SWINBURNE UNIVERSITY OF TECHNOLOGY

Week 10

Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.

More Security Issues

Incorrect Constructor Name

The "Incorrect Constructor Name" vulnerability pertains to smart contracts, especially those deployed on the Ethereum platform, and is typically associated with the contract's constructor function. A constructor is a special function that runs when a contract is deployed, often used to initialize the contract's state variables.

Incorrect Constructor Name

Vulnerability Details

In Solidity, the constructor function was named the same as the contract. If developers failed to update the constructor's name when copying or refactoring a smart contract, it could introduce the "Incorrect Constructor Name" vulnerability. This means that the code, which should run when the contract is created, won't be executed as it is no longer recognized as a constructor.

Incorrect Constructor Name

- **Constructor:**
- **A constructor in Solidity is a special function that is used to initialize state variables in a contract.**
- **The constructor is called when an contract is first created and can be used to set initial values.**
  - Constructors are an optional function
  - In case, no constructor is defined, a default constructor is present in the contract
  - The constructor is executed one time when the contract is first created and does not run again
  - A constructor can be either public or internal.

Incorrect Constructor Name

- **Constructor:**
  - A constructor is declared using the constructor keyword.
  - These functions can be public or internal.
  - If a constructor is not specified the contract will assume the default constructor which is equivalent to constructor() public {}.

```
pragma solidity ^0.8.0;

contract A {
	uint public a;

	//when creating this contract _a is passed in a a parameter
	//and sets the variable a
	constructor(uint _a) internal {
		a = _a;
	}
}
```

Incorrect Constructor Name

- Incorrect Constructor Name vulnerabilities occur when constructor is named incorrectly.
- If this disparity between contract name and constructor name occurs in development, the constructor will revert to a publicly callable function. (recall the issue Function Default Visibility).
- This allows malicious actors to perform unintended or unauthorized actions.
- This vulnerability affected Solidity version before 0.4.22, while more recent versions require the constructor to be defined with the constructor keyword, effectively mitigating this vulnerability.

Incorrect Constructor Name

- The following vulnerable contract designates the address which initializes it as the contract's owner.
- This is a common pattern adopted to grant special privileges, such as the ability to withdraw contract funds.

```
pragma solidity ^0.4.15;

contract Missing{
        address private owner;

        modifier onlyowner {
                require(msg.sender==owner);
                _;
        }

        // The name of the constructor should be Missing
        // Anyone can call the IamMissing once the contract is deployed
        function IamMissing() public
        {
                owner = msg.sender;
        }

        function withdraw() public onlyowner
        {
                owner.transfer(this.balance);
        }
}
```

Incorrect Constructor Name

- Fixed example:

  Developers should rewrite the contract to use the constructor keyword.
- Prevention

  Use constructor instead of a named constructor to prevent this issue.

```
constructor() public
{
      owner = msg.sender;
}
```

Incorrect Constructor Name

Potential Risks

- **Uninitialized State Variables**: The contract's state variables might not be initialized correctly since the constructor does not run, leading to logical issues in the contract.

- **Additional Gas Costs**: The incorrect constructor might become a regular function of the contract, consuming additional gas if it is called externally.

- **Logical Errors and Security Risks**: The non-running constructor might lead to unforeseen logical errors and security risks in the contract.

Insufficiently Random Values

- Ethereum has been used as a platform for a variety of applications of financial interest.
- Several of these have a need for randomness — e.g., to implement a lottery, a competitive game, or crypto-collectibles.

Insufficiently Random Values

- Difficulties of random number generation
  - Writing a random number generator on a public blockchain is hard;
  - Computation needs to be deterministic, so that it can be replayed in a decentralized way;
  - and all data that can serve as sources of randomness are also available to an attacker.
  - Several exploits of bad randomness have been discussed exhaustively in the past.

Insufficiently Random Values

- Ethereum Randomness Practices
  - The Ethereum Yellow Paper itself suggests "[approximating randomness with] pseudo-random numbers by utilising data which is generally unknowable at the time of transacting.
  - An attacker can get the same information as the victim contract by just having a transaction in the same block.
  - In this way, an attacker can replay the randomness computation of the attacked contract before deciding whether to take a random bet.

