their Exploitation

Bc. Vít Barták
Faculty of Information Technology
Brno University of Technology
Brno, Czech republic
xbarta47@stud.fit.vutbr.cz

***Abstract*—This paper serves as a practical introduction and summary of the basic exploitation of Ethereum smart contracts. We also present how to prevent such security flaws in Solidity source code.**

At first, we provide context to cryptocurrencies, namely Bitcoin and Ethereum, programming smart contracts in Solidity and the background of hacking Ethereum smart contracts. We then present one famous type of vulnerability in Ethereum smart contracts: reentrancy attack and how variants of this vulnerability work. We also briefly talk about other security flaws.

The paper then continues with a quick tutorial on how-to setup Solidity development environment for smart contract programming. Examples of Solidity code with vulnerable smart contract is presented, then the patched version is shown. We end the paper with a practical showcase of exploitation at runtime.

Keywords—ethereum, smart contracts, hacking, reentrancy attack, solidity

I. Introduction

A. Bitcoin

In 2009 an unknown person or group of people under the pseudonym Satoshi Nakamoto released the source code for Bitcoin. This peer-to-peer electronic cash was revolutionary in the way, that it solved the double-spending problem that haunted decentralized digital money that came before it (such as Bit Gold) [1]. Bitcoin solved the double-spending problem by introducing a new distributed database that heavily relied on cryptography to serve as a one source of truth for the distributed network of computers [2].

Bitcoin's blockchain is a distributed, public, append-only ledger that consists of *blocks* which contain transactions of Bitcoin between its users, timestamps, and other metadata.

These blocks are linked together by a way of cryptographic hashing: the next block in the line contains a hash of the previous block. This means that if one were to alter some block in the history, one would need to also alter every block that came after it, since the hashes in the subsequent blocks would be incorrect. This property alone does not make Bitcoin secure, because hashing is a cheap computing operation for current computers and could be altered quickly [2].

To provide one source of truth for the p2p network, there's so called *Nakamoto Consensus* algorithm in place that governs the addition of new blocks to the blockchain by the users. New blocks are to be found (mined) by the network every 10 minutes (in average). This mining is done by *miners*: computers, that solve artificial computational challenge (proof of work) to prove that work has been done on mining the block – resources were spent similarly to mining gold. The exact workings of the Nakamoto consensus can be found in [2] and are out of scope of this paper.

Important role of miners is also to gather transactions from the network and put them inside new blocks. For the miner's work, they get rewarded by newly minted bitcoins and transactional fees. By design, the blockchain can fork and so the clients consider the longest chain the true one – most computational work has been put into it [1].

Bitcoin also incorporates the idea of using asymmetric cryptography for accessing one's wallet on the blockchain – signing transactions as is classically done in digital signature applications [2]. This means that if a person loses their private key, they are locked out of the access to their Bitcoin forever.

While Bitcoin revolutionized p2p payment using public network and circumventing state-organized banking, creating the whole cryptocurrency industry, it still has its shortcomings. Bitcoin is slow for transacting between lots of users at once – it can on average process 7 transactions in a second. Bitcoins climate impact is extremely big, and electricity is wasted on Bitcoin mining [3].

Bitcoin comes with relatively simple scripting language that is by design Turing-incomplete. This means that users can program only basic transactional logic in it – such as multi-signature transactions.

B. Ethereum

In 2014 Vitalik Buterin and Gavin Wood invented Ethereum [4]. Ethereum is a distributed, open-source, public computational platform which heavily relies on the idea of blockchain that Bitcoin has brought. Ethereum is not just a cryptocurrency however. There is ether (ETH), a native coin that serves for paying fees and as a basic currency unit. But we can think of Ethereum as a global decentralized computer (state machine) that the public can use to work with (not only) decentralized finance, while paying fees for doing so [5].

Besides using it as a p2p cash (like Bitcoin), users can also use Ethereum to store arbitrary data on the blockchain in a general way. This comes with price. Also, important to note is that Ethereum is Turing-complete and can be programmed via smart contracts [5].

Smart contracts are computer programs that are embedded in the Ethereum blockchain and can be triggered (executed) when user (or another contract) interacts with them via a transaction (a remote procedure call). In the simplest terms, we can think of smart contracts as an intelligent automated vending machine [5].

