CONSENSYS Diligence

Diligence Security Tooling Guide: What To Use And When

A guide of the top blockchain security tools for building secure smart contracts

By ConsenSys Diligence

You see, but you do not observe.

> SHERLOCK HOLMES

INTRODUCTION

> For hackers by hackers

Building secure smart contracts can be difficult. There are guidelines for building safe applications in Web3 that include adhering to best practices for writing smart contracts and commissioning a smart contract audit before launching your product. Another recommended measure is to integrate tools for improving smart contract security into your project's development lifecycle.

The Diligence Security Tooling Guide was created to help builders navigate the emergent blockchain security space by outlining the top tools available to them at every stage of development. Not only do these tools help with detecting and fixing programming errors, but they also assist developers in building scalable, flexible, and robust smart contracts.

This guide explores the different smart contract security tools to use at every stage of development. Most of the software covered in the guide is open-source and freely available to use.

INDEX

> Contents

5 **Who We Are**

6 **ConsenSys Diligence**

7 **Development Phase**

8 **Unit testing**

9 Truffle

10 Foundry

11 Brownie

13 Vertigo

14 Solidity-coverage

15 **Property-based testing**

16 Diligence Scribble

18 Diligence Fuzzing

20 Silther

21 Mythril

23 Manticore

24 Echidna

25 MythX

27 **Preparing your smart contract for audits**

28 NatSpec

29 Ethlint

30 OpenZeppelin Contracts

31 **Post-development Phase**

32 **Monitoring and performing automated reactions**

33 OpenZeppelin Defender Sentinels

34 Tenderly Alerts

35 **Responding to security incidents**

36 Tenderly War Rooms

38 **Secure access control and administration**

39 OpenZeppelin Defender Admin

40 Safe

41 **Coordinating bug bounties and vulnerability discourses**

42 Immunefi

43 HackerOne

45 **Contact Us**

Who we are

WHO WE ARE

> ConsenSys Diligence

"I promised myself I would never again put money into something I hadn't audited myself."

These are the words of Diligence Co-Founder, Gonçalo Sa. You've likely seen him around the ecosystem or on Twitter. His relentless curiosity and passion for smart contract auditing led him to ConsenSys in April 2017, where he's been building ConsenSys Diligence ever since.

ConsenSys Diligence is a blockchain security company. Our industry-leading suite of blockchain security analysis tools, combined with hands-on audits from our veteran smart contract auditors, ensures that your application is ready for launch and built to protect users.

As you delve into this guide, you will learn more about two Diligence tools we're constantly evolving: Fuzzing and Scribble.

Development Phase

DEVELOPMENT PHASE

> Unit testing for smart contracts

Mistakes in smart contract design, however trivial, can often lead to considerable losses. To avoid costly errors, it is important that you perform tests to ensure your smart contract functions as expected.

Unit testing is one approach to assessing the functionality of smart contracts. A unit test suite will usually have multiple focused tests that evaluate specific components of your application for correctness. Good unit tests should be simple, easy to execute, and provide useful insights into errors encountered at runtime.

TRUFFLE

Truffle is a framework for building Ethereum smart contracts with ease, speed, and efficiency. In addition to utilities for writing, compiling, and deploying contracts, Truffle users can access resources for unit testing. You can either test contracts in JavaScript or Solidity depending on your preferences—although using both approaches is also ideal.

JavaScript-based tests mostly cover external interactions with a contract, such as checking if users can place and withdraw bids in an auction contract. Conversely, Solidity tests can help assess a contract's behavior in the blockchain environment (e.g., interactions with other contracts) and highlight details about its internal operations.

```
Contract: SimpleContract
  √ should return the name
  √ should return change the name (67ms)
  √ should execute only by the owner (45ms)
  √ should fail (39ms)
  √ should check the type of the event
  √ should emit with correct paremeters
  Events emitted in tx 0x8c048e45e8968270aa7db1ed29c9616d8f796928ac033ba825d7771f43d85851:
  --------------------------------------------------------------------------------
  NameEvent(evPram: hello event)
  --------------------------------------------------------------------------------

  √ should print the event paremeters


7 passing (313ms)
```

Truffle's unit testing framework checks multiple contract functions for correctness (source)

For testing in Javascript, Truffle uses Mocha for asynchronous test execution and Chai for creating assertions. It also integrates with Ganache and allows for running unit tests on a local version of the Ethereum blockchain. This gives you the flexibility of testing your application in a production-like environment without needing to deploy to Mainnet.

Foundry

Written in Rust, Foundry is a versatile framework for developing smart contracts targeted at the Ethereum Virtual Machine (EVM). Foundry also includes Forge, a framework for testing contracts, in its package for users.

While using Truffle might require some knowledge of Javascript, Forge can be used to write unit tests directly in the Solidity language. This might be ideal if your grasp of Javascript is low, or you want to avoid switching contexts when writing and testing code.

Using Forge's unit test feature, you can confirm that contract logic (e.g., access control and contract ownership) satisfy safety requirements. A summary of all failing and passing tests is available by default—but you can adjust the settings to extract detailed debugging information. This includes details about logs emitted during execution (to check assertion errors) and stack traces for failing tests.

Forge also offers advanced functionalities (cheatcodes) that can improve your testing workflow. One of the cheatcodes, for instance, lets you generate an arbitrary amount of tokens for any address—handy for running running tests on forked blockchains.

Brownie

Brownie is a Python-based framework for developing and testing EVM smart contracts. To provide unit test functionality, Brownie integrates with pytest, a Python framework that supports both small tests and complex functional testing for applications.

Like other unit testing tools, Brownie provides resources for unit-testing different parts of your smart contract's logic separately. It also comes with stack trace analysis for measuring code coverage. This feature works by analyzing transaction traces to detect the branches and statements executed for each test suite.

