Lesson 22 - Audits / Governance

Audit Process and Reporting

The audit process varies greatly from company to company, and between individuals as there is, as yet, no generally-accepted industry standard process.

Smart contract auditing is a niche information security service. It arose out of necessity.

Smart contracts audits aim to prevent the pain entrepreneurs, developers and users experience when Ethereum contracts are hacked or otherwise fail.
Immutability implies that repair may be difficult and costly, or impossible.
Immutability implies a requirement for debut production releases to be free of defects, but errors and oversights are likely to remain commonplace as new developers enter the space.
The EVM is an unfamiliar platform, blockchain is, at first, an unfamiliar paradigm, and Solidity is, at first, an unfamiliar language. It is not reasonable to expect perfection from new developers.

Observing projects getting killed by preventable problems increased general awareness of the importance of preventative quality-assurance.
Two approaches shaped the formative Ethereum code security industry.

1. Bug Bounties

The first of these is Bug Bounties. Bug bounties are a time-tested approach to reinforcing information security. Organisations such as HackerOne , organise bug bounties for corporate clients. Bug Bounties are a way of reaching out to large numbers of qualified developers, to possibly discover critical issues.

2. Formal Verification

Formal verification is the process by which one proves properties of a system mathematically. In order to do that one writes a formal specification of the application behaviour. The formal specification is analogous to our Statement of Intended Behaviour, but it is written in a machine-readable language. The formal specification is later proved (or not) using one of the available tools.

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

According to this article , the top auditing companies are
Trail of Bits
Open Zeppelin
Consensys Dilligence
Runtime Verification
Certora

They forget to mention
Extropy

Also when choosing a company, you might want to look at the Rekt News LeaderBoard

---

The Audit Process

> Auditing a smart contract entails a methodical review of the in-scope source code, in order to provide reasonable assurance that the code behaves as expected, and contains no vulnerabilities.

Reasonable assurance is important because it is impossible to ensure a piece of code contains no bugs. Beware of this when wording reports. Declaring a code base is bug free is irresponsible, and can lead to liability problems.

> The company receives a defense against possible liability. The auditor accepts reputational risk.

For emphasis, auditors should apply care to all forms of communication to avoid a situation in which the auditor appears to take on, perhaps unwittingly, liability for the project.

How Will They all Fit Together?

The best processes will mix and layer a number of approaches, increasing the probability of finding a bug, if one exists.

A recent example is MakerDAO's Multi Collateral DAI set of smart contracts. Most of the smart contracts were formally verified and an audit was conducted. This was the start of an excellent process. Even so, a USD $50,000 critical bug was awarded by their Bug Bounty program, demonstrating the value of a Bug Bounty even after audits and formal verification.

The process we recommend is an audit, or audits, followed by a well-funded bug bounty that is open for sufficient time to build confidence in the project and with significant rewards for finding critical bugs.

Code freeze

From a software engineering perspective, a Freeze is a period when the rules that govern changes become more strict. Freezes are used for a variety of reasons. For example a team might implement a Feature Freeze to prevent any new features being added so they can focus on testing, issue resolution, even documentation and marketing collateral. A Specifications Freeze might block further design changes so that implementation of the specification can proceed.

In our case, a Code Freeze is a full code freeze - no changes of any kind while the audit is performed. Smart contract audits are normally performed on repository containing the code, so no commits are permitted during the audit.

This means development is finished. The developers made their best effort to create an application that behaves exactly as specified and contains no bugs.

This is very important. The main reason is obvious: Auditors should look at the version that is going to be deployed. Smart contracts are immutable (we'll get to upgradeability hacks shortly). The audit can be thought of as a dress rehearsal for actual deployment. After deployment, remediation of defects will be either extremely costly or completely impossible. An audit is always about a precise deployment candidate. Future versions of that candidate (if any) must be considered unaudited, since any change is potentially a source of new problems.

The business world applies tremendous pressure on this process. Deadline pressure will invariably push against the ideals of thoroughness and process integrity. As the auditor who accepts reputational risk and endorses the audit finding, your duty is to defend the integrity of the process.

Always request a commit and stick to that during the audit, while also documenting it in the report. Never try to audit a moving target. The effectiveness of your work will be impaired, as will your reputation.

Specifying intended behaviour

The auditor is tasked with ensuring the application behaves as specified. Where, exactly, is application behavior specified? This will vary greatly from project to project, but ideally there should exist a succinct document outlining the goal of the application, what is allowed and what is prevented. We call this a Statement of Intended Behavior. It should be precise and unambiguous so auditors can compare what the developers want to happen and the code that is intended to make it happen.

A Statement of Intended Behaviour will be presented as a separate document, sometimes as part of the repository's wiki or readme.md. Sometimes the document is simply non-existent. In such a case, request that the developer, along with the rest of his team create a document before the audit starts. Input from business-focused professionals is valuable. Sometimes, they will have a clearer view of how the system should behave.

The size of the specification will be proportional to the complexity of the application. To generalize for any application, the specs should include:

- Goal of the application
- Main flows
- The actors / roles and what they do
- Access restrictions
- Failure states to be avoided

One caveat: You will stumble upon specifications that seem to be wrong, and in fact are. If you notice that the owner of the contract can drain the contract of user's funds, it seems obvious that it needs to be reported. But what if the client has specified this as intended behavior?

