Audit Techniques & Tools 101 1

Audit: is an external security assessment of a project codebase, typically requested and paid-for by the project team

a. It detects and describes (in a report) security issues with underlying vulnerabilities, severity/difficulty, potential exploit scenarios and recommended fixes.

b. It also provides subjective insights into code quality, documentation and testing.

c. The scope/depth/format of audit reports varies across auditing teams but they generally cover similar aspects.

Audit Techniques & Tools 101 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.

Audit Techniques & Tools 101 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.

Audit Techniques & Tools 101 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.

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.

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.

Audit Types: depend on the scope/nature/status of projects but generally fall into the following categories:

a. New audit: for a new project that is being launched

b. Repeat audit: for a new version of an existing project being revised with new/fixed features

c. Fix audit: for reviewing the fixes made to the findings from a current/prior audit

d. Retainer audit: for constantly reviewing project updates

e. Incident audit: for reviewing an exploit incident, root causing the incident, identifying the underlying vulnerabilities and proposing fixes.

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.

Audit Effort: typically involves more than one auditor simultaneously for getting independent, redundant or supplementary/complementary assessment expertise on the project.

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.

Audit Prerequisites should include:

a. 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

b. Public/private repository

c. Public/anonymous team

d. Specification of the project's design and architecture

e. Documentation of the project's implementation and business logic

f. Threat models and specific areas of concern

g. Prior testing, tools used, other audits

h. Timeline, effort and costs/payments

i. Engagement dynamics/channels for questions/clarifications, findings communication and reports

j. Points of contact on both sides

Audit Limitations: Audits are necessary (for now at least) but not sufficient:

a. 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.

b. 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.

c. 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.

Audit Techniques & Tools 101 — 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.

Audit Techniques & Tools 101 — 14 (1/2)

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:

a. Access Controls: Related to authorization of users and assessment of rights

b. Auditing and Logging: Related to auditing of actions or logging of problems

c. Authentication: Related to the identification of users

d. Configuration: Related to security configurations of servers, devices or software

e. Cryptography: Related to protecting the privacy or integrity of data

f. Data Exposure: Related to unintended exposure of sensitive information

g. Data Validation: Related to improper reliance on the structure or values of data

h. Denial of Service: Related to causing system failure

Audit Techniques & Tools 101 — 14 (2/2)

i. Error Reporting: Related to the reporting of error conditions in a secure fashion

j. Patching: Related to keeping software up to date

k. Session Management: Related to the identification of authenticated users

l. Timing: Related to race conditions, locking or order of operations

m. Undefined Behavior: Related to behavior triggered by the program

Audit Techniques & Tools 101 — 15 (1/2)

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:

a. Undetermined: The difficulty of exploit was not determined during this engagement

b. Low: Commonly exploited, public tools exist or can be scripted that exploit this flaw

c. Medium: Attackers must write an exploit, or need an in-depth knowledge of a complex system

d. 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

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.

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.

a. OWASP proposes a 3x3 Severity Matrix which combines the three Likelihood levels with the three Impact levels

b. 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;

c. 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

d. 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.

Audit Techniques & Tools 101 — 18 (1/2)

Audit Checklist For Projects (See here **for Trail of Bits recommendations):**

a. 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.

Audit Techniques & Tools 101 — 18 (2/2)

b. Document: 1) Describe what your product does, who uses it, why, and how it delivers. 2) Add comments about intended behavior in-line with the code. 3) Label and describe your tests and results, both positive and negative. 4) Include past reviews and bugs.

c. 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.

Audit Techniques & Tools 101 — 19 (1/2)

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.

Audit Techniques & Tools 101 — 19 (2/2)

a. Specification analysis (manual)

b. Documentation analysis (manual)

c. Testing (automated)

d. Static analysis (automated)

e. Fuzzing (automated)

f. Symbolic checking (automated)

g. Formal verification (automated)

h. 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.

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.

a. 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.

b. Analysing the specification of a project provides auditors with the above details and lets them evaluate the assumptions made and indicate any shortcomings

c. 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.

