Lesson 12 - Security / Auditing / Monitoring

Security / Best Practices

Consensys Best Practices

General

- Prepare for Failure
- Stay up to Date
- Keep it Simple
- Rolling out
- Blockchain Properties
- Simplicity vs. Complexity

Precautions

- General
- Upgradeability
- Circuit Breakers
- Speed Bumps
- Rate Limiting
- Deployment
- Safe Haven

Solidity Specific

- Assert, Require, Revert
- Modifiers as Guards
- Integer Division
- Abstract vs Interfaces
- Fallback Functions
- Payability
- Visibility
- Locking Pragmas
- Event Monitoring
- Shadowing
- tx.origin
- Timestamp Dependence
- Complex Inheritance
- Interface Types

- EXTCODESIZE Checks

Token Specific

- Standardization
- Frontrunning
- Zero Address
- Contract Address

Documentation

- General
- Specification
- Status
- Procedures
- Known Issues
- History
- Contact

Attacks

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

General Security Best Practices

- Before Using Yul, Verify YOUR assembly is better than the compiler's
- Using Vanity Addresses with lots of leading zeroes
  Why? Well if you have 2 addresses - 0×000000a4323... and 0×0000000000f38210 because of the leading zeroes you can pack them both into the same storage slot, then just prepend the necessary amount of zeroes when using them. This saves you storage when doing things such as checking the owner of a contract. (But dont use the profanity tool to create this, see the Wintermute hack below.)

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. Organizations such as HackerOne , organize 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 behavior. The formal specification is analogous to our Statement of Intended Behavior, 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

Open Zeppelin
Certik
Peckshield
Extropy

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

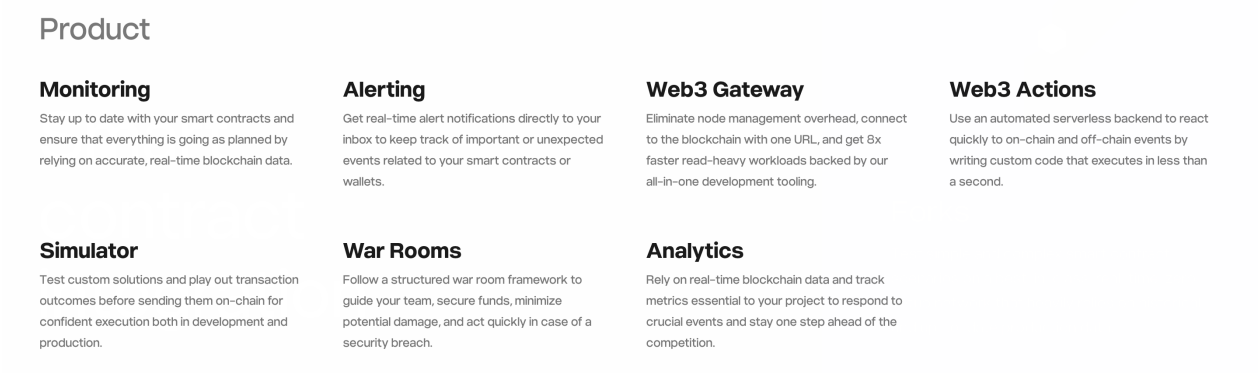
Document who requested the audit. It's acceptable if the client requested anonymity. Your report should indicate this explicitly. Also document the date the audit was published, the files reviewed, bugs and concerns discovered and an overall conclusion about the health of the application.

