title: A guide to smart contract security tools description: An overview of three different testing and program analysis techniques author: "Trailofbits" lang: en tags: ["solidity", "smart contracts", "security"] skill: intermediate published: 2020-09-07 source: Building secure contracts sourceUrl: https://github.com/crytic/building-secure-contracts/tree/master/program-analysis

We are going to use three distinctive testing and program analysis techniques:

- **Static analysis with Slither.** All the paths of the program are approximated and analyzed at the same time, through different program presentations (e.g. control-flow-graph)
- **Fuzzing with Echidna.** The code is executed with a pseudo-random generation of transactions. The fuzzer will try to find a sequence of transactions to violate a given property.
- **Symbolic execution with Manticore.** A formal verification technique, which translates each execution path to a mathematical formula, on which on top constraints can be checked.

Each technique has advantages and pitfalls, and will be useful inspecific cases:

Technique	Tool	Usage	Speed	Bugs missed	False Alarms
Static Analysis	Slither	CLI & scripts	seconds	moderate	low
Fuzzing	Echidna	Solidity properties	minutes	low	none
Symbolic Execution	Manticore	Solidity properties & scripts	hours	none*	none

* if all the paths are explored without timeout

Slither analyzes contracts within seconds, however, static analysis might lead to false alarms and will be less suitable for complex checks (e.g. arithmetic checks). Run Slither via the API for push-button access to built-in detectors or via the API for user-defined checks.

Echidna needs to run for several minutes and will only produce true positives. Echidna checks user-provided security properties, written in Solidity. It might miss bugs since it is based on random exploration.

Manticore performs the "heaviest weight" analysis. Like Echidna, Manticore verifies user-provided properties. It will need more time to run, but it can prove the validity of a property and will not report false alarms.

Suggested workflow {#suggested-workflow}

Start with Slither's built-in detectors to ensure that no simple bugs are present now or will be introduced later. Use Slither to check properties related to inheritance, variable dependencies, and structural issues. As the codebase grows, use Echidna to test more complex properties of the state machine. Revisit Slither to develop custom checks for protections unavailable from Solidity, like protecting against a function being overridden. Finally, use Manticore to perform targeted verification of critical security properties, e.g., arithmetic operations.

- Use Slither's CLI to catch common issues
- Use Echidna to test high-level security properties of your contract
- Use Slither to write custom static checks
- Use Manticore once you want in-depth assurance of critical security properties

A note on unit tests. Unit tests are necessary to build high-quality software. However, these techniques are not the best suited to find security flaws. They are typically used to test positive behaviors of code (i.e. the code works as expected in the normal context), while security flaws tend to reside in edge cases that the developers did not consider. In our study of dozens of smart contract security reviews, unit test coverage had no effect on the number or severity of security flawswe found in our client's code.

Determining Security Properties {#determining-security-properties}

To effectively test and verify your code, you must identify the areas that need attention. As your resources spent on security are limited, scoping the weak or high-value parts of your codebase is important to optimize your effort. Threat modeling can help. Consider reviewing:

- Rapid Risk Assessments (our preferred approach when time is short)
- Guide to Data-Centric System Threat Modeling (aka NIST 800-154)
- Shostack thread modeling
- STRIDE / DREAD

- PASTA
- Use of Assertions

Components {#components}

Knowing what you want to check will also help you to select the right tool.

The broad areas that are frequently relevant for smart contracts include:

- **State machine.** Most contracts can be represented as a state machine. Consider checking that (1) No invalid state can be reached, (2) if a state is valid that it can be reached, and (3) no state traps the contract.

- Echidna and Manticore are the tools to favor to test state-machine specifications.

- **Access controls.** If your system has privileged users (e.g. an owner, controllers, ...) you must ensure that (1) each user can only perform the authorized actions and (2) no user can block actions from a more privileged user.

- Slither, Echidna and Manticore can check for correct access controls. For example, Slither can check that only whitelisted functions lack the onlyOwner modifier. Echidna and Manticore are useful for more complex access control, such as a permission given only if the contract reaches a given state.

- **Arithmetic operations.** Checking the soundness of the arithmetic operations is critical. Using`SafeMath` everywhere is a good step to prevent overflow/underflow, however, you must still consider other arithmetic flaws, including rounding issues and flaws that trap the contract.

- Manticore is the best choice here. Echidna can be used if the arithmetic is out-of-scope of the SMT solver.

- **Inheritance correctness.** Solidity contracts rely heavily on multiple inheritance. Mistakes such as a shadowing function missing a `super` call and misinterpreted c3 linearization order can easily be introduced.

- Slither is the tool to ensure detection of these issues.

- **External interactions.** Contracts interact with each other, and some external contracts should not be trusted. For example, if your contract relies on external oracles, will it remain secure if half the available oracles are compromised?

- Manticore and Echidna are the best choice for testing external interactions with your contracts. Manticore has an built-in mechanism to stub external contracts.

- **Standard conformance.** Ethereum standards (e.g. ERC20) have a history of flaws in their design. Be aware of the limitations of the standard you are building on.

- Slither, Echidna, and Manticore will help you to detect deviations from a given standard.

Tool selection cheatsheet {#tool-selection-cheatsheet}

Component	Tools	Examples
State machine	Echidna, Manticore	
Access control	Slither, Echidna, Manticore	Slither exercise 2, Echidna exercise 2
Arithmetic operations	Manticore, Echidna	Echidna exercise 1, Manticore exercises 1 - 3
Inheritance correctness	Slither	Slither exercise 1
External interactions	Manticore, Echidna	
Standard conformance	Slither, Echidna, Manticore	`slither-erc`

Other areas will need to be checked depending on your goals, but these coarse-grained areas of focus are a good start for any smart contract system.

Our public audits contain examples of verified or tested properties. Consider reading the`Automated Testing and Verification` sections of the following reports to review real-world security properties:

- 0x
- Balancer