Zellic

Sequence Wallet

Smart Contract Security Assessment

March 9, 2023

Prepared for:

Peter Kieltyka

Horizon

Prepared by:

Daniel Lu and Filippo Cremonese

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Horizon from February 6th to February 10th, 2023. During this engagement, Zellic reviewed Sequence Wallet's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the wallet safe against unauthorized use?
- Is the wallet safe against replay attacks?
- Is the wallet safe against denial-of-service attacks?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope
- Issues deriving from user error or insecure configurations

1.3 Results

During our assessment on the scoped Sequence Wallet contracts, we discovered one informational finding. No critical issues were found.

Additionally, Zellic recorded its notes and observations from the assessment for Horizon's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	1

2 Introduction

2.1 About Sequence Wallet

Sequence is a smart contract wallet that implements batch transactions, full multisig compatibility, modules, and more, while remaining well-optimized with regards to gas usage. Some of the features of the wallets include

- Merkle signatures
- Lazy configuration updates
- Minimal deployment (proxy + counterfactual address)
- Multiple types of calls (delegatecall, batched, atomic batches, expirable, etc.)
- Signature replacement support

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash

loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Sequence Wallet Modules

Repository	https://github.com/0xsequence/wallet-contracts
Versions	`a1a6368152b02cea6d4776c7cc43c1364546a3d2`
Programs	• MainModule • MainModuleUpgradable and inherited modules • SequenceBaseSig • SequenceChainedSig • SequenceDynamicSig • SequenceNoChainIdSig
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Daniel Lu, Engineer
daniel@zellic.io

Filippo Cremonese, Engineer
fcremo@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

February 3, 2023	Kick-off call
February 6, 2023	Start of primary review period
February 10, 2023	End of primary review period

3 Detailed Findings

3.1 Integer overflow

- **Target**: `SequenceBaseSig::recoverBranch`
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Informational

Description

While the signature tree is being verified, `recoverBranch` computes the cumulative weight of the nodes.

All node types increase the `weight` variable by the appropriate value, with the exception of nodes of type `FLAG_SUBDIGEST`. When processing this specific node type, the `weight` variable is directly set to `type(uint256).max`.

A signature tree containing a node of type `FLAG_SUBDIGEST` authorizing the transaction digest followed by another signature node with nonzero weight would cause an integer overflow in the `weight` variable.

Impact

This issue could cause a low-impact accidental denial of service. We believe this issue cannot be triggered by an attacker under common usage scenarios; therefore, this issue is reported as informational. We do not believe any reasonable configuration would lead to a permanent denial-of-service condition that would not exist if not for this issue.

We note that the `recoverBranch` function body is wrapped in an `unchecked` block. This eliminates the overflow checks normally inserted by the Solidity compiler, which would otherwise lead to the transaction being reverted. Instead, the `weight` will overflow to zero, possibly preventing the weight from becoming greater than the `threshold` required to authorize a transaction.

Recommendations

One possibility to mitigate this issue would be to use a saturating addition function for incrementing the `weight` variable, preventing overflows.

Remediation

Horizon acknowledged this finding, opting to not change the code as a signature that contains both `FLAG_SUBDIGEST` as well as other types of signatures is considered to be malformed. We believe Horizon choice is reasonable, since this issue cannot cause the execution of an unauthorized transaction or a permanent denial of service. We report a verbatim quote from Horizon, explaining their reasoning:

> We believe that using FLAG_SUBDIGEST alongside other flags that also increase the total weight should result in a wrongly encoded signature, leading to a failure in the signature verification process, as expected.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Signature validation

To execute any transaction using the wallet authority, users must call `ModuleCalls::execute`:

```
function execute(
  Transaction[] calldata _txs,
  uint256 _nonce,
  bytes calldata _signature
)
```

The `execute` function takes the list of transactions to execute, and a nonce and signature are required to authorize the execution. The authorization check can be summarized in two main steps:

1. The nonce is validated to prevent replay attacks.
2. The signature is validated to ensure the wallet owners have authorized the transaction.

The signature is tied to the given transaction and nonce by signing the hash of both. Signature verification ensures that

- The signature corresponds to the transaction and nonce.
- The signature corresponds to the signer configuration (`imageHash`).
- The signers that actually signed have enough weight to authorize execution of a transaction.

