Lesson 7

This Week

- The Audit Process
- Audit Part 2
- Gas Optimisation Review
- Gas Optimisation game

Underhanded Solidity Contest

Details

The goal of the Underhanded Soldity Contest is to write seemingly innocent and straightforward-looking Solidity code which actually contains malicious behavior or backdoors.

In this year, we would like you to build a simple decentralized exchange where people can trade their hard-earned NFTs, DAO governance tokens, or dog coins of their choice.

The Audit Process

What is an Audit

An audit is:

• An assessment of your secure development process.
• The best option available to identify subtle vulnerabilities.
• A systematic method for assessing the quality and security of code.

An opportunity to:

• Learn from experts
• Identify gaps in your process
• Identify underspecified areas of your system

An audit can not:

• Replace internal quality assurance
• Overcome excessive complexity or poor architecture
• Guarantee no bugs or vulnerabilities

Audit Companies

Open Zeppelin
Certik
Peckshield
Extropy

When choosing a company, yo might want to look at the Rekt News LeaderBoard

Auditing Techniques

It isn't a rule book to follow religiously but it's good to have these things in mind when you feel stuck in a particular project. The actual process of auditing is somewhat personal and you'll probably develop your own as you get more experienced, but here are a few guidelines:

High-Level Understanding

The first time you look into code, you don't necessarily need to be analytically looking for bugs or wrong implementations. You should aim to build a good mental model of how the whole system fits together. Unless some particular vulnerability jumps in front of you, don't focus too much on bugs on your first pass, just try and understand the system as a whole.

For this, a good practice is to skim over each file and read functions names and signatures. In most cases, although not always, the interface alone provides a good representation of functionality as well as the entry points of an application. Pay close attention to the inheritance scheme as it helps clarify the relationship between contracts.

Read the specification. Or not.

This topic is somewhat controversial. Some auditors do read the provided specification as a first step in an audit, as it helps to understand the intended behavior and save some time reasoning about the contracts. The counter argument is that most specifications are written by the developers themselves, and when you read their intentions, you will develop bias which might blind you to the objective facts of the code.

The detailed inspection

There is a multitude of approaches to this. For example, you could look through each `.sol` file individually or you could pick a functionality, say a deposit, and follow its flow, doing a kind of a mental transaction graph. Ideally, you should do both as each provides different kinds of insights.

A good practice is to take some time to actually run the code. Compile it if you can, run tests if they are present or even throw it on remix and use it a little just to get yourself familiar with it.

Preparing for an Audit

Following these steps to prepare for an audit will go a long way to helping you get the best results.

1. Documentation
2. Clean code
3. Testing
4. Automated Analysis
5. Frozen code
6. Use a checklist

- We have a finite amount of time to audit your code.
- Preparation will help you get the most value from us.
- We must first understand your code, before we can identify subtle vulnerabilities.
- Imagine we're a new developer hired to join your team, but we only have a few days to ramp up.

1. Documentation

The less time we spend trying to understand your system, the faster we can get deep into your code, and the more time we can spend finding bugs. This is why the number one thing you can do to improve the quality of your audit is provide good documentation.

Good documentation starts with a *plain English* description of what you are building, and why you are building it. It should do this both for the overall system *and* for each unique contract within the system.

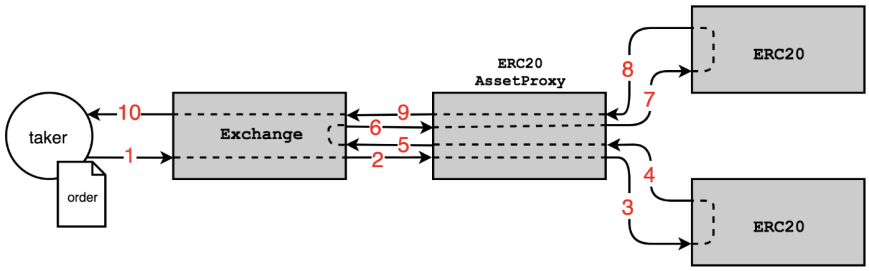
Another marker of good documentation is that it includes a specification of your system's intended functionality. For each contract, it should describe the most important properties or behaviors that should be maintained. It should also describe the actions and states that should not be possible.

One of the best examples we've seen is the protocol spec for the 0xProject. In particular, their use of flow charts nicely illustrates how the system fits together.

