SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Private
Prepared for:	Telcoin
Prepared by:	Sherlock
Lead Security Expert:	0xadrii
Dates Audited:	June 4 - June 9, 2024
Prepared on:	June 27, 2024

Introduction

Telcoin creates low-cost, high-quality financial products for every mobile phone user in the world. This audit is to ensure constant security for all of our user's wallets.

Scope

Repository: telcoin/telcoin-audit

Branch: main

Commit: 97c24d868172afa88790230f87b9a2028d5abda0

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
2	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0xadrii KupiaSec

Issue M-1: Not cleaning upper bits of certain data could lead to wallet being unaccessible forever

Source: https://github.com/sherlock-audit/2024-05-assembly-judging/issues/6

Found by

0xadrii, KupiaSec

Summary

Lack of cleaning upper bits of certain variables might lead to setting incorrect values in the protocol, leading to improper behavior of the contract.

Vulnerability Detail

All values in the EVM are stored as 256 bit values. When a value with a type smaller than 256 bits is used, it is necessary to clean the remaining bits. This is performed by the Solidity compiler by default. However, in yul this cleaning is **not performed,** so values need to be cleaned manually.

However, the Telcoin Wallet implementation lacks performing such data cleaning, which makes the wallet susceptible of accepting incorrect data as valid. This step is extremely important, as it can lead to unexpected outcomes if data is not propably formatted.

Note that this situation is more sensible than the typical incorrect input. Even if owners/gatekeepers are trusted, it is easy to leave dirty bits that might affect the code. For example, expressions of type `address(uint160(packed))` that aim at converting packed data to an address **will leave dirty upper bits.**

This could lead to several consequences affecting the wallet, as it is unknown how interaction with this wallets will be performed, and nothing guarantees that calldata passed to it will be 100% clean. The following situations could arise due to not cleaning dirty upper bits:

- Making the wallet be locked forever. In the situation where there are no owners in the wallet and only the two gatekeepers have access to it, the gatekeepers could decide to change one of the gatekeeper's address in order to add a different gatekeeper. This can be done via the `replaceGatekeeper` function:

```
function replaceGatekeeper() {
    let _old := calldataload(0x4)
    let _new := calldataload(0x24) // @audit-issue [HIGH-01] - Not cleaning
↪   upper bits of certain data could lead to wallet being unaccessible forever
```

```
    ...
    // Note that at this point we are guaranteed that _old != _new since we could
    // not have both storage[_old] == 3 and storage[_new] == 0 at the same time.

    sstore(_old, 0) // Delete the old mapping.
    sstore(_new, 3) // Add the new mapping.

    stop()
}
```

This could lead to a critical situation where if the `_old` address is passed without dirty upper bits but the `_new` address is passed with actual dirty upper bits (maybe due to a different way of fetching both addresses when interacting with the wallet and building the calldata), a wrong `_new` address would be set as the gatekeeper. This would make the wallet **remain locked forever**, given that the `addOwner` and `replaceGatekeeper` functions in the wallet are expected to be triggered via `execute` , which requires **at least two signers.** In this situation, because one of the gatekeepers has been improperly updated due to dirty upper bits, the wallet will remain locked forever, leading to unrecoverable stuck funds.

This could also happen when adding an incorrect owner (although the impact would be smaller). When triggering the `addOwner` function, the `_owner` is directly extracted from calldata and stored to storage. If `_owner` contains dirty upper bits, an incorrect value will be set as the owner of the wallet:

```
// TelcoinWallet.sol
function addOwner() {
    let _owner := calldataload(0x4)
    ...

}
```

- Transactions failing. The `execute` function will trigger the internal `__ecrecover` function so that the signer's address can be recovered from a signature:

```
function __ecrecover(h, v, r, s) -> a {
    // The builtin ecrecover() function is stored in a builtin contract deployed
↪   at
    // address 0x1, with a gas cost hard-coded to 3000 Gas. It expects to be
↪   passed
    // exactly 4 words:
    // (offset 0x00) keccak256 hash of the signed data
    // (offset 0x20) v value of the ECDSA signature, with v==27 or v==28
    // (offset 0x40) r value of the ECDSA signature
    // (offset 0x60) s value of the ECDSA signature
    // Since we will receive signatures with v values of 0 or 1, we can
↪   unconditionally
    // add 27 to transform them into the format expected by ecrecover().
    v := add(v, 27)

    mstore(0, h)
    mstore(0x20, v)
    mstore(0x40, r)
    mstore(0x60, s)

    // Instead of sending 3000 == 0x0bb8 Gas, we will send a little more with
↪   0x0c00
    // since this will have the same result and save us some Gas when deploying
↪   the
    // contract (0x00 bytes are cheaper to deploy).
    if iszero(staticcall(0x0c00, 0x1, 0, 0x80, 0, 0x20)) {
        revert(0, 0)
    }

    a := mload(0)
}
```