This is always a tough call, and has been discussed many times such as in Adam Kolar's article and recently in the unsolicited audit of Compound Finance's contracts.

When in doubt, document the issue in the report. The whole purpose of our industry is to create systems where trust is not required, or its role is greatly minimized.

Estimating and price quotes

The goal of an estimate is to efficiently assess the key factors that tend to affect actual effort / hours. In this context, “efficiently” means to limit oneself to a superficial perusal of the code that won’t take too long. The key is to know what to look for.

Many companies quote based on lines of code. In our experience, line count (quantity) is a very poor indicator. Complexity is a better indicator of the actual time required for the audit process. A very large, monolithic smart contract will often be easier to audit than a handful of very small smart contracts that interact in multiple ways.

In our experience, good indicators to note include:

- The count of external calls: The number of external calls is a good indicator because they impact the code base complexity in a number of ways.
  Even simple implementations such as an ERC20 token can have an impact on a calling smart contract: USDT and OMG tokens do not return true for successful transfers, for example.
  Contracts can be maliciously altered too, so if you are calling untrusted contracts, this has to be accounted for. Recently SpankChain was hacked and the attacker used a rogue ERC20 token implementation. The rogue contract implemented the ERC20 standard interface, but when called for a transfer would re-enter SpankChain's contract.
- The count of public / external functions: These are the points of entry. Execution starts here. They will determine the number of paths possible during execution.
- Use of Solidity Assembly: Solidity Assembly takes a lot longer to audit. Code is harder to read, several opcodes that are not accessible via Solidity are at the developer’s disposal and none of Solidity’s usual safeguards apply.
- Code Smell
- Other signs of cleverness, novel solutions: Anything not idiomatic

---

When the Client Proposes the Scope

- To audit only certain files in the overall project
- To audit an amended version of something that was audited before, possibly by someone else.

There are important considerations to keep in mind in these cases.

- **Treat all out-of-scope contracts as untrusted contracts**. This may be counter-intuitive to the client, because they trust them. Again, your duty is to safeguard the integrity of the process and your audit team's reputation. If you do not review them, treat them (and most importantly, calls to them from the in-scope contracts) as interactions with untrusted contracts.
- **Treat all audits as full audits**. It is not uncommon that clients request a follow-up audit on code that has previously been audited and changed just a bit. If you were not the first auditor, make sure to quote a full audit of the code.
  Lastly, if you notice important parts of the code base are out of scope, take time to guide your client to understand the risks involved. Remember, clients and readers of your report are depending on you to identify and raise concerns.

The Process

Extropy uses a very particular process, that we feel is ideal for auditing smart contracts. All audits include three auditors in the team, with the exception of some very low complexity audits, in which case we allow teams of two.

We schedule a debrief meeting close to the delivery day.
lit's not uncommon that a vulnerability will be found by say only two out of three.
This is, itself, an advantage of layering independent audits, diverse sets of experience, and uniquely personal work processes.

We do not require the auditors to follow a prescribed process. Auditors are encouraged to audit using the tools they know and trust, inspecting code in the ways that best suit them. In that debrief meeting, the reports are merged into the final Extropy report that is delivered to the client.

Remediation Period

After the report is delivered the project enters a phase in which the client can report fixes that will be verified and documented by the team. The effectiveness of the fixes is verified by the audit team. This is to confirm that the fixes actually work and, importantly, do not create new issues.

The commits in which each issue was fixed are included, as well as a last-reviewed version both in the summary and in the conclusion of the report.

Our reports can be public, at the discretion of the client.

After the Audit

We encourage clients to proceed to a bug bounty with significant rewards, as another way to layer mitigation of the risk of bugs and their impact.

In bug bounties, the hunters tend to look for critical bugs, but report whatever they see along the way. They tend to not look over the whole codebase, but they spend time in areas that appear to be high-risk. In combination with an audit, the entire code base is secured by an audit, and the high risk areas are further secured by more eyes and more imagination focused on the code. That’s more experts applying their experience, their imagination and their skills to mitigate the risk that something subtle has gone unnoticed.

Audit Report

The Audit Report is the deliverable of the engagement. As such, it's important that it includes defined sections and communicates the project completely. These are the normalized section headings of an audit report:

- Identification of the client
- Date
- Scope (list all files)
- Commit hash and repository address
- Bugs
- Audit Methodology
- Conclusion

Document who requested the audit. It’s acceptable if the client requested anonymity. Your report should indicate this explicitly. Also document the date the audit was published, the files reviewed, bugs and concerns discovered and an overall conclusion about the health of the application.

Be aware of the audience, for example, the client might be a Venture Capitalist with limited understanding of the technical details.

Not all Audit Reports are prepared for such diverse audiences. If delivering to developers on a confidential basis, it may be acceptable to be less didactic while ensuring that bugs are clearly and concisely described and that the introduction and conclusion can be understood by the average ethereum user (a technically literate user).

In particular, be sure to describe the potential impact (why it matters) in terms that are understandable by the widest possible audience, and explanations in terms a developer can parse to comprehend the precise nature of the bug without further explanation.

