CONSENSYS
Diligence

HOW TO PREPARE YOUR SMART CONTRACTS FOR AN AUDIT

August 14th 2019

Diligence Presenters

John Mardlin
Security Auditor

Brianna Montgomery
Business Development

Steve Marx
Security Auditor

About ConsenSys Diligence

Creating a safe, trustworthy and healthy Ethereum ecosystem since 2017.

0xProject

USDC
(Coinbase + Circle)

Aragon Network

komgo

Uniswap

GNOSIS

ETHEREUM
NAME SERVICE

About ConsenSys Diligence

The team behind:

MythX

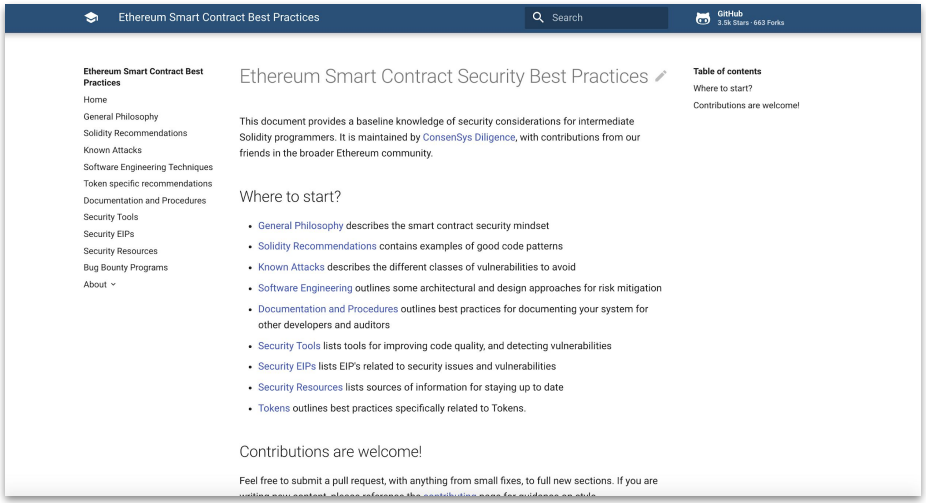
The Smart Contract Security Tool for Ethereum

