TECHNISCHE
UNIVERSITÄT
WIEN

SWC-124: Write to Arbitrary Storage Location

192.127 Seminar in Software Engineering (Smart Contracts)

Ivaylo Ivanov & Peter Millauer
January 8, 2024

Outline

SWC-124: Weakness Outline

SWC-124 is a weakness that allows attackers to write to places in the storage where they should not be able to. It can be used to gain unauthorized access, overwrite data, steal funds etc.

SWC-124: Weakness Outline

We generally differentiate three types of SWC-124:

- unchecked array write
- incorrect array length check
- unchecked assembly code

Examples follow, use in production at your own risk ;)

Unchecked Array Write

```
1  pragma solidity 0.4.25;
2
3  contract MyContract {
4    uint[] private arr;
5
6    constructor() public {
7      arr = new uint[](0);
8    }
9
10   function write(unit index, uint value) {
11     arr[index] = value;
12   }
13 }
```

Incorrect Array Length Check

```
1  pragma solidity 0.4.25;
2
3  contract MyContract {
4    uint[] private arr;
5
6    constructor() public {
7      arr = new uint[](0);
8    }
9
10   function push(value) {
11     arr[arr.length] = value;
12     arr.length++;
13   }
14
15   function pop() {
16     require(arr.length >= 0);
17     arr.length--;
18   }
19
20   function update(unit index, uint value) {
21     require(index < arr.length);
22     arr[index] = value;
23   }
24 }
```

Unchecked Assembly

```
1  pragma solidity 0.4.25;
2
3  contract MyContract {
4    address private owner;
5    mapping(address => bool) public managers;
6
7    constructor() public {
8      owner = msg.sender;
9      setNextUserRole(msg.sender);
10   }
11
12   function setNextManager(address next) internal {
13     uint256 slot;
14     assembly {
15       slot := managers.slot
16       sstore(slot, next)
17     }
18     bytes32 location = keccak256(abi.encode(160, uint256(slot)));
19     assembly {
20       sstore(location, true)
21     }
22   }
23
24   function registerUser(address user) {
25     require(msg.sender == owner);
26     setNextManager(user);
27   }
28 }
```

SWC-124: Detection Heuristics 1

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

SWC-124: Detection Heuristics 2

If heuristic 1 does not hold, we can then apply a second heuristic: 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.

SWC-124: Detection Heuristics 2.1

If the version of the contract is higher than 0.8.0, we examine whether unchecked arithmetic 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.

Note on assembly

Due to the nature of the examples given, we could not find reliable heuristics for unchecked assembly.

SWC-124: Detection Tools

- existing static analysis tools were useless - most of them had no support for SWC-124
- `solc-select` - for changing solidity compiler versions
- `slither` - for printing contract variable layout

Vulnerable Examples

Using the previously mentioned heuristics, we could not find a vulnerable contract from the dataset.

Future Work

- add heuristics to static analysis tool like `slither` or `mythril`
- develop additional vulnerable and non-vulnerable contracts and test against heuristics
- train a model against the resulting dataset
- fine-tune heuristics

Questions?