

Security And Auditing

1.07.28.34

What is an Audit?

An audit is a security focused code review for looking for issues with your code.

```
7 // Follow https://en.wikipedia.org/wiki/Programmer\_s\_dilemma
8 contract EtherStore {
9     mapping(address => uint256) public balances;
10
11     function deposit() external payable {
12         balances[msg.sender] += msg.value;
13     }
14
15     function withdraw() external {
16         uint256 balance = balances[msg.sender];
17         require(balance > 0);
18         (bool success, ) = msg.sender.call{value: balance}("");
19         require(success, "Failed to send Ether");
20         balances[msg.sender] = 0;
21     }
22
23     // Helper function to check the balance of this contract
24     function getBalance() external view returns (uint256) {
25         return address(this).balance;
26     }
27 }
```

For example:

We have same codes looks like this.

IN here , sends and then update balances

This code vulnerable to reentrancy attack.

And Audit is something that an auditor would catch.

Audit TIPS AND TRICKS

<https://twitter.com/tinchoabbate/status/1400170232904400897>

1/ You cannot imagine the countless hours you save an auditor by just stating what you intend to do with that crazy obscure low-level assembly math thing that just multiplies two numbers. So, add comments.

2/ Document everything that is part of the contracts' public API. If functions are private / internal but are super sensitive, document them as well. Big big big plus for using NatSpec format!

3/ Test! Countless critical vulns can be saved with simple unit tests. Also, tests let us understand intended behavior. A trick some auditors use: if a public function is not being called in the tests, that's where they aim their guns. Yes, it really works to find bugs

4/ An easy one, but so overlooked. Make sure the project compiles and test suite passes. Yeah, that basic. First thing we do before auditing is building and running the tests. It's just sooo disappointing when it fails.

5/ Don't assume that we'll know your code, or we've heard of that EIP you're implementing, or we've seen that crazy algorithm / data structure you're coding. We need documentation, and time to understand it. So have plenty of docs ready for us to start digging on the first day.

6/ Be ready to have an open comms channel. There'll be lots of things we don't understand, and we'll need you to help. Depending on the auditors' style we might be more quiet or more chatty, but knowing that you'll be there and responsive is SUCH a big plus.

7/ Don't try to be so clever with optimizations. First code for correctness and security. I've seen so many vulns coming from trying to optimize simple operations. And if you use some obscure trick to optimize, add comments explaining why.

8/ A crazy naming system doesn't make anyone a better dev. It just badly hinders onboarding of new people to the ecosystem. U cn b bttr thn ths.

9/ Clean up old comments, unused functions, etc. Make your code shiny and pretty. We'll be reading it for weeks, and it's tough having to spend so many hours reading something that is SO difficult to look at.

10/ Some projects to check dev practices: Uniswap, Augur, Compound, UMA, Optimism. Not only see their Solidity. See how they test, deploy, upgrade. See the open discussions in issues and PRs. They might not be perfect, agreed, but I've really enjoyed reading their code.

11/ Don't reinvent the wheel. It's 2021, and you shouldn't be coding your own ERC20 from scratch. Same for other ERCs, governance, access controls, etc. Use known secure libraries. It's safer, the audit's scope will be smaller, and you'll save money.

12/ Try not to be so secretive about your roadmap and future developments. We'll likely ask about it. The more you share, the easier it'll be to understand the big picture of what you're building, and what kind of attack vectors are more relevant.

13/ Auditing a moving target is so complicated - we need to constantly adapt and change our mental model of the system. So if you can keep that commit frozen, or at least plan in advance when you'll change it, oh dear we'll WORSHIP you.

14/ If you can run some high level code walkthroughs for us, specially on the first days, that's gonna be great. Imagine we're a dev you're onboarding to your code base: what'd you show ? what'd you warn ? what'd you highlight ?

19/ Don't put unrealistic expectations on us auditors! Instead let's work together and build trust, so we can become your security partners for the long run. Because yes, you can get along with these crazy Solidity nerds that are about to break your code for good.

15/ Ideally, I think should see a security audit as valuable process beyond finding or not a specific critical issue. Of course we want to do that, but we can also have insightful conversations just asking and entertaining ideas for attack vectors, design considerations, etc.

16/ From day one, think of the audit as a best-effort code review with a security-oriented adversarial mindset. Lots of misconception and false expectations come from confusions with the "real" legal / financial auditors. We're not that kind (nor plan to be).


17/ Considering that, don't assume we'll find every single vulnerability forever and that your code is perfect once the audit is done. Not even if after our report you managed to fix everything. Which means...

18/ Security incidents can happen after an audit. Be sensible, assume you can be hacked, get insurance, and set up monitoring infrastructure and recovery plans.

