Simple MultiSig Wallet Audit

1 Summary

ConsenSys Diligence conducted a security audit for the `SimpleMultiSig` contract. The contract will at first be used to store and transfer ether. It may be used to store and transfer tokens or other assets in the future.

Number of issues per severity

	Minor	Medium	Major	Critical
Open	1	0	0	0
Closed	3	4	0	0

1.1 Key Observations/Recommendations

- **Simple contract design**: The contract design is excellent. The simplicity of the contract means there's a very small attack surface.
- **Clear code**: The code is well written and commented.

The signatures used with this wallet lock down key parameters of a transaction, such as the recipient and payload, but they leave the following parameters unconstrained:

- the initiator of the transaction
- the gas limit supplied
- the timing of the transaction

These free parameters are the primary surface area for an attacker. The recommended changes detailed in section 3 mostly relate to **reducing the attack surface area** by constraining these remaining parameters. In some cases, reducing the surface area also reduces functionality. For a wallet that will control assets of considerable value, we believe that tradeoff is worthwhile.

1.2 Audit Dashboard

Audit Details

- **Project Name:** Simple MultiSig Wallet
- **Auditors:** Steve Marx, Dean Pierce
- **Internal Reviewer:** J. Maurelian
- **Languages:** Solidity
- **Date:** 2018-09-11 to 2018-09-24

1.3 Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

Security: Identifying security related issues within each contract and within the system of contracts.

Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:

- Correctness
- Readability
- Sections of code with high complexity
- Improving scalability
- Quantity and quality of test coverage

1.4 System Overview

Scope

The scope of this audit was the single file `SimpleMultiSig-v1.0.4` .

Design

The `SimpleMultiSig` contract is an m-of-n multisig wallet that uses minimal usable state. Signatures are gathered offline and submitted together to execute a transaction.

2 Issue Overview

The following table contains all the issues discovered during the audit. The issues are ordered based on their severity. More detailed description on the levels of severity can be found in Appendix 2. The table also contains the GitHub status of any discovered issue.

Chapter	Issue Title	Issue Status	Severity
3.1	Signed data should include the sender	Closed	Medium
3.2	Signed data should include the gas limit	Closed	Medium
3.3	Failed transactions should not be replayable	Closed	Medium
3.4	A reasonable threshold should be required	Closed	Medium
3.5	Consider whitelisting function calls to standard ERC20/ERC223 functions	Open	Minor
3.6	Use the latest Solidity compiler	Closed	Minor
3.7	Passing multiple arguments to keccak256 is deprecated	Closed	Minor
3.8	The execute function can be external rather than public	Closed	Minor

3 Issue Detail

3.1 Signed data should include the sender

Severity	Status	Link	Remediation Comment
Medium	Closed	issues/32	This was fixed in https://github.com/christianlundkvist/simple-multisig/commit/179a6834e80a39f5f709a6a92107bec2673aa89e.

Description

`execute()` can be called by anyone in possession of `threshold` valid signatures. This opens up a front-running opportunity where an attacker observes a desired call to `execute()` and sends their own transaction with the same signatures. This changes the `tx.origin` for the transaction, which may affect the result.

Furthermore, locking down the sender to a trusted (by the owners) party could mitigate cross-chain replay attacks by not allowing an attacker to submit the transaction on the other chain.

Example

Below is a contrived example of a target contract that an attacker could manipulate by front-running with their own transaction:

```
function() public payable {
    // Behavior differs according to tx.origin.
    if (tx.origin == owner) {
        // Bad attempt at a refund?
        tx.origin.transfer(msg.value);
    } else {
        // Track how much ether has been deposited
        balance += msg.value;
    }
}
```

Remediation

Add `msg.sender` to the signed data.

3.2 Signed data should include the gas limit

Severity	Status	Link	Remediation Comment
Medium	Closed	issues/31	This is considered mitigated by adding the `executor` to the signature in https://github.com/christianlundkvist/simple-multisig/commit/179a6834e80a39f5f709a6a92107bec2673aa89e. The gas limit was also included, but there is no check for sufficient gas, so this only prevents issues involving providing *too much* gas.

Description

`execute()` can be called by anyone in possession of `threshold` valid signatures. The signatures do not specify how much gas will be forwarded to the destination address. This allows an attacker, by front-running a legitimate call, to make the same call with a different amount of gas.

If the executed call succeeds but behaves differently based on the amount of gas provided, an attacker could use this technique to affect the outcome.