Brownie is especially useful for running “happy path” tests that check if contract functions return the expected results based on user inputs. For every reverted transaction, Brownie displays a detailed execution trace (via a GUI) to show where the error statement occurred. This makes it easier to debug your unit tests and fix issues in your code.

```
Brownie environment is ready.
>>> nft = NFToken.deploy( "", "", 10000000, {'from': a[0]})
Transaction sent: 0x9c68294565654e9f4cd4daec15c6c3c930e77205e4b8b5ddfdecdf665af239cf
  Gas price: 20.0 gwei   Gas limit: 2099342
  NFToken.constructor confirmed - Block: 1   Gas used: 2099342 (100.00%)
  NFToken deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87

>>> tx = nft.transfer(a[0], 1000, {'from': a[2]})
Transaction sent: 0x29e2d639a68133e53a762fc2eaf003e7081e7e3bd5a7af87ee258de97bef2bf3
  Gas price: 20.0 gwei   Gas limit: 6721975
  NFToken.transfer confirmed (dev: underflow) - Block: 2   Gas used: 23683 (0.35%)

>>> tx.traceback()
Traceback for '0x29e2d639a68133e53a762fc2eaf003e7081e7e3bd5a7af87ee258de97bef2bf3':
Trace step 74, program counter 1814:
  File "contracts/NFToken.sol", line 164, in NFToken.transfer:
    _transfer(msg.sender, _to, _value);
Trace step 125, program counter 2616:
  File "contracts/NFToken.sol", line 237, in NFToken._transfer:
    balances[_from].balance = balances[_from].balance.sub(_value);
Trace step 148, program counter 3204:
  File "contracts/SafeMath.sol", line 66, in SafeMath64.sub:
    require(_b <= _a); // dev: underflow
>>>
```

Traces in Brownie show where transaction reverts occur (source)

Honorable mentions for other unit testing tools: Hardhat, DappTools, Waffle Tests, Remix Tests, ApeWorx

Vertigo

Vertigo is a mutation testing tool for assessing the quality of your unit tests. In mutation testing, variations of a contract's source code (mutants) with changes to certain elements—for example, statements and variables—are created. These changes introduce subtle errors and bugs, and the objective is to check whether a test suite fails for the mutated code.

Mutation testing is important because the effectiveness of unit testing depends largely on the quality of the testing approach. Suppose all unit tests involving a contract's functions succeed, but the testing tool itself (or the test suite) is weak. This would result in a high incidence of "false negatives" (i.e., a test case passes while the bug the test was designed to catch is present in the contract's code).

Thus, "testing your tests" with a mutation testing tool like Vertigo is necessary for developing safer smart contracts. Not only does this increase confidence in the reliability of test results, but it also reduces the risk of missing out on bugs during development.

To generate mutants, Vertigo relies on 'mutation operators' that determine what parts of the source code must be modified. It also supplies a 'mutation score', which can be used to assess the quality of a test suite. The mutation score is calculated by dividing the mutants rejected ("killed") during tests by the total number of mutants (including "surviving" mutants that weren't rejected).

$$mutation\ score = \frac{dead\ mutants}{total\ surviving\ \&\ dead\ mutants} \times 100\%$$

Honorable mentions for other mutation testing tools: Universal Mutator

solidity-coverage

Running multiple tests is great, but not if those tests exercise only a small part of your codebase. A good rule of thumb is to ensure that tests cover all—or, at least, a majority—of a contract's source code. Since untested code is prone to executing in unexpected ways, optimal coverage can provide higher assurances of correctness.

Solidity-coverage is a code coverage tool that measures the percentage of lines, statements, and branches covered by tests in a test suite. It works by adding coverage annotations to Solidity code as executable statements, and tracking their execution.

As each annotation is a call to emit an event to signal that the subsequent line of code has been run, solidity-coverage can reliably determine which parts of the code were covered in tests or the frequency of coverage. You can see this feature in action in this snippet from solidity-coverage's HTML report below:

```
22          modifier onlyColonyOwners {
23    17×     I if (!this.userIsInRole(msg.sender, 0)) { throw; }
24    17×       _
25          }
```

Numbers in the margin indicate those lines are run 17 times during tests (source)

DEVELOPMENT PHASE

> Property-based testing for smart contracts

Properties describe the expected behavior of a smart contract and state logical assertions about its execution. A property (also described as an invariant) is a condition expected to hold true at all times. A “property violation” occurs when a smart contract’s execution fails to satisfy an invariant.

Property-based testing tools take a smart contract’s code and a collection of user-defined properties as inputs, and check if execution violates them at any point in time. Compared to unit testing, property testing can increase assurances that a smart contract’s business logic is sound.

DILIGENCE Scribble

Part of preparing a contract for property-based testing, including fuzzing, is writing properties that express assertions about the expected behavior of different contract functionalities. This is what allows the fuzzer to flag execution traces that violate properties.

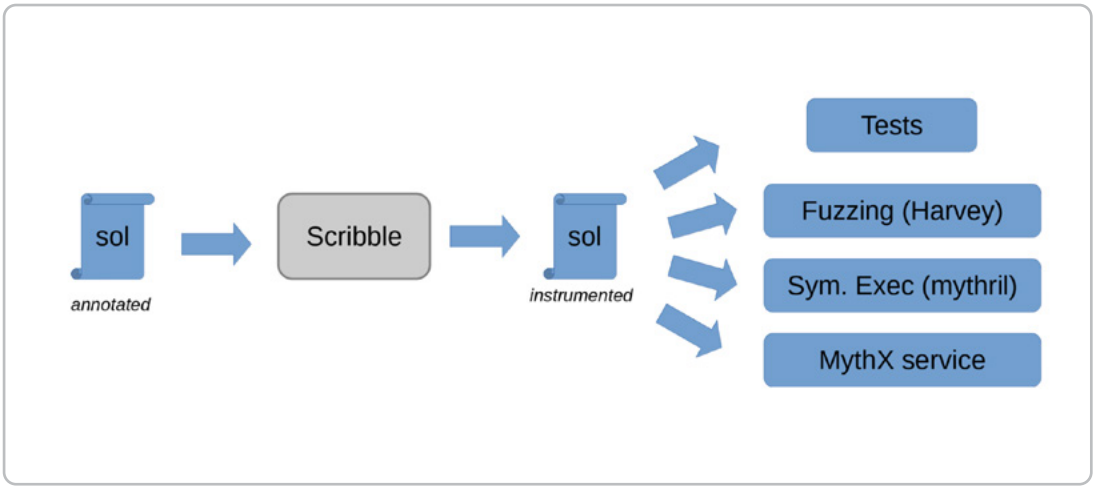
As a specification language, Scribble enables transforming informal assumptions about a contract's execution into formal statements that can be tested rigorously. This encourages precise reasoning about the correctness of smart contracts and reduces the possibility of your contract displaying unexpected behavior after deployment.

Say you make the following assertion about an ERC-20 token contract: "If a transfer succeeds then the sender will have `_value` subtracted from its balance." Below is how Scribble makes it easier to check if the assertion holds under different conditions:

01 You annotate the Solidity contract in question with the assertion on state variables using Scribble syntax.

02 Scribble takes the annotated source file as input and generates an instrumented version with the assertions added to the contract file added as executable code.

