A Large-Scale Study on Vulnerability Scanners for Ethereum Smart Contracts

Christoph Sendner, Lukas Petzi, Jasper Stang, Alexandra Dmitrienko
University of Wurzburg, Germany

Seminar for **Decentralized System** master course

Author:
Kevin Cattaneo – S4944382

a.y. 2024/25

UNIVERSITÀ DEGLI STUDI DI GENOVA

Index

In this seminar we will talk about:

An expensive recap

Cryptocurrency hacks have become increasingly common in recent years:

- **DAO hack -** the first → **60 million dollars** and a **hard fork** of Ethereum.
- **Safemoon Hack -** access control vulnerability → **8.9 million dollars.**
- **LendHub hack** - wrong update mechanism → **6 million dollars**.
- **Deus Finance hack -** access control issue → **13.4 million dollars**.

Paper contribution

“We acknowledge the efforts of previous works that have attempted already to compare various tools for vulnerability detection.

However, these works either provide a survey-like comparison without performing actual performance evaluations or evaluate a small subset of tools on a very limited dataset.

In addition, some of these works, only analyze tools that operate either on **bytecode** or **source code** and do not offer a comprehensive exploration of **both**. [...]”

Vulnerability detection

In recent years, many vulnerability detection tools have been developed for smart contracts that can aid smart contract developers in **pre-deployment** security testing.

Interestingly, even though some of these tools have been available for years and provided under **open-source** licenses, smart contracts continue to suffer from **vulnerabilities** that malicious actors can exploit.

But why?

Questions to answer

- Why, **despite the existence** of many effective vulnerability detection tools, does the problem of vulnerabilities in smart contracts still prevail?
- Is it due to difficulties of **setting up** and using those tools, which raises the adaption barrier?
- Is because they are **less effective** than the security research community believes?
- If some tools are better suitable for the detection of selected vulnerability types, can one achieve better detection by using **several tools**?

Vulnerability categories

The equivalent to CVE report is the **Smart Contract Weakness Classification Registry (SWC).**

These vulnerabilities can be splitted into three categories:

- **Software Errors:** software errors often resulting in bugs
- **Runtime Bugs:** bugs associated with the contract execution runtime.
- **Blockchain Characteristics:** the blockchain infrastructure itself can give rise to vulnerabilities within smart contract

Vulnerabilities/Bugs – Software Errors

- **arithmetic issues -** such as integer overflows and underflows. Notably, these vulnerabilities have been partly addressed through a compiler update in Solidity
- **suicide vulnerability -** can potentially allow an attacker to destroy a contract
- **assert violation -** arises when developers inadvertently leave a failing assert statement in live code
- **misuse of txOrigin -** if exploited in place of msg.sender, could allow attackers to manipulate a global variable to their advantage
- **use of block timestamps as a proxy for time**, which can be exploited by attackers aware of this time dependency
- **locked Ether -** arises from the absence of a mechanism to withdraw Ether from the contract

Vulnerabilities/Bugs – Runtime Bugs

- **reentrancy -** where an attacker can exploit the ability to reenter a function during runtime, even before the initial execution is completed
- **legacy callstack depth vulnerability –** legacy call stack was up to 1024 then the stack becomes exhausted → no more calls
- **use of DelegateCall -** this feature allows a contract to execute another contract’s code within its own context

Vulnerabilities/Bugs – Blockchain characteristic

- **Transaction Ordering Dependency (ToD) -** can result in financial losses when the code depends on the sequence of transactions, such as determining the first correct answer submitted

- **Usage of blockchain’s block values for time related purposes -** these values can be influenced by miners / validators and should not be relied upon

- **Creating randomness -** within smart contracts becomes challenging due to their deterministic execution nature

The scanners tested

- Analysis with **13** source code-based tools: Slither, SmartCheck, Maian, Oyente, Artemis, Osiris, Securify2, Mythril, TeEther, ConFuzzius, Smartian, sFuzz, GNNSCVulDetector and attempted to detect **8** vulnerability types in our source code-based dataset.
- **Four** vulnerability detection tools that operate on the byte-code level: Vandal, Maian, Oyente, Mythril. Their performance was evaluated using **5** vulnerability types.
- These tools can be classified into four categories:
  - **static analysis**
  - **symbolic analysis**
  - **fuzzing**
  - **machine learning approaches**
- The results of this evaluation similarly display **lack of consensus** among the tools, to the extent that questions their ability for accurate detection.

How scanners work – Static Analysis

Static Analysis involves examining source code or compiled output without requiring an execution environment or risking running vulnerable or malicious code.

- **Slither** is a static analysis framework for finding vulnerabilities in Solidity source files.
- **Vandal** is based on EVM bytecodes. It operates by converting the bytecode into semantic logic relations.
- **SmartCheck** it employs text parsing techniques to convert Solidity source code into an XML parse tree, which is subsequently examined for vulnerability patterns. From 2020 is **deprecated**.
- **EtherTrust** it abstracts EVM bytecode and employs static reachability analysis with Horn clauses (logics) for vulnerability detection.

How scanners work – Symbolic Execution

Symbolic Execution includes the execution of the specific source code or compiled code, including dynamic analysis.