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

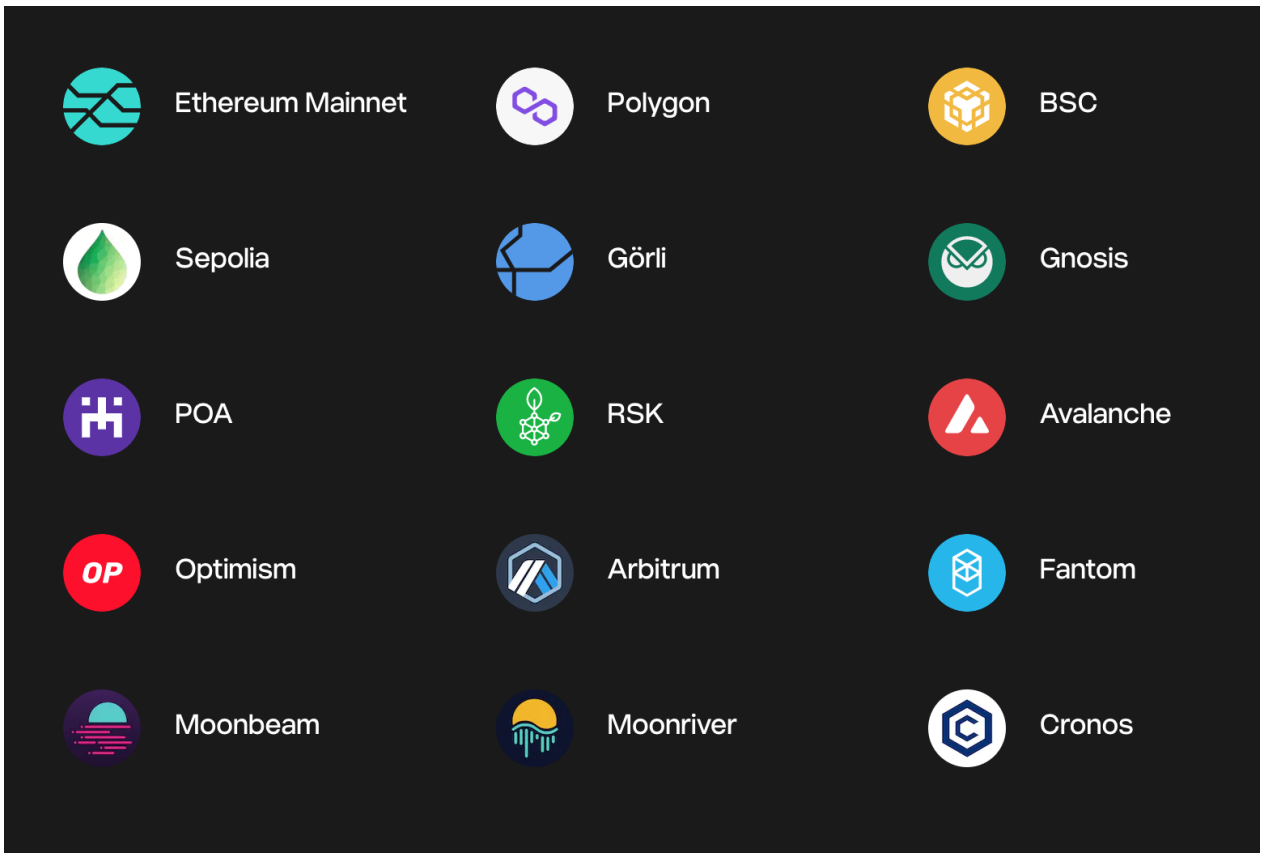
> Not all Audit Reports are prepared for such diverse audiences. If delivering to developers on a confidential basis, it may be acceptable to be less didactic while ensuring that bugs are clearly and concisely described and that the introduction and conclusion can be understood by the average ethereum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