Once a smart contract is deployed onto the blockchain its bytecode is public and it cannot be edited or taken down – same as Bitcoins transactions are immutable. This also means, that as long as Ethereum platform remains operational, the automatic contract remains operational too. Example of smart contracts can be decentralized exchanges, crowdfunding apps, lending apps, stable coins etc. [5].

While Bitcoin is a distributed ledger, Ethereum is a distributed state machine. The *State* is a data structure that holds all balances of all accounts, and a machine state. This State changes with each new block and rules of how the State can be changed is defined by the Ethereum Virtual Machine (EVM) [6][8].

At given block, Ethereum has one true *canonical state* and EVM is used to define rules of computing new valid state from block to block. There are several OPCODES (operations) that can tell how the State should change. For example, we can write data into the State, read data from the State, create transaction of tokens from Alice's account to Bob's etc. The EVM runs on every Ethereum node and modifies its State deterministically driven by transactions. This means that globally, all Ethereum nodes are on the same state at the same time (block) [6][8].

Bitcoin was by design Turing-incomplete because it had no way of preventing malicious or careless users of getting it into an infinite loop and crashing the whole network. Ethereum solves this problem by the idea of using *gas*. Every OPCODE has a gas price. This means, that for every operation in a smart contract that the users wants it to compute, they must pay price in ether for it. It then can happen that the computation of the contract "runs out of gas" and fails – not returning the spent gas to the user. This way if someone were to run for example an infinite loop, the contract would simply fail after running out of gas [7][8].

The depth of the formal Ethereum specification is out of this paper's scope and can be found in the Ethereum Yellow Paper [8].

C. Solidity

Ethereum smart contracts are normally programmed in Solidity: a statically typed programming language focused on smart contracts. Solidity is object oriented inspired by JavaScript, C++, and Python. It supports inheritance, libraries, complex user-defined types. In Solidity, classes are replaced with *contracts*. Solidity is using JavaScript-like syntax, which makes it easy to learn for current web developers [9].

The result after compiling code in solidity is an EVM *bytecode* (similarly to JVM) that can be deployed onto Ethereum via a special type of deployment transaction. The Ethereum node's EVM then runs this bytecode and interacts with the Ethereum State as defined by [8].

Solidity is also being used to write contracts on other EVM-compatible platforms such as Polygon, Binance Smart Chain, and Tron for example.

Ethereum smart contracts can also be programmed with Vyper, another contract-oriented Python-like language that also targets the EVM. In this paper, we shall deal with Solidity only.

D. ERC-20 token

On Ethereum, users can program their own cryptocurrency token. For this purpose, the ERC-20 Token Standard is used [10].

This standard defines the interface (API) of a smart contract that works as a custom token. This token can represent anything real or virtual that can be tokenized (anything from fiat currency through shares in a company to character experience in an online videogame). This type of smart contract is also one of the most deployed smart contracts on Ethereum (over 400 thousand by 2022) that hold hundreds of millions of dollars [11]. It is vital, then, that security is one of the most important things in the smart contract development life cycle.

The ERC-20 standard defines an interface (methods) that every ERC-20 token must implement and comply with due to the integration with Ethereum wallets and communication with other contracts. Note that the specific implementation of these methods is not strictly forced, and this is where the most vulnerabilities can spring up [10].

Once a token is deployed to Ethereum, there is no way (normally) to take it back and update it once a flaw is found. This means that developers must be sure that their contract does not contain vulnerabilities that can later be exploited. This is not always the case.

Note that upgradeable contracts already exist – contract developers are using tricks to evade Ethereum's native immutability – which is naturally related to trust. This can be achieved in various ways: Contract migration, Data Separation and Proxy patterns to name few (for detailed information about upgradeable tokens see [12]).

Upgradeable contracts however are harder to program, maintain and users of the contract can have obvious trust issues if their money is reliant on a contract that can change at any time [13].

E. Notable hacks on Ethereum

Most smart contracts on Ethereum hold some financial value in them (as do vending machines). These can be hundreds of thousands or millions of dollars worth of value in cryptocurrencies or other cryptocurrency assets (ERC-721 tokens etc.).

Ethereum and EVM compatible platform's smart contracts are notorious for being hacked. In Table 1 are some famous hacks to date (the list is cherry picked).