- **Oyente** is a symbolic execution tool created for detecting possible security flaws in the source code and bytecode of Ethereum smart contracts. Oyente is also **outdated** and no longer actively maintained. **Artemis** and **Osiris** are its extensions.
- **Securify2** performs in-depth analysis of EVM bytecode, examining it against **violation patterns**, which encompass the necessary conditions for ensuring the smart contract's **adherence to specific security requirements.**
- **Mythril** is a EVM bytecode analysis tool that combines symbolic execution, SMT (satisfy formula) solving, and **taint analysis** techniques to detect a wide range of vulnerabilities.
- **TeEther** does not primarily focus on general vulnerability detection but rather the **creation of practical exploits**. To achieve this, it specifically targets four low-level instructions closely tied to value transfer: (1) Call, (2) Selfdestruct, (3) Callcode, and (4) Delegatecall.

How scanners work – Fuzzing

Fuzzing involves the guided but random selection of input data.

- **ConFuzzius** is a fuzzer that applies **constraint resolving** and data dependency analysis to detect smart contract vulnerabilities.
- **Smartian** is a mutation-based fuzzer that integrates static and dynamic analyses to drive **input mutation** leveraging dynamic dataflow analysis, incorporating **bug oracles** into its testing process.
- **sFuzz** is a framework developed based on AFL used for C/C++ programs and introduces a novel metric to target **hard-to-cover branches** specifically.

How scanners work – Machine Learning

Machine Learning that exploits Graph Neural Networks (GNNs) and Recurrent Neural Networks (RNNs), for vulnerability detection purposes.

- **GNNSCVulDetector** is designed to detect vulnerabilities in smart contracts by utilizing a GNN. It analyzes the syntactic and semantic structures of the smart contract, which are represented as a **graph**. This graph is processed using a **degree-free graph convolutional neural network for classification**. It can identify also infinite loops.

Dataset for analysis #1

The build of dataset followed several phases:

- **Google BigQuery**
- **EtherScan** and **IPFS** (via CBOR decoder, accessing metadata and then link to IPFS)

The retrieved contracts are stored across various blockchains, primarily encompassing the **Ethereum Mainnet** (i.e., the primary production blockchain) and the **testnets (reason the numbers!)**, namely Goerli, Rinkeby, Ropsten, and Kovan.

The source codes has been preprocessed to remove duplications, using the Solidity compiler to generate an **Abstract Syntax Tree** (AST) from the source code file to eliminate noise.

Dataset for analysis #2

Four datasets have been generated:

- **Source Code Dataset** (SCD), which comprises a collection of smart contract source codes
- **Bytecode Dataset** (BCD), consisting of a hexadecimal representation of compiled bytecodes
- **Reentrancy Ground Truth** (RGT), specifically focused on identifying and labeling source code operations that are vulnerable to reentrancy. *Will not be treated in this seminar.*
- **Audits Ground Truth** (AGT), which includes labeled data obtained from security audits conducted on smart contracts by expert people. *Will not be treated in this seminar.*

The method

“We use the vulnerability scanners **as-is** and do not optimize them per smart contract. For instance, if the tool supports only a limited range of compiler versions, we don’t attempt to enhance it to other versions.”
→ so tools are not necessarily used at their **maximum potential**

“We also consider vulnerability scanners that **time out on a smart contract as non-vulnerable** since the outcome is the same for a developer – no vulnerability is found or reported.”

Later on within a robustness analysis we see the completion rate of the different scanners.

The method #2

Source code analysis

“We employed **13** tools to identify over **200** vulnerability types. For comparability and simplicity, our focus was on **eight** types detectable by at least **three** scanners. Results for other types are omitted due to space limitations.”

Bytecode analysis

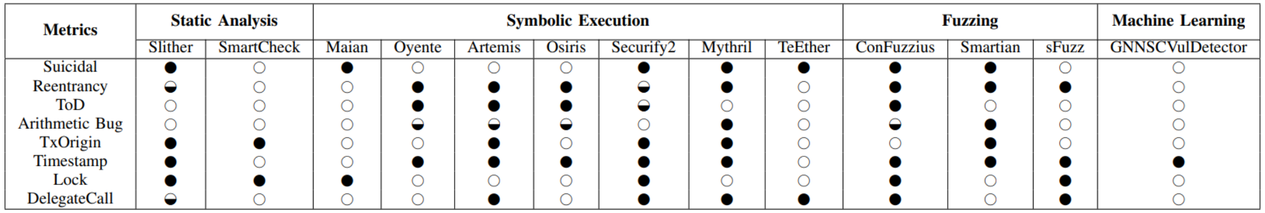
“We analyzed the bytecode-based datasets using four tools. Since the tools’ **density is lower** than that of those operating on source codes, we provided our analysis of tools based on four vulnerability types present across the tools.”

Analysis on SCD

Evaluated all 77,219 smart contracts available in SCD using 13 source code-based tools. Please read the paper to delve in those results :)

- **Suicide**: 7 tools, max agree 6/7, agreed 70 contracts by 3 tools (same result)
- **Reentrancy**: 9 tools, max agree 3/9, agreed 106 contracts by 3 tools
- **ToD**: 5 tools, max agree 2/5 → Osiris & Artemis both based on Oyente
- **Arithmetic bugs**: 6 tools, max agree 3/6, agree 2,622 contracts by 3 tools (again Osiris → Oyente)
- **TxOrigin**: 5 tools, max agree 5/5, agree usually between 2 tools
- **Time Dependency:** 10 tools, max agree 8/10, agree 7 contracts by 8 tools
- **Locked Ether**: 6 tools, max agree 2/6, agree 20 contracts by 2 tools
- **DelegateCall**: 6 tools, max agree 3/6, agree usually between 3 tools