Example

Below is a contrived example of a target contract that fails to transfer funds if insufficient gas is provided. An attacker maliciously lowering the provided gas could result in locked ether or simply a denial of service attack against the wallet owners.

```
function withdraw(address destination) external {
    uint256 amount = balance;
    balance = 0;

    // Silently fails with insufficient gas
    destination.call.value(amount)();
}
```

Remediation

Add the gas limit as a parameter to `execute()` and include that parameter in the signature.

3.3 Failed transactions should not be replayable

Severity	Status	Link	Remediation Comment
Medium	Closed	issues/30	The project team considers this unnecessary due to the addition of an `executor` in the signed data in https://github.com/christianlundkvist/simple-multisig/commit/179a6834e80a39f5f709a6a92107bec2673aa89e.

Description

If a call to `execute()` with proper signatures fails because the resulting `call` does, the `nonce` is not incremented. Until the `nonce` is used for a different transaction, anyone can attempt that original transaction again. If the transaction has different behavior in the future (e.g. because the destination contract was destroyed or a crowdsale moved to a different phase), this may be an opportunity for an attacker to make mischief.

Example

Below is a contrived example of a target contract that reveals this vulnerability. The initial call to `reverts` with attached ether will fail, but if the contract is later `selfdestruct` ed, replaying that transaction would result in lost ether, as there would be no code to revert the transaction.

```
function reverts() external payable {
    revert();
```

```
  }

  function kill() external {
      require(msg.sender == owner);
      selfdestruct(msg.sender);
  }
```

Remediation

We suggest two changes to address this vulnerability:

1. As suggested in other issues, specify the transaction sender in the signature. This means that only the sender can retry a failed transaction. Depending on the trust model and security of the sender, this may be sufficient.
2. Increment the nonce even when the `call` fails. (Remove `require(success)` .) Note that this should *only* be done if the sender and/or gas limit have been locked down. (If those changes are not made, then increasing the nonce on failed transactions makes it *easier* for an attacker to mount a denial of service attack by front-running with insufficient gas and forcing the `call` to fail, consuming the nonce.)

If suggestion 2 above is not taken, then we recommend both of the following additional remediations:

1. Specify a maximum block number parameter and include that parameter in the signature. Reject any transactions if the maximum block number has passed. This limits an attacker's window of opportunity to an arbitrarily small number of blocks.
2. As an operational procedure, any time a transaction fails, submit a new "no-op" transaction with the same nonce to replace the failed one and prevent future replay attempts. (A good "no-op" transaction would be a transfer of 0 ether to the wallet contract.)

3.4 A reasonable threshold should be required

Severity	Status	Link	Remediation Comment
Medium	Closed	issues/26	This was fixed in https://github.com/christianlundkvist/simple-multisig/commit/1d8ad609e57a4326230a05250a3b72b7356c2ba7. The threshold is now required to be positive.

Currently, the code allows for a `threshold` of `0`, which is somewhat nonsensical and would allow anyone to call `execute` successfully without any signatures.

SimpleMultiSig-v1.0.4.sol:L11

```
    require(owners_.length <= 10 && threshold_ <= owners_.length && threshold_ >= 0);
```

A threshold greater than 0 should be required, and consideration should be given to requiring something stricter (like a majority: `threshold > owners_.length / 2`).

3.5 Consider whitelisting function calls to standard ERC20/ERC223 functions

Severity	Status	Link	Remediation Comment
Minor	Open	issues/29	The issue is currently under review

As it exists now, the contract can make arbitrary calls to arbitrary contracts. To reduce the potential attack surface, it might be useful to whitelist which functions can be called on remote contracts or even what contracts can be called, if at all possible. We recommend that the whitelist be hardcoded and unchangeable once deployed. If requirements change, a new multisig contract can be deployed with updated capabilities.

3.6 Use the latest Solidity compiler

Severity	Status	Link	Remediation Comment

Severity	Status	Link	Remediation Comment
Minor	Closed	issues/25	The compiler version was bumped to 0.4.24 in https://github.com/christianlundkvist/simple-multisig/commit/f63ef72e448ecef85dd61ad5a3727c7dba4e4377.

The codebase uses pragmas that are set to an older version of Solidity. Use the most recent version of Solidity (0.4.25 at the time of this audit).

3.7 Passing multiple arguments to keccak256 is deprecated

Severity	Status	Link	Remediation Comment
Minor	Closed	issues/24	The latest version of the code uses `abi.encodePacked`.