Year	**Smart Contracts**		
	Name	***Type of hack***	***Value stolen (at the time)***
2016	The DAO Hack	Reentrancy Attack	$70 million
2020	Lendf.me	Reentrancy Attack	$25 million

Year	Smart Contracts		
	Name	***Type of hack***	***Value stolen (at the time)***
2021	Cream Finance	Reentrancy Attack	$19 million
2021	Ronin Network	Stolen private keys	$614 million
2022	Qubit Bridge	Wrong input sanitization	$80 million

Table 1 – Famous hacked smart contracts

The DAO hack was so big at its time, that Ethereum Foundation decided to alter the blockchain and do a fork, where the hack never happened. This spawned a lot of controversy about the "decentralization" of Ethereum and new cryptocurrency Ethereum Classic was created, where the hack was not removed [14].

We can see that the reentrancy attack is still quite popular because developers after all these years still do not watch out for it in their code (it's not always easily visible as is in our example). We will show how the reentrancy attack works in the next chapter.

II. Reentrancy Attack

A. The Basic Mechanism

Reentrancy attack works on the principle, that the exploit contract calls the exploited contract's method repeatedly before the first call of a method is finished [15]. This means that the computation of a method is interrupted, the State has not been changed and the control flow is given to external contract. Let's demonstrate this on an example (see Fig 1.)

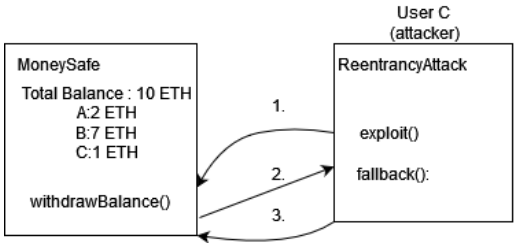

Figure 1- Reentrancy attack mechanism

Imagine a hypothetical legitimate smart contract called *MoneySafe*, where users can deposit their ether, they can also withdraw their ETH or check their balance. The withdraw function can be called by user and works as such:

1. Check if the user has any balance in MoneySafe.
2. Withdraw all the user's funds (send it to their Ethereum address).
3. Set user's balance equal to 0.

Now imagine a second smart contract called *ReentrancyAttack (RA)*. In its simplest form, this contract has two methods, exploit() method and fallback() method.

The exploit method basically calls the withdrawBalance() method from the MoneySafe contract.

The fallback method is called everytime a message with no arguments is received by a contract, or when no other function matches. The reentrancy attack works as follows:

1. User C calls exploit() method in the ReentrancyAttack contract.
2. exploit() deposits 1 ETH from its own balance to MoneySafe. Then requests to withdraw the users balance from MoneySafe by calling the withdrawBalance() method (step 1 in Fig 1.).
3. MoneySafe contract begins to compute the withdrawBalance() method by checking if the User C (RA) has any balance – they do, they just deposited 1 ETH beforehand.
4. The MoneySafe contract withdraws all of the User C's balance by sending it (we will see how in the code section) to the ReentrancyAttacks address (step 2 in Fig. 1).
5. While the withdraw function is computed, the MoneySafe contract computation holds and waits for the reaction of the ReentrancyAttack contract. But the Ether is already sent onto the address of the attacker (address of the RA contract).
6. While MoneySafe waits for the withdrawal function to finish, the fallback() function is triggered by the call from the withdrawal function. The fallback() function calls the MoneySafe again and asks to withdrawBalance() (this can be seen in step 3 of Fig 1.).
7. Because the balance of the user in the withdrawal function is set to zero only after the ether is already sent and the original function call is stuck before this, the MoneySafe contract sees that the User C still has balance and withdraws it again (from the native ether balance) triggering the fallback() function again and thus getting into a death spiral that drains all its funds.

The more obvious fix for this is to set the balance of the user to zero just before we send them the ETH from the balance of the contract. This is called *checks-effect-interaction* (CEI) pattern [15] and is the preferred way of writing smart contracts in general. Less sophisticated defence is to just set *gaslimit* of the contract – the reentrancy loop will fail after only so many iterations – this is not to be preferred.

A sophisticated defence is using the CEI pattern in combination with other countermeasures: for example, using the so called ReentrancyGuard *modifier*. Modifiers are a Solidity property that make changes to how a function behaves. Modifiers are reusable and general (we will show this in the code chapter). We can think of a modifier as a shell that wraps the function and adds prerequisites to it.

