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

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

WT 2023/24

Abstract

This paper outlines different forms of the common smart contract weakness with the SWC number 124, commonly referred to as "Write to Arbitrary Storage Location". While this paper focuses on applications within the context of Ethereum's EVM and higher-level language Solidity, we will also briefly touch on other research that deals with the Hyperledger Fabric environment. We will begin with a gentle introduction to the Solidity storage layout design that allows this weakness to occur, followed by common forms of exploit, alongside their associated consequences. Finally, we will outline the code characteristics that are detectable by automated tools as well as an exploit sketch.

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 used 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 low-level instructions, a programmer can construct not only weaknesses similar to the ones described previously, but also others, such as overwriting map locations, contract variables etc.

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

[1]`https://docs.soliditylang.org/en/latest/assembly.html`, accessed: Oct. 30th 2023

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 is intended to 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 does not grow, but the first element is always 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. If social engineeering is applied, an attacker can persuade the owner to set them as a manager, which would result in the weakness being exploited directly and the owner giving up their own management rights.

1.3 Consequences

The consequences of exploiting an arbitrary storage access weakness can be of different types and severity. An attacker may gain read-write access to private contract data, which should only be accessible to owners, maintainers etc. They may also exploit the contract to circumvent authorization checks and drain the contract funds. According to Li Duan et al. [3], an attacker may also be able to destroy the contract storage structure and thus cause unexpected program flow, abnormal function execution or contract freeze.

1.4 Similar yet safe code example

Using dynamic arrays is naturally not inherently dangerous, as long as they're used properly. The following version of Algorithm 2 correctly checks for array length, and thereby prevents the integer underflow of the length value. This code example is not vulnerable to the techniques shown in this paper.

Algorithm 4: Correctly 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;
  }
}
```

2 Vulnerable contracts in literature

One example for vulnerable contracts, which is similar to Algorithm 2, is mentioned in the paper by Li Duan et al. [3]:

Algorithm 5: Arbitrary write as per Li Duan et al.

```
function PopBonusCode() public {
  require(0 <= bonusCodes.length);
  bonusCodes.length--;
}

function UpdateBonusCodeAt(uint idx, uint c) public {
  require(idx < bonusCodes.length);
  bonusCodes[idx] = c;
}
```

We will not go into a detailed explanation, as we already did this in the previous section. A more sophisticated example is presented in the paper by Sukrit Kalra et al. [4]:

Algorithm 6: Arbitrary read as per Sukrit Kalra et al.

```
    uint payout = balance/participants.length;
    for (var i = 0; i < participants.length; i++)
      participants[i].send(payout);
```

The vulnerability here is an integer overflow - as the variable `i` is dynamically typed, it will get the smallest possible type that will be able to hold the value 0 - that being `uint8`, which is able to hold positive integers up to 255.

Because of this, if the length of the `participants` arrays is greater than 255, the integer overflows on the 256th iteration and instead of moving on to `participants[255]`, it reverts back to the first element in the array. As a result, the first 255 participants will split all the balance of the contract, whereas the rest will get nothing.

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 [7] and teEther [5] 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 [7], 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 [5] 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 [6], a multitude of these fuzzing libraries can be used, although its application is limited to the Hyperledger Fabric permissioned blockchain. The problem at hand is, 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 inferring the available APIs from the bytecode. There are multiple approaches to decide the order of available fuzzing steps, including a heuristic based on code complexity (i.e. nested conditions, loops, array operations, etc.), random sequences, and user-generated strategies.

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

An exploitation sketch to Algorithm 2 and to Algorithm 5 is available from Doughoyte [2].

Checkpoint A We assume that the following events have occurred:

(a) the contract MerdeToken[2] has been created;

(b) the investor has set a withdrawal limit of 1 ether, which only they can change;

(c) an investor has invested 50 ETH;

(d) the owner is malicious.

At this point, an example storage layout as per Doughoyte would be:

Algorithm 7: Exploit - Memory at Checkpoint A

```
    "storage": {
        // The address of the contract owner:
        "0000000000000000000000000000000000000000000000000000000000000000": "94b898c1a30adcff67208fd79b9e5a4d339f3cc6d2",
        // The address of the trusted third party:
        "0000000000000000000000000000000000000000000000000000000000000001": "948bc7317ad44d6f34f0f0b6e3c8c7bf739ba666fa",
        // The amount deposited (50 ETH):
        "0000000000000000000000000000000000000000000000000000000000000003": "8902b5e3af16b1880000",
        // The withdrawal limit (1 ETH):
        "0000000000000000000000000000000000000000000000000000000000000004": "880de0b6b3a7640000",
        // the legth of the array would normaly stay here, if it was not zero at init time
        // balanceOf[investorAddress] (50 MDT):
        "dd87d7653af8fba540ea9ebd2d914ba190d975fcfa4d8d2927126a5decdbff9e": "8902b5e3af16b1880000"
    }

```

Checkpoint B Afterwards, the malicious owner calls the vulnerable function `popBonusCode()` and the length of the array is set to the max value. This happened, because prior to the underflow, the array length was zero and, to save space, it was omitted from the memory:

[2]`https://github.com/Arachnid/uscc/blob/master/submissions-2017/doughoyte/MerdeToken.sol`, accessed: Oct. 30th 2023

Algorithm 8: Exploit - Memory at Checkpoint B

