---

title: How to use Echidna to test smart contracts description: How to use Echidna to automatically test smart contracts author: "Trailofbits" lang: en tags: ["solidity", "smart contracts", "security", "testing", "fuzzing"] skill: advanced published: 2020-04-10 source: Building secure contracts sourceUrl: https://github.com/crytic/building-secure-contracts/tree/master/program-analysis/echidna

---

Installation {#installation}

Echidna can be installed through docker or using the pre-compiled binary.

Echidna through docker {#echidna-through-docker}

```
bash docker pull trailofbits/eth-security-toolbox docker run -it -v "$PWD":/home/training trailofbits/eth-security-toolbox
```

The last command runs eth-security-toolbox in a docker that has access to your current directory. You can change the files from your host, and run the tools on the files from the docker

Inside docker, run :

```
bash solc-select 0.5.11 cd /home/training
```

Binary {#binary}

https://github.com/crytic/echidna/releases/tag/v1.4.0.0

Introduction to property-based fuzzing {#introduction-to-property-based-fuzzing}

Echidna is a property-based fuzzer, we described in our previous blogposts (1, 2, 3).

Fuzzing {#fuzzing}

Fuzzing is a well-known technique in the security community. It consists of generating inputs that are more or less random to find bugs in the program. Fuzzers for traditional software (such as AFL or LibFuzzer) are known to be efficient tools to find bugs.

Beyond the purely random generation of inputs, there are many techniques and strategies to generate good inputs, including:

- Obtain feedback from each execution and guide generation using it. For example, if a newly generated input leads to the discovery of a new path, it can make sense to generate new inputs closes to it.
- Generating the input respecting a structural constraint. For example, if your input contains a header with a checksum, it will make sense to let the fuzzer generates input validating the checksum.
- Using known inputs to generate new inputs: if you have access to a large dataset of valid input, your fuzzer can generate new inputs from them, rather than starting from scratch its generation. These are usually called *seeds*.

Property-based fuzzing {#property-based-fuzzing}

Echidna belongs to a specific family of fuzzer: property-based fuzzing heavily inspired byQuickCheck. In contrast to classic fuzzer that will try to find crashes, Echidna will try to break user-defined invariants.

In smart contracts, invariants are Solidity functions, that can represent any incorrect or invalid state that the contract can reach, including:

- Incorrect access control: the attacker became the owner of the contract.
- Incorrect state machine: the tokens can be transferred while the contract is paused.
- Incorrect arithmetic: the user can underflow its balance and get unlimited free tokens.

Testing a property with Echidna {#testing-a-property-with-echidna}

We will see how to test a smart contract with Echidna. The target is the following smart contract `token.sol`:

```
solidity contract Token{ mapping(address => uint) public balances; function airdrop() public{
balances[msg.sender] = 1000; } function consume() public{ require(balances[msg.sender]>0); balances[msg.sender]
-= 1; } function backdoor() public{ balances[msg.sender] += 1; } }
```

We will make the assumption that this token must have the following properties:

- Anyone can have at maximum 1000 tokens
- The token cannot be transferred (it is not an ERC20 token)

Write a property {#write-a-property}

Echidna properties are Solidity functions. A property must:

- Have no argument
- Return `true` if it is successful
- Have its name starting with `echidna`

Echidna will:

- Automatically generate arbitrary transactions to test the property.
- Report any transactions leading a property to return `false` or throw an error.
- Discard side-effect when calling a property (i.e. if the property changes a state variable, it is discarded after the test)

The following property checks that the caller has no more than 1000 tokens:

```
solidity function echidna_balance_under_1000() public view returns(bool){ return balances[msg.sender] <= 1000; }
```

Use inheritance to separate your contract from your properties:

```
solidity contract TestToken is Token{ function echidna_balance_under_1000() public view returns(bool){ return
balances[msg.sender] <= 1000; } }
```

`token.sol` implements the property and inherits from the token.

Initiate a contract {#initiate-a-contract}

Echidna needs a constructor without argument. If your contract needs a specific initialization, you need to do it in the constructor.

There are some specific addresses in Echidna:

- `0x00a329c0648769A73afAc7F9381E08FB43dBEA72` which calls the constructor.
- `0x10000`, `0x20000`, and `0x00a329C0648769a73afAC7F9381e08fb43DBEA70` which randomly calls the other functions.

We do not need any particular initialization in our current example, as a result our constructor is empty.