Monitoring

Tenderly

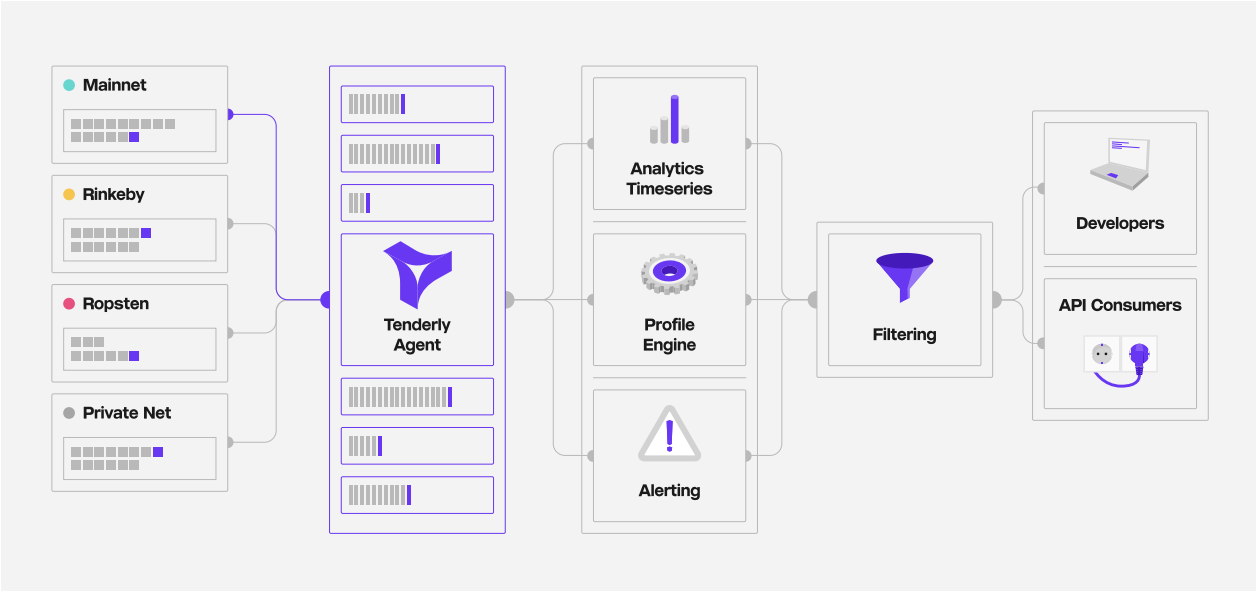


Supported Networks

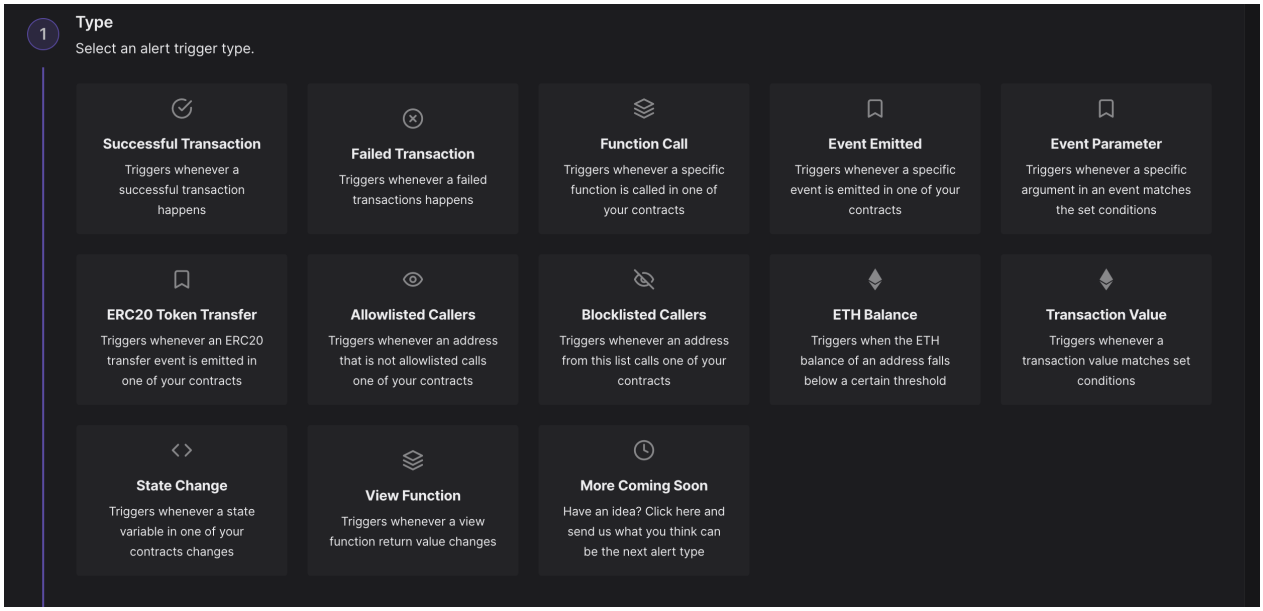


See [Documentation] (https://docs.tenderly.co/)

Monitoring


Mainnet
Rinkeby
Ropsten
Private Net
Tenderly Agent
Analytics Timeseries
Profile Engine
Alerting
Filtering
Developers
API Consumers