Insufficiently Random Values

- The contract performs an extra store in the case of a winning outcome.

```
contract Victim {
        mapping (address => uint32) winners;
        …
        function draw(uint256 betGuess) public payable {
        require (msg.value >= 1 ether);
        uint16 outcome = badRandom(betGuess);
        if (winning(outcome))
        winners[msg.sender] = outcome;
        }
}
```

Insufficiently Random Values

- The attacker can trivially exploit this to leak information about the outcome, before the transaction even completes:

```
contract Attacker {
        function test() public payable {
        Victim v = Victim(address(<address of victim>));
        v.draw.value(msg.value)(block.number); // or any guess
        require (gasleft() < 253000); // or any number that will
                                      // distinguish an extra store
                                      // relative to the original gas
        }
}
```

Insufficiently Random Values

To summarize, the recommendation for on-chain random number generation is to follow a pattern such as:

- Accept a bet, with payment, register the block number of the bet transaction.
- The bettor has to not only place the bet but also invoke the contract in a future transaction (within the next 256 blocks).
- If the bettor is too late (or too early) the outcome should favor the contract, not a potential attacker.
- The expected value of the random trial for all bets in a single block should be lower than the reward for mining a block.

Griefing

- This attack may be possible on a contract which accepts generic data and uses it to make a call another contract (a 'sub-call') via the low level address.call() function, as is often the case with multi-signature and transaction relayer contracts.
- If the call fails, the contract has two options:
  - revert the whole transaction
  - continue execution.

Griefing

- Take the following example of a simplified Relayer contract which continues execution regardless of the outcome of the subcall:

```
contract Relayer {
	mapping (bytes => bool) executed;

	function relay(bytes _data) public {
		// replay protection; do not call the same transaction twice
		require(executed[_data] == 0, "Duplicate call");
		executed[_data] = true;
		innerContract.call(bytes4(keccak256("execute(bytes)")), _data);
	}
}
```

Griefing

- This contract allows transaction relaying.
- Someone who wants to make a transaction but can't execute it by himself (e.g. due to the lack of ether to pay for gas) can sign data that he wants to pass and transfer the data with his signature over any medium.
- A third party "forwarder" can then submit this transaction to the network on behalf of the user.

```
contract Relayer {
	mapping (bytes => bool) executed;

	function relay(bytes _data) public {
		// replay protection; do not call the same transaction twice
		require(executed[_data] == 0, "Duplicate call");
		executed[_data] = true;
		innerContract.call(bytes4(keccak256("execute(bytes)")), _data);
	}
}
```

Griefing

- If given just the right amount of gas, the Relayer would complete execution recording the _data argument in the executed mapping, but the subcall would fail because it received insufficient gas to complete execution.
- An attacker can use this to censor transactions, causing them to fail by sending them with a low amount of gas.

```
contract Relayer {
	mapping (bytes => bool) executed;

	function relay(bytes _data) public {
		// replay protection; do not call the same transaction twice
		require(executed[_data] == 0, "Duplicate call");
		executed[_data] = true;
		innerContract.call(bytes4(keccak256("execute(bytes)")), _data);
	}
}
```

Griefing

- This attack is a form of "griefing": It doesn't directly benefit the attacker, but causes grief for the victim.
- A dedicated attacker, willing to consistently spend a small amount of gas could theoretically censor all transactions this way, if they were the first to submit them to Relayer.

```
contract Relayer {
	mapping (bytes => bool) executed;

	function relay(bytes _data) public {
		// replay protection; do not call the same transaction twice
		require(executed[_data] == 0, "Duplicate call");
		executed[_data] = true;
		innerContract.call(bytes4(keccak256("execute(bytes)")), _data);
	}
}
```

Griefing

- One way to address this is to implement logic requiring forwarders to provide enough gas to finish the subcall.
- If the miner tried to conduct the attack in this scenario, the require statement would fail and the inner call would revert.
- A user can specify a minimum gasLimit along with the other data