→ mythx.io

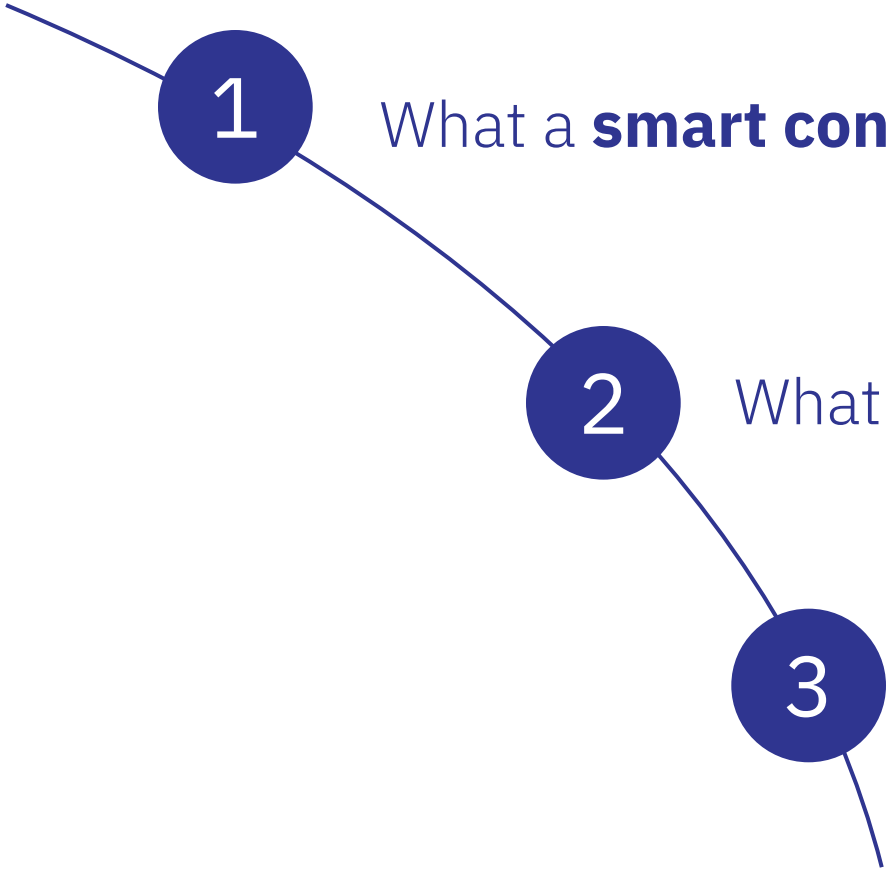
About ConsenSys Diligence

Authors of the industry standard guide to secure development practices

The Ethereum Smart Contract Best Practices Guide



Agenda



1. What a **smart contract audit** is (and what it is not)
2. What **you can do** to get the most out of an audit
3. What to **expect** during the process

First,
a poll

What is an audit?

What is an audit?

A security 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.

What is an audit?

An opportunity to:

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

The limitations of an audit

An audit can not:

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

The value of an audit

AUDIT

Systematic review

Qualitative assessment

Suggestions for improvement

Thorough coverage of code in scope

Should leave your team better educated

VS

BUG BOUNTY

Reviews tend to be shallower

Typically identify well known standard bug classes

Rarely identify subtle logic bugs or unspecified behaviour

Requires substantial rewards and promotional effort.

Might result in no review at all!

Do you really need an audit?

What is at risk?

How complex is your code?

Preparing for an audit

The Preparedness Mindset

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.

Steps to prepare

1

Document it

2

Make it easy to run

3

Clean up the code

4

Run a linter

5

Run analysis tools

6

Freeze the Code

Documentation

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

Documentation

Documenting your code will also help you to improve it.

Documentation

Example:

0x Protocol Specifications

0x protocol 2.0.0 specification

Table of contents

0xProject

Documentation

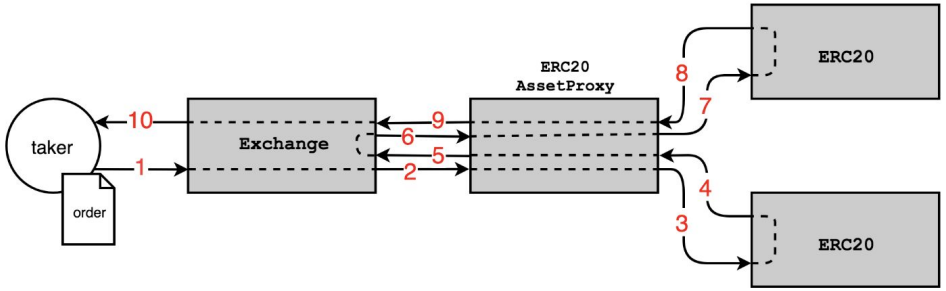
Example:

0x Protocol Specifications

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`)

0xProject

Make it easy to install and run the tests

Setup

The smart contracts are written in Solidity and tested/deployed using Truffle version 4.1.0. The new version of Truffle doesn't require testrpc to be installed separately so you can just run the following:

```
# Install Truffle package globally:
$ npm install --global truffle

# (Only for windows) set up build tools for node-gyp by running below command in powershell:
$ npm install --global --production windows-build-tools

# Install local node dependencies:
$ yarn
```

Testing

To test the code simply run:

```
# on *nix systems
$ npm run test

# on windows systems
$ npm run wintest
```

Source: Polymath

Clean up the code

Add helpful comments

```
// Derive maker asset amounts for left & right orders, given store taker assert amounts
uint256 leftTakerAssetAmountRemaining = _safeSub(leftOrder⬆.takerAssetAmount, leftOrderTakerAssetFilled
uint256 leftMakerAssetAmountRemaining = _safeGetPartialAmountFloor(
    leftOrder⬆.makerAssetAmount,
    leftOrder⬆.takerAssetAmount,
    leftTakerAssetAmountRemaining
);
```