Although the `r` and `s` values are `32-byte` values, `v` consists of only 1 byte. This makes it susceptible of containing dirty upper bits, which could lead to valid signed transactions failing due to an incorrect value of `v` containing garbage in the top bits.

- Storage clashing: Telcoin Wallet uses the beacon proxy pattern, where the `ClonableBeaconProxy` inherits from `Initializable`, a contract that helps preventing the `initialize` function from being called more than once:

```
// ClonableBeaconProxy.sol
contract ClonableBeaconProxy is Proxy, Initializable {
    /**
     * @dev Initializes the proxy with `beacon`.
     *
     * If `data` is nonempty, it's used as data in a delegate call to the
↪   implementation returned by the beacon. This
     * will typically be an encoded function call, and allows initializing the
↪   storage of the proxy like a Solidity
     * constructor.
     *
     * Requirements:
     *
     * - `beacon` must be a contract with the interface {IBeacon}.
     * - If `data` is empty, `msg.value` must be zero.
     */
    function initialize(
        address beacon,
        bytes memory data
    ) external initializer {
        ERC1967Utils.upgradeBeaconToAndCall(beacon, data);
    }
```

The `Initializable` contract will set the `INITIALIZABLE_STORAGE` slot to 1 when the `initializer` modifier is triggered in the `initialize` function.

Because of the lack of cleaning upper bits, it is theoretically possible to pass the `INITIALIZABLE_STORAGE` slot to the `removeOwner` function as the owner to be removed. Because the data stored in `INITIALIZABLE_STORAGE` has a value of 1, the `iszero(eq(sload(_owner), 1))` would pass, and the initializable value would be effectively set to zero.

```
// TelcoinWallet.sol

function removeOwner() {
    let _owner := calldataload(0x4)

    // Check if the provided address is equal to 0xaa
    if eq(_owner, 0xaa) {
        revert(0, 0)
    }

    // Invalid if:
    //   `caller() != address() || state[_owner] != 1`
    if or(
        // Checks whether the currently executing code was called by the
        // contract itself, and reverts if that's not the case.
        iszero(eq(caller(), address())),
        // _owner must currently be an owner.
        //
        // Note that if _owner == 0 this will always trip since state is
        // guaranteed to be >3.
        iszero(eq(sload(_owner), 1))
    ) {
        revert(0, 0)
    }

    sstore(_owner, 0) // Delete the mapping.

    stop()
}
```

This would make the ClonableBeaconProxy's function callable again, allowing a malicious actor to change the beacon address and make the wallet be unaccessible forever (although due to considering gatekeepers and owners as TRUSTED roles, this storage clashing situation shouldn't be considered for Sherlock judging as it actually requires maliciously crafted calldata).

Impact

Medium.

From Sherlock's latest judging criteria:

"The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines."

Considering the following excerpt from the contest's README:

"Data that is not properly signed by an authorized gatekeeper or wallet owner should be rejected by the wallet"

As described in the README, data submitted to the wallet with dirty upper bits (which falls in the category of **NOT** properly signed data, even if it is submitted by a gatekeeper/owner) should be rejected. However, the wallet doesn't actually reject this data and will instead accept it, effectively breaking the expected functionality of the wallet of rejecting data that is not properly signed.

Because of this, this bug is of medium impact. Not performing a crucial step when dealing with low-level code will lead to unexpected outcomes, with different impacts ranging from the wallet remaining locked forever to transactions failing.