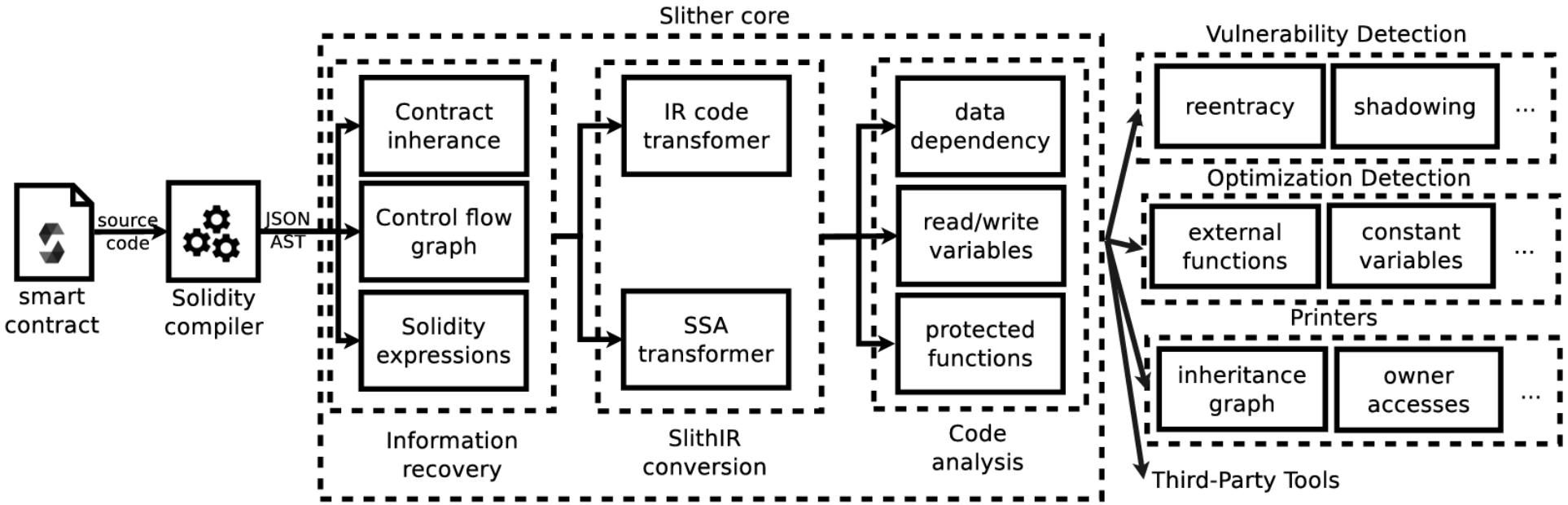
```
// contract called by Relayer
contract Executor {
        function execute(bytes _data, uint _gasLimit) {
                require(gasleft() >= _gasLimit);
                ...
        }
}
```

Auditing Tools

SWINBURNE UNIVERSITY OF TECHNOLOGY

What is Slither ?

Slither is a Solidity static analysis framework written in Python3. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses.



SWINBURNE UNIVERSITY OF TECHNOLOGY

What is Slither ?

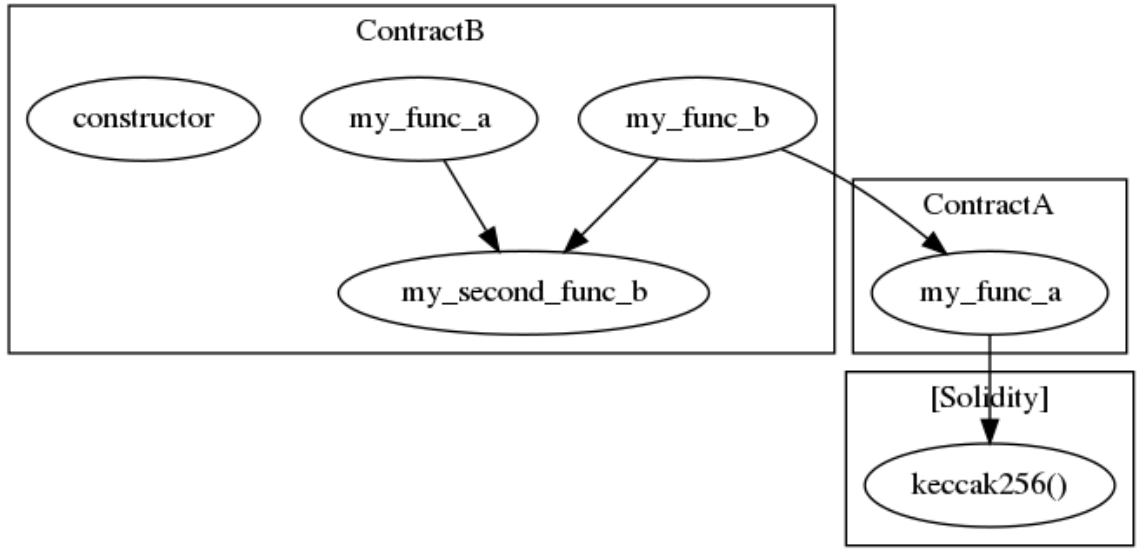
A call graph typically illustrates how the execution flows from one function to another, including both internal and external function calls. It helps in understanding the overall structure and interactions within a smart contract system