<https://learn.openzeppelin.com/security-audits/readiness-guide>



A Comprehensive Smart Contract Audit Readiness Guide

 [DOWNLOAD PDF VERSION](#)


 [EXPLORE DEFENDER](#)

Table of Contents

[Preparing for a Smart Contract Security Audit](#)

[Understanding Audits](#)

[Optimal Audit Timing](#)

[Audit Readiness Checklist](#)

[What Audit Clients Can Expect](#)

Preparing for a Smart Contract Security Audit

Understanding Audits

Smart contract audits provide Web3 developers valuable feedback to address the novel security challenges of distributed systems. Web3 developers must contend with the extreme variability of code, as well as the ongoing evolution of both individual projects and the surrounding ecosystem. Moreover, accumulating value in smart contract systems and DAOs (decentralized autonomous organizations) only makes security considerations more salient. Audits allow a project's developers to demonstrate that their code has been thoroughly inspected and battle-tested, thereby fostering trust and encouraging the project's adoption by prospective users.



Help your auditors!

Add comments

This will help your auditors understand what you're doing.

Use natspec

Document your functions. DOCUMENT YOUR FUNCTIONS.

Test

If you don't have tests, and test coverage of all your functions and lines of code, you shouldn't go to audit.

If your tests don't pass, don't go to audit.

Be ready to talk to your auditors

The more communication, the better.

Be prepared to give them plenty of time.

They literally pour themselves over your code.

"At this time, there are 0 good auditors that can get you an audit in under a week. If an auditor says they can do it in that time frame, they are either doing you a favor or they are shit. " - Patrick Collins, March 4th, 2022

Process

An auditors process looks like this:

1. Run tests
2. Read specs/docs
3. Run fast tools (like slither, linters, static analysis, etc)
4. Manual Analysis
5. Run slow tools (like echidna, manticore, symbolic execution, MythX)
6. Discuss (and repeat steps as needed)
7. Write report ([Example report](#)) ([example report](#))

Typically, you organize reports in a chart that looks like this:

	Likelihood			
		Low	Medium	High
Impact	Low	Low	Low	Medium
	Medium	Low	Medium	High
	High	Medium	High	Critical

STATIC ANALYSIS : Is the process of just running some program to read over all your code and look for commonly known bugs. One of the most popular static analysis tools is going to be SLITHER.

Project folder : hardhat-security-fcc

We will do here static analysis.

Clone Patrick's github repo

```
git clone https://github.com/PatrickAlphaC/hardhat-security-fcc.git .
```

These contract have different vulnerabilities.

In contracts/BadRNG.sol

```
function pickWinner() external {  
    uint256 randomWinnerIndex = uint256(  
        keccak256(abi.encodePacked(block.difficulty, msg.sender))  
    );  
    address winner = s_players[randomWinnerIndex % s_players.length];  
    (bool success, ) = winner.call{value: address(this).balance}("");  
    require(success, "Transfer failed");  
}
```

That picks a random winner of a raffle using `block.difficulty` and `msg.sender`. This isn't truly random. Ad the miners can influence the `block.difficulty` and people can cancel transactions. And there is a ton of different vulnerabilities with creating randomness in this way.

We also have `contracts/LiquidityPoolAsOracle.sol`.

Two most common types of attacks are;

1- Reentrancy

2- Oracle Manipulation. Which luckily for you, we have taught you about decentralized oracle's and working with chain link. Which should make you a lot safer.

In this contract;
contracts/LiquidityPoolAsOracle.sol

We are using a liquidity pool as an Oracle and this is kind of some advanced defi stuff in contract. This is a minimalistic decentralized Exchange example where people can buy and sell and swap different assets.

Now using this singular Exchange to get the swap price (`getSwapPrice()`) is a terrible idea. Because this is a single protocol for a single price. The price from this protocol is a single centralized location and we don't want to get our price from a single centralized exchange. We want to get it from many exchanges. Getting price of any asset from a single decentralized Exchange is not decentralized. As somebody manipulates the market doing some crazy advanced defi things that will run in the price of your assets. So getting the price of your assets from a centralized location is a terrible idea.

```
function getSwapPrice(  
    address from,  
    address to,  
    uint256 amount  
) public view returns (uint256) {  
    return ((amount * IERC20(to).balanceOf(address(this))) /  
        IERC20(from).balanceOf(address(this)));  
}
```


We have a metamorphic Proxy in contracts/MetamorphicContract.sol

```
contract MetamorphicContract is Initializable{
    address payable owner;

    function kill() external{
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

The issue here is that its initialized double and we dont guarantee that the contract has been initialized.

We have classic reentrancy issue here.