03 A fuzzer (e.g., Diligence Fuzzing) or symbolic execution tool executes the instrumented contract while checking for execution traces that violate the supplied assertion.



Different tools can use Scribble specifications for property testing (source)

Scribble is also useful for defining assertions related to preconditions (e.g., input values and pre-execution contract states) and postconditions (e.g., transaction results and post-execution contract states) for functions, contract invariants, and more. Moreover, the syntax is Solidity-based and fairly easy to grasp—beneficial for developers new to writing formal specifications.

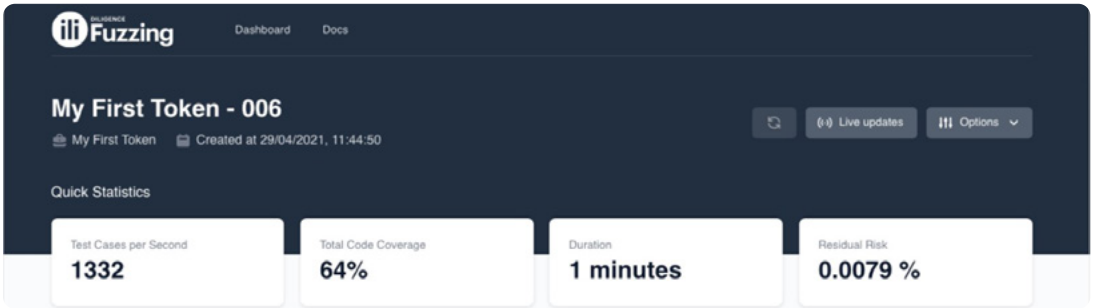
DILIGENCE Fuzzing

Diligence Fuzzing, based on the HARVEY, is a cutting-edge fuzzing tool that executes smart contracts with mutated inputs to detect bugs and vulnerabilities.

Harvey is an advanced fuzzer with tons of techniques incorporated like efficient fuzzing of multiple transactions and input prediction to name a few. Harvey uses an advanced input generation technique that improves the efficiency and effectiveness of fuzzing campaigns.

Consider a blackbox fuzzer that randomly generates inputs: in most cases, the tested program can detect if such inputs are syntactically invalid and reject them. This technique also rarely guarantees better code coverage, as multiple inputs may explore the same paths during execution.

Greybox fuzzing (as implemented in Harvey) relies on analysis of code coverage to guide the generation of new test cases. The fuzzer takes a "seed value" (a piece of valid data) as input and introduces small changes to produce mutated versions. While these mutated values (mostly) keep inputs valid, they can exercise interesting behaviors in contract code—increasing the possibility of detecting unexpected execution behavior.



Reports from from an example Diligence Fuzzing campaign

The contract's execution is also instrumented to allow the fuzzer to collect coverage information such as identifying what inputs explore new paths. By mutating inputs with high code coverage, the fuzzer efficiently generates new test cases that explore deeper paths in code. When combined with Scribble, Diligence Fuzzing can be a powerful tool for verifying your smart contract's security properties.

Fuzzing smart contracts

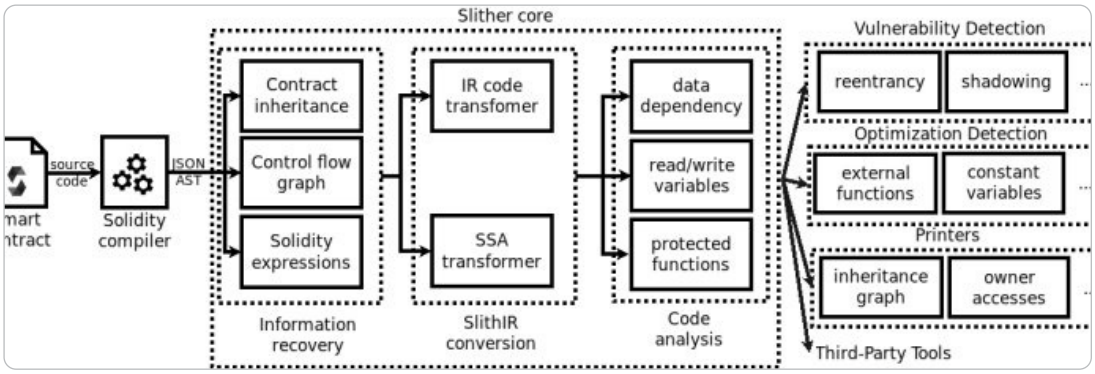
Fuzzing is a method of testing an application by feeding it random or unexpected data inputs and observing its operation at runtime. One use case for fuzzing is gauging the ability of a program to validate user inputs.

Invalid inputs to a program can break its functionality and trigger unintended behavior if handled improperly. By running a contract with multiple variations of input values, fuzzers can identify inputs that trigger property violations in code.

Slither

Slither is a Python-based static analysis framework for smart contracts written in Solidity. By analyzing low-level representations of a contract, particularly the abstract syntax tree (AST) generated by the Solidity compiler, Slither can automatically detect code paths that result in error conditions.

The detection framework used in Slither covers well-known smart contract vulnerabilities, including reentrancy, state variable shadowing, and uninitialized storage variables. But you can also write custom analyses using the Detector API to suit your application's needs.



Overview of Slither's architecture (source)

Slither also comes out of the box with printers that visualize important contract information (e.g., inheritance hierarchy, control flow, and data dependencies) in a human-readable format. This feature aids code comprehension for developers and makes reasoning about the correctness of the overall system easier for others.