Run Echidna {#run-echidna}

Echidna is launched with:

```
bash echidna-test contract.sol
```

If contract.sol contains multiple contracts, you can specify the target:

```
bash echidna-test contract.sol --contract MyContract
```

Summary: Testing a property {#summary-testing-a-property}

The following summarizes the run of echidna on our example:

```
solidity contract TestToken is Token{ constructor() public {} function echidna_balance_under_1000() public view
returns(bool){ return balances[msg.sender] <= 1000; } }
```

```bash echidna-test testtoken.sol --contract TestToken ...

echidna_balance_under_1000: failed!                    Call sequence, shrinking (1205/5000): airdrop() backdoor()

... ```

Echidna found that the property is violated if `backdoor` is called.

# Filtering functions to call during a fuzzing campaign {#filtering-functions-to-call-during-a-fuzzing-campaign}

We will see how to filter the functions to be fuzzed. The target is the following smart contract:

```solidity contract C { bool state1 = false; bool state2 = false; bool state3 = false; bool state4 = false;

function f(uint x) public { require(x == 12); state1 = true; }

function g(uint x) public { require(state1); require(x == 8); state2 = true; }

function h(uint x) public { require(state2); require(x == 42); state3 = true; }

function i() public { require(state3); state4 = true; }

function reset1() public { state1 = false; state2 = false; state3 = false; return; }

function reset2() public { state1 = false; state2 = false; state3 = false; return; }

function echidna_state4() public returns (bool) { return (!state4); } } ```

This small example forces Echidna to find a certain sequence of transactions to change a state variable. This is hard for a fuzzer (it is recommended to use a symbolic execution tool like Manticore). We can run Echidna to verify this:

```
bash echidna-test multi.sol ... echidna_state4: passed!   Seed: -3684648582249875403
```

Filtering functions {#filtering-functions}

Echidna has trouble finding the correct sequence to test this contract because the two reset functions (`reset1` and `reset2`) will set all the state variables to `false`. However, we can use a special Echidna feature to either blacklist the reset function or to whitelist only the `f`, `g`, `h` and `i` functions.

To blacklist functions, we can use this configuration file:

```
yaml filterBlacklist: true filterFunctions: ["reset1", "reset2"]
```

Another approach to filter functions is to list the whitelisted functions. To do that, we can use this configuration file:

```
yaml filterBlacklist: false filterFunctions: ["f", "g", "h", "i"]
```

- `filterBlacklist` is `true` by default.
- Filtering will be performed by name only (without parameters). If you have `f()` and `f(uint256)`, the filter `"f"` will match both functions.

Run Echidna {#run-echidna-1}

To run Echidna with a configuration file `blacklist.yaml`:

```
bash echidna-test multi.sol --config blacklist.yaml ... echidna_state4: failed!                  Call sequence:
f(12) g(8) h(42) i()
```

Echidna will find the sequence of transactions to falsify the property almost immediately.

Summary: Filtering functions {#summary-filtering-functions}

Echidna can either blacklist or whitelist functions to call during a fuzzing campaign using:

```
yaml filterBlacklist: true filterFunctions: ["f1", "f2", "f3"]

bash echidna-test contract.sol --config config.yaml ...
```

Echidna starts a fuzzing campaign either blacklisting `f1`, `f2` and `f3` or only calling these, according to the value of the `filterBlacklist` boolean.

How to test Solidity's assert with Echidna {#how-to-test-soliditys-assert-with-echidna}

In this short tutorial, we are going to show how to use Echidna to test assertion checking in contracts. Let's suppose we have a contract like this one:

```solidity contract Incrementor { uint private counter = 2**200;

function inc(uint val) public returns (uint){ uint tmp = counter; counter += val; // tmp <= counter return (counter - tmp); } } ```

### Write an assertion {#write-an-assertion}

We want to make sure that `tmp` is less or equal than `counter` after returning its difference. We could write an Echidna property, but we will need to store the `tmp` value somewhere. Instead, we could use an assertion like this one:

```solidity contract Incrementor { uint private counter = 2**200;

function inc(uint val) public returns (uint){ uint tmp = counter; counter += val; assert (tmp <= counter); return (counter - tmp); } } ```

### Run Echidna {#run-echidna-2}

To enable the assertion failure testing, create an Echidna configuration file `config.yaml`:

```
yaml checkAsserts: true
```