Reporting Bugs

Bug reports are the main product of both audits and bug hunts. A bug report is only as good as the understanding it provokes in the mind of the receiver. The central task of a bug report is to make the issue crystal clear to other people.

Also keep in mind that the audience of a bug report is often the very people who either wrote the code or audited it. They have looked at it from many angles and your task is to change their minds about something they thought was correct.
Explain in a matter-of-fact, non-accusatory tone and include sufficient information to support your claims.

Clearly describe the problem, the consequences, steps to get there, impact, severity and optionally a suggested direction for the fix.

Keep in mind that bugs are subjective. Indeed, considerable controversy can swirl around exactly what is and what is not a bug. For example, under certain conditions that probably cannot possibly exist, something terrible could happen. Or, under everyday conditions, something odd can happen but it is of no serious consequence.

Some good examples:

The ERC20 Approval Attack:

This is how the original ERC20 approval attack was described. Although the format is unusual, it has everything that a well-described bug should have: the context, the steps to the exploit, a brief analysis and a possible workaround.

CryptoKitties empty fallback:

This was found by Nick Johnson during the initial crypto kitties bug bounty. It follows a format more likely to be found in bug bounty and audit reports with concise explanation and consequences.

Categorization of Severity

Risk ratings, as well as the processes we've just seen, vary greatly between organizations. Each company will have its own way to classify bugs. Even when the familiar categories of critical, major, and minor are used, the definitions of what's included are inconsistent between firms.

This is another example of organizations in the space working independently on their own processes in a standards-free setting. When one finds a bug, it's important to categorize it properly, according to the local customs of the audit team or bug bounty program.

> You need to be prepared to defend your classification of the bug as well as your description of the bug.

Several industry standards (in the wider security industry, not smart contract audits) address the topic. The most prominent of these is the OWASP (Open Web Application Security Project) risk classification standard, which is used by the Ethereum Foundation and many others. Another is the CVSS (Common Vulnerability Scoring System).

OWASP

Risk = Likelihood * Impact

Risk equals the Likelihood of something materializing (or, in our case, the likelihood of the bug being exploited) times the Impact caused when it happens. This formulation is pervasive. It applies to everyone assessing risk across all domains.

With this understanding in mind, let's look at how OWASP breaks down likelihood and impact, making a previously purely interpretative assessment more objective.

Likelihood

OWASP breaks likelihood into two sub-dimensions. The final score is usually a simple average of all the values. The factors are:

Threat Agent Factors

Threat agent is the possible attacker. The goal here is to estimate the likelihood of a successful attack by this group of threat agents. Use the worst-case threat agent.

- **Skills**: How technically skilled is this group of threat agents?
- **Motive**: How motivated is this group of threat agents to find and exploit this vulnerability?
- **Opportunity**: What resources and opportunities are required for this group of threat agents to find and exploit this vulnerability?
- **Size**: How large is this group of threat agents?

Vulnerability Factors

The next set of factors are related to the vulnerability involved. The goal here is to estimate the likelihood of the particular vulnerability involved being discovered and exploited. Assume the threat agent selected above.

- Ease of discovery: How easy is it for this group of threat agents to discover this vulnerability?
- Ease of exploit: How easy is it for this group of threat agents to actually exploit this
- Awareness: How well-known is this vulnerability to this group of threat agents?
- Intrusion detection: How likely is an exploit to be detected?

Impact

Impact is usually measured in financial terms, in OWASP's case it also derives from a number of factors:

Technical Impact Factors

Technical impact can be broken down into factors aligned with the traditional security areas of concern: confidentiality, integrity, availability, and accountability. The goal is to estimate the magnitude of the impact on the system if the vulnerability were to be exploited.

- Loss of confidentiality
- Loss of integrity
- Loss of availability
- Loss of accountability

Business Impact Factors

The business impact stems from the technical impact, but requires a deep understanding of what is important to the company running the application. In general, you should be aiming to support your risks with business impact. The business risk is what justifies investment in fixing security problems.

The factors below are common areas for many businesses, but this area is even more unique to a company than the factors related to threat agent, vulnerability, and technical impact.

- **Financial damage**
- **Reputation damage**
- **Non-compliance**
- **Privacy violation**

OWASP provides a nice spreadsheet so we don't have to reinvent the wheel.

Although OWASP's model is the industry standard, when we look at our niche (Ethereum smart contracts), we'll find simpler models.
The model below is very simple, but can be applied to most smart-contract-only bug bounties:

- Critical: Stealing user funds, freezing funds in the smart contracts.
- Major: A user obtains advantage over others in an unintended way.
- Minor: Bugs that can cause friction to users, but put no funds at risk and create no unfair advantages for particular users.
- Informational : A suggested better approach or optimisation

---

Auditing Techniques in more detail

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.

Auditing - the client perspective

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.