Mythril

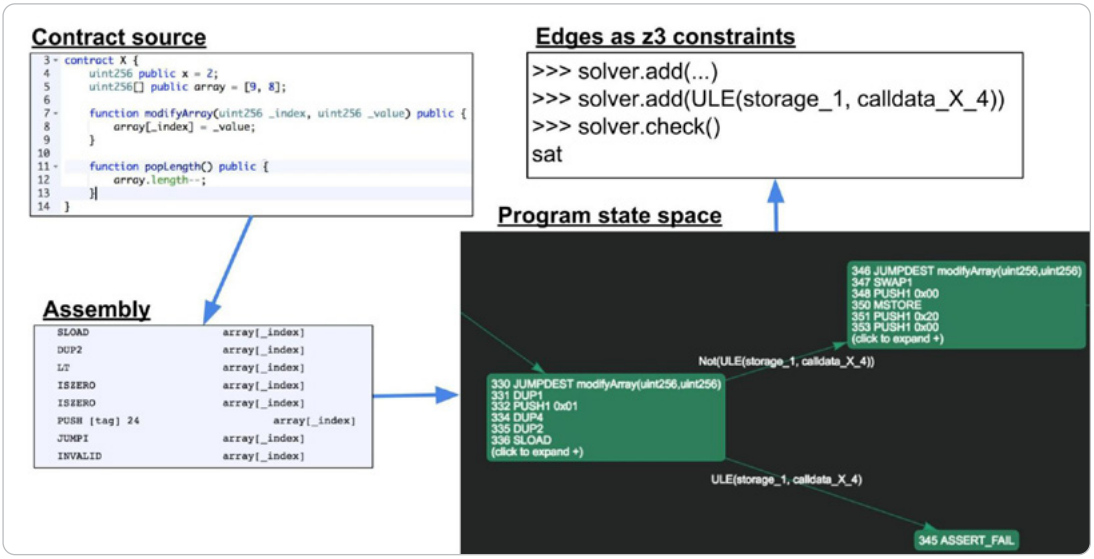
Mythril uses symbolic execution, which represents each execution path as a mathematical formula over symbolic input values that trigger a program to follow that path. While unit testing uses concrete input values ("withdraw five ether"), symbolic execution uses symbolic values (e.g., "withdraw n ether") that can represent multiple classes of concrete values.

Mythril's symbolic execution engine uses LASER (an implementation of the EVM in Python) to simulate the execution of smart contracts. This allows it to run a contract's bytecode and produce a control flow graph that captures all possible execution states.

Call graph for functions produced during a Mythril test run (source)

Each node on the CFG represents a sequence of instructions that execute and alter the contract's state. Meanwhile, edges are constraints on conditions that trigger transitions between various states.

If a problematic state (one that results in an assertion violation) is reachable, Mythril uses the underlying Z3 Solver to check the satisfiability of path constraints. Where such path constraints are satisfiable (i.e., they can be solved), the Z3 solver can generate values for parameters that satisfy them.



Workflow of Mythril's symbolic execution engine (source)

The combination of LASER and Z3 ensures Mythril can detect error conditions in your smart contract's code and identify concrete inputs to functions that cause the problem. It also means you can perform unit tests (using those concrete values) to check if post-testing fixes have solved the problem.

For a more in-depth introduction to Mythril, read Bernhard Mueller's article on formally verifying contract properties using symbolic execution.

Manticore

Manticore is another useful symbolic execution tool for verifying properties and performing general analysis of Ethereum smart contracts. Manticore's verification targets Ethereum Virtual Machine (EVM) bytecode, allowing for a more precise evaluation of a contract's low-level operations.

The Manticore verifier comprises an emulated blockchain environment with accounts that send symbolic transactions invoking a smart contract's functions. By exploring all reachable states, Manticore can discover contract operations that violate properties. Like Mythril, Manticore uses an SMT solver to calculate inputs that drive the execution of a smart contract to specific states.

Below is a simple example of a test run in Manticore to check if arithmetic operations from a function can overflow:

```
pragma solidity ^0.4.15;
contract Overflow {
    uint private sellerBalance=0;

    function add(uint value) returns (bool, uint){
        sellerBalance += value; // complicated math with possible overflow

        // possible auditor assert
        assert(sellerBalance >= value);
    }
}
```

Call graph for functions produced during a Mythril test run (source)

Manticore is particularly ideal for developers who desire fine-grained control of property-based testing. Besides configuring accounts used for sending symbolic transactions to the target contract, you can also control the duration of property tests. The latter is primarily achieved by defining conditions for terminating state space exploration, such as a maximum number of transactions or a minimum level of code coverage.

Echidna

Echidna is a property-based fuzzing tool for Ethereum smart contracts. You can write properties for Echidna tests using Slither's property generator, Scribble, or Echidna's built-in property writing feature. Based on these invariants, Echidna will attempt to generate user inputs that violate properties in your code.

In particular, Echidna uses information from a contract's ABI (Application Binary Interface) when generating transactions for property tests. This makes sure that only inputs that can plausibly come from a real-world user are used for fuzzing functions. Echidna can also instrument Solidity code to collect and visualize code coverage.

Integrating Echidna into your development workflow is quite straightforward. For example, you can easily use the crytic-compile plugin to test smart contracts compiled with build frameworks such as Hardhat and Truffle with Echidna.

```
                                    Echidna 2.0.0
Tests found: 11
Seed: -7920091280766187841
Unique instructions: 1355
Unique codehashes: 1
Corpus size: 13
                                        Tests
assertion in checkAnInvariant(): FAILED! with ErrorUnrecognizedOpcode

Call sequence:
1.lock(1)
2.voteSlate("\161\231\t\DLE\225+\SUB
  \SUBe>\GSr2s\158\250\147pDL\223\170\215\"1\241\155\152Iv\161")
3.etch(0x30000)
4.checkAnInvariant()

AssertionFailed(..): fuzzing (28325/50000)

assertion in deposits(address): fuzzing (28325/50000)

assertion in voteSlate(bytes32): fuzzing (28325/50000)

assertion in lock(uint256): fuzzing (28325/50000)

assertion in free(uint256): fuzzing (28325/50000)

assertion in votes(address): fuzzing (28325/50000)

assertion in slates(bytes32): fuzzing (28325/50000)
```

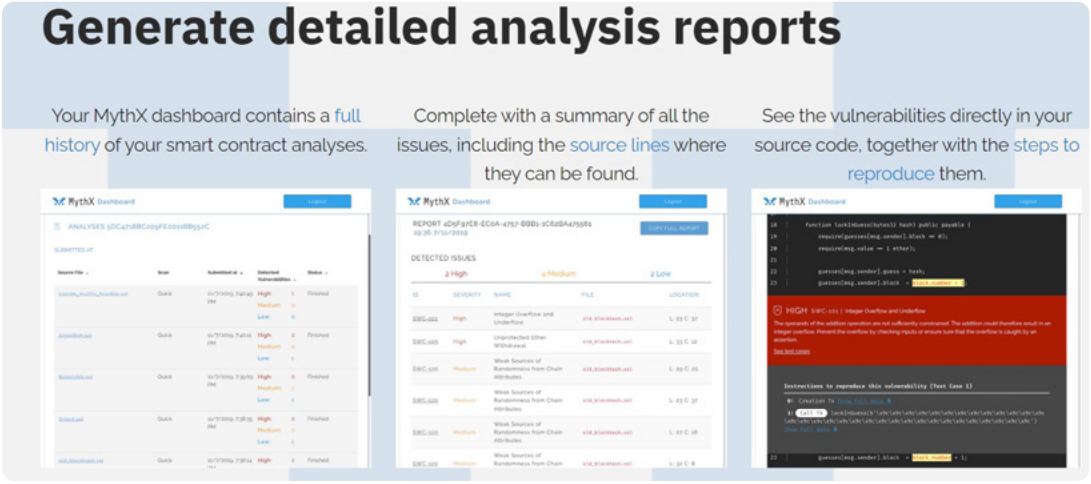
Echidna's analysis engine produces reports of assertion violations for smart contracts

MythX

MythX is a cloud-based testing tool that combines multiple property-based testing techniques, including fuzzing, symbolic execution, and static analysis. The MythX analysis engine packs three tools in one package: Mythril (symbolic execution), Harvey (greybox fuzzing), and Maru (static code analysis).