Clean up the code

Use NatSpec comments

```
/// @dev Calculates fill amounts for the matched orders.
///      Each order is filled at their respective price point. However, the calculations are
///      carried out as though the orders are both being filled at the right order's price point.
///      The profit made by the leftOrder order goes to the taker (who matched the two orders).
/// @param leftOrder First order to match.
/// @param rightOrder Second order to match.
/// @param leftOrderTakerAssetFilledAmount Amount of left order already filled.
/// @param rightOrderTakerAssetFilledAmount Amount of right order already filled.
/// @param shouldMaximallyFillOrders A value that indicates whether or not this calculation should use
///                                  the maximal fill order matching strategy.
/// @param matchedFillResults Amounts to fill and fees to pay by maker and taker of matched orders.
function calculateMatchedFillResults(
```

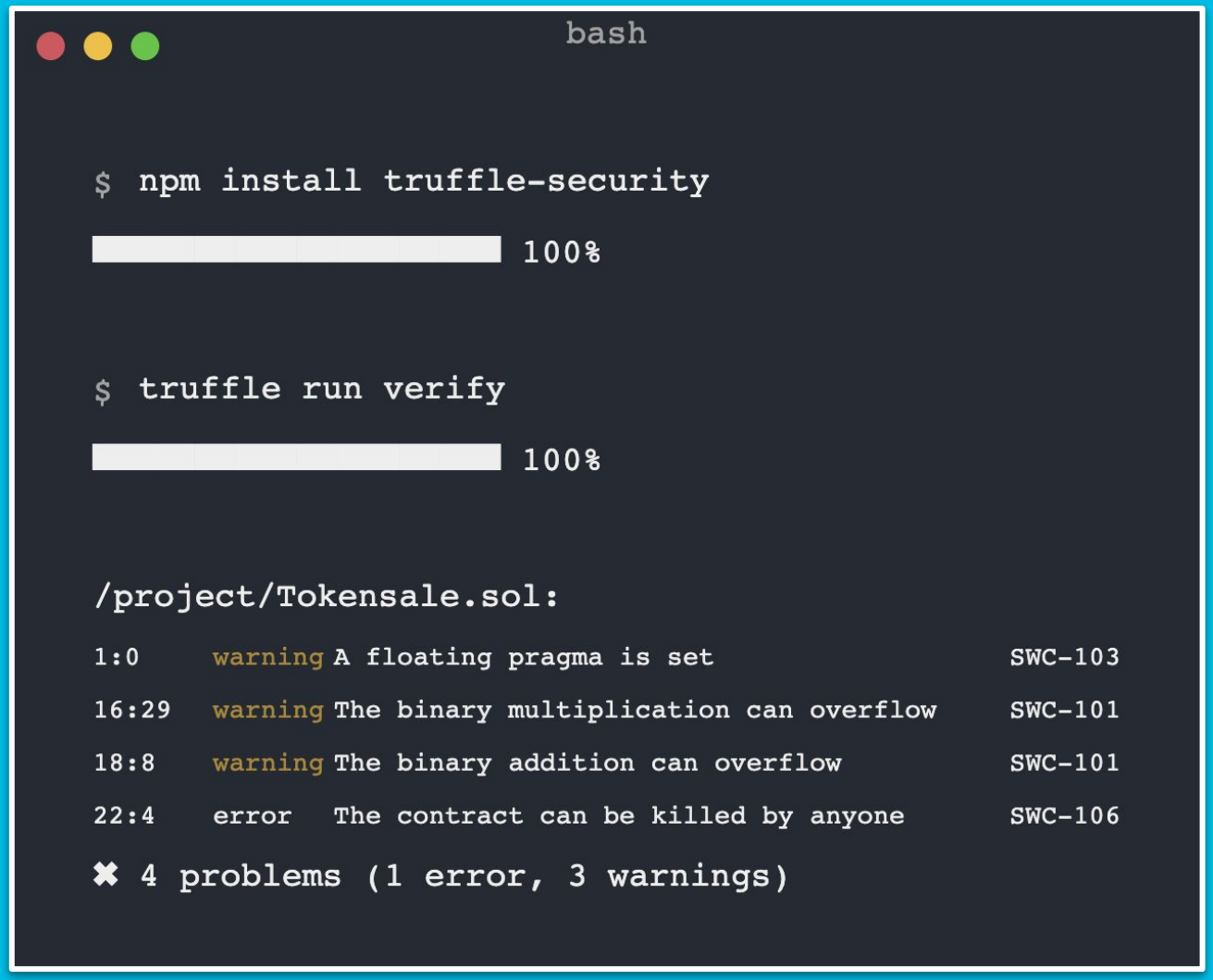
Clean up the code

Resolve TODO/FIXME comments

Delete commented-out blocks of code

Run analysis tools

MythX