B. Other vulnerabilities in Smart Contracts

In 2022, researchers found a novel reentrancy attack type called *Read Only reentrancy* [16]. This attack is bit more complex than the basic type. It can be exploited when different protocols interact between each other. In basic terms: instead of draining the victim of the attack directly, it uses two contracts, one "legitimate" contract that works in the correct way with the victim, but makes it get stuck on a method call. And the second contract takes advantage of this and reads some property from the yet not updated victim contract. This can be used to get incorrect prices / exchange rates on DeFi protocols (because the total supply of a token is changed, but not yet propagated everywhere).

Another type of attack is *Oracle manipulation* [17]. Oracles are providers of *off-chain* (real-world) information,

and they themselves are mostly smart-contracts. Contracts connected to oracle can rely on it for source of truth, but if attackers can find a vulnerability in such oracle, they can manipulate lots of smart contracts at once.

Frontrunning [17] is an exploitation of the fact that all transactions are public and can be seen before being included in a block (new block is mined every 12 seconds on average in Ethereum). But there are transactions in the *mempool* already. This can be exploited in decentralized exchanges (such as Uniswap) when a large buy or sell order transaction is executed, attackers can see it in the mempool and frontrun it – for example selling large quantity and crashing the price before the front ran transaction goes through, then quickly buying the large sell order for cheap. This attack is usually performed by bots that react quickly and are well financially stocked to make big moves. Defence against this type of attack is hard. However, revenge for this can be so called Bad Sandwich: poisoning frontrunning bots and getting their money [18].

Insecure Arithmetic [17] bugs can have devastating consequences. Similarly to "conventional" computing, overflows and underflows can happen, developers must be extra cautious because they are directly dealing with users money in the case of smart contracts.

III. Developing Solidity Smart Contracts

A. The Tools

There are several tools that a developer can use to develop smart contracts in Solidity, one of the most popular is Remix. It's the all-embracing IDE for Web3 development, it comes with Solidity compilers, blockchain virtual machines (so you don't need to test your contracts on the test net), debugger etc. It can be run in your browser at https://remix.ethereum.org. There are several more tools for smart contract development, but we will use Remix for showcases in this paper. We are going to showcase the text-book code of the basic reentrancy attack [17] and the text-book defences against it (already talked about in the second chapter of this paper).

Appended to this paper should be two source code files: *ReentrantUnsafe.sol* and *ReentrantSafe.sol* which are Solidity smart contract source codes. We can edit these in Remix.

Let's start by looking at the code of *ReentrantUnsafe.sol*. We can see that there are two contracts: *MoneySafe* and *ReentrancyAttack*. The vulnerable *MoneySafe* contract can be seen on Fig 2.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.13;
3
4  contract MoneySafe {
5      mapping(address => uint) public userBalances;
6
7      function getBalance() public view returns (uint) {···
9      }
10
11     function getUserBalance() public view returns (uint) {···
13     }
14
15     function deposit() public payable {
16         userBalances[msg.sender] =  userBalances[msg.sender] + msg.value;
17     }
18
19     function withdrawBalance() public {
20         // Withdraw the whole balance of user
21         uint amountToWithdraw = userBalances[msg.sender];
22         (bool successfulWithdraw, ) = msg.sender.call{value:amountToWithdraw}("");
23         require(successfulWithdraw, "Failed to withdraw ether");
24         userBalances[msg.sender] = 0;
25     }
26
27 }
```

Figure 2- MoneySafe smart contract

Every .sol source code must have *pragma solidity xxx;* specified; this tells the compiler which version of the standard to use.

B. ReentrantUnsafe.sol

We use *mapping* to create a list of pairs of addresses and uint (unsigned integer), we will use this as *userBalances*: how much ether does the user with the specified address have from the total balance of ether of the smart contract.

getBalance() and *getUserBalance()* are only helper functions (for debugging) and we shall not pay attention to them. All they do is return the native ether balance of the contract and users balance of deposited ether in the contract.

The function *deposit()* is *public* and *payable* which means that external contracts or wallets (users) can call it and that native ether can be sent to it. This native ether will be saved on the *address* of the smart contract (we can think of this as the total balance of the contract).

In Solidity, the *msg* object are the arguments that the external contract or user (wallet) passed to the RPC. *msg.sender* is the address of the method's invoker. *msg.value* is the value of native ether sent with the method invoking transaction – we can think of the method invoking transaction as paying a coffee machine and selecting a choice.

The function *withdrawBalance()* is the vulnerable one. As specified in the II. chapter of this paper: the method withdraws all of the user's (*msg.sender*) balance and send it to his address. At line 22 in Fig 2., we can see that call to the msg.sender is being made with *value* of *amountToWithdraw*, this effectively means that we are sending native ether to the *msg.sender* (the address of the user that called the withdraw function) from the total balance of this contract.

This is where the computation halts and gives the control flow to the external contract or wallet. That is when the fallback() function would be called. If everything went right and the ether was successfully sent, the userBalance of the caller is set to zero.

Now for the *ReentrancyAttack* contract, we can see it in Fig 3. This contract has an instance of the MoneySafe inside it, the address of the MoneySafe is not yet known and must be filled in at deployment of the RA contract (we will see how in the next chapter). This is how we establish connections between contracts in Solidity.