Using MythX requires submitting the Solidity source file of your smart contract to the API. The MythX client then passes artifacts output by the Solidity compiler (e.g., contract bytecode and source mappings) and the source code to the various tools for analysis. Results from tests are filtered, combined, and presented in a single, easy-to-read report containing discovered vulnerabilities and test-cases for reproducing them.

Adopting a microservices-based architecture helps MythX deliver better results than a standalone testing tool. Beyond detecting well-known vulnerabilities (particularly ones that appear in the SWC Registry), MythX can also verify formal properties of a smart contract and check if assertions about contract behavior hold true under execution.



MythX provides a powerful suite of testing tools that integrate seamlessly into development workflows.

When used individually, techniques like lightweight static analysis and input fuzzing may generate false positives for certain issue classes (for instance, arithmetic overflows). MythX solves this problem through post-processing: results from each testing tool are evaluated to identify duplicates and false positives. This increases the accuracy of results and minimizes inefficiency for developers analyzing contracts.

You can use MythX by running the command-line interface (CLI) tool or using the MythX security plugin (available in popular tools like Remix). While MythX is a paid software-as-a-service tool, pricing plans are flexible enough to choose a package that fits your needs.

Honorable mentions for other property testing tools: Forge (via proptest), DappTools (hevm), and Brownie (via Hypothesis), Etheno.

DEVELOPMENT PHASE

> Preparing your smart contracts for audits

Getting an audit prior to deploying on mainnet can improve your smart contract's security guarantees—but only if approached properly. The tools in this section prepare your codebase for independent review and improve the productivity and effectiveness of auditors working on your project.

NatSpec

As a developer, it is important to keep your code clean, readable, and easy to understand—**clean code is good code**. You can write cleaner code by properly documenting functions using comments with NatSpec.

NatSpec (Ethereum Natural Language Specification Format) is used in Solidity code to provide rich documentation to developers, end-users, or both. NatSpec comments are human-readable statements that explain what a smart contract—and different elements defined therein—do. Comments can be used to explain contracts, libraries, interfaces, functions, variables, return values, and more.

Documenting your code with NatSpec equips auditors with the knowledge needed to properly review your smart contracts. Instead of wasting time trying to figure out the intent behind a function or event, auditors can focus on reviewing code for vulnerabilities and bugs.

ETHLINT

Analyse Solidity for Style and Security

Linting is a static analysis technique that checks the quality of a program's source code for stylistic errors, violation of programming conventions, and use of unsafe constructs. Admittedly, linting is designed to catch minor issues in your source code. But even "minor" issues can result in bugs that affect users' security or loopholes that can be exploited.

Ethlint is a linting tool designed for analyzing smart contracts and enforcing security rules in Solidity code. The Ethlint analysis engine takes a Solidity file and configuration as input and compares it against a set of rule implementations. While these rules mostly cover styling conventions (mostly taken from the Solidity style guide), they also cover more critical security issues.

For example, the linter will output warnings for use of tx.origin for authentication or call.value for sending Ether (ETH) prone to re-entrancy vulnerabilities. A comprehensive list of conventions enforced by the Ethlint linter can be found here.

Running the automatic code formatting command on Ethlint's CLI automatically applies fixes for lint where possible. In other cases, you'll need to manually review the warnings, fix those issues, and run the code through the linter until no warnings are emitted. Overall, linting can remedy persistent issues in Solidity code that auditors would otherwise waste valuable time on solving.

OpenZeppelin | Contracts

OpenZeppelin Contracts is a suite of secure code libraries that supply utilities to simplify your workflow as an Ethereum developer. This includes contract logic for implementing access control, pause functions, upgrades, and more. As OpenZeppelin Contracts benefit from extensive testing and audits, using them can reduce the possibility of introducing errors into your code.

More importantly, using popular and vetted libraries reduces cognitive overhead for auditors reviewing your code. Auditors are typically familiar with the structure and security properties of such contracts, making it easier to audit contracts that use them.

Creating custom code—for example, writing your own ERC20 token contract or contract ownership logic—increases the scope of an audit. As the logic used may be unfamiliar to auditors, it takes more effort to reason about the correctness of system components.

Honorable mentions for other contract libraries: Solmate Contracts, useful for optimizing gas usage

Post-development Phase

POST-DEVELOPMENT PHASE

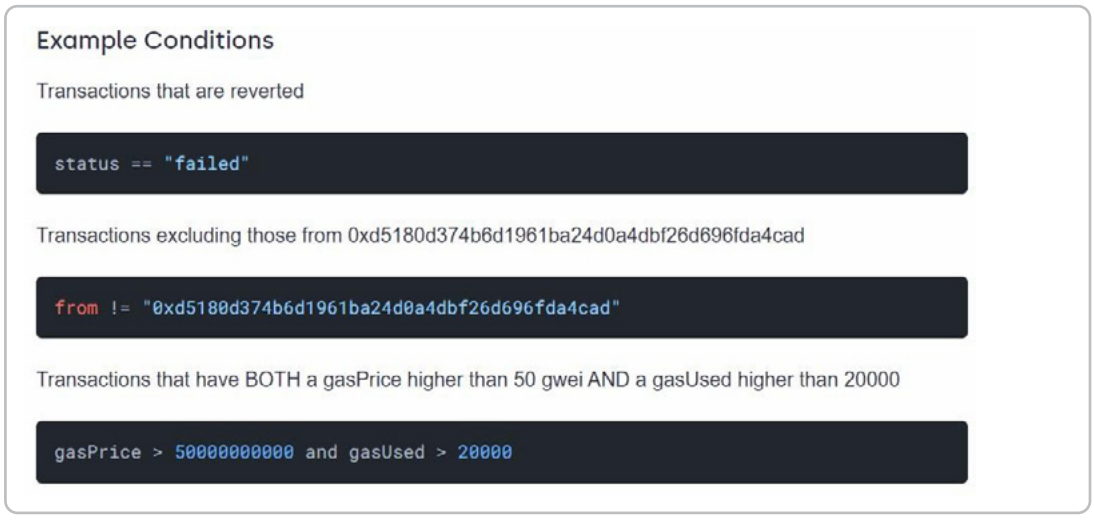
> Monitoring contracts and performing automated reactions

On a public blockchain, such as Ethereum, you have little control over who does what with your smart contract. Having a monitoring and alerting system in place ensures you have updated information about on-chain smart contracts—enough to respond proactively to avert problematic situations.