Sequence wallets accept four kinds of signatures: `LEGACY_TYPE`, `DYNAMIC_TYPE`, `NO_CHAIN_ID_TYPE`, and `CHAINED_TYPE`. The type of the signature is encoded in the first byte of `_signature`.

Signature types

`LEGACY_TYPE`, `DYNAMIC_TYPE`, and `NO_CHAIN_ID_TYPE` signatures

The code path for validating those signatures is as follows:

- `ModuleCalls::execute` - user invokes this function.
  - `ModuleCalls::_validateNonce` validates the nonce and updates the stored one.
  - `ModuleAuth::_signatureValidation` performs signature validation.
- `ModuleAuth::signatureRecovery` ensures subsignatures correspond to the TX data and nonce and returns signature weight, threshold and `imageHash`.
  - `SequenceBaseSig::recover` is code specific to the two signature types being discussed and computes signature weight, threshold and `imageHash`.
- `SequenceBaseSig::recoverBranch` is described below.
  - `ModuleAuthFixed`, `ModuleAuthUpgradable`, or `ModuleExtraAuth` `_isValidImage`, depending on the wallet configuration, ensures the `imageHash` computed from the provided signature corresponds to the expected one.

The three signatures types follow virtually identical code paths, all invoking `SequenceBaseSig::recover`. There are two minor but important differences between legacy signatures and `DYNAMIC_TYPE` and `NO_CHAIN_ID_TYPE` signatures. First, in the case of both latter signatures, one byte is stripped from the `_signature` array before passing it to `recover`. Second, in the case of `NO_CHAIN_ID_TYPE`, the digest of the transaction data to be signed is computed without including the chain ID. This allows reuse of the same signature across multiple chains.

`CHAINED_TYPE` signatures

This signature type allows to delegate signer authority to a different `imageHash` by chaining multiple signatures together. Each signature in the chain is validated by calling `ModuleAuth::signatureRecovery`.

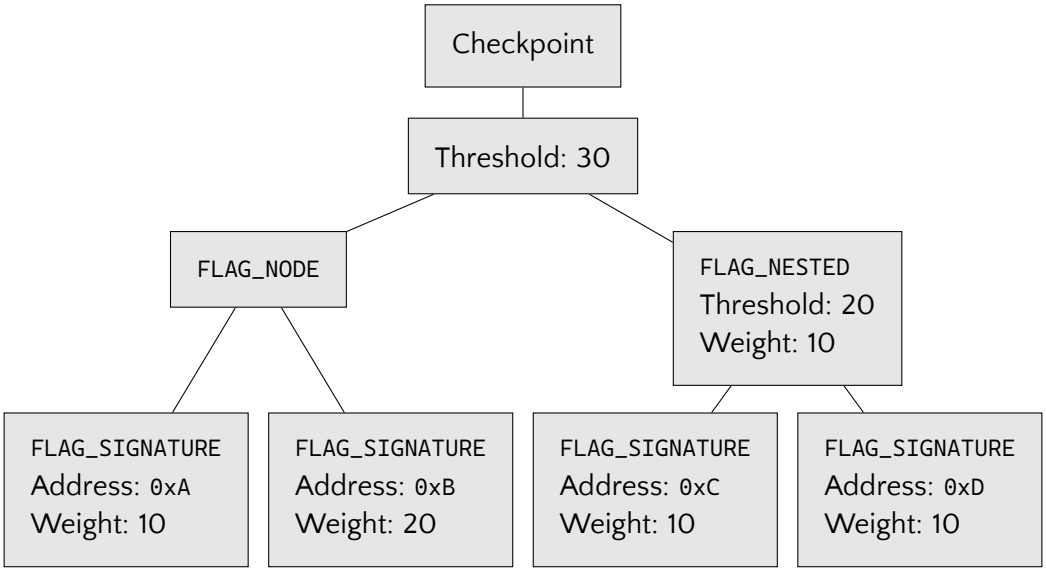
The first signature in the chain is computed against the hash of the transaction data and nonce, exactly like the other signature types. Each following signature is computed against the `imageHash` derived from the preceding signature in the chain. The `imageHash` of the last signature in the chain must correspond to the `imageHash` that is authorized to execute transactions.