Trade settlement

A trade is initiated when an order is passed into the `Exchange` contract. If the order is valid, the `Exchange` contract will attempt to settle each leg of the trade by calling into the appropriate `AssetProxy` contract for each asset being exchanged. Each `AssetProxy` accepts and processes a payload of asset metadata and initiates a transfer. To simplify the trade settlement diagrams below, we assume that the orders being settled have zero fees.

ERC20 <> ERC20



Transaction #1

1. `Exchange.fillOrder(order, value)`
2. `ERC20Proxy.transferFrom(assetData, from, to, value)`
3. `ERC20Token(assetData.address).transferFrom(from, to, value)`
4. ERC20Token: (revert on failure)
5. ERC20Proxy: (revert on failure)
6. `ERC20Proxy.transferFrom(assetData, from, to, value)`
7. `ERC20Token(assetData.address).transferFrom(from, to, value)`
8. ERC20Token: (revert on failure)
9. ERC20Proxy: (revert on failure)
10. Exchange: (return `FillResults`)

Good documentation requires a lot of effort.
It can be useful for the auditors to document the code.
Writing our own documentation of the code's behavior is an excellent way to understand it.
It can even lead us to discover vulnerabilities and unexpected edge cases.

What about a pseudocode spec? I placed an emphasis on "plain English" above (as opposed to rigid/formal English) because plain English more clearly expresses what you *want* the code to do. By contrast, the actual code is often so similar to the pseudocode specification that it can be hard to see when they both describe something you do not actually want.

Pseudocode does have its place and can be especially helpful for precisely describing complex mathematics, but it should always be accompanied by some English about what the math is meant to achieve.

The less time we spend trying to understand your system, the more time we can spend finding bugs.

GOOD DOCUMENTATION:

- Describes the overall system and its objectives
- Describes what should not be possible
- Lists which contracts are derived/deployed, and how they interact with one another

Documenting your code will also help you to improve it.

Example of good documentation: 0x Protocol Specifications

Example from Polymath:

2. Clean up the code

Polished, well-formatted code is easier to read, which reduces the cognitive overhead needed to review it. A little bit of cleanup will go a long way towards allowing us to focus our energy on finding bugs.

1. Run a linter on your code. Fix any errors or warnings unless you have a good reason not to. For Solidity, we like Ethlint. Remix also has a linter integrated at compile time. The [Solidity template](Solidity Template bundles together some useful tools
2. If the compiler outputs any warnings, address them.
3. Remove any comments that indicate unfinished work (ie. `TODO` or `FIXME`). *(This is assuming it's your final audit before deploying to mainnet. If not, exercise your judgement about what makes sense to leave in.)*
4. Remove any code that has been commented out.
5. Remove any code you don't need.

• Add helpful comments: explain the intent, i..e. what are you trying to do

• Using NatSpec (natural specification) comments:

3. Testing

Write tests! A good goal is a test suite with 100% code coverage.

Review the list of test cases for gaps. Are your tests mostly focused on making sure the the 'happy path' works? Write some tests to verify undesirable actions are properly protected against, and that the contract fails properly instead of landing in an undesired state.

Important: Your README should give clear instructions for running the test suite. If any dependencies are not packaged with your code (e.g. Truffle), list them and their **exact** versions.

4. Automated Analysis

Ethereum has many good security analysis tools to help find some of the most common issues. We use some of these during our audits, though you can also run them in advance, which will allow us to spend our time looking for trickier bugs.

The MythX suite, which runs several kinds of analysis at once, is a great place to start. There are many ways to submit your contracts for analysis, including CLI tools for JavaScript and Python as well as plugins for Remix and Truffle.
You can find more security tools listed in Smart Contract Best Practices.

There are useful plugins for Remix such as Mythx
In VSCode Solidity Metrics gives useful information.

It's not essential to do this, but it helps. A caveat is that you will often get warnings about issues that don't actually exist.

5. Freeze the code

> *We can't audit a moving target*

An audit is an investment in the security of your smart contract system. Besides selecting a high quality auditor for the work, there are several things you can do to make sure you get the most out of your investment.

At the start of our audit, confirm that you've "frozen the code" (i.e. halted development), and provide a specific git commit hash to be the target of our audit.

If a change comes in halfway through an audit, it means the auditors wasted time on old code. In addition, the auditors would have to stop and incorporate the change, which can have wide-ranging impacts on things like the threat model and other code that interacts with the changed code.

If your code won't be ready by the scheduled start date It's better to delay altogether than try to complete an audit while you continue development.

6. Use A Checklist

These steps are summarized in a markdown checklist that you can copy and paste for use in your own project.