OpenZeppelin | Defender Sentinels

Defender Sentinels from OpenZeppelin provide real-time security monitoring of smart contracts and alerts users based on predefined conditions. Compared to manually logging events, Sentinels are a more flexible and convenient mechanism for tracking interactions with your smart contracts.

Using Sentinels ensures you can detect suspicious transactions targeting a contract and respond immediately. Alerts can be created with different parameters depending on what works for your security needs. Options include getting alerts if withdrawals cross a specific threshold, someone performs critical action like calling transferOwnership, or a blacklisted address attempts to interact with your contract.



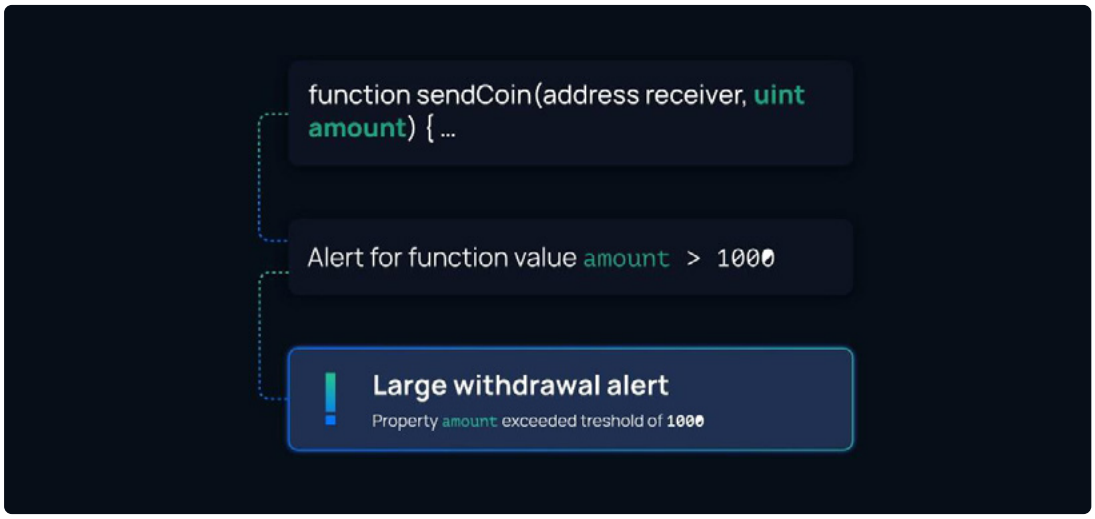
Sentinel trackers can be programmed with custom parameters

If you have a Defender Admin project, information from Sentinel trackers can inform security decisions, such as pausing a contract or revoking access to a privileged account. Security intervention can also be automated by using a Sentinel to trigger a Defender Autotask. You can also choose your preferred means for receiving notifications—options include Discord, Slack, Telegram, and email.

Tenderly Alerts

Tenderly's Real-Time Alerting simplifies the monitoring of on-chain contracts and wallets. It supplies a collection of custom triggers to choose from, including conditions on functions, events, and transactions.

As an example, you can choose to receive an alert if an EOA's transactions targeting your contract have higher-than-normal gas values or cross a limit within a time window. Other options include getting alerts on transactions from blacklisted/whitelisted addresses, state-change operations, token transfers, and transactions with abnormal values.



Detecting irregular contract transactions with Tenderly Alerts

Like Defender Sentinels, Tenderly Alerts can be configured to deliver notifications via multiple communication channels, including Slack, PagerDuty, and Telegram. You're advised to choose a frequency for getting notifications based on your preferences. Also, Tenderly Alerts allows users to filter alerts to extract important information, such as events emitted, current contract balances, and so on.

Honorable mentions for other monitoring tools: CertiK Skynet, Dune Analytics.

POST-DEVELOPMENT PHASE

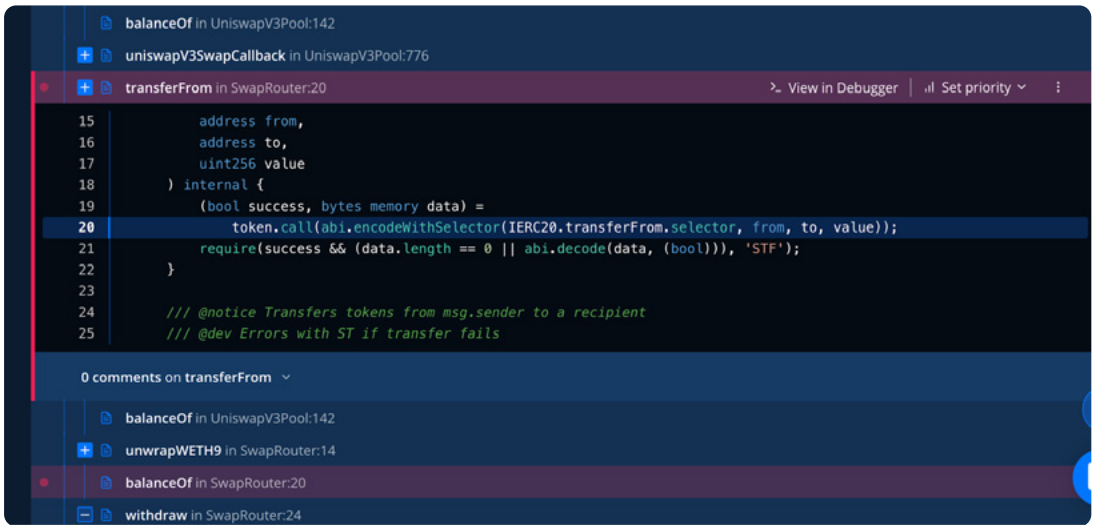
> Responding to security incidents

A useful security recommendation for smart contract developers is to prepare for failure. While extensive testing and auditing during the development phase can reduce errors, it cannot rule out the possibility of a problem appearing later. This makes it necessary to have tools that can help you handle critical failures in smart contracts gracefully.