Analysis on SCD



Metrics	Static Analysis		Symbolic Execution							Fuzzing			Machine Learning
	Slither	SmartCheck	Maian	Oyente	Artemis	Osiris	Securify2	Mythril	TeEther	ConFuzzius	Smartian	sFuzz	GNNSCVulDetector
Suicidal	●	○	●	○	○	○	●	●	●	●	●	○	○
Reentrancy	◒	○	○	●	●	●	◒	●	○	●	●	●	○
ToD	○	○	○	●	●	●	◒	○	○	●	○	○	○
Arithmetic Bug	○	○	○	◒	◒	◒	○	●	○	◒	●	○	○
TxOrigin	●	●	○	○	●	○	●	●	○	○	●	○	○
Timestamp	●	○	○	●	●	●	●	●	○	●	●	●	●
Lock	●	●	●	○	○	○	●	○	○	●	○	●	○
DelegateCall	◒	○	○	○	●	○	●	●	●	●	○	●	○

TABLE 1: Overview of vulnerability scanners with detectable vulnerability types on source code.

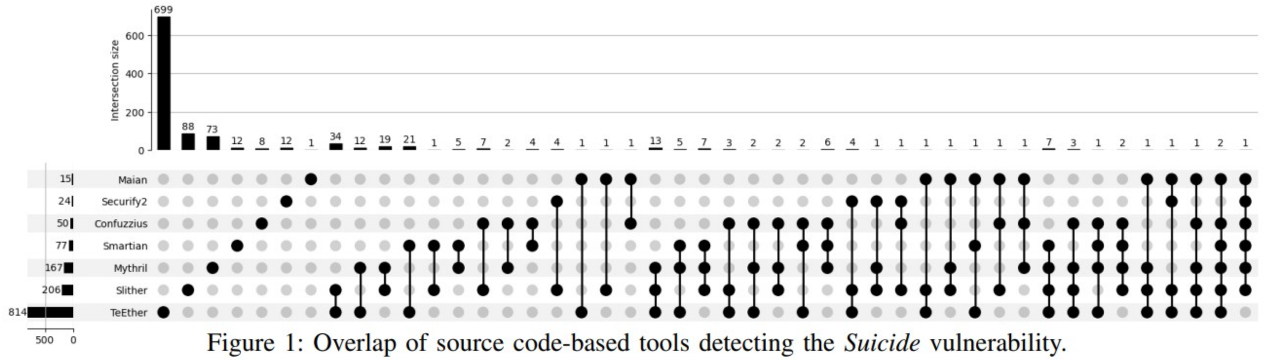


Figure 1: Overlap of source code-based tools detecting the *Suicide* vulnerability.

Figure1. On the left side, the total number of vulnerable samples found by each tool.
On the top, we show the total number of samples that overlap among the tools. If there is a single dot, it signifies that these samples do not overlap with any other tool. Conversely, when a column contains multiple dots, it signifies that the respective tools agree in their analysis of these specific samples.

Still not sure about "semi-points", maybe something regarding false positives…

Positive comment: even if "the best" scanner could be deduced, the paper never tell that, but focuses on on the overlapping

Analysis on BCD

Evaluated all 4,062,844 smart contracts available in BCD using 4 byte code-based tools.

- **Suicide**: 3 tools, max agree 2/3, agree 14,991 contracts by 2 (*)
- **Reentrancy**: 4 tools, max agree 3/4, low agreement
- **TxOrigin**: 2 tools, max agree 2/2, agree 2,372 by 2 tools
- **Time Dependency**: 2 tools, max agree 2/2, agree 8,735 contracts by 2 tools… still those disagree for the 90%

(*) that case has to take in account Vandal scanner, that reports also “susceptible” contracts, identifying 1,771,577 instances as vulnerable

Source-code VS Byte-code

Considering **Maian** scanner, that allows both source-code and bytecode as input, we observe something interesting:

Maian flags:

- Suicidal: 4 smart contracts in SCD and 93 in BCD, overlapping in **3** cases
- Locked Ether: 80 contracts in SCD and 3,223 in BCD, with **66** common detections.

Maian shows a higher positive rate in bytecode analysis.

We don’t know if **all** codes analysed in one dataset were present **also in the other one**...

If yes (hopefully), this means that same tools **can disagree within themselves**!

Robustness analysis

The following plot shows the percentage of contracts successfully **scanned** by tools, not necessarily meaning that on them tools detected something.

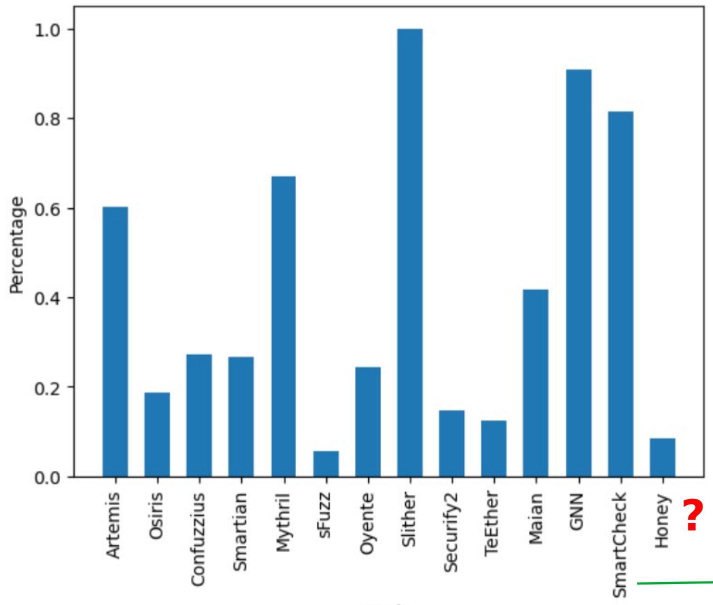


Figure 13: Robustness of Tool Detection

?