Alerts


1
Type
Select an alert trigger type.
Successful Transaction
Triggers whenever a successful transaction happens
Failed Transaction
Triggers whenever a failed transactions happens
Function Call
Triggers whenever a specific function is called in one of your contracts
Event Emitted
Triggers whenever a specific event is emitted in one of your contracts
Event Parameter
Triggers whenever a specific argument in an event matches the set conditions
ERC20 Token Transfer
Triggers whenever an ERC20 transfer event is emitted in one of your contracts
Allowlisted Callers
Triggers whenever an address that is not allowlisted calls one of your contracts
Blocklisted Callers
Triggers whenever an address from this list calls one of your contracts
ETH Balance
Triggers when the ETH balance of an address falls below a certain threshold
Transaction Value
Triggers whenever a transaction value matches set conditions
State Change
Triggers whenever a state variable in one of your contracts changes
View Function
Triggers whenever a view function return value changes
More Coming Soon
Have an idea? Click here and send us what you think can be the next alert type

Web3 Gateway

See Docs

Tenderly Web3 Gateway is a production node that offers reliable, fast, and consistent access to the blockchain.

Use the Tenderly node to:

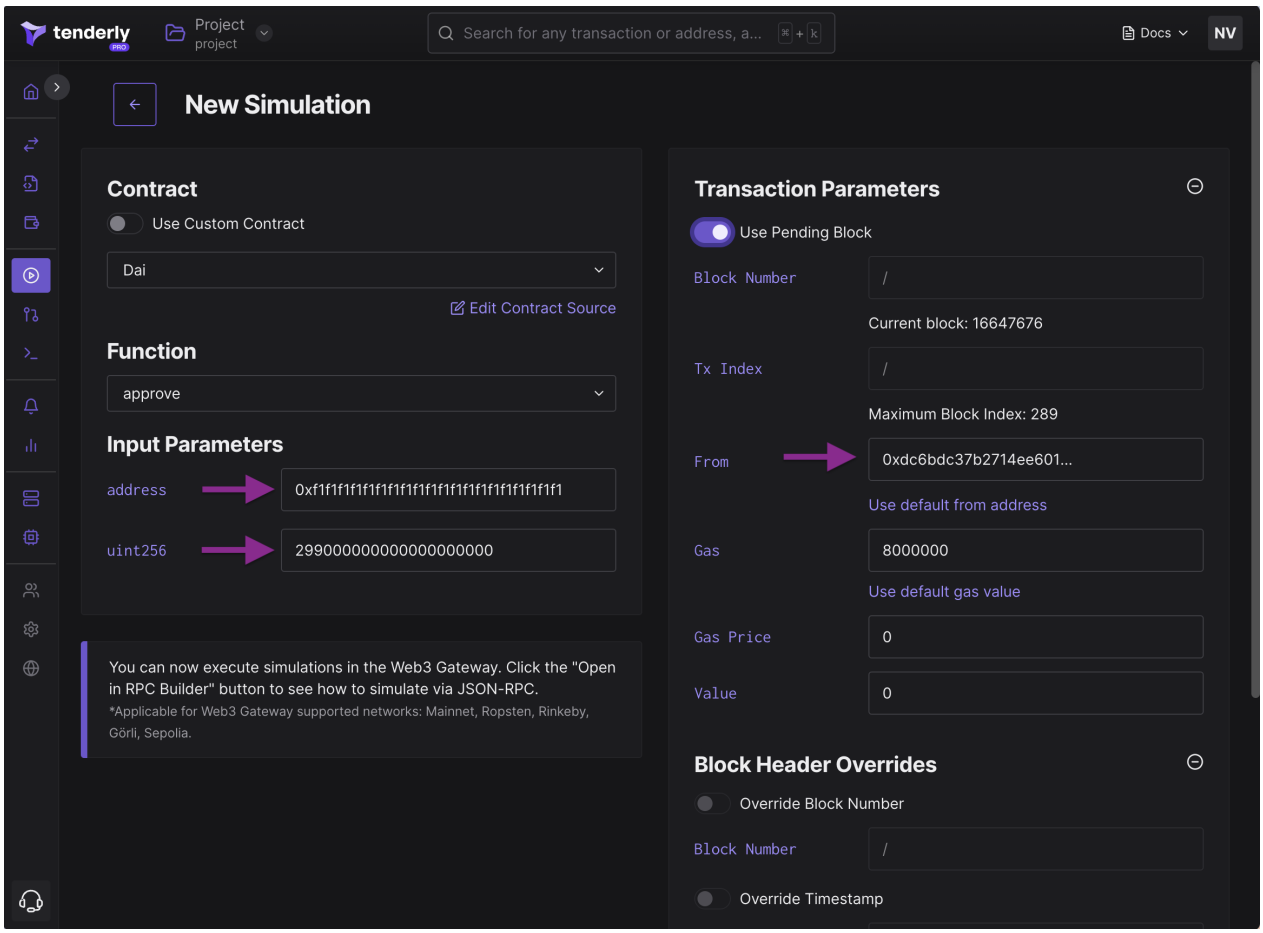
- Read, stream, and analyze blockchain data with 100% consistency.
- Run tx simulations before sending them on-chain using a single RPC URL.