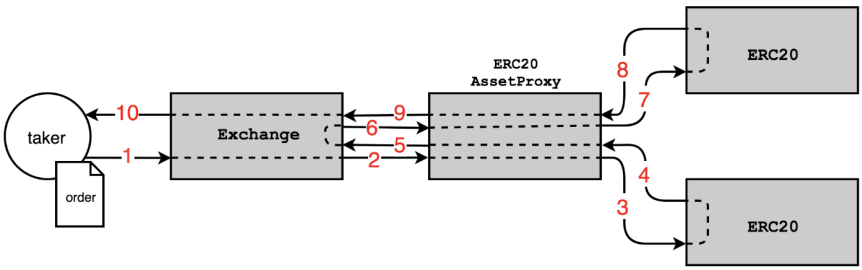
Another marker of good documentation is that it includes a specification of your system's intended functionality. For each contract, it should describe the most important properties or behaviors that should be maintained. It should also describe the actions and states that should not be possible.

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.
The Solidity 2 UML tool is good for visualisation.

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.

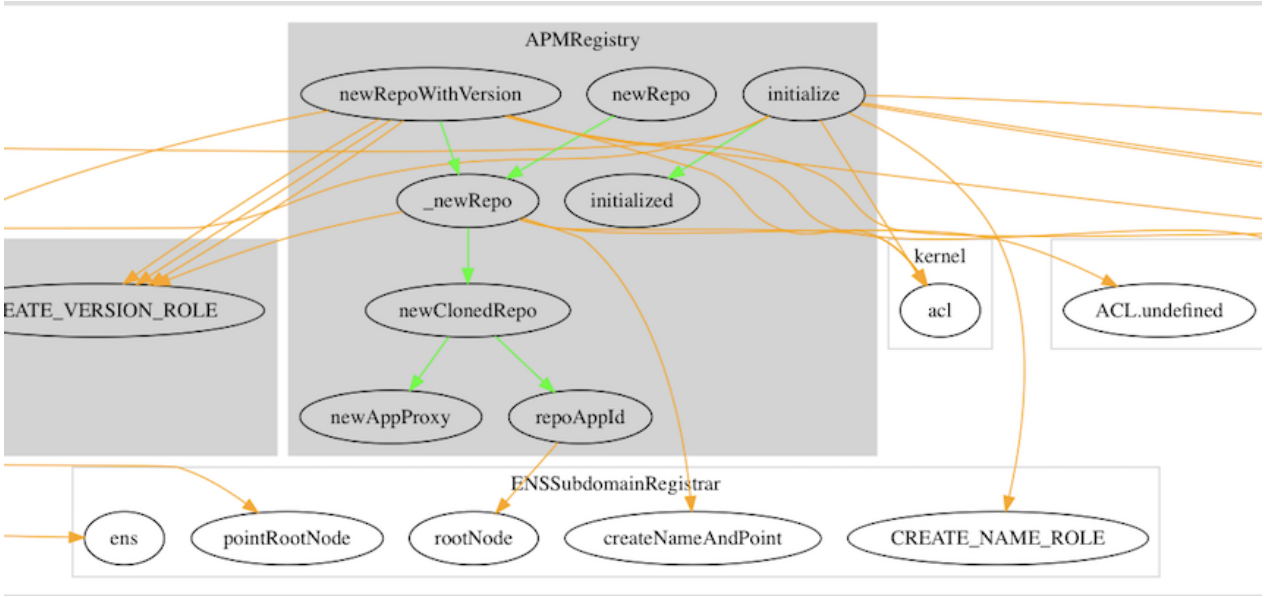
6. Use A Checklist

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

Static Analysis and Visualisation Tools

Visualisation

- EVM Lab
- Surya



- Piet
- Solidity Metrics
- Solidity 2 UML tool

Static and Dynamic Analysis

- [Mythx] (https://mythx.io/) - also available as a remix plugin
- Slither
  - List of Detectors
- Echidna
  -Vertigo - Mutation testing framework
  -Manticore

Checklists

SWC Registry
SCSVS
Smart Contract Security Verification Standard 14-part checklist created to standardize the security of smart contracts.
Example Token Checklist

Consensys Known Attacks

- Reentrancy
- Oracle Manipulation
- Frontrunning
- Timestamp Dependence
- Insecure Arithmetic
- Denial of Service
- Griefing
- Force Feeding

Development recommendations

Token Checklist

Solidity Bugs by version

Audit Competitions

You may want to submit your code to Code Arena or practice your skills by entering their competitions.

Governance

“The greatest challenge that new blockchains must solve isn’t speed or scaling, it’s governance”

- Kai Sedgwick - Why Governance is the Greatest Problem for Blockchains To Solve

We are concerned with how blockchain protocols develop and can adapt to circumstances, rather than how blockchains are used in say administrative settings.

It is useful to think of governance in the following areas

- Consensus
  Who is involved and how do they come to consensus ?

- Information
  How does relevant information reach the participants ?

- Incentives
  How are the incentives aligned to ensure
  - Correct Behaviour
  - There is a sufficient level of participation

- Procedures
  In a decentralised system how are
  - Proposals made
  - Votes submitted
  - Consensus reached

On Chain

Explicit on-chain governance is typically touted as having several major advantages.

- First, unlike the highly conservative philosophy espoused by Bitcoin, it can evolve rapidly and accept needed technical improvements.