...and it was deprecated!

Remember: "[...] We also consider vulnerability scanners that **time out on a smart contract as non-vulnerable**"

That means that potential detection may have been ignored, and by so possible overlappings

Also the paper doesn't specify those timeouts...

Disagreement of tools

The question that may arise is why this big disagreement among tools?

- **Compiler Version changes tools**, particularly those analyzing source code, are often designed for specific versions of the Solidity compiler; as the compiler evolves, these tools may not be updated.
- **Evolution of Smart Contract** Programming, in which the practices in smart contracts can shift over time, in this context also influenced by **gas limit alterations** for function calls, that have led to programmers being advised against their use.
- **Attack Variations** with built-in defenses within the attack

Do we have a solution?

Can this disagreement be solved?

Conclusion from the paper

Using a **combination** of tools to identify vulnerabilities in a smart contract might seem like a promising strategy.

The differing results from various tools suggest that their combined use could **potentially offer** a more thorough analysis. However, this method has its drawbacks, primarily the increase of **false positives**.

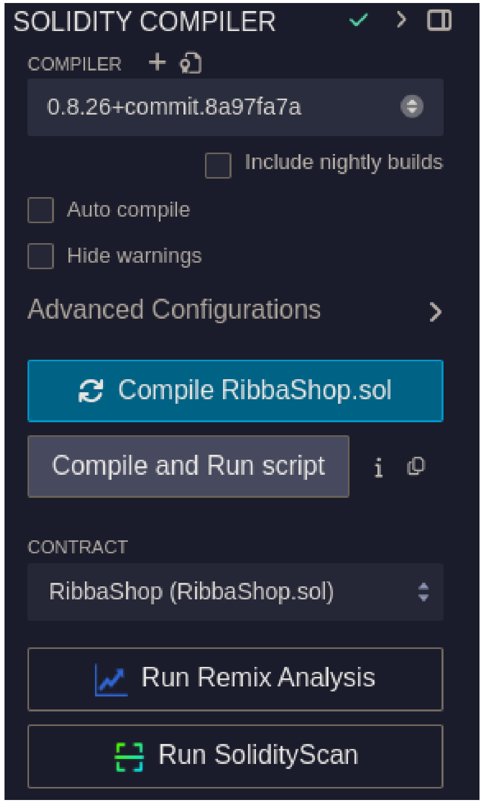
Moreover, the challenge lies in selecting an appropriate **mix of tools**.

There’s **no universally** effective combination that guarantees the successful detection of all vulnerabilities.

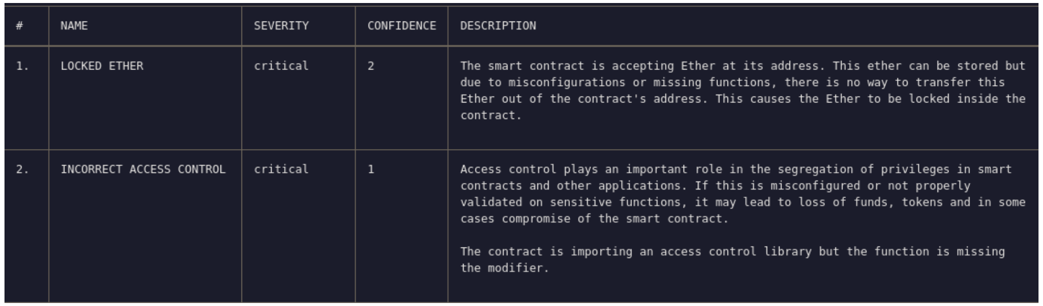
Remix scanners

In this section I investigate a little more on the the compilation section of Remix.

- **Solidity Scan**: uses a third party service cloud&AI-based static-analyzer (still in BETA) that shows on terminal bug detection and suggestion on security and gas consuming
- **Remix Analysis**: allow to use static analysis scanner as Remix own scanner, Solhint and Slither

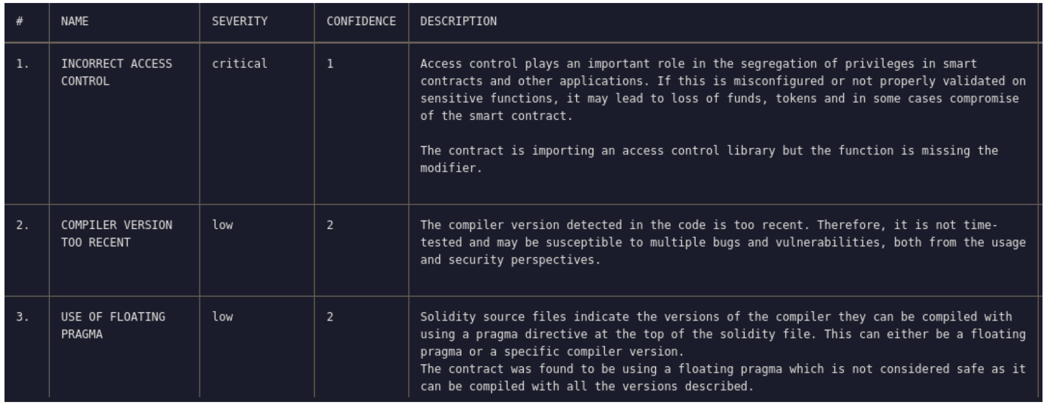


Solidity Scan



#	NAME	SEVERITY	CONFIDENCE	DESCRIPTION
1.	LOCKED ETHER	critical	2	The smart contract is accepting Ether at its address. This ether can be stored but due to misconfigurations or missing functions, there is no way to transfer this Ether out of the contract's address. This causes the Ether to be locked inside the contract.
2.	INCORRECT ACCESS CONTROL	critical	1	Access control plays an important role in the segregation of privileges in smart contracts and other applications. If this is misconfigured or not properly validated on sensitive functions, it may lead to loss of funds, tokens and in some cases compromise of the smart contract. The contract is importing an access control library but the function is missing the modifier.

