

# 192.127 Seminar in Software Engineering (Smart Contracts)

## SWC-124: Write to Arbitrary Storage Location

\*\*\* YOUR NAME AND STUDENT ID \*\*\*

WT 2023/24

## 1 Weakness and consequences

### 1.1 Solidity storage layout

Any contract's storage is a continuous 256-bit address space consisting of 32-bit values. In order to implement dynamically sized data structures like maps and arrays, Solidity distributes their entries in a pseudo-random location. Due to the vast 256-bit range of addresses collisions are statistically extremely improbable and of no practical relevance.

In the case of a dynamic array at variable slot  $p$ , data is written to continuous locations starting at  $keccak(p)$ . The array itself contains the length information.

For maps stored in variable slot  $p$  the data for index  $k$  can be found at  $keccak(k.p)$  where  $.$  is the concatenation operator.

### 1.2 The Weakness

Any unchecked array write is potentially dangerous, as the storage-location of all variables is publicly known and an unconstrained array index can be reverse engineered to target them.

---

**Algorithm 1:** A completely unchecked array write

---

```
1  pragma solidity 0.4.25;
2
3  contract MyContract {
4      address private owner;
5      uint[] private arr;
6
7      constructor() public {
8          arr = new uint[](0);
9          owner = msg.sender;
10     }
11
12     function write(uint index, uint value) {
13         arr[index] = value;
14     }
15 }
16
```

---

In the following example the *pop* function incorrectly checks for an array *length*  $\geq 0$ , thereby allowing the value to underflow when called with an empty array. Once this weakness is exploited *update* in Algorithm 2 behaves just like *write* did in Algorithm 1.

---

**Algorithm 2:** An incorrectly managed array length

---

```
1  pragma solidity 0.4.25;
2
3  contract MyContract {
4      address private owner;
5      uint[] private arr;
6
7      constructor() public {
8          arr = new uint [] (0);
9          owner = msg.sender;
10     }
11
12     function push(value) {
13         arr[arr.length] = value;
14         arr.length++;
15     }
16
17     function pop() {
18         require(arr.length >= 0);
19         arr.length--;
20     }
21
22     function update(uint index, uint value) {
23         require(index < arr.length);
24         arr[index] = value;
25     }
26 }
27
```

## 2 Vulnerable contracts in literature

collect vulnerable contracts used by different papers to motivate/illustrate the weakness

## 3 Code properties and automatic detection

summarize the code properties that tools are looking for so that they can detect the weakness

## 4 Exploit sketch

sketch ways to potentially exploit the different variants of the weakness.

[4] [3] [2] [1]

## References

- [1] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239, 2021.
- [2] Johannes Krupp and Christian Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, Baltimore, MD, August 2018. USENIX Association.
- [3] Siddhasagar Pani, Harshita Vani Nallagonda, Vigneswaran, Raveendra Kumar Medicherla, and Rajan M. Smartfuzzdrivergen: Smart contract fuzzing automation for golang. In *Proceedings of the 16th*

*Innovations in Software Engineering Conference*, ISEC '23, New York, NY, USA, 2023. Association for Computing Machinery.

- [4] Petar Tsankov, Andrei Dan, Dana Drachler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.