```
acts >  Reentrancy.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// This is the age-old reentrancy attack that should make yo
// https://solidity-by-example.org/hacks/re-entrancy/ for fu
// Follow https://twitter.com/programmersmart

contract EtherStore {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external {
```

And we have a vault here where some password is stored on chain and we are crossing our fingers that nobody reads this password to unlock it.

```
contracts > Vault.sol
// Security module

pragma solidity ^0.8.0;

contract Vault {
    bool public s_locked;
    bytes32 private s_password;

    constructor(bytes32 password) {
        s_locked = true;
        s_password = password;
    }

    function unlock(bytes32 password) external {
        if (s_password == password) {
            s_locked = false;
        }
    }
}
```


So we are going to run some static analysis on these contracts. See if that static analysis can spot some of the bad things in here.

Slither

1.07.35.31

<https://github.com/crytic/slither>

<https://github.com/trailofbits>

auditors

To use slither;

install Python 3

pip3 install solc-select

solc-select use 0.8.7

(to install solc version)

solc-select install 0.8.7

pip3 install slither-analyzer

Check slither:

slither --help

The second way is using Docker.

RUN SLITHER:

yarn

yarn slither

```
MetamorphicContract.owner (contracts/MetamorphicContract.sol#6) is never initialized. It is used in:  
- MetamorphicContract.kill() (contracts/MetamorphicContract.sol#8-11)  
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

The first line is issue and there is a reference to that issue.

Red lines means there is a high issue. And reference gives how to correct it.

The green lines low issues.

Slither tool caught issues in
reeantrancy.sol
and metamorphic.sol.

But not caught
vault.sol and
liquidityPool and
badRNG.sol

Fuzzy and eth-security-toolbox

1.07.41.23

MANUAL ANALYSIS:

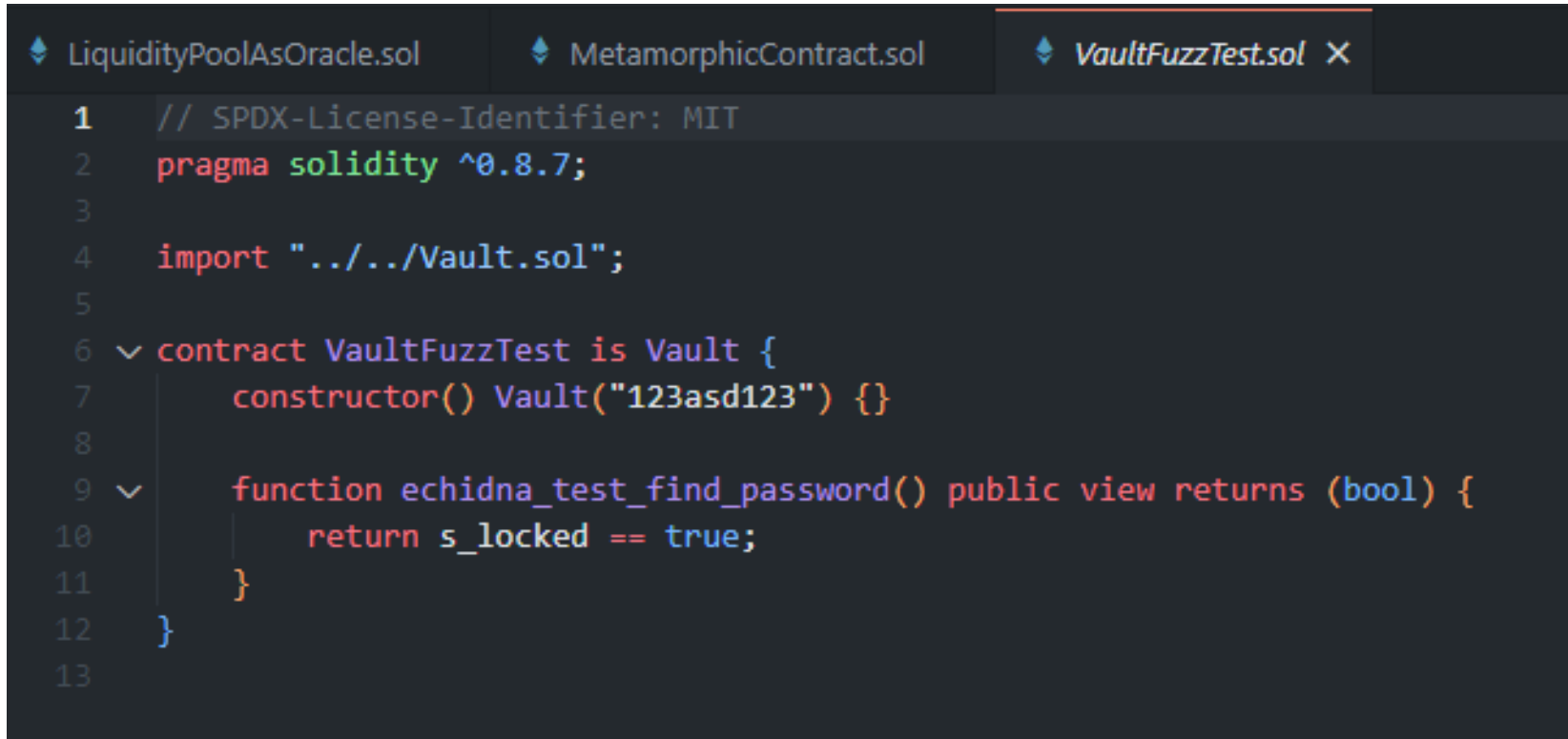
5. Run slow tools (like echidna, manticore, symbolic execution, MythX)