Code Snippet

https://github.com/sherlock-audit/2024-05-assembly-user/blob/main/telcoin-audit/contracts/TelcoinWallet.sol#L174

https://github.com/sherlock-audit/2024-05-assembly-user/blob/main/telcoin-audit/contracts/TelcoinWallet.sol#L421

(Note the url user has been changed to avoid showing Sherlock's username)

Tool used

Manual Review

Recommendation

Clean the upper bits of all the code variables that store data smaller than 32 bytes, especially the addresses in `addOwner`, `removeOwner`, `replaceGatekeeper` and the `v` value in `__ecrecover`.

Discussion

iamckn

Medium. I am of the opinion that this goes beyond standard input validation.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/telcoin/telcoin-audit/pull/56

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: Signature hash is wrongly computed and leads to signatures not being properly verified

Source: https://github.com/sherlock-audit/2024-05-assembly-judging/issues/7

The protocol has acknowledged this issue.

Found by

0xadrii, KupiaSec

Summary

The current approach to computing the hash to be signed does not properly encode EIP191's validator address, leading to properly signed EIP-191 signatures being rejected by the wallet.

Vulnerability Detail

Telcoin Wallet incorporates the `execute` and `transferERC20` functions, both of which require a specific hash to be signed by at least one gatekeeper and an additional gatekeeper or owner. The hash to be signed is specific to each of the functions, and is computed given the calldata passed when calling the function. However, the standard to follow for the hash to sign is EIP-191 with 0x00 version (*Data with intended validator*), which expects the following format to be signed:

- EIP-191 prefix (0x19).
- Version of EIP-191 signature (0x00 in Telcoin's case, corresponding to "*Data with intended validator*").
- Intended validator address (a **20-byte field** set to `address(this)`).
- Data to sign. As per the EIP, "*The data to sign could be any arbitrary data*".

Note how the third field requires specifically a **20-byte address**, and not a 32-byte field holding data corresponding to an address left-padded with zeroes. Because of this, if one was to build a hash following EIP-191 using plain solidity, the encoding would be performed in the following way (note `abi.encodePacked` is used, which won't pad the `validator` address with zeroes, and instead of using `abi.encode`, which would incorrectly pad `validator` with 12 zeroes on the left):

```
function toDataWithIntendedValidator(address validator, bytes memory dataToSign)
↪  internal pure returns (bytes32) {
    return keccak256(abi.encodePacked("\x19\x00", validator, dataToSign));
}
```

Telcoin attempts to perform an operation similar to `toDataWithIntendedValidator` using plain assembly. In order to build the hash to be signed, `execute` will store the data to be hashed in memory so that later it can be hashed using the `keccak256` opcode. The memory layout chosen by Telcoin to hash the data is the following:

- `0x80`-`0x81`: EIP-191 prefix (0x19).
- `0x81`-`0x82`: 0x00 (version of EIP-191 signature).
- `0x82`-`0xa2`: `address(this)`. It is important to note that `mstore(0x82, address())` is used in order to store this data, so the address will be stored left-padded with zeroes from byte `0x82` to byte `0xa2` in memory, where bytes from `0x82` to `0x1e` correspond to the padding zeroes, and bytes from `0x1e` to `0xa2` correspond to the actual address. This is wrong, as EIP-191 **does not expect the validator address to be zero-padded.**
- `0xa2`-onwards: Actual data to sign.

```
// TelcoinWallet.sol

function execute() {
    ...

    // Set up EIP191 prefix.
    mstore8(0x80, 0x19)
    mstore8(0x81, 0x00)
    mstore(0x82, address())

    // Copy method signature + _identifier + _destination + _value to memory.
    calldatacopy(0xa2, 0, 0x64)

    // Copy _data (without offset or length) after that.
    calldatacopy(0x106, 0x164, _dataLength)

    // Hash all user data except the signatures.
    //
    // The second argument cannot overflow due to an
    // earlier check limiting the maximum value of the
    // length variable.
    let hash := keccak256(0x80, add(0x86, _dataLength))

    ...

}
```

As mentioned, the main problem is that the current approach followed by Telcoin will add padding to the address stored, so the hash computed will be calculated as if `abi.encode` was used, instead of `abi.encodePacked`.

Impact

Medium. Valid EIP-191 signatures will be rejected by the code, making transactions that should be accepted to always fail.

Proof of concept

The following proof of concept shows how data is improperly encoded with an example.