Documentation analysis: Documentation is a description of what has been implemented based on the design and architectural requirements.

a. Documentation answers ‘how’ something has been designed/architected/implemented without necessarily addressing the ‘why’ and the design/requirement goals

b. 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.

c. Documentation in many cases serves as a substitute for specification and provides critical insights into the assumptions, requirements and goals of the project team

d. 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

Audit Techniques & Tools 101 21 (2/2)

e. Mismatches between the documentation and the code could indicate stale/poor documentation, software defects or security vulnerabilities

f. 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

Audit Techniques & Tools 101 22 (1/2)

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.

a. 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

b. Smart contract development platforms (Truffle, Embark, Brownie, Waffle, Hardhat etc.) are relatively new with different levels of support for testing

c. Projects, in general, have very little testing done at the audit stage. Testing integrations and composability with mainnet contracts and state is non-trivial

d. 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

Audit Techniques & Tools 101 22 (2/2)

e. 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

f. "Program testing can be used to show the presence of bugs, but never to show their absence!" - E.W. Dijkstra

Audit Techniques & Tools 101 23

Static analysis: is a technique of analyzing program properties without actually executing the program.

a. This is in contrast to software testing where programs are actually executed/run with different inputs

b. 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.

c. Static analysis typically is a combination of control flow and data flow analyses

Audit Techniques & Tools 101 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

a. 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)

b. Echidna and Harvey are two popular tools for smart contract fuzzing



Audit Techniques & Tools 101 25 (1/2)

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

a. 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)

b. 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.

c. A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure

Audit Techniques & Tools 101 25 (2/2)

d. 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.

e. 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

f. 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

Audit Techniques & Tools 101 26 (1/2)

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

a. Formal verification is effective at detecting complex bugs which are hard to detect manually or using simpler automated tools

Audit Techniques & Tools 101 26 (2/2)

b. Formal verification needs a specification of the program being verified and techniques to translate/compare the specification with the actual implementation

c. 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.



Audit Techniques & Tools 101 27 (1/2)

Manual analysis: is complimentary to automated analysis using tools and serves a critical need in smart contract audits

Audit Techniques & Tools 101 27 (2/2)

a. 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

b. 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.

c. Manual analysis is however the only way today to infer and evaluate business logic and application-level constraints which

Audit Techniques & Tools 101 28 (1/2)

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.

a. False positives require further manual analysis on findings to investigate if they are indeed false or true positives

b. High number of false positives increases manual effort in verification and lowers the confidence in the accuracy of the earlier automated/manual analysis

c. True positives might sometimes be classified as false positives which leads to vulnerabilities being exploited instead of being fixed

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.

a. False negatives, per definition, are not reported or even realised unless a different analysis reveals their presence or the vulnerabilities are exploited

b. High number of false negatives lowers the confidence in the effectiveness of the earlier manual/automated analysis.

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.

(QR codes omitted)

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.

Categories of security tools: tools for testing, test coverage, linting, disassembling, visualization, static analysis, dynamic analysis and formal verification of smart contracts.

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).



Slither features:

a. Detects vulnerable Solidity code with low false positives

b. Identifies where the error condition occurs in the source code

c. Easily integrates into continuous integration and Truffle builds

d. Built-in 'printers' quickly report crucial contract information

e. Detector API to write custom analyses in Python

f. Ability to analyze contracts written with Solidity >= 0.4

g. Intermediate representation (SlithIR) enables simple, high-precision analyses

h. Correctly parses 99.9% of all public Solidity code

i. Average execution time of less than 1 second per contract

Slither bugs and optimizations detection can run on a Truffle/Embark/Dapp/Etherlime/Hardhat application or on a single Solidity file:

a. Slither runs all its detectors by default. To run only selected detectors, use *--detect detector1,detector2*. To exclude detectors, use *--exclude detector1,detector2*.

b. To exclude detectors with an informational or low severity, use *--exclude-informational* or *--exclude-low*

c. *--list-detectors* lists available detectors

Audit Techniques & Tools 101 — 36 (1/2)