```
29 contract ReentrancyAttack {
30     MoneySafe public safe;
31
32     constructor(address _safeAddress) {
33         safe = MoneySafe(_safeAddress);
34     }
35
36     function getBalance() public view returns (uint) {···
38     }
39
40     fallback() external payable {
41         if (address(safe).balance >= 1 ether) {
42             safe.withdrawBalance();
43         }
44     }
45
46     function exploit() external payable {
47         require(msg.value >= 1 ether); // Send atleast 1 ETH to the attack contract
48         safe.deposit{value: 1 ether}(); // Deposit it in the MoneySafe
49         safe.withdrawBalance(); // Withdraw it back from the MoneySafe
50     }
51 }
```

Figure 3- ReentrancyAttack smart contract

We can see that the exploit() function does what has been specified in the II. chapter. We call the methods of the MoneySafe to deposit 1 ETH, so that we can withdraw it in the next step. We then tell the MoneySafe to withdrawBalance

and get our 1 ETH back. This makes the MoneySafe to execute and call the msg.sender at line 22 of Fig 2. (send ETH to the attacking contract). This triggers fallback() function in Fig 3., our *RA* contract receives 1 ETH while the original call is halted.

In fallback() we check that the *safe* has at least 1 ETH, and if it does, we request to withdraw our balance again, the MoneySafe contract is still halted in the previous call and has not yet updated the msg.sender's balance and thinks that it still has 1 ETH, thus sending it on line 22 again and triggering the death spiral, because fallback() is called yet again in the *RA* contract.

C. *ReentrantSafe.sol*

By implementing the defence mechanisms specified in chapter II. we can prevent this drainage of funds. These defences have been implemented in *ReentrantSafe.sol* source code file. The reentrant safe withdrawBalance() function will look like this then (see Fig 4.).

```
6     bool internal re_lock;