Simulators

Transaction Simulator allows you to see a transaction's execution without sending it to the blockchain. It gives you the results of running a transaction against any point in blockchain history, including the latest block.

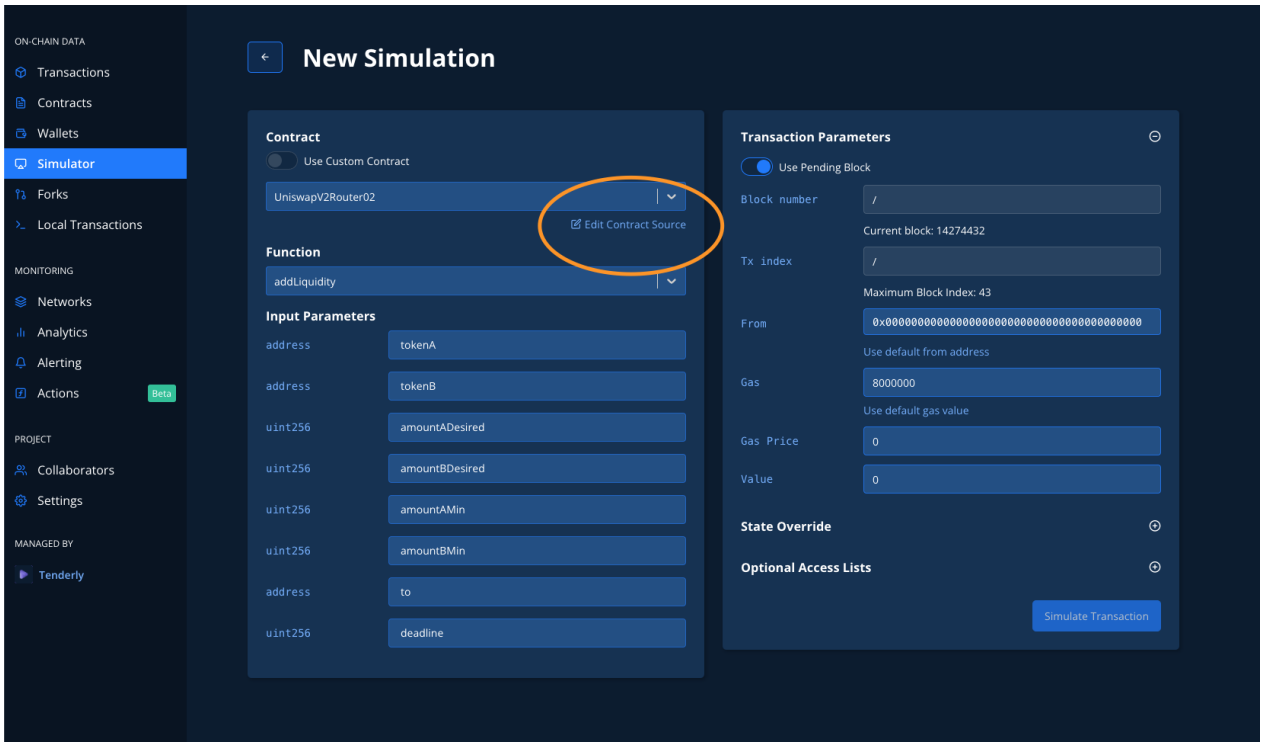
With Transaction Simulator, your transactions are run in a lightweight simulation environment, delivering detailed information about state changes, emitted events (logs), gas usage, and all calls that a simulated transaction made.