Symbolic execution is where we simulate executing transactions on the blockchain. ECHIDNA tool is one of them.

<https://github.com/crytic/echidna> This is doing fuzzy testing. Fuzzy testing is an automated software testing technique that involves providing invalid unexpected or random data as inputs to a computer program.

In a lot of our code, ofentimes, we are going to get people interacting with them in ways that we will never think about.

So we want to be able to provide random data and random information to our test to see if something weird happens that we were not expecting. So we can actually build our own fuzzy tests in our hard hat projects and run these fuzzy tests.



```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.7;
3
4 import "../Vault.sol";
5
6 contract VaultFuzzTest is Vault {
7     constructor() Vault("123asd123") {}
8
9     function echidna_test_find_password() public view returns (bool) {
10         return s_locked == true;
11     }
12 }
13
```

We have Vault.sol contract. And we think that at first glance, hey, nobody should ever be able to know the password and no one should ever be able to unlock this contract. Which obviously, we know is ridiculous, because we know that anybody can read anything in a storage variable. So we know that this should fail, but it might be hard to write a test to catch that this actually would fail.

A good approach to testing this would be to just send a ton of random bytes32 objects to this unlock function to see if we can unlock it.

We can write a fuzzy test to do exactly that.

To make this test we are going to install eth-security-toolbox and we will use echidna tool

<https://github.com/trailofbits/eth-security-toolbox>

For this we need DOCKER INSTALL.

<https://docs.docker.com/go/wsl2/>

yarn toolbox

```
eemcs@DESKTOP-LJJC06I:~/freecodecamp/hardhat-security-fcc$ yarn toolbox
```

```
yarn run v1.22.19
```

```
warning ../../package.json: No license field
```

```
$ docker run -it --rm -v $PWD:/src trailofbits/eth-security-toolbox
```

```
Unable to find image 'trailofbits/eth-security-toolbox:latest' locally
```

```
latest: Pulling from trailofbits/eth-security-toolbox
```

```
88736512a147: Pulling fs layer
```

```
c563e9c0639d: Pulling fs layer
```

```
.
```

```
.
```

```
.
```

Security Tools and Resources Installed:

<https://github.com/trailofbits/echidna>

<https://github.com/trailofbits/etheno>

<https://github.com/trailofbits/manticore>

<https://github.com/trailofbits/slither>

<https://github.com/trailofbits/rattle>

<https://github.com/trailofbits/not-so-smart-contracts>

Use `solc-select` to switch between different versions of `solc`

ethsec@927590328d3a:~\$

With yarn toolbox a docker Shell begins:

ethsec@927590328d3a:~\$

To test run :

```
echidna-test /src/contracts/test/fuzzing/VaultFuzzTest.sol --contract VaultFuzzTest --config  
/src/contracts/test/fuzzing/config.yaml
```

VaultFuzzTest.sol comes with a config file as well. contracts/test/fuzzing/config.yaml

```
# testLimit is the number of test sequences to run; default is 50000
testLimit: 10000
# maximum time between generated txs; default is one week
maxTimeDelay: 0
# maximum number of blocks elapsed between generated txs; default is expected increment in
one week
maxBlockDelay: 60480
# additional arguments to use in crytic-compile for the compilation of the contract to
test.
cryticArgs:
  [
    "--solc-remaps",
    "@chainlink/contracts=/src/node_modules/@chainlink/contracts",
    "--solc-args",
    "--allow-paths /src/contracts",
  ]
```

DEBUG CONSOLE

It found the password, which means anybody else could immediately find the password. And this would be an indicator that what we are doing there is not a good setup.

Closing Security Thoughts

1.07.47.06

Other resources

<https://github.com/PatrickAlphaC/hardhat-security-fcc#usage>

The end