- Second, by creating an explicit decentralized framework, it avoids the perceived pitfalls of informal governance, which is viewed to either be too unstable and prone to chain splits, or prone to becoming too de-facto centralized

Off Chain

The mechanism to change the protocol are external to the system

The process is often

- ad hoc
- may be poorly specified
- communication and coordination can be problematic

Developers may have a key role in deciding and implementing changes to the protocol

Bitcoin

Actors :

- Miners
- Developers
- Users (Exchanges / Wallets)

Governance mainly off chain through improvement proposals
A high degree of coordination is needed, done via mailing lists

Results of the nature of Bitcoin Governance :
“This results in a self-reinforcing cycle of more power becoming concentrated in a small group of early core developers, slower technological advancement, and conservatism. Developers are at risk of being bribed since they have a lot of power but weak economic incentives. “

“Similarly, asymmetries in ability to coordinate give miners disproportionate power. Communication amongst miners is easier because they are a small and concentrated group. Since mining is a business with economies of scale, we’d expect a continued trend towards natural monopoly in mining and even greater coordination advantage. “

From : article**

Bitcoin Cash hash wars in late 2018.

“Jihan (Bitmain’s CEO) does have a lot of control for now, and much of that is simply due to mining centralization. As Bitmain is so vertically integrated, from selling ASICs, to operating mining farms, to running mining pools, he can prevent network upgrade and attempt to hijack the Bitcoin brand with things like Bitcoin Cash”

- Samson Mow (CSO of Blockstream - http://fortune.com/2017/08/25/bitcoin-mining/)

Ethereum

- Similar to Bitcoin
- Ethereum founder Vitalik Buterin seen as a “benevolent dictator”

- Some on chain governance over system parameters, e.g. Miners can vote on gas price. See article

Tezos

'Self Amending Ledger'

- Proof of Stake Consensus
- Governance Process
- Code updates are open to anyone
- On chain vote pushes change to test network
- Confirming vote pushes change to the live network
- Contributions are rewarded with tokens
- Power moves away from miners and developers
- Allows delegated democracy

paper 1
abstract

Tezos white paper

- A blockchain protocol can be decomposed into three distinct protocols:
- The network protocol discovers blocks and broadcasts transactions.
- The transaction protocol specifies what makes a transaction valid.
- The consensus protocol forms consensus around a unique chain.
- Tezos implements a generic network shell. This shell is agnostic to the transaction protocol and to the consensus protocol.
- There is the ability to replace the current protocol by one on the test network
- Amendments are adopted over election cycles lasting 131 072 blocks each. Given the a one minute block interval, this is about three calendar months.
- The election cycle is itself divided in four quarters of 32 768 blocks.

Terzos implemented their first on chain governance in May 2019

There was a series of stages

Proposal Stage (gas limit)

```
Athens A: 71% (102 bakers)
Athens B: 29% (68 bakers)
```

Exploration Period

Yay/Nay/Pass vote:

```
Yay: 57.86% (178 bakers)
Nay: .02% (3 bakers)
Pass: 42.12% (13 bakers)
```

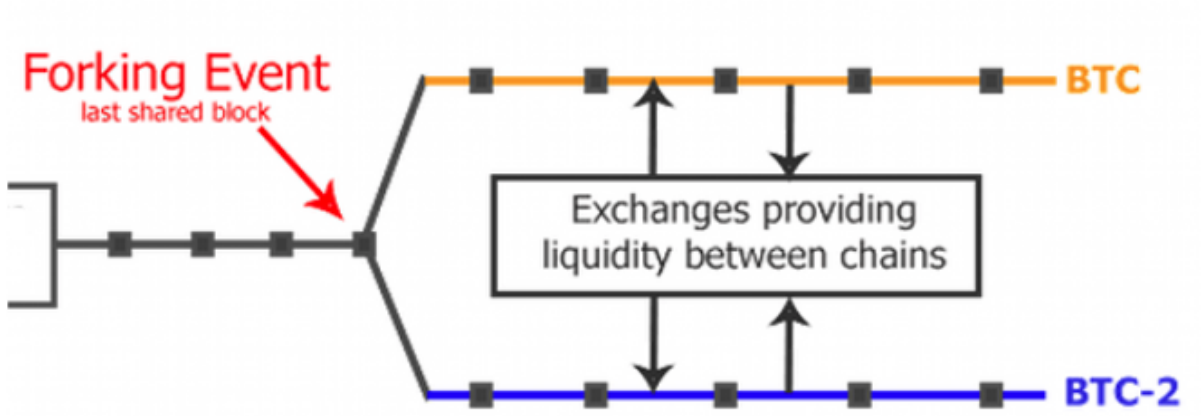
Testing Period

Promotion period

```
Yay: 64.94% (200 bakers)
Nay: .07% (3 bakers)
Pass: 34.98% (12 bakers)
```

When all else fails : Exit Strategies

- Hard and Soft Forks
- Software Forks



See https://fork.lol/

Governance Tokens

Governance is non trivial , as seen on Ethereum and Bitcoin
Various attempts at governance have been tried with on chain / off chain or hybrid models