Let's say we wanted to hash the following data:

- EIP-191 prefix (0x19).
- 0x00 (version of EIP-191 signature).
- `0x82-0xa2`: Address of validator, for this example it will be 0x7c8999dc9a822c1f0df42023113edb4fdd543266
- Data to sign:
  - Signature of the `execute` function (0x9d55b53f)
  - Identifier, with value 115792089237316195423569452513147041964411626318613413730273965268791
  - A destination with address 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
  - A value of 1 ether
  - Empty calldata

With this data, the expected calldata to be signed would be the following:

0x19005615deb798bb3e4dfa0139dfa1b3d433cc23b72f9d55b53ffffffffffffffffff000000000000

However, the actual data encoded by Tecloin Wallet is the following: (Note the added 12-byte zero padding between the 0x1900 prefix and the start of the wallet's address 0x5615...)
0x19000000000000000000000000005615deb798bb3e4dfa0139dfa1b3d433cc23b72f9d55b5

Let's dissect it:

- 0x1900 —> the 0x1900 prefix
- 0000000000000000000000005615deb798bb3e4dfa0139dfa1b3d433cc23b72 —> the validator address (incorrectly padded with 12 bytes on the left)

- f9d55b53 —> the `execute` selector
- ffffffffffffffff0000000000000000000000000000000000000000000400010 —> The identifier
- 0000000000000000000000000007c8999dc9a822c1f0df42023113edb4fdd54326 —> The destination address
- 6000000000000000000000000000000000000000000000000000de0b6b3a764000 —> The 1 ETH value, encoded in hex
- 0 —> The dynamic data (was set to empty)

This calldata can be obtained by adding the following `computeDataToHash` function to `TelcoinWallet`, which performs the exact same computations to calculate the hash in `execute`, and serves as a helper to visualize the calldata returned:

```
// TelcoinWallet.sol

case 0xc19d93fb /* bytes4(keccak256("state()")) */ {
      state()
      stop()
  }
  case 0x56bdcd27 { /* computeDataToHash(uint256,address,uint256,bytes,uint8,byt
↪   es32,bytes32,uint8,bytes32,bytes32) */
      computeDataToHash() <---- add this case
      stop()
  }
  default {
      // We stop the transaction here and accept any ETH that was passed in.
      stop()
  }

  function computeDataToHash() { <---- add this function
      let _dataLength := calldataload(0x144)

      // Set up EIP191 prefix.
      mstore8(0x80, 0x19)
      mstore8(0x81, 0x00)
      mstore(0x82, address())

      // Copy method signature + _identifier + _destination + _value to memory.
      mstore8(0xa2, 0x9d)
      mstore8(0xa3, 0x55)
      mstore8(0xa4, 0xb5)
      mstore8(0xa5, 0x3f)
      calldatacopy(0xa6, 0x04, 0x60)

      // Copy _data (without offset or length) after that.
      calldatacopy(0x106, 0x164, _dataLength)

      return(0x80, add(0x86, _dataLength))

  }
```