20    modifier nonReentrant() {
21        require(!re_lock, "Reentrancy detected.");
22        re_lock = true;
23        _;
24        re_lock = false;
25    }
26
27    function withdrawBalance() public nonReentrant {
28        // Withdraw the whole balance of user
29        uint amountToWithdraw = userBalances[msg.sender];
30        userBalances[msg.sender] = 0;
31        (bool successfulWithdraw, ) = msg.sender.call{value:amountToWithdraw}("");
32        require(successfulWithdraw, "Failed to withdraw ether");
33    }
```

Figure 4- Defence against reentrancy attack

Let's first point out that the CEI template has been applied and we first set the user's balance to zero before proceeding to withdraw ETH. Secondly, we can see that new internal boolean variable is instantiated at the creation of the contract. This *re_lock* is used in the modifier *nonReentrant()*.

What this modifier does is as follows: it first checks if the *re_lock* is false, e.g. it's the first call, not the reentrant one. If this goes through, the modifier then locks the contract by setting the lock to true. By using the _; special notation of Solidity, we tell the runtime to execute the function on which this modifier has been applied. After executing the function on line 23 in Fig. 4, we unlock the contract for next use. The header of the *withdrawBalance()* function has been modified; the *nonReentrant* keyword has been added; the modifier is applied to the function and is called always when the function is called.

IV. Showcase of Reentrancy Attack at Runtime

We will now focus on how to deploy the example smart contracts onto a Virtual Machine Blockchain and see the exploit as it would play out in the real world. Fortunately, Remix has already integrated debugging options and Virtual Blockchain providers to ease up the development of smart contracts.

After opening *ReentrantUnsafe.sol* file in the Remix, we can see "Deploy & run transactions" on the left (4th icon from the top see Fig 5.). In this tab, we can select the specifications of deploying our contract to the London JavaScript virtual machine (virtual blockchain). In *accounts*, we can see dummy accounts with 100 ETH each. We shall use the first account as the owner of the contract, the second account as the unsuspecting victim and the third as the attacker.

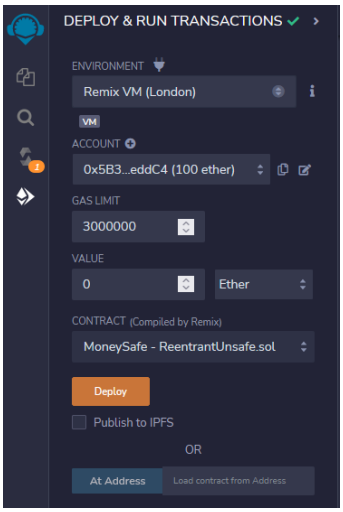

Figure 5- Deploy & run transactions

After selecting the first account, we select MoneySafe in the *contract* dropdown menu and we press *deploy*. If everything went well, our MoneySafe (with reentrancy vulnerability) is now deployed to the Ethereum VM. We select the second account, set the *value* to 10 Ether, scroll down and interact with the deployed contract (Fig 6.).

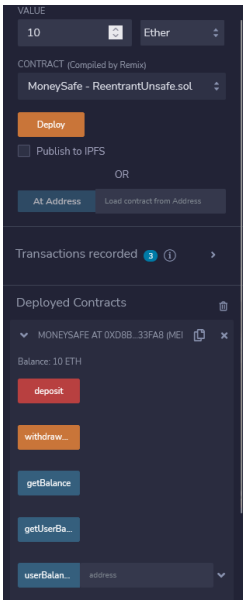

Figure 6-Interacting with the contract

After clicking *deposit* we can see that Account 2 now has 10 ETH stored in the MoneySafe (we can see total and user balance by calling the relevant function).

We will now switch to Account 3 and exploit the reentrancy vulnerability. We first need to deploy the ReentrancyAttack contract. Select the ReentrancyAttack in the *contract* dropbox and before clicking *deploy*, we need to get the address of the MoneySafe that we just deployed (so we can instantiate it in the constructor of the *RA* contract). This address can be copied by clicking on the copy icon from the *Deployed Contracts* where it says something like "MONEYSAFE DEPLOYED AT 0x...". We copy this address, paste it next to the *Deploy* button and deploy. At this

time, we have deployed our *RA* contract, that we now may use to exploit and drain the MoneySafe.

Important is to set the *value* to atleast 1 ETH so that way the deployed contract will have it in its balance. After setting the value to atleast 1, call the *exploit* function from *Deployed Contracts* -> "REENTRANCY ATTACK AT 0x…".

After calling the *RA exploit* method, we can see that the balance of the *RA* contract is now *total_MoneySafe_balance*, we have successfully exploited the reentrancy vulnerability (in real life we would withdraw these funds to our wallet from the *RA contract* by having prepared some withdraw function beforehand).

To check that the defences against the RA work, we shall do the same process but with the *ReentrantSafe.sol* source code. We can see that the *exploit* call fails and that the defence mechanism works.

We could also deploy these contracts (or any other) onto the Ethereum mainnet (or testnet). But we would need to have set-up Ethereum wallet, like for example Metamask.

Note that for practical purposes, smart contract developers use templates, because lot of the code in the contracts (ERC-20 ERC-721 tokens) is repeated and these templates are modified for the concrete use. Most used templates for this purpose are OpenZeppelin [19], it's good to use these and not try to invent the wheel – in a space where wrong arithmetic operation can cost millions of dollars.