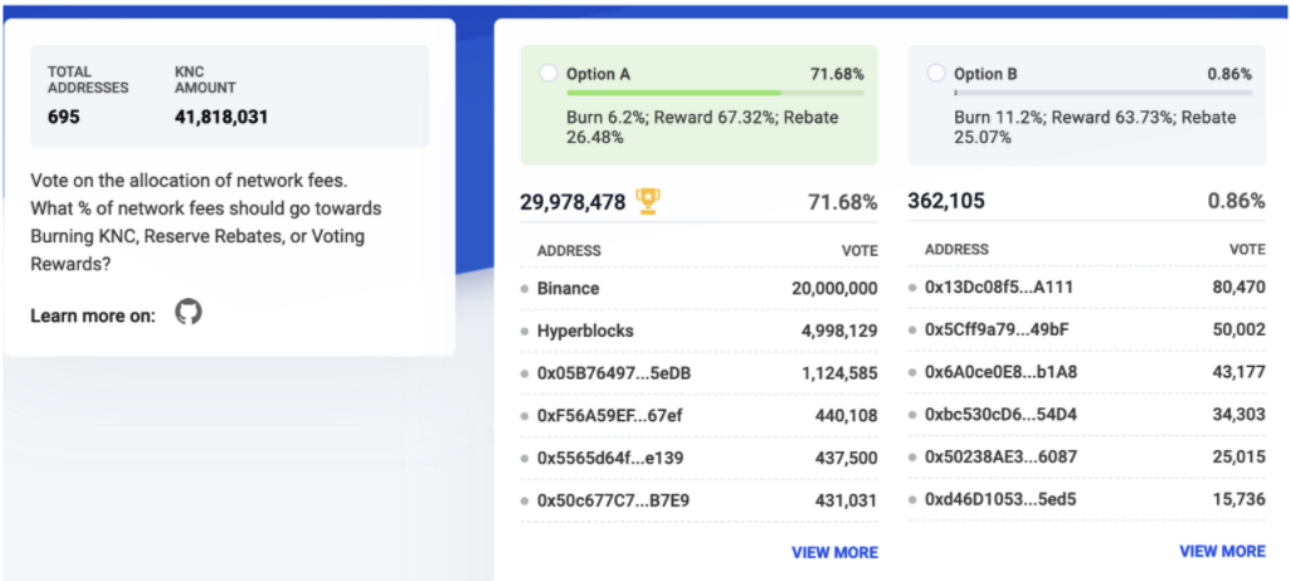
Incentives are (needed) given for participants in the governance process

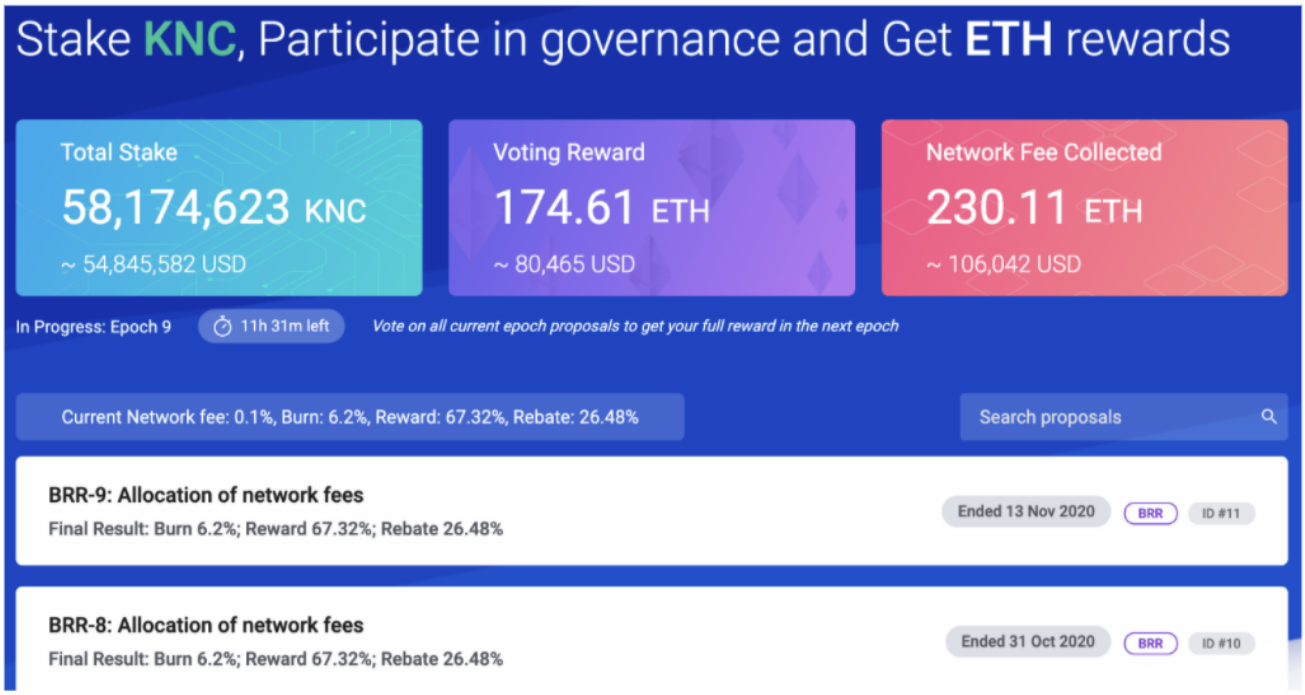
Many DeFi projects issue governance tokens, though with a different purpose.