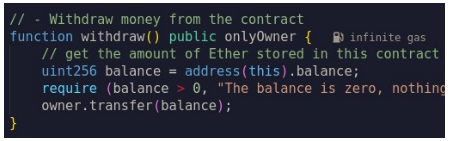
Ribba Shop



#	NAME	SEVERITY	CONFIDENCE	DESCRIPTION
1.	INCORRECT ACCESS CONTROL	critical	1	Access control plays an important role in the segregation of privileges in smart contracts and other applications. If this is misconfigured or not properly validated on sensitive functions, it may lead to loss of funds, tokens and in some cases compromise of the smart contract. The contract is importing an access control library but the function is missing the modifier.
2.	COMPILER VERSION TOO RECENT	low	2	The compiler version detected in the code is too recent. Therefore, it is not time-tested and may be susceptible to multiple bugs and vulnerabilities, both from the usage and security perspectives.
3.	USE OF FLOATING PRAGMA	low	2	Solidity source files indicate the versions of the compiler they can be compiled with using a pragma directive at the top of the solidity file. This can either be a floating pragma or a specific compiler version. The contract was found to be using a floating pragma which is not considered safe as it can be compiled with all the versions described.

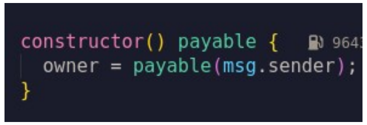
Wish of Day

LOCKED ETHER shows that the Ether can't be transferred to other addresses and just flows into the address of the contract.
This is not true, because there is a transfer function to the owner of the contract, so the scanner has hallucinated and this is a false positive.

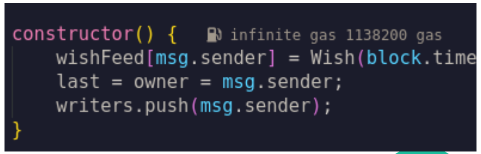


```
// - Withdraw money from the contract
function withdraw() public onlyOwner {    infinite gas
    // get the amount of Ether stored in this contract
    uint256 balance = address(this).balance;
    require (balance > 0, "The balance is zero, nothing
    owner.transfer(balance);
}
```

The common error is an INCORRECT ACCESS CONTROL, that may be caused by a library that is imported (or is part of the compiler), that contains the proper modifier function to handle owner permissions.
However this kind of modifier is already implemented in the contract, but maybe is not considered a best practice by the scanner.



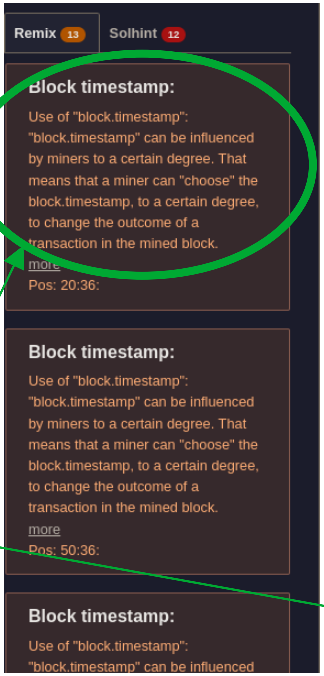
```
constructor() payable {    964
  owner = payable(msg.sender);
}
```



```
constructor() {    infinite gas 1138200 gas
    wishFeed[msg.sender] = Wish(block.time
    last = owner = msg.sender;
    writers.push(msg.sender);
}
```

Remix & Solhint scanners

Wish of Day



Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
more
Pos: 20:36:

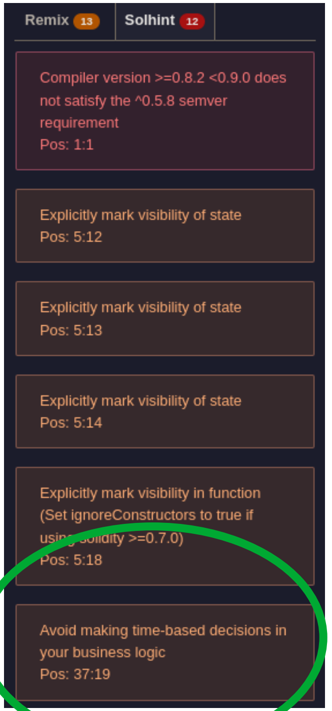
Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
more
Pos: 50:36:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced

Overlap!



