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

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

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 little practical relevance in safely implemented contracts.

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 as an $uint256$ value. Even enormous arrays are unlikely to produce collisions due to the vast address space, although an improperly managed array may store data to an unbounded user-controlled offset, thereby allowing arbitrary overwriting of data.

For maps stored in variable slot p the data for index k can be found at $keccak(k.p)$ where . is the concatenation operator. This is a statistically safe approach, as the chance of intentionally finding a value for $keccak(k.p)$ s.t. for a known stored variable x, $keccak(k.p) == storage\_address(x)$ is about one in 2^{256} and $keccak$ is believed to be a cryptographically secure hash function.

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. This can be achieved by using the known array storage location p, target-variable x, and computing the offset-value o such that $keccac(p) + o == storage\_address(x)$.

A trivial example of such a vulnerable write operation is shown in Algorithm 1.

Algorithm 1: A completely unchecked array write

```
pragma solidity 0.4.25;

contract MyContract {
  address private owner;
  uint[] private arr;

  constructor() public {
    arr = new uint[](0);
    owner = msg.sender;
  }

  function write(unit index, uint value) {
    arr[index] = value;
  }
}
```

In the following example (Algorithm 2) the *pop* function incorrectly checks for an array *length* >= 0, thereby allowing the *length* value to underflow when called with an empty array. Once this weakness is triggered, *update* in Algorithm 2 behaves just like *write* did in Algorithm 2.

Algorithm 2: An incorrectly managed array length

```
pragma solidity 0.4.25;

contract MyContract {
  address private owner;
  uint[] private arr;

  constructor() public {
    arr = new uint[](0);
    owner = msg.sender;
  }

  function push(value) {
    arr[arr.length] = value;
    arr.length++;
  }

  function pop() {
    require(arr.length >= 0);
    arr.length--;
  }

  function update(unit index, uint value) {
    require(index < arr.length);
    arr[index] = value;
  }
}
```

Another weakness that allows arbitrary storage access is unchecked assembly code. Assembly is a powerful tool that allows the developers to get as close to the EVM as they can, but it may also be very dangerous when not tested correctly. As per the documentation[1]: *"this [inline assembly] bypasses important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it."* When given access to such lowlevel structures, a programmer can built-in not only weaknesses similar to the ones described previously, but also others, such as overwriting map locations, contract variables

[1] https://docs.soliditylang.org/en/latest/assembly.html

etc.

An example for such a weakness is given in Algorithm 3.

Algorithm 3: An unchecked assembly write to mapping

```
pragma solidity 0.4.25;

contract MyContract {
  address private owner;
  mapping(address => bool) public managers;

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

  function setNextManager(address next) internal {
    uint256 slot;
    assembly {
      slot := managers.slot
      sstore(slot, next)
    }

    bytes32 location = keccak256(abi.encode(160, uint256(slot)));
    assembly {
      sstore(location, true)
    }
  }

  function registerUser(address user) {
    require(msg.sender == owner);
    setNextManager(user);
  }

  function cashout() {
    require(managers[msg.sender]);
    address payable manager = msg.sender;
    manager.transfer(address(this).balance);
  }
}

```

The contract has a manager mapping, which should be used as a stack. The developer has added the `setNextManager` function, which should set the top of the stack to the latest user as a manager. The issue is that the function is implemented in such a way, that the stack would not grow, but the first element would always be overwritten - this arises from the fact that the memory slot of the managers mapping does not point to the memory address on the top of the stack, but instead to the base of it. The function is then using this slot address directly, without calculating any offset, overwriting the base of the stack.

2 Vulnerable contracts in literature

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

3 Code properties and automatic detection

Automatic detection tools can be broadly categorized into ones employing static analysis and those who use fuzzing, i.e. application of semi-random inputs. Notable static analysis tools include Securify [5] and teEther [3] which both function in a similar manner:

Initially, the given EVM byte-code is disassembled into a control-flow-graph (CFG). In the second step, the tools identify potentially risky instructions. In the case of arbitrary writes, the instruction of note is $sstore(k, v)$ where both k and v are input-controlled. The tools differ in the way they identify whether or not the values are input-controlled.

In the case of Securify [5], the CFG is translated into what the authors call "semantic facts" to which an elaborate set of so-called security patterns is applied. These patterns consist of building blocks in the form of predicates, which allows the tool to simply generate output based on the (transitively) matched patterns.

teEther [3] employs a similar approach, but instead the authors opt to build a graph of dependent variables. If the graph arrives at a $sstore(k, v)$ instruction and a path can be found leading to user-controlled inputs, the tool infers a set of constraints which are then used to automatically generate an exploit.

The fuzz-driven approach to vulnerability detection is more abstract, as general-purpose fuzzing tools generally don't have knowledge of the analysed program. For the tool SmartFuzzDriverGenerator [4], a multitude of these fuzzing libraries can be used. The problem at hand is, however, that the technique cannot interface with a smart contract out of the box. The "glue" between fuzzer and program is called a driver, hence the name of "driver-generator".

SmartFuzzDriverGenerator aims to automatically generate such a driver by

The Smartian tool [1] attempts to find a middle-ground between static and dynamic analysis by first transforming the EVM bytecode into control-flow facts. Based on this information, a set of seed-inputs is generated that are expected to have a high probability of yielding useable results. Should no exploit be found, the seed-inputs are then mutated in order to yield a higher code coverage.

4 Exploit sketch

[2]

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] doughoyte. Merdetoken: It's some hot shit. `https://github.com/Arachnid/uscc/tree/master/submissions-2017/doughoyte` [Accessed: Oct. 27th 2023].

[3] 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.

[4] 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.

[5] Petar Tsankov, Andrei Dan, Dana Drachsler-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.