DeFi Governance Tokens

Holding the token gives the holder the right to vote on aspects of the protocol, typically economic settings, inclusion of assets

The tokens may have a yield





DeFi and Governance

See article

Compound developers turned over the operation and ownership of the network to the community.
The Compound Governance DAO gave the community members control of the protocol's reserve assets that are generated via fees from borrowers. These cash flows were at the time the highest revenues ever generated by an on-chain protocol.

Their mechanism is now

"Screenshot 2022-03-22 at 16.40.50.png" is not created yet. Click to create.

The Compound protocol is governed and upgraded by COMP token-holders, using three distinct components; the COMP token, governance module (Governor Bravo), and Timelock. Together, these contracts allow the community to propose, vote, and implement changes through the administrative functions of a cToken or the Comptroller. Proposals can modify system parameters, support new markets, or add entirely new functionality to the protocol.

COMP token-holders can delegate their voting rights to themselves, or an address of their choice. Addresses delegated at least 65,000 COMP can create governance proposals; any address can lock 100 COMP to create an Autonomous Proposal, which becomes a governance proposal after being delegated 65,000 COMP.

When a governance proposal is created, it enters a 2 day review period, after which voting weights are recorded and voting begins. Voting lasts for 3 days; if a majority, and at least 400,000 votes are cast for the proposal, it is queued in the Timelock, and can be implemented 2 days later. In total, any change to the protocol takes at least one week.

Governance token Value

Protocols may try to claim to their token has no value

Yield Finance
“In further efforts to give up this protocol (mostly because we are lazy and don’t want to do it), we have released YFI, a completely valueless 0 supply token. We re-iterate, it has 0 financial value. There is no pre-mine, there is no sale, no you cannot buy it, no, it won’t be on Uniswap, no, there won’t be an auction. We don’t have any of it.”

Within a week it was worth $3000 and was giving returns of 35,000%

YFI demonstrated that the promise of governance alone could bootstrap network adoption. The fair-launch model, and its use of initial token distribution to target the ideal future users, has since become prevalent.

NFT DAOs

Some DAOs use their DAO governance token to manage their treasury, perform asset sales (including proceeds from fractionalization), and for asset curation.
DAO tokenholders have the right to vote on these issues and in many cases, the outcomes of these votes are directly executed on-chain algorithmically using DeFi protocols such as Fractional or Uniswap.

Gaming DAOs

Unlike in traditional gamer guilds play-to-earn mechanics found within games like Axie Infinity can encourage cooperative strategies and revenue sharing amongst participants. These mechanics make them more like DeFi DAOs — participation in the network earns rewards while also boosting the network's prospects — but to this point the governance of the networks are less tied to pure financial metrics and more tied to game performance and social metrics.

See also survey of DAOs

DAO name	DAO platform	#Funds in USD	#Members
PieDAO	Aragon	73,829,906$	2,881
mStable	Aragon	38,263,266$	8
dxDAO	DAOstack	17,581,208$	444
Airalab	Aragon	13,263,696$	11
Aragon Trust	Aragon	7,015,477$	5
Aragon Network Budget	Aragon	5,903,309$	3
MetaCartel Ventures	DAOhaus	5,619,718$	99
Aavegotchi	Aragon	5,059,662$	3
API3 DAOv1	Aragon	2,991,833$	30
Aragon Network	Aragon	2,932,121$	5

Table 5: Top 10 DAOs by a total of cryptocurrencies in USD, as of 1st December2020.

(Faqir-Rhazoui Y., et al., 2021)

DAO name	DAO platform	# Funds in USD	# Members	% Voter Participation
Uniswap	Compound	5.1 B	1204	0.5%
Compound	Compound	1.7 B	987	0.6%
Radicle	Compound	653.9 M	60	1.1%
Rarible	Gnosis Safe/Snapshot	369.8 M	2,067	8.3%
Badger DAO	Aragon	179.9 M	4	0.01%
Kusama	Substrate	165.6 M	1,106	37.1%
Balancer	Gnosis Safe/Snapshot	159.5 M	5,841	16.7%
API3 DAOv1	Aragon	124.3 M	9	29.0%
Fei	Compound	93.9 M	592	4.1%
Barnbridge	Independent	89.3 M	13	0.01%

(Retrieved August 2021)

Open Zeppelin Governance Contracts

Meta governance

See article.
It is commonly defined as holding one DAO's token in order to influence decisions in another DAO(s). The benefits of meta governance are clear - DAO2DAO relationships are positive-sum incentive-alignment mechanisms that amplify the voices of individuals.
According to the article there has been a change over time

1. Token holders believing they can participate in all governance decisions
2. Token holders realizing they can't participate in all governance decisions
3. Token holders delegating to individuals with perceived specialized expertise and bandwidth
4. Token holders and individual delegates realizing delegate models have been constructed ineffectively

While the trend of governance delegation to individuals had all the best of intentions, it is clear that it has fallen short of expectations. The combination of the time-commitment and depth required for participation, misaligned incentives and accountability mechanisms, and legal complexity has made it impossible for governance delegation to fulfill its promise.

Because of this underperformance, it is clear that the rising prevalence of metagovernance committees is the next logical experiment to drive meaningful progress within DAOs.
Metagovernance committees are better positioned to create aligned incentives with stakeholders and have structures suited to provide scaled governance impact.