Tenderly War Rooms

When confronted with an exploit or bug, it is ideal to have a defined and structured process for responding effectively and minimizing damage. Tenderly War Rooms is a highly recommended resource for coordinating and streamlining your team's response to security issues that might surface at any time.

Among other things, the War Rooms aid kit provides utilities for debugging transaction information and analyzing execution traces. This is useful since identifying what error(s) an attacker exploited is a critical step when dealing with an exploit. Users can access the Tenderly Debugger (including the advanced Evaluate Expression tool) to replay execution traces and perform detailed analysis of transactions.



War Rooms offers a rich UI for collectively debugging and inspecting transactions (source)

As you'll likely have multiple participants in a war-room situation, clear and effective communication is important. Which is why the annotation and prioritization feature in War Rooms is useful.

Using this feature, you can leave comments on code snippets from debugged transactions to review or review comments from other team members. You can also set priority flags (Low, Medium, High) on different transaction traces, so your team has direction when approaching the debugging process.

Honorable mentions for other transaction debuggers: Foundry Transaction Replay Trace/Debugger

POST-DEVELOPMENT PHASE

> Secure access control and smart contract administration

Managing smart contracts can be tricky. You want to avoid centralized control and its attendant risks, while retaining the ability to implement emergency measures when needed. The tools in this section help you securely manage your smart contracts on-chain without introducing new risks or burdening users with trust assumptions.

Defender

OpenZeppelin | Defender Admin

OpenZeppelin's Defender Admin is a versatile tool designed to ensure simple and secure administration of smart contracts. Using Defender Admin streamlines the process of taking sensitive administrative actions, such as upgrading a proxy contract to a new implementation or modifying an 'allowed list' of accounts with privileged access.

Defender Admin enables support for multisignature contract accounts that control the execution of actions targeting a specific contract. A multisig account requires approval from a minimum number of address owners—for example, 3-of-5—to execute transactions. Using a multisig to manage your contract reduces risks associated with unilateral control by a single owner.

Decentralized and secure management of access control with Defender Admin (source)

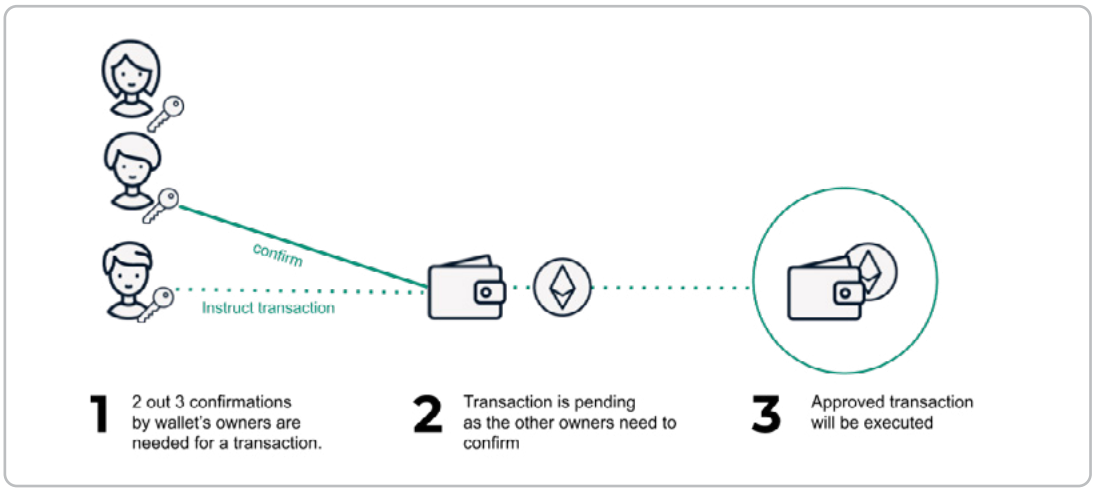
The Defender Admin interface also uses timelocks which delay the implementation of certain contract actions until the predefined window elapses. A timelock acts as a speed bump and gives both users and developers time to respond to changes in a protocol. Delaying critical actions (e.g., minting new tokens) is recommended—especially in cases where attackers can compromise a privileged account and execute those actions.

Safe

As a multisignature wallet, Safe (formerly known as Gnosis Safe), is mostly used for treasury management by protocols. But it can also be ideal for managing smart contracts on-chain, especially if you prefer the Ownable pattern—where a single EOA "owns" the smart contract and controls admin functions—to a role-based access control scheme.

While certainly useful, distributing administrative powers to different privileged accounts (as implemented in role-based permissions) can have drawbacks. Owners of privileged accounts may be compromised (or even go rogue), and having too many roles in a system could complicate coordination and administration.

A better alternative is setting a Safe multisig address as the Owner of your contract. Although the contract is still managed by one account, transactions that trigger contract functions cannot execute on-chain unless the threshold is reached. This gives you the benefit of managing your contract from one same place while reducing single points of failure.



Safe's n-out-of-m access control scheme reduces centralized control of smart contracts (source)

POST-DEVELOPMENT PHASE

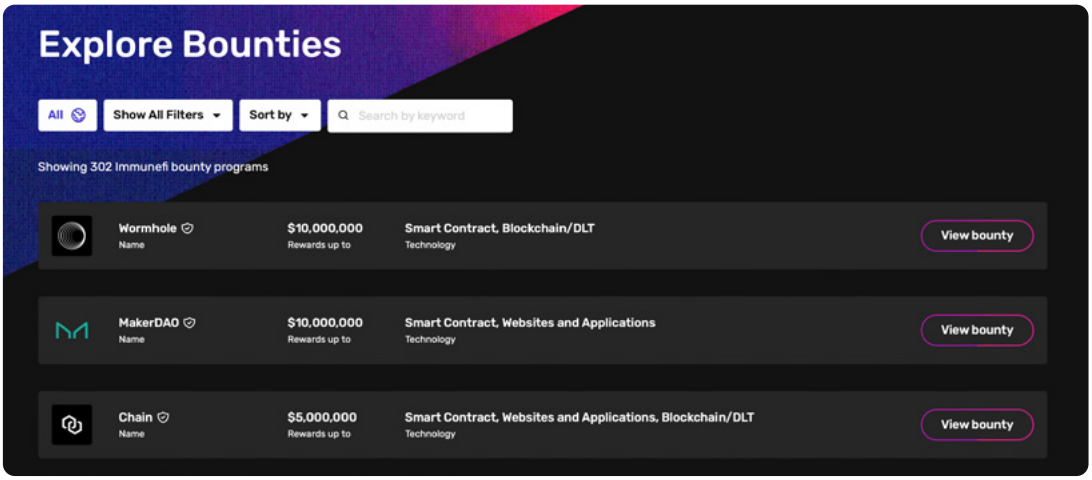
Coordinating bug bounties and vulnerability disclosures

Where testing and audits fail to catch vulnerabilities, the most feasible option is relying on an independent party to find them for you. The tools in this section are useful for coordinating bug bounty programs and vulnerability disclosures.