When we run this contract in Echidna, we obtain the expected results:

```bash echidna-test assert.sol --config config.yaml Analyzing contract: assert.sol:Incrementor assertion in inc: failed! Call sequence, shrinking (2596/5000): inc(217110167319967866419195596891289827224881221248076057573982970014837118074 88) inc(723700557733226221397318656304299424082937404160253525246609900049457060249 6) inc(868440669279871465676782387565159308899524884992304230295931880059348472299 52)

Seed: 1806480648350826486 ```

As you can see, Echidna reports some assertion failure in the `inc` function. Adding more than one assertion per function is possible, but Echidna cannot tell which assertion failed.

### When and how use assertions {#when-and-how-use-assertions}

Assertions can be used as alternatives to explicit properties, specially if the conditions to check are directly related with the correct use of some operation `f`. Adding assertions after some code will enforce that the check will happen immediately after it was executed:

```solidity function f(..) public { // some complex code ... assert (condition); ... }

```

On the contrary, using an explicit echidna property will randomly execution transactions and there is no easy way to enforce
```

exactly when it will be checked. It is still possible to do this workaround:

`solidity function echidna_assert_after_f() public returns (bool) { f(..); return(condition); }`

However, there are some issues:

- It fails if `f` is declared as `internal` or `external`.
- It is unclear which arguments should be used to call `f`.
- If `f` reverts, the property will fail.

In general, we recommend following John Regehr's recommendation on how to use assertions:

- Do not force any side effect during the assertion checking. For instance: `assert(ChangeStateAndReturn() == 1)`
- Do not assert obvious statements. For instance `assert(var >= 0)` where `var` is declared as `uint`.

Finally, please **do not use** `require` instead of `assert`, since Echidna will not be able to detect it (but the contract will revert anyway).

Summary: Assertion Checking {#summary-assertion-checking}

The following summarizes the run of echidna on our example:

```solidity contract Incrementor { uint private counter = 2**200;

function inc(uint val) public returns (uint){ uint tmp = counter; counter += val; assert (tmp <= counter); return (counter - tmp); } } ```

```bash echidna-test assert.sol --config config.yaml Analyzing contract: assert.sol:Incrementor assertion in inc: failed! Call sequence, shrinking (2596/5000): inc(217110167319967866419195596891289827224881221248076057573982970014837118074 88) inc(72370055773322622139731865630429942408293740416025352524660990004945706024 96) inc(86844066927987146567678238756515930889952488499230423029593188005934847229 952)

Seed: 1806480648350826486 ```

Echidna found that the assertion in `inc` can fail if this function is called multiple times with large arguments.

## Collecting and modifying an Echidna corpus {#collecting-and-modifying-an-echidna-corpus}

We will see how to collect and use a corpus of transactions with Echidna. The target is the following smart contract `magic.sol`:

```solidity contract C { bool value_found = false; function magic(uint magic_1, uint magic_2, uint magic_3, uint magic_4) public { require(magic_1 == 42); require(magic_2 == 129); require(magic_3 == magic_4+333); value_found = true; return; }

function echidna_magic_values() public returns (bool) { return !value_found; }

} ```

This small example forces Echidna to find certain values to change a state variable. This is hard for a fuzzer (it is recommended to use a symbolic execution tool like Manticore). We can run Echidna to verify this:

```bash echidna-test magic.sol ...

echidna_magic_values: passed!

Seed: 2221503356319272685 ```

However, we can still use Echidna to collect corpus when running this fuzzing campaign.

### Collecting a corpus {#collecting-a-corpus}

To enable the corpus collection, create a corpus directory:

```
bash mkdir corpus-magic
```

And an Echidna configuration file `config.yaml`:

```
yaml coverage: true corpusDir: "corpus-magic"
```

Now we can run our tool and check the collected corpus:

```
bash echidna-test magic.sol --config config.yaml
```

Echidna still cannot find the correct magic values, but we can take look to the corpus it collected. For instance, one of these files was:

```
json [ { "_gas'": "0xffffffff", "_delay": ["0x13647", "0xccf6"], "_src":
"00a329c0648769a73afac7f9381e08fb43dbea70", "_dst": "00a329c0648769a73afac7f9381e08fb43dbea72", "_value": "0x0",
"_call": { "tag": "SolCall", "contents": [ "magic", [ { "contents": [ 256,
"93723985220345906694500679277863898678726808528711107336895287282192244575836" ], "tag": "AbiUInt" }, {
"contents": [256, "334"], "tag": "AbiUInt" }, { "contents": [ 256,
"68093943901352437066264791224433559271778087297543421781073458233697135179558" ], "tag": "AbiUInt" }, { "tag":
"AbiUInt", "contents": [256, "332"] } ] ] }, "_gasprice'": "0xa904461f1" } ]
```

Clearly, this input will not trigger the failure in our property. However, in the next step, we will see how to modify it for that.