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):

a. call-graph: Export the call-graph of the contracts to a dot file

b. cfg: Export the CFG of each functions

c. constructor-calls: Print the constructors executed

d. contract-summary: Print a summary of the contracts

e. data-dependency: Print the data dependencies of the variables

f. echidna: Export Echidna guiding information

g. evm: Print the evm instructions of nodes in functions

h. function-id: Print the keccack256 signature of the functions

i. function-summary: Print a summary of the functions

Audit Techniques & Tools 101 — 36 (2/2)

l. inheritance-graph: Export the inheritance graph of each contract to a dot file

m. modifiers: Print the modifiers called by each function

n. require: Print the require and assert calls of each function

o. slithir: Print the slithIR representation of the functions

p. slithir-ssa: Print the slithIR representation of the functions

q. variable-order: Print the storage order of the state variables

r. vars-and-auth: Print the state variables written and the authorization of the functions

Audit Techniques & Tools 101 — 37 (1/2)

Slither upgradeability checks helps review contracts that use the delegatecall proxy pattern using ***slither-check-upgradeability*** tool with following options:

a. became-constant: Variables that should not be constant

b. function-id-collision: Functions ids collision

c. function-shadowing: Functions shadowing

d. missing-calls: Missing calls to init functions

e. missing-init-modifier: initializer() is not called

f. multiple-calls: Init functions called multiple times

g. order-vars-contracts: Incorrect vars order with the v2

Audit Techniques & Tools 101 — 37 (2/2)

h. order-vars-proxy: Incorrect vars order with the proxy

i. variables-initialized: State variables with an initial value

j. were-constant: Variables that should be constant

k. extra-vars-proxy: Extra vars in the proxy

l. missing-variables: Variable missing in the v2

m. extra-vars-v2: Extra vars in the v2

n. init-inherited: Initializable is not inherited

o. init-missing: Initializable is missing

p. initialize-target: Initialize function that must be called

q. initializer-missing: initializer() is missing

Slither code similarity detector (a research-oriented tool) uses state-of-the-art machine learning to detect similar (vulnerable) Solidity functions

a. It uses a pre-trained model from etherscan_verified_contracts with 60,000 contracts and more than 850,000 functions

b. It uses FastText, a vector embedding technique, to generate compact numerical representations of every function

c. 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

Slither contract flattening tool ***slither-flat*** produces a flattened version of the codebase with the following features:

a. 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

b. Supports circular dependency

c. Supports all the compilation platforms (Truffle, embark, buidler, etherlime, ...).

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:

a. unused-state

b. solc-version

c. pragma

d. naming-convention

e. external-function

f. constable-states

g. constant-function

Slither ERC conformance tool ***slither-check-erc*** checks the following for ERC's conformance for ERC20, ERC721, ERC777, ERC165, ERC223 and ERC1820:

a. All the functions are present

b. All the events are present

c. Functions return the correct type

d. Functions that must be view are view

e. Events' parameters are correctly indexed

f. The functions emit the events

g. Derived contracts do not break the conformance

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:

a. Transferable - Test the correct tokens transfer

b. Pausable - Test the pausable functionality

c. NotMintable - Test that no one can mint tokens

d. NotMintableNotBurnable - Test that no one can mint or burn tokens

e. NotBurnable - Test that no one can burn tokens

f. Burnable - Test the burn of tokens. Require the "burn(address) returns()" function

Slither new detectors: Slither's plugin architecture lets you integrate new detectors that run from the command line. The skeleton for a detector has:

a. *ARGUMENT*: lets you run the detector from the command line

b. *HELP*: is the information printed from the command line

c. *IMPACT*: indicates the impact of the issue. Allowed values are INFORMATIONAL|LOW|MEDIUM|HIGH

d. *CONFIDENCE*: indicates your confidence in the analysis. Allowed values are LOW|MEDIUM|HIGH

e. *WIKI*: constants are used to generate automatically the documentation.

f. *_detect()* is the function that implements the detector logic and needs to return a list of findings.

Manticore is a symbolic execution tool for analysis of Ethereum smart contracts (besides Linux binaries & WASM modules). See tutorial for details.

a. Program Exploration: Manticore can execute a program with symbolic inputs and explore all the possible states it can reach

b. Input Generation: Manticore can automatically produce concrete inputs that result in a given program state

c. Error Discovery: Manticore can detect crashes and other failure cases in binaries and smart contracts

d. Instrumentation: Manticore provides fine-grained control of state exploration via event callbacks and instruction hooks

e. Programmatic Interface: Manticore exposes programmatic access to its analysis engine via a Python API