Immunefi

Immunefi is a leading Web3 bug bounty platform that encourages responsible disclosure of bugs and vulnerabilities found in smart contracts. Immunefi's scaling bug bounty model incentivizes ethical hackers and independent security researchers to assist Web3 developers in fixing critical errors in code as opposed to exploiting them for personal gain.

Although it is quite possible to run a vulnerability disclosure program in-house, using a bug bounty platform like Immunefi has some advantages. For example, users get a dashboard that makes it convenient and easy to receive, evaluate, and respond to bug reports—all from the same place.



Immunefi is home to some of Web3's biggest bug bounty programs (source)

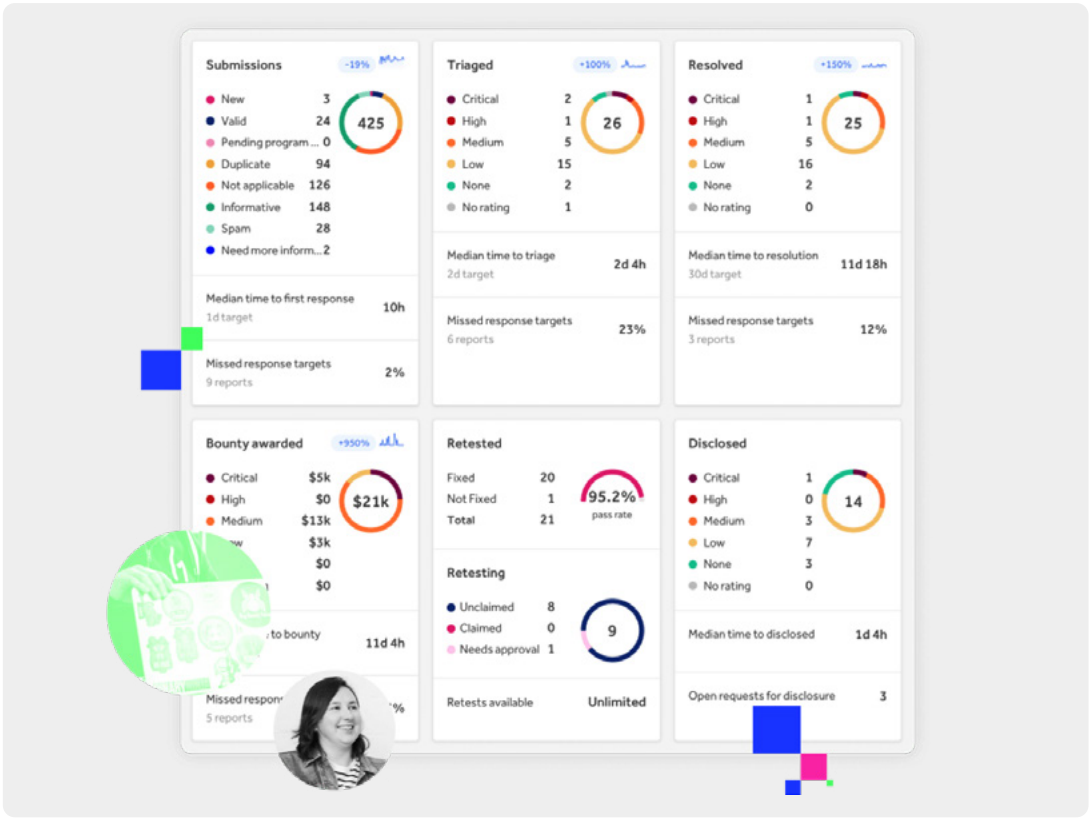
Furthermore, Immunefi offers assistance with refining details of the bug bounty, such as acceptable reports, communication channels, payout amounts, response times, etc., before launching. Publishing this information publicly ensures transparency and could help improve interactions with members of the hacker community.

Registering a bug bounty on Immunefi is free, but you'll pay a 10% "performance fee" on top of any reward paid to a bug bounty hunter. Immunefi also includes support for managing external communications in the aftermath of a vulnerability disclosure, which can help with the non-trivial task of restoring users' trust in your application.

hackerone

HackerOne is a platform for getting your code reviewed by some of the best minds in the ethical hacker community. Formed in 2012, HackerOne offers a full-featured suite of solutions designed to provide continuous security risk identification and mitigation by tapping into a pool of vetted independent security researchers and experts.

HackerOne provides a more developed framework for integrating independent security reviews into your security process. Besides running a bug bounty program, you can also schedule penetration tests for your assets. As a HackerOne user, you're automatically connected with whitehat hackers with skills and experience that fit your project's needs.



Immunefi is home to some of Web3's biggest bug bounty programs (source)

The HackerOne bug bounty platform dashboard provides an array of features to make running bug bounties easier. For example, users can access data that classifies disclosed vulnerabilities according to severity and impact using classifications like Common Weaknesses Enumeration (CWE) and Common Vulnerability Scoring System (CVSS).

Other useful features include the ability to hire hackers for retesting (after fixing the vulnerability), managing hacker payouts (send tax forms and pay out in multiple currencies), and integration with workflow tools like Slack and Microsoft Teams. Although less of a blockchain-focused bug bounty platform, HackerOne counts several Web3 companies (MetaMask, Crypto.org, Chainlink, Hyperledger) as users.

Honorable mentions for other bug bounty platforms: Code4arena, Hackenproof, Bugcrowd

CONTACT US

> Don’t be shy

We hope this guide has helped you navigate the top security tools on the market today and when to use them in your development cycle to ensure your smart contracts are as secure as possible.

While this is the end of the Diligence Security Tooling Guide, we hope it’s the beginning of your longstanding journey to achieve top security for your smart contracts with ConsenSys Diligence.

You can connect with our team at any time on the following channels:

 Send us an e-mail

 Follow us on Twitter

 Check out our website


CONSENSYS
Diligence