Seeding a corpus {#seeding-a-corpus}

Echidna needs some help in order to deal with the `magic` function. We are going to copy and modify the input to use suitable parameters for it:

```
bash cp corpus/2712688662897926208.txt corpus/new.txt
```

We will modify `new.txt` to call `magic(42,129,333,0)`. Now, we can re-run Echidna:

```bash echidna-test magic.sol --config config.yaml ... echidna_magic_values: failed! Call sequence: magic(42,129,333,0)

Unique instructions: 142 Unique codehashes: 1 Seed: -7293830866560616537

```

This time, it found that the property is violated immediately.

Finding transactions with high gas consumption {#finding-transactions-with-high-gas-consumption}

We will see how to find the transactions with high gas consumption with Echidna. The target is the following smart contract:

```solidity contract C { uint state;

function expensive(uint8 times) internal { for(uint8 i=0; i < times; i++) state = state + i; }

function f(uint x, uint y, uint8 times) public { if (x == 42 && y == 123) expensive(times); else state = 0; }

function echidna_test() public returns (bool) { return true; }

} ```

Here `expensive` can have a large gas consumption.

Currently, Echidna always need a property to test: here `echidna_test` always returns `true`. We can run Echidna to verify this:

``` echidna-test gas.sol ... echidna_test: passed!

Seed: 2320549945714142710 ```

## Measuring Gas Consumption {#measuring-gas-consumption}

To enable the gas consumption with Echidna, create a configuration file`config.yaml`:

```
yaml estimateGas: true
```

In this example, we will also reduce the size of the transaction sequence to make the results easier to understand:

```
yaml seqLen: 2 estimateGas: true
```

Run Echidna {#run-echidna-3}

Once we have the configuration file created, we can run Echidna like this:

```bash echidna-test gas.sol --config config.yaml ... echidna_test: passed!

f used a maximum of 1333608 gas Call sequence: f(42,123,249) Gas price: 0x10d5733f0a Time delay: 0x495e5 Block delay: 0x88b2

Unique instructions: 157 Unique codehashes: 1 Seed: -325611019680165325

```

- The gas shown is an estimation provided by HEVM.

Filtering Out Gas-Reducing Calls {#filtering-out-gas-reducing-calls}

The tutorial on **filtering functions to call during a fuzzing campaign** above shows how to remove some functions from your testing.
This can be critical for getting an accurate gas estimate. Consider the following example:

```
solidity contract C { address [] addrs; function push(address a) public { addrs.push(a); } function pop() public { addrs.pop(); } function clear() public{ addrs.length = 0; } function check() public{ for(uint256 i = 0; i < addrs.length; i++) for(uint256 j = i+1; j < addrs.length; j++) if (addrs[i] == addrs[j]) addrs[j] = address(0x0); } function echidna_test() public returns (bool) { return true; } }
```

If Echidna can call all the functions, it won't easily find transactions with high gas cost:

```
echidna-test pushpop.sol --config config.yaml ... pop used a maximum of 10746 gas ... check used a maximum of 23730 gas ... clear used a maximum of 35916 gas ... push used a maximum of 40839 gas
```

That's because the cost depends on the size of`addrs` and random calls tend to leave the array almost empty. Blacklisting `pop` and `clear`, however, gives us much better results:

```
yaml filterBlacklist: true filterFunctions: ["pop", "clear"]
```

```
echidna-test pushpop.sol --config config.yaml ... push used a maximum of 40839 gas ... check used a maximum of 1484472 gas
```

Summary: Finding transactions with high gas consumption {#summary-finding-transactions-with-high-gas-consumption}

Echidna can find transactions with high gas consumption using the`estimateGas` configuration option:

```
yaml estimateGas: true
```

```
bash echidna-test contract.sol --config config.yaml ...
```

Echidna will report a sequence with the maximum gas consumption for every function, once the fuzzing campaign is over.