```
"storage": {
    "0000000000000000000000000000000000000000000000000000000000000000": "94b898c1a30adcff67208fd79b9e5a4d339f3cc6d2",
    "0000000000000000000000000000000000000000000000000000000000000001": "948bc7317ad44d6f34f0f0b6e3c8c7bf739ba666fa",
    "0000000000000000000000000000000000000000000000000000000000000003": "8902b5e3af16b1880000",
    "0000000000000000000000000000000000000000000000000000000000000004": "880de0b6b3a7640000",
    // The array length has underflowed:
    "0000000000000000000000000000000000000000000000000000000000000005": "a0ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
    "dd87d7653af8fba540ea9ebd2d914ba190d975fcfa4d8d2927126a5decdbff9e": "8902b5e3af16b1880000"
}
```

Increasing the length of the array to the maximum allowed by `uint256` was important, as this will now allow the owner to pass the requirement set in `modifyBonusCode` and still use the function for storage modification.

Checkpoint C The owner is then able to use `modifyBonusCode` to increase the fixed withdraw limit to the max `uint256` value. Had the contract not have this vulnerability, this action should only have been possible through the `setWithdrawLimit`, which is only available to the investor.

In order to overwrite the withdrawal limit, the owner must calculate the hex value to use as a first argument (index) to the function. Since the array `bonusCodes` underflow is defined in the sixth place in the contract storage, its length is in the fifth storage slot (counting from zero)

The limit is defined at the fourth storage slot. Then, in order to manipulate the withdrawal limit, the owner must convert the address of the length to hexadecimal:

Algorithm 9: Exploit - Convert length address to hex

```
$ web3.sha3("0x0000000000000000000000000000000000000000000000000000000000000005", {encoding: 'hex' })
> "0x036b6384b5eca791c62761152d0c79bb0604c104a5fb6f4eb0703f3154bb3db0"
```

and then just calculate the array index that will wrap around using the formula $2^{256} - H + 4$, where 2^{256} is the max `uint256` value, H is the hex obtained from the previous command and 4 is the offset of the withdrawal limit storage slot from the base of the contract. This, converted to hex, will give the owner the address to use with `modifyBonusCode`. The Perl snippet below does that:

Algorithm 10: Exploit - Convert limit offset to address

```
$ perl -Mbigint -E 'say ((2**256 - 0x036b6384b5eca791c62761152d0c79bb0604c104a5fb6f4eb0703f3154bb3db0 + 4)->as_hex)'
> 0xfc949c7b4a13586e39d89eead2f38644f9fb3efb5a0490b14f8fc0ceab44c254
```

As a result, the memory now looks like this:

Algorithm 11: Exploit - Memory at Checkpoint C

```
"storage": {
    "0000000000000000000000000000000000000000000000000000000000000000": "94b898c1a30adcff67208fd79b9e5a4d339f3cc6d2",
    "0000000000000000000000000000000000000000000000000000000000000001": "948bc7317ad44d6f34f0f0b6e3c8c7bf739ba666fa",
    "0000000000000000000000000000000000000000000000000000000000000003": "8902b5e3af16b1880000",
    // The withdrawal limit is now really high:
    "0000000000000000000000000000000000000000000000000000000000000004": "a0ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
    "0000000000000000000000000000000000000000000000000000000000000005": "a0ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
    "dd87d7653af8fba540ea9ebd2d914ba190d975fcfa4d8d2927126a5decdbff9e": "8902b5e3af16b1880000"
}

```

Checkpoint D The owner can now call `withdraw()` with the full amount of ether in the contract and drain it. The investor has not increased the limit at any point.

5 Conclusion

We presented different forms of the common weakness SWC-124: Write to Arbitrary Storage Location and how they might be detected using automated tools. We have shown how a possible exploit may be constructed, and how this can lead to the complete compromise of a smart contract's storage and control flow. We have given multiple attackable and benign code examples to illustrate this weakness. We believe this weakness to be of particular practical relevance, as it is very easy to introduce by accident, and hard to for a developer to spot without advanced knowledge of the underlying mechanisms that cause it.

As for preventative measures, we would recommend developers not to interact with low-level building blocks like an array's length value or inline assembly instructions if possible, and instead to employ standard library functions when ever available.

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] Li Duan, Yangyang Sun, Ke-Jia Zhang, and Yong Ding. Multiple-layer security threats on the ethereum blockchain and their countermeasures. *Security and Communication Networks*, 2022, 02 2022.

[4] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Network and Distributed System Security Symposium*, 2018.

[5] 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.

[6] 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.

[7] 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.