bash
$ npm install truffle-security
100%
$ truffle run verify
100%
/project/Tokensale.sol:
1:0 warning A floating pragma is set SWC-103
16:29 warning The binary multiplication can overflow SWC-101
18:8 warning The binary addition can overflow SWC-101
22:4 error The contract can be killed by anyone SWC-106
✖ 4 problems (1 error, 3 warnings)

MythX

MythX

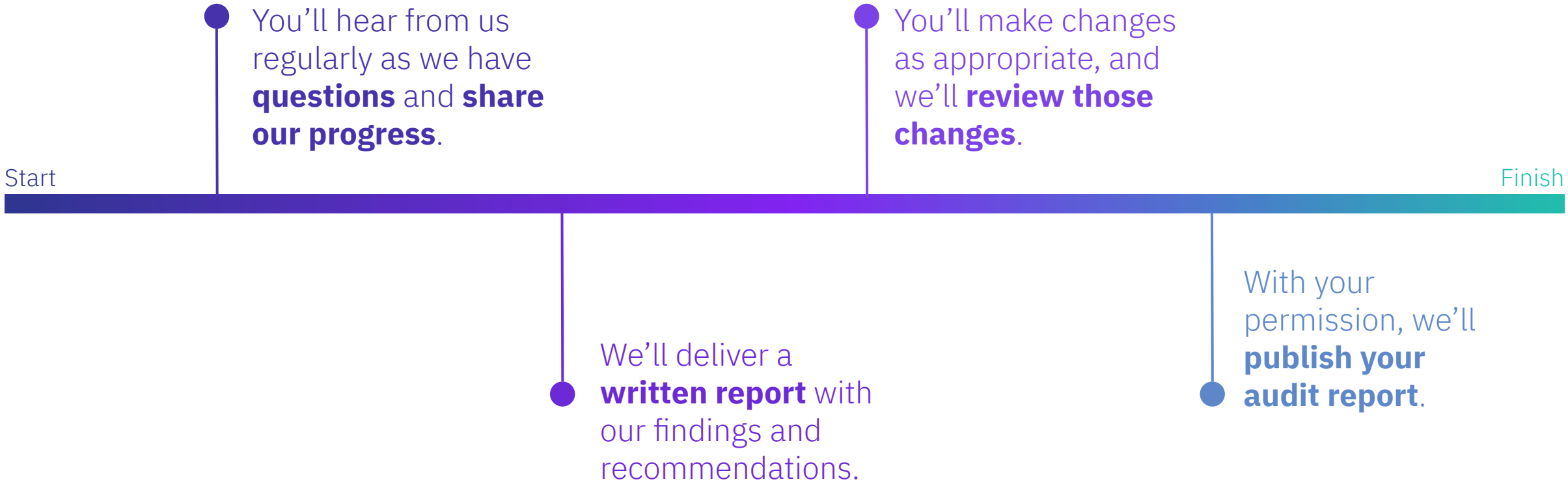
MythX

Code freeze

We can’t audit a **moving target**.

What to expect from an audit

What to expect during an audit



Example Findings: ENS

Front-running due to flawed commit/reveal scheme

Good documentation explained how things were intended to work

Hard to detect with tests or tools without already knowing the bug

3.3 `ETHRegistrarController.register` is vulnerable to front running Critical ✓ Fixed

Resolution

Issue has been closed in ensdomains/ethregistrar#18

Description

`commit()` and then `register()` appears to serve the purpose of preventing front running. However, because the commitment is not tied to a specific owner, it serves equally well as a commitment for a front-running attacker.