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.*

Echidna Features:

a. Generates inputs tailored to your actual code

b. Optional corpus collection, mutation and coverage guidance to find deeper bugs

c. Powered by Slither to extract useful information before the fuzzing campaign

d. Source code integration to identify which lines are covered after the fuzzing campaign

e. Curses-based retro UI, text-only or JSON output

f. Automatic test case minimization for quick triage

g. Seamless integration into the development workflow

h. Maximum gas usage reporting of the fuzzing campaign

i. Support for a complex contract initialization with Etheno and Truffle

Echidna Usage (see tutorial **for details):**

a. 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.

b. Writing invariants: Invariants are expressed as Solidity functions with names that begin with echidna_, have no arguments, and return a boolean.

c. 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.

Eth-security-toolbox is *a Docker container preinstalled and preconfigured with all of Trail of Bits' Ethereum security tools*. **This includes:**

a. Echidna property-based fuzz tester

b. Etheno integration tool and differential tester

c. Manticore symbolic analyzer and formal contract verifier

d. Slither static analysis tool

e. Rattle EVM lifter

f. Not So Smart Contracts repository

Ethersplay is *a Binary Ninja plugin which enables an EVM disassembler and related analysis tools*.

a. Takes as input the evm bytecode in raw binary format

b. Renders control flow graph of all functions

c. Shows Manticore coverage

Pyevmasm is *an assembler and disassembler library for the Ethereum Virtual Machine (EVM)*. It includes a command line utility and a Python API.

Rattle is *an EVM binary static analysis framework designed to work on deployed smart contracts* (not actively developed anymore).

a. Takes EVM byte strings and uses a flow-sensitive analysis to recover the original control flow graph

b. Lifts the control flow graph into an SSA/infinite register form, and optimizes the SSA – removing DUPs, SWAPs, PUSHs, and POPs

c. 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

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*.

a. Reliably recovers a Control Flow Graph (CFG) from EVM bytecode using a dedicated Value Set Analysis

b. Recovers functions names

c. Recovers attributes (e.g., payable, view, pure)

d. Outputs the CFG to a dot file

e. Library API

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:**

a. Slither

b. Echidna

c. Manticore

d. evm-cfg-builder

Solc-select is *a script to quickly switch between Solidity compiler versions*.

a. solc-select: manages installing and setting different solc compiler versions

b. solc: wrapper around solc which picks the right version according to what was set via solc-select

c. solc binaries are downloaded from https://binaries.soliditylang.org/ which contains official artifacts for many historical and modern solc versions for Linux and macOS

Etheno is *the Ethereum testing Swiss Army knife. It's a JSON RPC multiplexer, analysis tool wrapper, and test integration tool*.

a. 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

b. 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

c. 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

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.

MythX process:

a. 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.

b. Activate a full suite of analysis techniques: The longer MythX runs, the more it can detect more security weaknesses.

c. 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.

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.

a. A static analyzer that parses the Solidity AST

b. a symbolic analyzer that detects possible vulnerable states, and

c. a greybox fuzzer that detects vulnerable execution paths

MythX coverage: extends to most SWCs found in the SWC Registry with the 46 detectors listed here along with the type of analysis used.

MythX is based on a SaaS (Security as a Service) platform based on the premise that:

a. Higher performance compared to running security tools locally

b. Higher vulnerability coverage than any standalone tool

c. Benefit from continuous improvements to our security analysis technology with new and improved security tests as the smart contract security landscape evolves.

MythX privacy guarantee for the smart contract code submitted using their SaaS APIs:

a. Code analysis requests are encrypted with TLS

b. 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.

c. It keeps the results of your analysis so you can retrieve them later, but the report can be accessed by you only.

MythX running time: Quick scan runs for 5 minutes, Standard scan runs for 30 minutes, and Deep scan runs for 90 minutes.

MythX official integrations, tools and libraries include:

a. MythX CLI: Unified tool to use MythX as a Command Line Interface (CLI) now with full Truffle Projects support.

b. MythX-JS: Typescript library to integrate MythX in your JS or TS projects.

c. PythX: Python library to integrate MythX in your Python projects.

d. MythX VSCode: VSCode extension which allows you to scan smart-contracts and view results directly from your code editor.