Compiler version >=0.8.2 <0.9.0 does not satisfy the ^0.5.8 semver requirement
Pos: 1:1

Explicitly mark visibility of state
Pos: 5:12

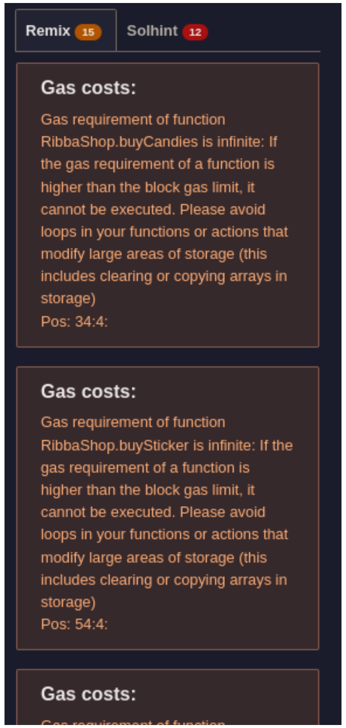
Explicitly mark visibility of state
Pos: 5:13

Explicitly mark visibility of state
Pos: 5:14

Explicitly mark visibility in function (Set ignoreConstructors to true if using solidity >=0.7.0)
Pos: 5:18

Avoid making time-based decisions in your business logic
Pos: 37:19

Ribba Shop



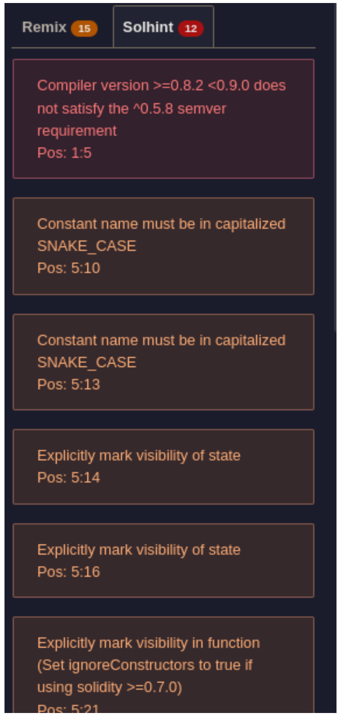
Gas costs:

Gas requirement of function RibbaShop.buyCandies is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 34:4:

Gas costs:

Gas requirement of function RibbaShop.buySticker is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 54:4:

Gas costs:



Compiler version >=0.8.2 <0.9.0 does not satisfy the ^0.5.8 semver requirement
Pos: 1:5

Constant name must be in capitalized SNAKE_CASE
Pos: 5:10

Constant name must be in capitalized SNAKE_CASE
Pos: 5:13

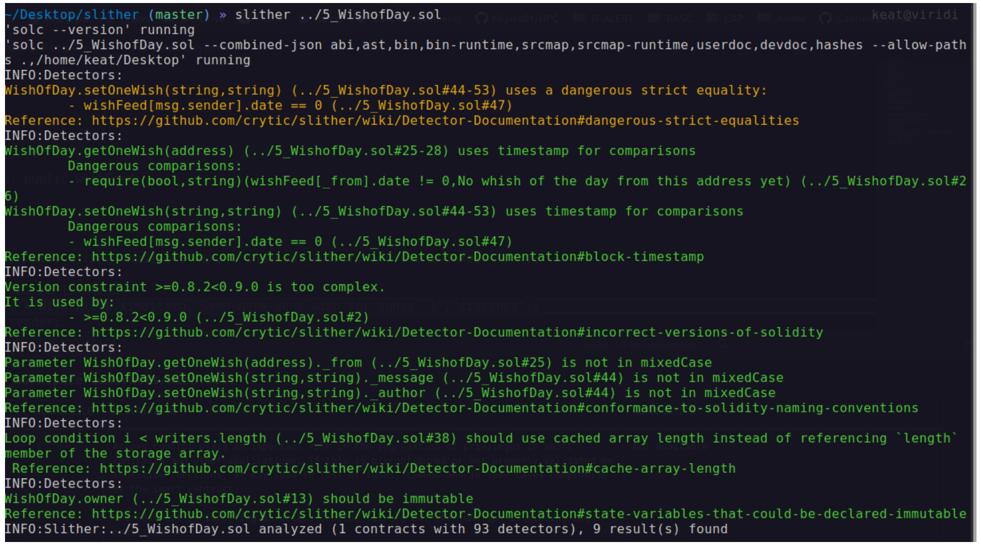
Explicitly mark visibility of state
Pos: 5:14

Explicitly mark visibility of state
Pos: 5:16

Explicitly mark visibility in function (Set ignoreConstructors to true if using solidity >=0.7.0)