`slither file.sol --print call-graph`

Export the call-graph of the contracts to a dot file

Example

```
$ slither examples/printers/call_graph.sol --print call-graph
```



The output format is dot. To vizualize the graph:

```
$ xdot examples/printers/call_graph.sol.dot
```

To convert the file to svg:

```
$ dot examples/printers/call_graph.sol.dot -Tpng -o examples/printers/call_graph.sol.png
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

What is Slither ?

Data dependency is crucial to understand because it affects the behavior and correctness of smart contracts. Changes or updates to one data element can have consequences for other dependent data elements. When data dependencies are not handled properly, it can lead to unexpected or unintended

SWINBURNE UNIVERSITY OF TECHNOLOGY

Data Dependencies

Print the data dependencies of the variables `slither file.sol --print data-dependency`

Example

```
$ slither examples/printers/data_dependencies.sol --print data-dependency
```

```
Contract MyContract
+----------+----------------------+
| Variable |     Dependencies     |
+----------+----------------------+
|    a     |     ['input_a']      |
|    b     | ['input_b', 'input'] |
|    c     |          []          |
+----------+----------------------+

Function setA(uint256,uint256)
+--------------+--------------+
|   Variable   | Dependencies |
+--------------+--------------+
|   input_a    |      []      |
|   input_b    |      []      |
| MyContract:a | ['input_a']  |
| MyContract:b |      []      |
| MyContract:c |      []      |
+--------------+--------------+
Function setB(uint256)
+--------------+----------------------+
|   Variable   |     Dependencies     |
+--------------+----------------------+
|    input     |     ['input_b']      |
| MyContract:a |          []          |
| MyContract:b | ['input_b', 'input'] |
| MyContract:c |          []          |
+--------------+----------------------+
```

What is Slither ?

an inheritance graph represents the hierarchical relationship between different smart contracts that utilize inheritance. Inheritance is a feature in many programming languages, including Solidity (the language commonly used for Ethereum smart contracts), that allows a contract to inherit properties and functionality from another contract.

`slither file.sol --print inheritance-graph`

Output a graph showing the inheritance interaction between the contracts.

Example

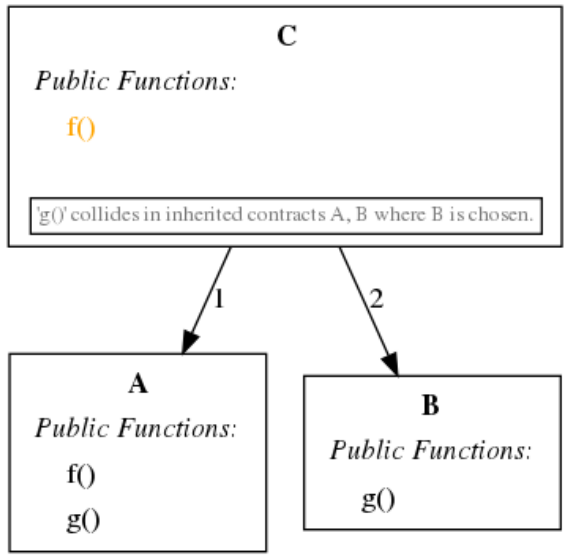
```
$ slither examples/printers/inheritances.sol --print inheritance-graph
[...]
INFO:PrinterInheritance:Inheritance Graph: examples/DAO.sol.dot
```

The output format is dot. To vizualize the graph:

```
$ xdot examples/printers/inheritances.sol.dot
```

To convert the file to svg:

```
$ dot examples/printers/inheritances.sol.dot -Tsvg -o examples/printers/inheritances.sol.png
```



SWINBURNE UNIVERSITY OF TECHNOLOGY

What is Slither ?

https://github.com/crytic/slither/wiki/Printer-documentation#cfg

Try each printer on your smart contract files.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Printer documentation

Feist Josselin edited this page on Jan 4, 2021 · 36 revisions

Slither allows printing contracts information through its printers.

Num	Printer	Description
1	`call-graph`	Export the call-graph of the contracts to a dot file
2	`cfg`	Export the CFG of each functions
3	`constructor-calls`	Print the constructors executed
4	`contract-summary`	Print a summary of the contracts
5	`data-dependency`	Print the data dependencies of the variables
6	`echidna`	Export Echidna guiding information
7	`evm`	Print the evm instructions of nodes in functions
8	`function-id`	Print the keccack256 signature of the functions
9	`function-summary`	Print a summary of the functions
10	`human-summary`	Print a human-readable summary of the contracts
11	`inheritance`	Print the inheritance relations between contracts
12	`inheritance-graph`	Export the inheritance graph of each contract to a dot file
13	`modifiers`	Print the modifiers called by each function
14	`require`	Print the require and assert calls of each function
15	`slithir`	Print the slithIR representation of the functions
16	`slithir-ssa`	Print the slithIR representation of the functions
17	`variable-order`	Print the storage order of the state variables
18	`vars-and-auth`	Print the state variables written and the authorization of the functions

How to write a smart contract audit report ?

Writing a professional smart contract audit report requires a combination of technical expertise, meticulous analysis, and clear communication.

1. Introduction

- **Objective:** Clearly state the purpose of the audit.
- **Scope:** Define what is and isn't covered in the audit.
- **Methodology:** Briefly describe the approach and tools used during the audit.

2. Executive Summary

- **Overview:** Provide a brief summary of the findings, risks, and recommendations.
- **Audit Highlights:** Mention any critical issues, vulnerabilities, or commendable practices observed.

SWINBURNE UNIVERSITY OF TECHNOLOGY

How to write a smart contract audit report ?

3. Contract Overview

- **Contract Description:** Provide a brief description of the smart contract and its functionality.
- **Codebase:** Mention the programming language, frameworks, and libraries used.
- **Dependencies:** List and describe external contracts and libraries that the code interacts with.

4. Methodology

- **Audit Process:** Describe the steps taken during the audit, such as code review, testing, and analysis.
- **Tools and Technologies:** Mention the tools and technologies used for testing and analysis.
- **Assumptions:** State any assumptions made during the audit.

SWINBURNE UNIVERSITY OF TECHNOLOGY

How to write a smart contract audit report ?

5. Findings

Organize findings into different categories based on severity:

- **Critical Issues:** Describe issues that pose severe risks and require immediate attention.
- **Major Issues:** Detail significant problems that could impact the functionality or security of the contract.
- **Minor Issues:** Mention less severe issues that do not pose immediate risks.
- **Notes/Observations:** Include any additional observations or suggestions.

For each issue, provide the following details:

- **Description:** Explain the issue and its potential impact.
- **Location:** Specify where in the code the issue was found.
- **Recommendation:** Suggest possible fixes or improvements.
- **Status:** Indicate whether the issue has been resolved.

SWINBURNE UNIVERSITY OF TECHNOLOGY

How to write a smart contract audit report ?

6. Security Assessment

- **Security Posture:** Provide an overall assessment of the contract's security.
- **Vulnerabilities:** Detail any vulnerabilities found and their potential impact.
- **Security Best Practices:** Highlight areas where the contract adheres to security best practices.

7. Code Quality Assessment

- **Coding Standards:** Evaluate the code for readability, maintainability, and adherence to coding standards.
- **Documentation:** Assess the quality and completeness of code documentation.
- **Test Coverage:** Evaluate the extent and effectiveness of test coverage.

8. Conclusion

- **Summary:** Summarize the key findings and overall assessment.
- **Final Thoughts:** Provide any additional comments or insights.
- **Recommendations:** Offer final recommendations for improving the contract.

SWINBURNE UNIVERSITY OF TECHNOLOGY