MythX pricing:

a. On Demand (US$9.99/3 scans): All scan modes and Prepaid scan packs

b. Developer (US$49/mo): Quick and Standard scan modes; 500 scans/month

c. Professional (US$249/mo): All scan modes; 10 000 scans/month

d. Enterprise (Custom pricing): Custom plans for your team's specific needs; Custom Verification Service; Retainer for Custom Support

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).

a. Principles/Goals: 1) Specifications are easy to understand by developers and auditors 2) Specifications are simple to reason about 3) Specifications can be efficiently checked using off-the-shelf analysis tools 4) A small number of core specification constructs are sufficient to express and reason about more advanced constructs

b. Transforms annotations in the Scribble specification language into concrete assertions

c. With these instrumented but equivalent contracts, one can then use Mythril, Harvey, MythX


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.

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.*

Audit Techniques & Tools 101 68 (1/2)

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:**

a. Automatic smart contract scanning which generates a list of possible exploits

b. Sending transactions to exploit a smart contract

c. Transaction pool monitor

d. Web3 console

Audit Techniques & Tools 101 68 (2/2)

e. Frontrunning and backrunning transactions

f. Waiting for a list of transactions and sending out others

g. Estimating gas for transactions means only successful transactions are sent

h. Disabling gas estimation will send transactions with a fixed gas quantity.

Audit Techniques & Tools 101 69 (1/2)

Visual Auditor is *a Visual Studio Code extension that provides security-aware syntax and semantic highlighting for* Solidity *and* Vyper.

a. 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

b. Semantic Highlighting: highlights StateVars (constant, inherited), detects and alerts about StateVar shadowing, highlights function arguments in the function body

c. 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

d. Views: Cockpit vs Outline

Audit Techniques & Tools 101 69 (2/2)

e. 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

f. 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

Audit Techniques & Tools 101 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.*

a. Integrated with Visual Auditor

b. Commands: graph, ftrace, flatten, describe, inheritance, dependencies, parse, mdreport

Audit Techniques & Tools 101 71 (1/2)

SWC Registry: The Smart Contract Weakness Classification Registry (SWC Registry) is an implementation of the weakness classification scheme proposed in EIP-1470.

a. 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

b. This repository is maintained by the team behind MythX and currently contains 37 entries

Audit Techniques & Tools 101 71 (2/2)

c. The goals of this project are as follows: 1) Provide a straightforward way to classify security issues in smart contract systems. 2) Define a common language for describing security issues in smart contract systems' architecture, design, or code. 3) Serve as a way to train and increase performance for smart contract security analysis tools.

Audit Techniques & Tools 101 72

Securify: is a security scanner for Ethereum smart contracts which Implements static analysis written in Datalog and supports 38 vulnerabilities

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: reduction of temporal safety verification to reachability checking, an efficient symbolic execution engine used to compute precise symbolic states within a transaction, and delayed abstraction which approximates symbolic states at the end of transactions into abstract states.

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.

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.

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.

a. 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.

b. 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

c. 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

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.

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:

a. 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.

b. 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

c. Paradigm CFT: is a set of seventeen challenges created by samczsun at Paradigm.

d. 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.

Audit Techniques & Tools 101 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.

Audit Techniques & Tools 101 80 (1/2)

Audit Process could be thought of as a ten-step process as follows:

a. Read specification/documentation of the project to understand the requirements, design and architecture

b. Run fast automated tools such as linters or static analyzers to investigate common Solidity pitfalls or missing smart contract best-practices

c. Manual code analysis to understand business logic and detect vulnerabilities in it

d. 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

e. Discuss (with other auditors) the findings from above to identify any false positives or missing analyses

Audit Techniques & Tools 101 80 (2/2)

f. Convey status to project team for clarifying questions on business logic or threat model

g. Iterate the above for the duration of the audit leaving some time for report writing

h. Write report summarizing the above with details on findings and recommendations

i. Deliver the report to the project team and discuss findings, severity and potential fixes

j. Evaluate fixes from the project team and verify that they indeed remove the vulnerabilities identified in findings.

Audit Techniques & Tools 101 81 (1/2)

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:

a. 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.

b. 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

c. Documentation is a description of what has been implemented based on the design and architectural requirements.

d. 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.

e. 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.

f. 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.