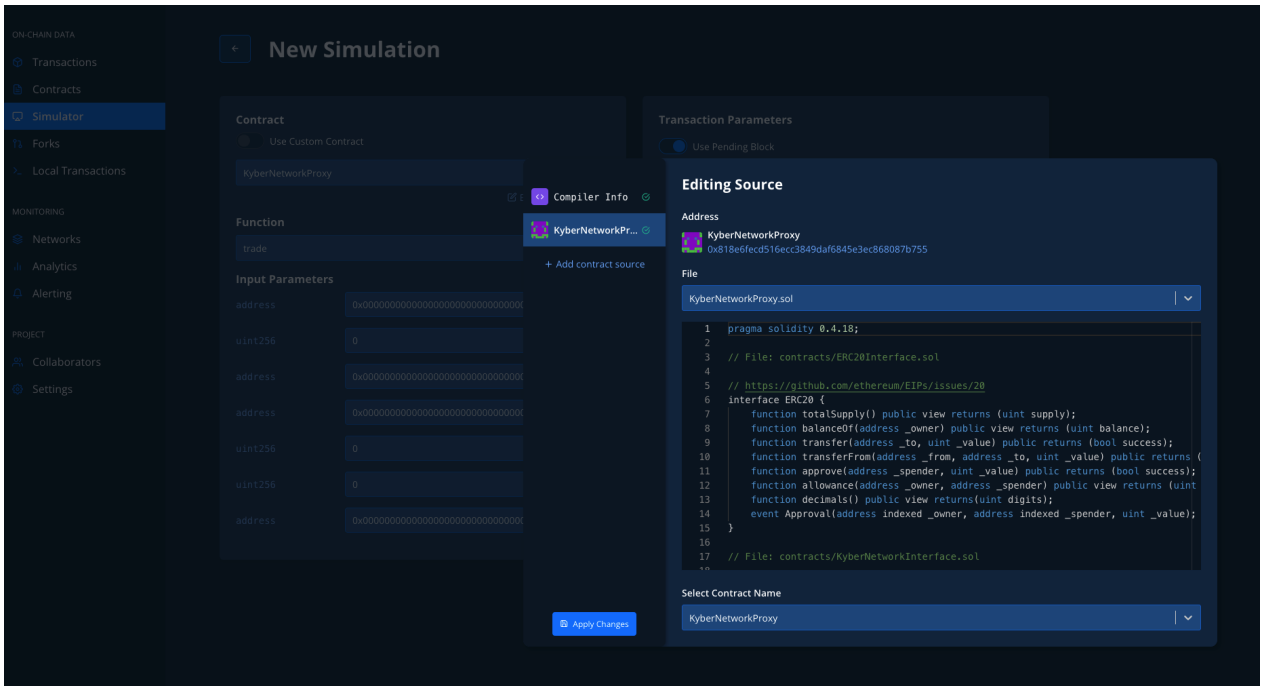
For example you can simulate a DAI approval



Editing contract Source in a simulation

You can edit the contract source on the fly while setting up your simulation or re-simulating, by showing the entire code and giving access to any line or parameter you would want to change.





You can also change the following compiler parameters for the simulation execution:

- Compiler Version
- Optimization Used
- Optimization Count
- EVM Version

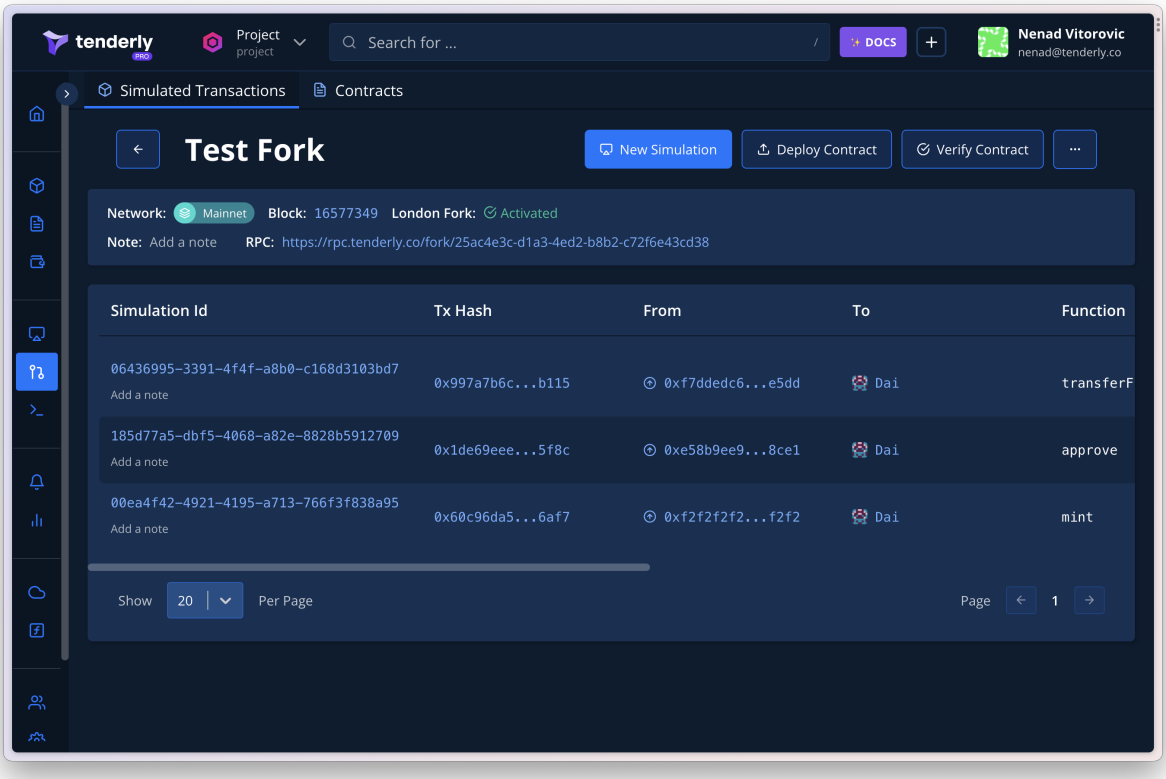
You can add a custom contract source to any contract you have chosen to edit or any address.

Forks

Tenderly Forks are a lightweight blockchain replica where you can run transaction simulations. You can base your Fork on one of Tenderly's supported networks and any of the blocks in a specific network's history. All transactions you simulate through Forks are recorded in an isolated timeline.

Additionally, you can do various custom actions on a Fork:

- Advance/mine a block (`evm_increaseBlocks`)
- Advance time on the Fork (setting the `timestamp`)
- Move the head of the Fork (`evm_snapshot` and `evm_revert`)
- Manage account balances (`tenderly_setBalance` and `tenderly_addBalance`)
- Override smart contract storage (`tenderly_setStorageAt`)



> CI / CD
>
> See Docs
> The docs have examples of using a fork for your CI, and an example github action workflow yaml file.