SimpleMultiSig-v1.0.4.sol:L29

```
    bytes32 txHash = keccak256(byte(0x19), byte(0), this, destination, value, data, nonce);
```

should be updated accordingly.

```
SimpleMultiSig-v1.0.4.sol:30:22: Warning: This function only accepts a single "bytes" argument. Please use
"abi.encodePacked(...)" or a similar function to encode the data.
    bytes32 txHash = keccak256(byte(0x19), byte(0), this, destination, value, data, nonce);
                     ^-------------------------------------------------------------------^
```

3.8 The execute function can be external rather than public

Severity	Status	Link	Remediation Comment
Minor	Closed	issues/23	Marking the function `external` would leave the `data` in call data (rather than memory), requiring other changes to the function. It was decided that leaving this function `public` is simpler.

Because it is never called internally, this function can be made `external`. This likely has no security implications but will save gas.

4 Threat Model

The creation of a threat model is beneficial when building smart contract systems as it helps to understand the potential security threats, assess risk, and identify appropriate mitigation strategies.

A threat model was created during the audit process in order to analyze the attack surface of the contract system and to focus review and testing efforts on key areas that a malicious actor would likely also attack.

4.1 Overview

The multisig wallet is intended to have two important properties:

1. It allows authorized access.
2. It disallows unauthorized access.

It's a common mistake to only focus on the second property, but the first property is also critical. The owners of the wallet lose just as much from having their access blocked as they lose from an attacker transferring the assets away. An attacker can mount a denial of service attack simply to make mischief or as a way to extort the owners.

4.2 Threat Analysis

Threat	Consequence	Mitigation
Owners' private keys are lost.	If no more than *threshold* private keys are available, the wallet owners can no longer access the controlled funds.	Using a sufficiently large number of owner accounts means that many keys would have to be lost before losing access to the wallet.
An attacker executes an unauthorized transaction.	All assets could be transferred to an attacker's account.	Only transactions signed by `threshold` keys can be executed. The signature specifies the destination, value, and data for the transaction, so these cannot be tampered with. Suggestions made in section 3 mitigate other changes that an attacker may make without changing the signature.
An attacker replays an executed transaction.	At a minimum, more assets could be transferred away than intended (e.g. paying multiple times for the same service). Given that the contract can make arbitrary calls, the risks of replay attacks cannot be fully enumerated.	Replay attacks are mitigated by the use of a nonce and EIP191 signatures, which include the contract address as part of the signature. Replaying a failed transaction can be mitigated by replacing the failed transaction with a no-op (like a transfer of 0 ether) to consume the nonce. Suggestions in section 3 further reduce the risk of replay attacks.

5 Tool based analysis

The issues from the tool based analysis have been reviewed and the relevant issues have been listed in chapter 3 - Issues.

5.1 Mythril

Mythril is a security analysis tool for Ethereum smart contracts. It uses concolic analysis to detect various types of issues. The tool was used for automated vulnerability discovery for all audited contracts and libraries. More details on Mythril's current vulnerability coverage can be found here.

The raw output of the Mythril vulnerability scan has been reviewed, and none of the findings have been deemed a legitimate threat.

5.2 Solhint

This is an open source project for linting Solidity code. The project provides both Security and Style Guide validations. The issues of Solhint were analyzed for security relevant issues only. It is still recommended to use Solhint during development to improve code quality while writing smart contracts.

5.3 Surya

Surya is an utility tool for smart contract systems. It provides a number of visual outputs and information about structure of smart contracts. It also supports querying the function call graph in multiple ways to aid in the manual inspection and control flow analysis of contracts.

6 Test Coverage Measurement

The Solidity-Coverage tool was used to measure the portion of the code base exercised by the test suite and identify areas with little or no coverage.

The existing tests at the time of this audit achieve 100% test coverage.

It's important to note that "100% test coverage" is not a silver bullet. The issues outlined in section 3 exist despite 100% coverage and passing tests.

Appendix 1 - File Hashes

The SHA1 hashes of the source code files in scope of the audit are listed in the table below.

File Name	SHA-1 Hash
SimpleMultiSig-v1.0.4.sol	bd300504603a75bb4c5f4e0fba283c11e555dad0

Appendix 2 - Severity

A.2.1 - Minor

Minor issues are generally subjective in nature, or potentially deal with topics like "best practices" or "readability". Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgment as to whether addressing these issues improves the codebase.

A.2.2 - Medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

A.2.3 - Major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable, or may require a certain condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

A.2.4 - Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.

Appendix 3 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.