Example

1. Alice calls `commit(makeCommitment("mydomain", <secret>))`.
2. 10 minutes later, Alice submits a transaction to `register("mydomain", Alice, ..., <secret>)`.
3. Eve observes this transaction in the transaction pool.
4. Eve submits `register("mydomain", Eve, ..., <secret>)` with a higher gas price and wins the race.

Remediation

Commitments should commit to `owner`s in addition to `name`s. This way an attacker can't repurpose a previous commitment. (They would have to buy on behalf of the original committer.)

As an alternative, if it's undesirable to pin down `owner`, the commitment could include `msg.sender` instead (only allowing the original committer to call `register`).

E.g. the following (and corresponding changes to callers):

```
function makeCommitment(
    string memory name,
    address owner, /* or perhaps committer/sender */
    bytes32 secret
)
    pure
    public
    returns(bytes32)
{
    bytes32 label = keccak256(bytes(name));
    return keccak256(abi.encodePacked(label, owner, secret));
}
```

ETHEREUM NAME SERVICE

Example Findings: Uniswap

Possible malicious reentrancy from non-malicious tokens

Requires broad knowledge of token design and ERC standards

Hard to anticipate when writing tests

3.1 Liquidity pool can be stolen in some tokens (e.g. ERC-777) (#29)

Severity	Status	Link	Remediation Comment
Major	Open	issues/29	The issue is currently under review

Description

If token allows making reentrancy on `transferFrom(address from, address to, uint tokens)` function by someone except the recipient, then all the liquidity funds might be stolen. For example, if token calls callback function of `from` address. It's irrelevant if reentrancy is done before or after the balances update.

Attack

Let's imagine we have a token that calls a callback function of `from` address on `transferFrom(address from, address to, uint tokens)` and allows `from` address to make a reentrancy. We will consider the case when reentrancy is made after the token balances are updated. If token balances are updated after the reentrancy (e.g. ERC-777), the algorithm is even easier and requires fewer funds to steal liquidity pool.

In `tokenToTokenInput` we have the following 2 lines of code

```
assert self.token.transferFrom(buyer, self, tokens_sold)
tokens_bought: uint256 = Exchange(exchange_addr).ethToTokenTransferInput(min_tokens_bought, deadline, reci
```

Attacker(`buyer`) can make reentrancy on the first line here.

1. Assume we have an exchange with a token that worth equally to ETH with liquidity pool equals (100 tokens, 100 ETH)
2. An attacker creates a fake Exchange (it will be the second exchange in `tokenToToken` transfers) that will receive ETH from the first exchange and behave like a normal exchange.
3. The attacker can buy 50 ETH for 100 tokens by using `tokenToTokenInput` function.
4. New liquidity pool should be (200 tokens, 50 ETH) but since the attacker makes reentrancy on `assert self.token.transferFrom(buyer, self, tokens_sold)` it will still be (200 tokens, 100 ETH).
5. While making reentrancy the attacker can buy 49.999 ETH for about 200 tokens using `tokenToEthSwapInput` .
6. After that, the liquidity pool should look like (400 tokens, 0.001 ETH)
7. Now the attacker can buy all the tokens for a very small amount of ETH.

This is not a very accurate algorithm that does not include fees and gas cost, but logic stays the same.

Remediation

Add mutex to all functions that make trades in order to prevent reentrancy.

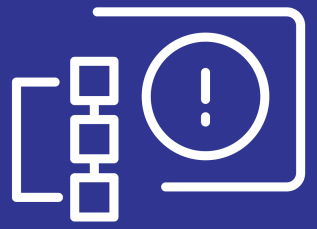
Uniswap

We can help!

Our services

Launch Assessments

Security Audits



Engineering Advisory

Contact us

Brianna Montgomery

brianna.montgomery@consensys.net

bpm6867 BriannaPageM Brianna Montgomery

CONSENSYS Diligence

Questions?