192.127 Seminar in Software Engineering (Smart Contracts)
SWC-124: Write to Arbitrary Storage Location
Report Exploits

Ivanov, Ivaylo (11777707) & Millauer, Peter (01350868)

WT 2023/24

1 Introduction

1.1 Paper introduction

The aim of this report is to present an analysis of the SWC-124 weakness and its detection. First, we will decribe the setup we did for performing the analysis. Then, in section 2 we will give a brief overview of the weakness and present heuristics for detecting it. Afterwards, in section 3 we will present the results of the security analysis of the assigned contracts[1] and finally, in section 4 we will wrap up, give an overview about what we learned and provide future research questions.

1.2 Setup

We attempted to implement an automatic weakness detection pipeline by using a multitude of tools. The software used includes:

- **solc**: Using **solc-select** we select the correct solc version and compile the associated *.sol file. This way we gather compiler hints and warnings.
- **Mythril**: A tool used for analysis of EVM bytecode, one of the de-facto standards for such work. It is unfortunately not able to detect SWC-124, but there is an existing feature request[2] for support available.
- **teEther**: A tool we found in the last research step of this seminar, teether is a dynamic analysis tool for smart contracts. It is unfortunately hardly documented and has been unmaintained for several years. We were unable to generate usable results with it.
- **Securify v2.0**: Another tool we found that could theoretically detect SWC-124. Unfortunately, it works only with old solidity versions and we were not able to start it.
- **Slither**: A highly useful tool that offers a large static analysis toolkit for solidity, it not only allows the extraction of contract data like storage layouts but also automatic scanning for common weaknesses. Although it did not seem to be able to detect SWC-124, the storage layout functionality was used extensively by our team.

[1] `https://git.logic.at/ethereum/vulnerabilities_23w/-/tree/swc-124?ref_type=heads`
[2] `https://github.com/Consensys/mythril/issues/861`

2 Exploit Creation

2.1 Short recap of weakness definitions

SWC-124 attempts to use the underlying mechanisms that govern dynamic array allocation to deterministically overwrite arbitrary data in smart contracts. An analogous weakness can be created by employing unchecked assembly instructions, although this is a less common attack vector due to its unusual structure.

2.2 Exploit heuristics

The workflow of determining the vulnerability of a given contract is straightforward and follows the general approach of automatic detection mechanisms from our last paper. Since SWC-124 requires the presence of very specific code, it is relatively easy to develop heuristics to exclude non-vulnerable contracts:

1. any contract without dynamic arrays (or mappings with integer keys) or raw assembly including a SSTORE instruction can immediately be considered non-vulnerable;

2. if heuristic 1 does not hold and there is a dynamic array or mapping with an integer key available, we can then apply a second heuristic and namely: checking the solidity compiler version, specified at the top of the contract. Solidity version 0.8.0+ introduced integer under- and overflow protection, which are enabled per default and require extra work to be disabled. As such, we have the following "sub-heuristic":

   (a) if the version of the contract is higher than 0.8.0, we examine whether unchecked arithmetic[3] has been used for modifying the arrays. If this is not the case, which it is not most of the time, we can then determine that the contract is non-vulnerable. Applying this heuristic, we found a contract that could have been vulnerable had it been compiled with a lower solidity version.

The presence of dynamic arrays can be determined using **slither –print variable-order**. A sample output looks as follows:

```
$slither cans.sol --print variable-order

Cans:
+-----------------------------+----------------------------------------------+------+--------+
| Name                        |                                         Type | Slot | Offset |
+-----------------------------+----------------------------------------------+------+--------+
| ERC1155._balances           |               mapping(address => uint32[7]) |    0 |      0 |
| ERC1155.tokens              |                                   uint256[] |    1 |      0 |
| ERC1155._operatorApprovals  | mapping(address => mapping(address => bool)) |    2 |      0 |
| ERC1155._uri                |                                       string |    3 |      0 |
| Ownable._owner              |                                      address |    4 |      0 |
| Functional._reentryKey      |                                         bool |    4 |     20 |
| Cans.START_TIME             |                                      uint256 |    5 |      0 |
| Cans.END_TIME               |                                      uint256 |    6 |      0 |
| Cans.amountClaimed          |                                uint8[9998] |    7 |      0 |
| Cans.CLAIM_ENABLED          |                                         bool |  320 |      0 |
| Cans.soda                   |                                 SODAContract |  320 |      1 |
| Cans.SODA_CONTRACT          |                                      address |  321 |      0 |
| Cans.baseURI                |                                       string |  322 |      0 |
+-----------------------------+----------------------------------------------+------+--------+
```

[3]https://docs.soliditylang.org/en/v0.8.0/control-structures.html#checked-or-unchecked-arithmetic

In this example the **ERC1155.tokens** array is the only potential weakess present. We would then look for the presence of what-where writes to this array in order to confirm the potential presence of SWC-124. What-where writes are of the form

```
someArray[where] = what;
```

and are a necessary code snippet for SWC-124. If such an instruction is present, we then attempt to reverse engineer a sequence of inputs to trigger the exploit, or formulate a reason why we believe this not to be possible.

3 Results

Applying the heuristics, mentioned in the previous section, we gathered the following data about the contracts.

3.1 Vulnerable contracts

Using the heuristics above, we were not able to find a contract that is vulnerable to SWC-124.

3.2 Non-exploitable contracts

Solidity files that contained no contracts, just libraries, that would not introduce SWC-124 to the contracts using them as per the heuristics:

- AuctionLib.sol
- LibRegion.sol
- LToken.sol

Contracts that were discarded due to the heuristics holding:

- DCU.sol
- ERC20_Asset_Pool.sol
- FacelessNFT.sol
- GElasticTokenManager.sol
- GoldToken.sol
- GovernmentAlpha.sol
- HedgeSwap.sol
- HermesImplementation.sol
- IMETACoin223Token_13.sol - had this contract been compiled with solidity under 0.8.0, it would have been vulnerable.
- UniswapV3PoolAdapter.sol
- UserDeposit.sol
- WPCMainnetBridge.sol

4 Discussion

4.1 Conclusions

We have proposed initial heuristics which can show us whether a contract is vulnerable to SWC-124. They are easy to understand and apply even in large contracts. We have demonstrated an example workflow that uses the tool Slither as a data-gathering aid and applies the heuristics. As a result of this workflow, we were unable to find vulnerable specimen from the examples provided.

4.2 Lessons learned: what works, what doesn't

During the analysis that we performed, we were able to learn that none of the state-of-the-art tools have support for SWC-124 as of the time of writing. Because of this, currently the only way of detecting this vulnerability is the application of the heuristics from section 2. The easiest way to protect yourself from this vulnerability is to use solidity 0.8.0+ and its built-in under- and overflow protection whenever using dynamic arrays (and in general), to avoid using assembly for trivial tasks and, if the need arises, to make sure that the assembly calls that modify the storage are formally correct and non-exploitable.

4.3 Open challenges

The initial heuristics are easy to add to a static code analysis tool, such as Slither. Afterwards, it can be used to develop a dataset of vulnerable or non-vulnerable samples, which, alongside with manually verified contracts, can be used to improve or expand the heuristics.