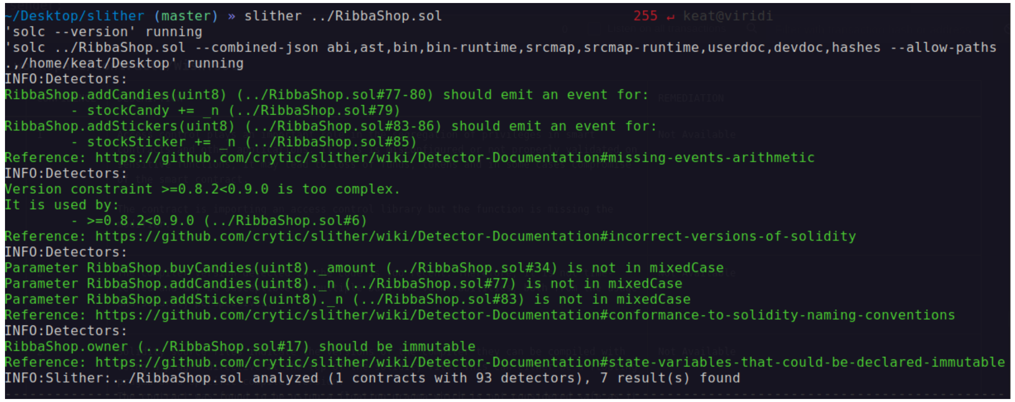
Local scanning - Slither

I run also by myself the **Mythral** and **Slither** static-code based scanner on Wish and Shop contracts.
We can see the overlapping with the previous scanners: even if this may resemble that all our tools are performing great, we also need to take in account the **simplicity** of those contracts.



```
~/Desktop/slither (master) » slither ../5_WishofDay.sol                                           keat@viridi
'solc --version' running
'solc ../5_WishofDay.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes --allow-path
s .,/home/keat/Desktop' running
INFO:Detectors:
WishOfDay.setOneWish(string,string) (../5_WishofDay.sol#44-53) uses a dangerous strict equality:
        - wishFeed[msg.sender].date == 0 (../5_WishofDay.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
WishOfDay.getOneWish(address) (../5_WishofDay.sol#25-28) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(wishFeed[_from].date != 0,No whish of the day from this address yet) (../5_WishofDay.sol#2
6)
WishOfDay.setOneWish(string,string) (../5_WishofDay.sol#44-53) uses timestamp for comparisons
        Dangerous comparisons:
        - wishFeed[msg.sender].date == 0 (../5_WishofDay.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Version constraint >=0.8.2<0.9.0 is too complex.
It is used by:
        - >=0.8.2<0.9.0 (../5_WishofDay.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter WishOfDay.getOneWish(address)._from (../5_WishofDay.sol#25) is not in mixedCase
Parameter WishOfDay.setOneWish(string,string)._message (../5_WishofDay.sol#44) is not in mixedCase
Parameter WishOfDay.setOneWish(string,string)._author (../5_WishofDay.sol#44) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Loop condition i < writers.length (../5_WishofDay.sol#38) should use cached array length instead of referencing `length`
member of the storage array.
 Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cache-array-length
INFO:Detectors:
WishOfDay.owner (../5_WishofDay.sol#13) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Slither:../5_WishofDay.sol analyzed (1 contracts with 93 detectors), 9 result(s) found
```

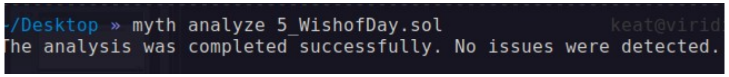
Orange one is a false positive, doesn't apply to our contract: no user input can change that



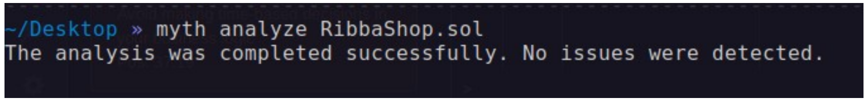
```
~/Desktop/slither (master) » slither ../RibbaShop.sol                              255 ↵ keat@viridi
'solc --version' running
'solc ../RibbaShop.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes --allow-paths
.,/home/keat/Desktop' running
INFO:Detectors:
RibbaShop.addCandies(uint8) (../RibbaShop.sol#77-80) should emit an event for:
        - stockCandy += _n (../RibbaShop.sol#79)
RibbaShop.addStickers(uint8) (../RibbaShop.sol#83-86) should emit an event for:
        - stockSticker += _n (../RibbaShop.sol#85)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic
INFO:Detectors:
Version constraint >=0.8.2<0.9.0 is too complex.
It is used by:
        - >=0.8.2<0.9.0 (../RibbaShop.sol#6)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter RibbaShop.buyCandies(uint8)._amount (../RibbaShop.sol#34) is not in mixedCase
Parameter RibbaShop.addCandies(uint8)._n (../RibbaShop.sol#77) is not in mixedCase
Parameter RibbaShop.addStickers(uint8)._n (../RibbaShop.sol#83) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
RibbaShop.owner (../RibbaShop.sol#17) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Slither:../RibbaShop.sol analyzed (1 contracts with 93 detectors), 7 result(s) found
```

… no vulnerabilities found on Ribba Shop :(

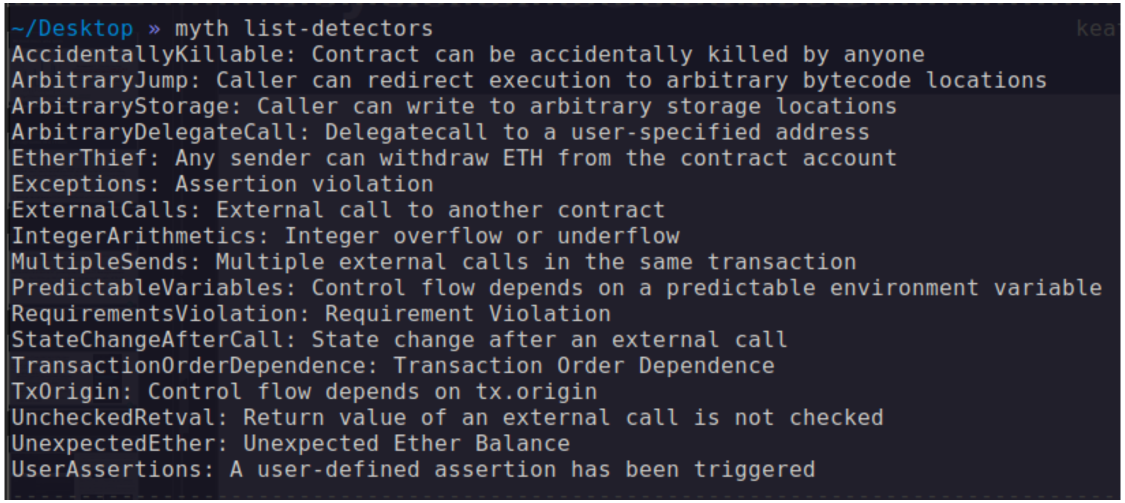
Local scanning - Mythril



```
~/Desktop » myth analyze 5_WishofDay.sol                keat@virid
The analysis was completed successfully. No issues were detected.
```