Running static analyzers: Automated tools such as linters or static analyzers help investigate common Solidity pitfalls or missing smart contract best-practices

a. 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.

b. 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.

c. False positives are possible among some of the detector findings and need to be verified manually if they are true/false positives

Manual code review: is required to understand business logic and detect vulnerabilities in it.

a. Automated analyzers do not understand application-level logic and their constraints. They are limited to constraints/properties of Solidity language, EVM or Ethereum blockchain.

b. Manual analysis of the code is required to detect security-relevant deviations in implementation vis-a-vis the specification or documentation.

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.

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.

Audit Techniques & Tools 101 84 (2/2)

a. 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

b. Analyzing the output of these tools requires significant expertise with the tools themselves, their domain-specific language and sometimes even their inner workings

c. 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

Audit Techniques & Tools 101 85 (1/2)

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.

Audit Techniques & Tools 101 85 (2/2)

a. 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.

b. 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.

Audit Techniques & Tools 101 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.

a. Findings may also be shared with the project team immediately on a private repository to discuss impact, fixes and other implications.

b. 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.

Audit Techniques & Tools 101 87 (1/2)

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.

a. 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

Audit Techniques & Tools 101 87 (2/2)

b. 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

c. 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

Audit Techniques & Tools 101 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.

a. 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

b. The project team typically takes some time to review the audit report and respond back with any counterpoints on findings, severities or suggested fixes

c. 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

Audit Techniques & Tools 101 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

a. 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

b. 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

c. 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.

Audit Techniques & Tools 101 90

Manual review approaches: Auditors have different approaches to manual reviewing smart contract code for vulnerabilities.

a. Starting with access control

b. Starting with asset flow

c. Starting with control flow

d. Starting with data flow

e. Inferring constraints

f. Understanding dependencies

g. Evaluating assumptions

h. Evaluating security checklists

Audit Techniques & Tools 101 91 (1/2)

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.)

a. 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.

b. 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.

Audit Techniques & Tools 101 91 (2/2)

c. 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

Audit Techniques & Tools 101 92 (1/2)

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.

a. Who: Assets should be withdrawn/deposited only by authorised/specified addresses as per application logic

b. When: Assets should be withdrawn/deposited only in authorised/specified time windows or under authorised/specified conditions as per application logic (when)

c. Which: Assets, only those authorised/specified types, should be withdrawn/deposited as per application logic

d. Why: Assets should be withdrawn/deposited only for authorised/specified reasons as per application logic

e. Where: Assets should be withdrawn/deposited only to authorised/specified addresses as per application logic

f. What type: Assets, only of authorised/specified types, should be withdrawn/deposited as per application logic

g. How much: Assets, only in authorised/specified amounts, should be withdrawn/deposited as per application logic

Evaluating control flow: Control flow analyzes the transfer of control, i.e. execution order, across and within smart contracts.

a. 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

b. Intraprocedural (i.e. within a function) control flow is dictated by conditionals (if/else), loops (for/while/do/continue/break) and return statements.

c. Both intra and interprocedural control flow analysis help track the flow of execution and data in smart contracts

Evaluating data flow: Data flow analyzes the transfer of data across and within smart contracts

a. Interprocedural data flow is evaluated by analyzing the data (variables/constants) used as argument values for function parameters at call sites

b. Intraprocedural data flow is evaluated by analyzing the assignment and use of (state/memory/calldata) variables/constants along the control flow paths within functions.

c. Both intra and interprocedural data flow analysis help track the flow of global/local storage/memory changes in smart contracts

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 ERC-721 token to the address when it makes a certain deposit of ERC-20 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.

a. 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.

b. 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.

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.

a. 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.

b. 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.

c. This is especially of interest/concern in DeFi protocols that rely on other related protocols for stablecoins, yield generation, borrowing/lending, derivatives, oracles etc.

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:

a. Only admins can call these functions

b. Initialization functions will only be called once by the contract deployer (e.g. for upgradeable contracts)

c. Functions will always be called in a certain order (as expected by the specification)

d. Parameters can only have non-zero values or values within a certain threshold e.g. addresses will never be zero valued

e. 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)

f. Function calls will always be successful and so checking for return values is not required

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.

a. 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."

b. 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.

c. 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.

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

a. 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

b. 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

c. 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

Audit Techniques & Tools 101 101

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.