Then, create a foundry project with the `TelcoinWallet` source file and paste the following test:

```
// TestPoc.t.sol

// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/TelcoinWallet.sol";
import "forge-std/console.sol";

interface ITelcoinWallet {
    function transferErc20(
        uint256,
        address,
        address,
        uint256,
        uint256,
        address,
        uint256,
        uint256,
        address,
        uint8,
        bytes32,
        bytes32,
        uint8,
        bytes32,
        bytes32,
        uint8,
        bytes32,
        bytes32
    ) external;
    function execute(
        uint256,
        address,
        uint256,
        bytes calldata,
        uint8,
        bytes32,
        bytes32,
        uint8,
        bytes32,
        bytes32
    ) external;
    function addOwner(address) external;
    function removeOwner(address) external;
    function replaceGatekeeper(address, address) external;
    function initialize(uint256, address, address, address) external;
    function isOwner(address) external view returns (bool);
    function state() external;
}

contract ContractTest is Test {
    ITelcoinWallet public wallet;
    uint256 public currState;
    address public gatekeeperA;
    address public gatekeeperB;
    address public owner;
    uint256 public gatekeeperAPrivKey;
    uint256 public gatekeeperBPrivKey;
    uint256 public ownerPrivKey;

    function setUp() public {
        wallet = ITelcoinWallet(address(new TelcoinWallet()));

        assembly {
            sstore(currState.slot, shl(180, not(0)))
        }

        (gatekeeperA, gatekeeperAPrivKey) = makeAddrAndKey("gatekeeperA");
        (gatekeeperB, gatekeeperBPrivKey) = makeAddrAndKey("gatekeeperB");
        (owner, ownerPrivKey) = makeAddrAndKey("owner");

        // Initialize wallet
        wallet.initialize(currState, gatekeeperA, gatekeeperB, owner);
        assertFalse(wallet.isOwner(gatekeeperA));
        assertFalse(wallet.isOwner(gatekeeperB));
        assertTrue(wallet.isOwner(address(0xaa)));
        assertTrue(wallet.isOwner(owner));
    }

    function testWrongEIP191Implementation() external {
        uint256 identifier;
        assembly {
            identifier := add(identifier, shl(8, 1)) // set nonce
            identifier := add(identifier, shl(26, timestamp())) // set timestamp
            identifier := add(sload(currState.slot), identifier) // add to
 ↪  cuerrent identifier
        }

        (, bytes memory dataToHash) = address(wallet).call(
            abi.encodeWithSignature(
                "computeDataToHash(uint256,address,uint256,bytes,uint8,bytes32,b⌋
 ↪  ytes32,uint8,bytes32,bytes32)",
                identifier,
                owner,
                1 ether,
                ""
            )
        );


        console.logBytes(dataToHash);

    }

}
```

As we can see, the validator address is wrongly padded with 12 zero bytes, which breaks EIP-191 compatibility.

Code Snippet

https://github.com/sherlock-audit/2024-05-assembly-user/blob/main/telcoin-audit/contracts/TelcoinWallet.sol#L606

https://github.com/sherlock-audit/2024-05-assembly-user/blob/main/telcoin-audit/contracts/TelcoinWallet.sol#L781

(Note the url user has been changed to avoid showing Sherlock's username)

Tool used

Manual Review, foundry

Recommendation

When encoding the validator address, perform the following change:

```
function execute() {
  // When executing this function, the calldata is intended to be:
  //
  //   start | description                     | length in bytes
  // --------+---------------------------------+------------------
  //   0x00  | Method signature                | 0x4
  //   0x04  | _identifier                     | 0x20
  //   0x24  | _destination                    | 0x20
  //   0x44  | _value                          | 0x20
  //   0x64  | _dataOffset                     | 0x20
  //   0x84  | _sig1V                          | 0x20
  //   0xa4  | _sig1R                          | 0x20
  //   0xc4  | _sig1S                          | 0x20
  //   0xe4  | _sig2V                          | 0x20
  //   0x104 | _sig2R                          | 0x20
  //   0x124 | _sig2S                          | 0x20
  //   0x144 | _dataLength                     | 0x20
  //   0x164 | _data                           | _dataLength
  //
  // We will copy these in memory using the following layout:
  //
  //   start | description                     | length in bytes
  // --------+---------------------------------+------------------
  //   0x00  | Scratch space for __ecrecover | 0x80
  //   0x80  | EIP191 prefix 0x1900            | 0x2          \
- //   0x82  | EIP191 address                  | 0x20         |
+ //   0x76  | EIP191 address                  | 0x20         |
  //   0xa2  | _methodSignature                | 0x4          |
  //   0xa6  | _identifier                     | 0x20         | sig1 & sig2
  //   0xc6  | _destination                    | 0x20         |
  //   0xe6  | _value                          | 0x20         |
  //   0x106 | _data                           | _dataLength  /
  //
  // This memory layout is set up so that we can hash all the operation data
↪  directly.
  //
  // Note that the hash includes the method signature itself, so that a blob
↪  signed
  // for execute() cannot be used for transferErc20(), or the opposite.
  //
  // Note that the hash also includes the wallet address() so that an operation
↪  signed
```

```
  // for this wallet cannot be applied to another wallet that would happen to
↪  have the
  // same owners/gatekeepers.

      ...

      // Set up EIP191 prefix.
-     mstore(0x82, address())
+     mstore(0x76, address())
      mstore8(0x80, 0x19)
      mstore8(0x81, 0x00)
```

Discussion

iamckn

Medium. The address should be packed immediately after the 0x1900 prefix but instead it is left padded with zeros.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.