V. Conclusion

We have introduced Ethereum and the broader context of smart contract hacking. We have shown how a simple bug can cost millions of dollars and how this flaw can be (and still is today) exploited in the real world. We presented how to get started in programming in Solidity by showing this text-book exploit. We also showed how easy it is to deploy contracts for on-chain testing.

The Ethereum (and EVM-compatible) space is large today and there are hundreds of smart contracts being deployed daily with billions of dollars in staked value. This also means that there is a lot of work to be done for cybersecurity experts, and lot of smart contract auditing companies are popping up lately, with famous names like Certik etc.

The benefit of Ethereum is its openness and transparency, so that contracts can be verified and audited by independent developers. This however means, that they can also be exploited by virtually anyone. The less nice aspect of the openness is that anyone can copy and paste a contract that somebody has put a lot of work in. The only thing, in the end, that differentiates the crypto projects is the people and capital standing behind it.

References

[1] TEAM INNERQUEST ONLINE. *How Does a Blockchain Prevent Double-Spending of Bitcoins?* [online]. 25 August 2018 [cit. 2022-11-11]. Available at: https://medium.com/innerquest-online/how-does-a-blockchain-prevent-double-spending-of-bitcoins-fa0ecf9849f7

[2] NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2009, 1-9 [cit. 2022-11-11]. Available at: https://bitcoin.org/bitcoin.pdf

[3] OSBORNE, Margaret. *Bitcoin Could Rival Beef or Crude Oil in Environmental Impact* [online]. October 3 2022 [cit. 2022-11-11]. Available at: https://www.smithsonianmag.com/smart-news/bitcoin-could-rival-beef-or-crude-oil-in-environmental-impact-180980877/

[4] BUTERIN, Vitalik. *Ethereum: A Next-Generation Cryptocurrency and Decentralized Application Platform* [online]. January 23 2014 [cit. 2022-11-11]. Available at: https://web.archive.org/web/20140413130750/http://bitcoinmagazine.com/9671/ethereum-next-generation-cryptocurrency-decentralized-application-platform/

[5] THE ETHEREUM FOUNDATION. *What is Ethereum* [online]. 18 November 2022 [cit. 2022-11-11]. Available at: https://ethereum.org/en/what-is-ethereum/

[6] *EVM* [online]. November 4 2022 [cit. 2022-11-11]. Available at: https://ethereum.org/en/developers/docs/evm/

[7] *Gas* [online]. [cit. 2022-11-11]. Available at: https://ethereum.org/en/developers/docs/gas/

[8] WOOD, Gavin. *Ethereum: A secure decentralised generalised transaction ledger* [online]. 2022, 24 october 2022 [cit. 2022-11-11]. Available at: https://ethereum.github.io/yellowpaper/paper.pdf

[9] *Solidity Documentation* [online]. [cit. 2022-11-11]. Available at: https://docs.soliditylang.org/en/latest/

[10] BUTERIN, Vitalik and Fabian VOGELSTELLER. *EIP-20: Token Standard* [online]. 2015-11-19 [cit. 2022-11-11]. Available at: https://eips.ethereum.org/EIPS/eip-20

[11] YOUNG, Martin. *Over 44 million contracts deployed to Ethereum Since Genesis* [online]. Aug 4, 2022 [cit. 2022-11-11]. Available at: https://cryptopotato.com/over-44-million-contracts-deployed-to-ethereum-since-genesis-research/

[12] VARIOUS. *Upgrading Smart Contract* [online]. October 18 2022 [cit. 2022-11-11]. Available at: https://ethereum.org/en/developers/docs/smart-contracts/upgrading/

[13] SALEHI, Mehdi, Jeremy CLARK and Mohammad MANNAN. *Not so immutable: Upgradeability of Smart Contracts on Ethereum* [online]. Montreal, Canada, 2022 [cit. 2022-11-11]. Available at: https://users.encs.concordia.ca/~mmannan/publications/upgradability-WTSC-2022.pdf. Concordia University.

[14] CRYPTOPEDIA. *The DAO: What Was The DAO?* [online]. March 17 2022 [cit. 2022-11-11]. Available at: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

[15] BEHNKE, Rob. *What is a Re-Entrancy Attack* [online]. 05.17.2022 [cit. 2022-11-19]. Available at: https://halborn.com/what-is-a-re-entrancy-attack/

[16] SACHINOGLOU, Ioannis. *Read Only Reentrancy: A Novel Vulnerability Class Responsible for 100m Funds At Risk* [online]. October 2022 [cit. 2022-11-19]. Available at: https://archive.devcon.org/archive/watch/6/read-only-reentrancy-a-novel-vulnerability-class-responsible-for-100m-funds-at-risk/?tab=YouTube

[17] *Ethereum Smart Contract Best Practices: Attacks* [online]. [cit. 2022-11-19]. Available at: https://consensys.github.io/smart-contract-best-practices/attacks/

[18] FOXLEY, William. *Bad Sandwich: DeFi Trader 'Poisons' Front-Running Miners for $250K Profit* [online]. Mar 22, 2021 [cit. 2022-11-19]. Available at: https://www.coindesk.com/tech/2021/03/22/bad-sandwich-defi-trader-poisons-front-running-miners-for-250k-profit/

[19] *Open Zeppelin: Contracts* [online]. [cit. 2022-11-19]. Available at: https://docs.openzeppelin.com/contracts/4.x/