Optimistic Governance

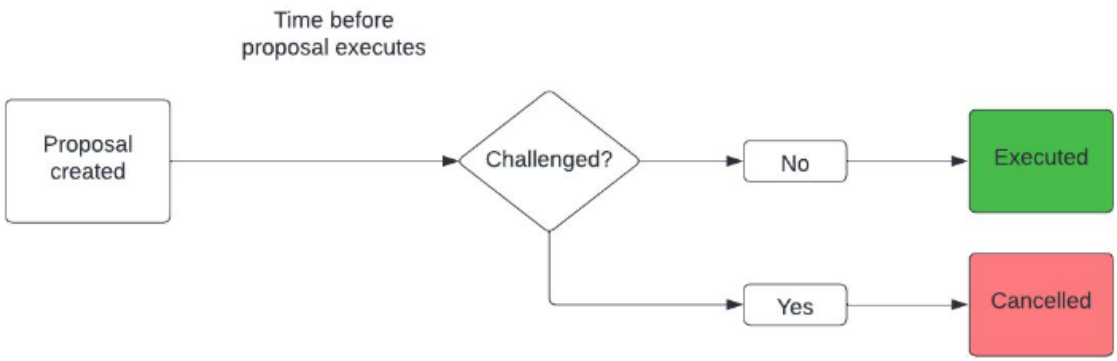
Based on a talk from Tom Waite from Fei / Tribe DAO

Models of Governance in DeFi

- Direct Democracy - token holders vote in a DAO
- Delegated Democracy - others vote on your behalf
- Multisig Governance - High context individuals make the decisions
- Optimistic Governance - Assent is assumed, but the community can veto

With Direct Democracy there is a trade off between building consensus and deadlines for a decision.

Optimistic Governance



Tribe DAO

See docs

There are 4 layers

1. Tribe DAO

- The highest level entity and has ultimate control over managing the protocol.

2. Optimistic governance pods

- These are working groups of community members and protocol experts sitting on a Gnosis safe connected to a timelock. They are able to optimistically govern specific parts of the protocol.
- Specifically, there is a top level Tribal Council pod that oversees other pods and is able to run the protocol on an operational day to day basis. Other pods will be created in the future to manage different aspects of the protocol

3. Nope DAO

- A sub-DAO with a low quorum that is specifically able to veto governance pod proposals.

4. Guardian

- An emergency multisig operated by the Core teams which can take limited safety and security actions in the event of a protocol emergency.

Tribe DAO

The Tribe DAO has ultimate control over the Tribe ecosystem. It has the highest level access control roles, including:

- Arbitrarily moving PCV
- Minting FEI
- Creating and granting new access roles

The Tribe DAO is controlled by Tribe token holders and in order to perform an action it requires a proposal to be created and passed. The proposal threshold is 2.5M TRIBE, with quorum being 25M TRIBE.

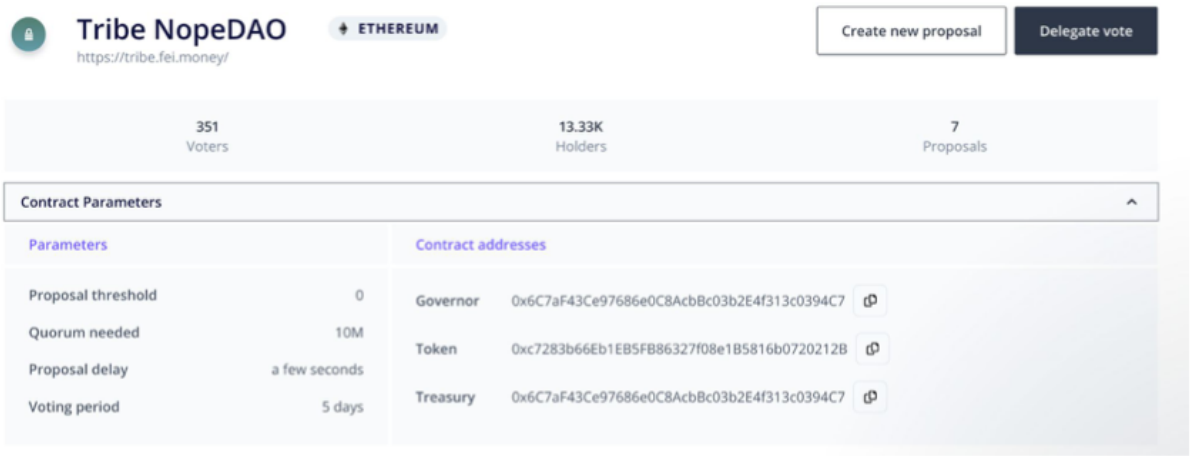
- Voting period: 2 days
- Timelock period: 1 day

Optimistic Governance Pods

The governance pods are the core of the optimistic governance process within the Tribe ecosystem. They are the primary way in which the protocol is managed on an operational, day to day basis. Most proposals do not require a full expensive DAO vote, and instead they can be approved in an optimistic fashion via a pod.

Optimistic pods: Built as an Orca pod (Gnosis safe with an NFT membership wrapper) + timelock

Veto: Built as a sub DAO, whose only role is the veto



Putting it together...