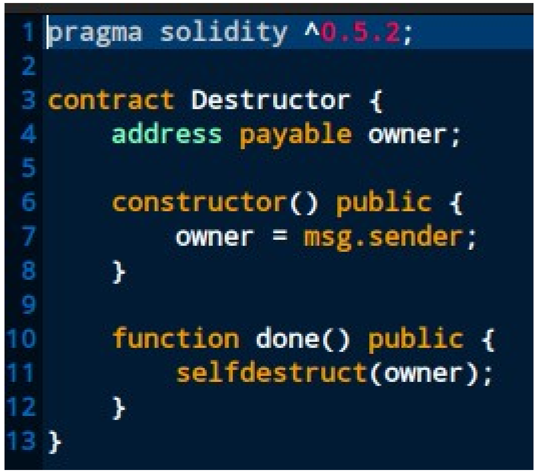
```
~/Desktop » myth analyze RibbaShop.sol
The analysis was completed successfully. No issues were detected.
```



```
~/Desktop » myth list-detectors                                              kea
AccidentallyKillable: Contract can be accidentally killed by anyone
ArbitraryJump: Caller can redirect execution to arbitrary bytecode locations
ArbitraryStorage: Caller can write to arbitrary storage locations
ArbitraryDelegateCall: Delegatecall to a user-specified address
EtherThief: Any sender can withdraw ETH from the contract account
Exceptions: Assertion violation
ExternalCalls: External call to another contract
IntegerArithmetics: Integer overflow or underflow
MultipleSends: Multiple external calls in the same transaction
PredictableVariables: Control flow depends on a predictable environment variable
RequirementsViolation: Requirement Violation
StateChangeAfterCall: State change after an external call
TransactionOrderDependence: Transaction Order Dependence
TxOrigin: Control flow depends on tx.origin
UncheckedRetval: Return value of an external call is not checked
UnexpectedEther: Unexpected Ether Balance
UserAssertions: A user-defined assertion has been triggered
```

Again, the simplicity of the contracts may not trigger those detectors.

Mythral: Triggering detectors

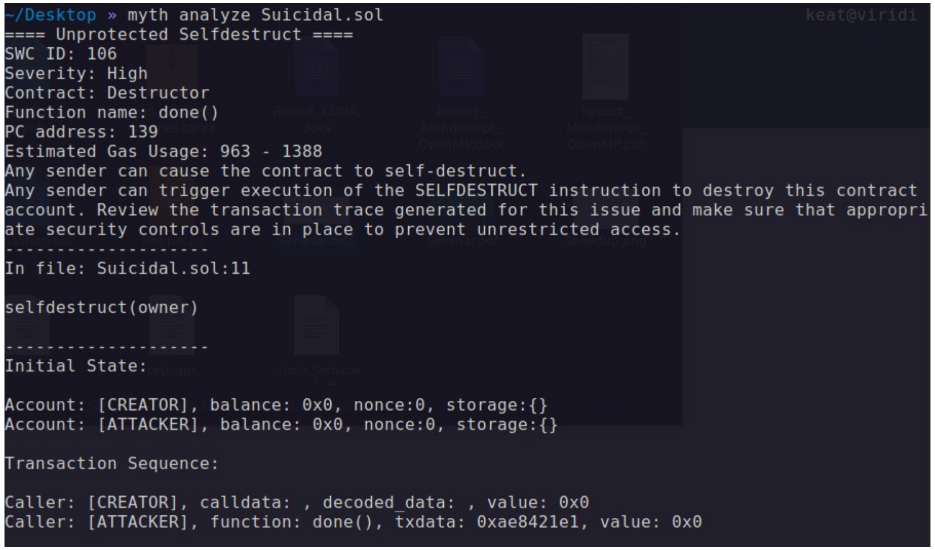


```solidity
pragma solidity ^0.5.2;

contract Destructor {
    address payable owner;

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

    function done() public {
        selfdestruct(owner);
    }
}
```



```
~/Desktop » myth analyze Suicidal.sol                                                    keat@viridi
==== Unprotected Selfdestruct ====
SWC ID: 106
Severity: High
Contract: Destructor
Function name: done()
PC address: 139
Estimated Gas Usage: 963 - 1388
Any sender can cause the contract to self-destruct.
Any sender can trigger execution of the SELFDESTRUCT instruction to destroy this contract
account. Review the transaction trace generated for this issue and make sure that appropri
ate security controls are in place to prevent unrestricted access.
--------------------
In file: Suicidal.sol:11

selfdestruct(owner)

--------------------
Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0
Caller: [ATTACKER], function: done(), txdata: 0xae8421e1, value: 0x0
```

While an explicit one about Sucidal vulnerability triggers Mythral

Thank you

Questions?

Bibliography

- **Paper link: https://arxiv.org/pdf/2312.16533**
- **Mythril scanner: https://github.com/Consensys/mythril**
- **Slither scanner: https://github.com/crytic/slither**
- **Remix: https://remix.ethereum.org/**
- **Meme image: https://imageresizer.com/meme-generator/edit/well-yes-but-actually-no**