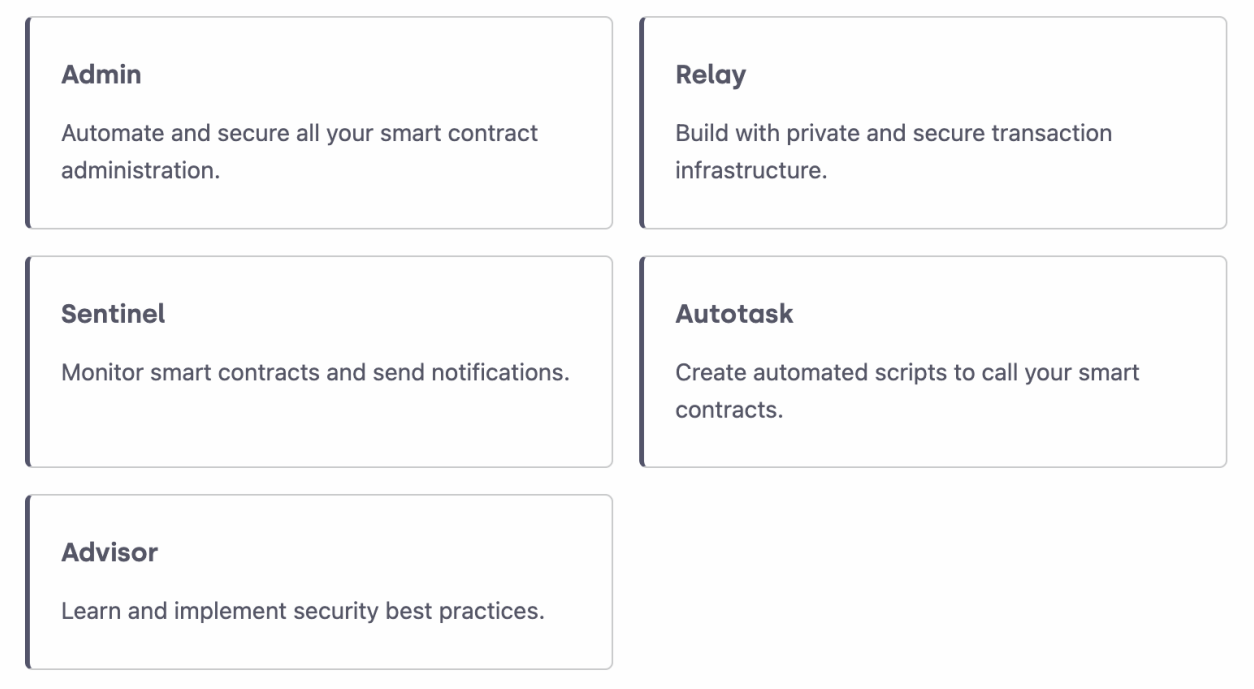
OZ Defender

See Docs

Available Networks

- **Polygon** (Matic) and **Mumbai**.
- **Arbitrum**, **Arbitrum Nova**, **Arbitrum Rinkeby**, and **Arbitrum Goerli**.
- **Optimism**, **Optimism Kovan**, and **Optimism Goerli**.
- **Moonbeam** and **Moonriver**.
- **xDai** and **Sokol**.
- **Binance Smart Chain** and **BSC testnet**.
- **Avalanche C** and **FUJI C-Chain**.
- **Fuse**.
- **Fantom** and **Fantom Testnet**.
- **Celo** and **Alfajores**.
- **Harmony** Shard 0 and **Harmony Testnet Shard 0**.
- **Aurora** and **Aurora Testnet**.
- **Hedera** and **Hedera Testnet**.
- **zkSync 2.0 Goerli**

Components



Admin

The Defender Admin service acts as an interface to **manage your smart contract project** through secure **multisig contracts** or **timelocks**. Defender Admin holds no control at all over your system, which is fully controlled by the keys of the signers.

Relay

The Defender Relay service allows you to send transactions via a regular HTTP API, and takes care of **private key secure storage**, **transaction signing**, **nonce management**, **gas pricing estimation**, and **resubmissions**. This way you don't need to worry about securing your private keys in your backend scripts, or by monitoring your transactions to ensure they get mined.

Sentinels

The Defender Sentinel service offers 3 types of Sentinels, **Contract Sentinels**, **Forta Sentinels** and **Forta Local Mode Sentinels**. Contract Sentinels allow you to **monitor transactions** to a contract by **defining conditions** on events, functions, transaction parameters. Forta Sentinels allow you to **monitor Forta Alerts** by **defining conditions** on Forta Bots, contract addresses, alert IDs and severity. If a Sentinel matches a transaction or a Forta Alert based on your defined conditions it will notify you via email, slack, telegram, discord, Autotasks, and more.

Autotasks

The Defender Autotasks service allows you to **run code snippets** on a **regular basis**, **via webhooks**, or in **response to a transaction**. Thanks to tight integration

to Relay and Sentinels, you can use Autotasks to automate regular actions by easily sending transactions or reacting to events from your contracts.

Logging

Defender generates log trails of every potentially relevant event in the system. This includes manual actions, such as modifying an Autotask or Sentinel, as well as automated actions, such as sending a transaction or firing a notification. Logs can be optionally forwarded to Datadog or Splunk for aggregation.

Advisor

The Defender Advisor service contains a **knowledge base of security best practices** curated by the OpenZeppelin team. The best practices cover development, testing, monitoring and operations. Defender Advisor can be used as a checklist to prioritize efforts in implementing project security.