Signer configuration and `imageHash` computation

Sequence wallets allow flexible configuration, supporting a wide array of multisig setups. For efficiency, the actual signer configuration is not stored on chain. A cryptographic hash (`imageHash`) of the configuration is stored instead. The signer configuration can be thought of as a Merkle tree. In Merkle trees, each node has a corresponding hash, and the hash of each node is derived from the information contained in the node and the hashes of all the underlying child nodes. This implies that the hash of the root of the tree is tied to all the information contained in, and uniquely identifies, the spe-

cific tree instance. The `imageHash` of the configuration is the hash of the root node of the configuration tree.

This is one example of a configuration requiring the signature of either address A and B or address B, C, and D to authorize a transaction.



Signatures should also be thought of as a tree replicating the structure of the configuration tree. This allows to compute the `imageHash` of the configuration corresponding to a given signature. A valid signature has to match the `imageHash` expected by the wallet, thus only allowing the intended signers to authorize transactions.

A signature tree can contain a number of different node types:

- `FLAG_ADDRESS` contains just an address and a weight, but no signature. The address and weight contribute to the `imageHash` computed for the signature. This node does not count towards the actual weight of the signature, since the address is not a signer.
- `FLAG_SIGNATURE` contains a weight and a fixed-length signature. The weight and the signer address recovered from the signature contribute to the `imageHash` of the signature.
- `FLAG_DYNAMIC_SIGNATURE` contains a weight, a signature, and the signature length. This allows to embed `SIG_TYPE_EIP712` and `SIG_TYPE_ETH_SIGN` signatures produced by an actual private key, as well as `ERC1271` signatures, which are validated by invoking the corresponding smart contract.
- `FLAG_NODE` contains 32 bytes that contribute directly to the `imageHash`. This node does not count against the effective weight of the signature.

- `FLAG_BRANCH` marks the beginning of a subtree. The subtree is recursively processed, and the `imageHash` of the subtree contributes to the `imageHash` of the parent tree.
- `FLAG_NESTED` marks the beginning of a subtree with a fixed weight and internal threshold. If the cumulated weight of the signatures in the subtree is greater or equal to the internal threshold, the external weight is added to the weight of the parent tree.
- `FLAG_SUBDIGEST` is a special node that allows to pseudo-sign a fixed digest. If the digest being signed matches the one encoded in this node, the signature weight is set to `type(uint256).max`.

Initial `imageHash` validation

When initially deployed, wallets use `ModuleAuthFixed::_isValidImage(bytes32 _imageHash)` to validate the `imageHash` derived from a signature against the expected `imageHash` identifying the signer configuration.

The initial `imageHash` is provided as the salt used by `create2` to derive the address of the deployed wallet. This allows to derive the address of a wallet that uses a given `imageHash` and even to send funds (ETH as well as, for example, ERC20 or ERC721 tokens) before deploying it, while still being secure against attackers deploying and taking control of a wallet to the given address. Also, `ModuleAuthFixed` verifies that the address of the contract corresponds to the address derived using the `imageHash` computed from the signature as a salt.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 File: `Factory.sol`

Function: `deploy`

Deploys a new instance of a wallet proxy using the given `_salt` and `_mainModule` as, respectively, the initial `imageHash` and the address of the implementation contract.

The given `_salt` is used as salt for the `create2` instruction. The salt is included in the material hashed to obtain the address of the newly deployed contract, which is calculated as follows:

```
keccak256(
  abi.encodePacked(
    hex"ff",
    FACTORY, // Address invoking `create2`
    _imageHash, // salt
    INIT_CODE_HASH // hash of the contract code
  )
)
```

It is therefore possible to derive the address of a wallet and send funds to it before deploying it while still being secure against attackers front-running the legitimate wallet deployment. See the section 4.1 for a more in-depth discussion of this aspect.

Branches and code coverage

Intended branches:

- Packs the bytecode used by the new instance and invokes `create2` to deploy it.
  ☑ Test coverage

Negative behavior:

- Wallet already exists.
  ☐ Negative test

Inputs

- `_mainModule`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the implementation contract called by the proxy.
- `_salt`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Represents the `imageHash` initially accepted by the wallet.

5.2 File: `ModuleAuth.sol`

Function: `signatureRecovery`

Given a signature tree and the digest signed, this function returns the signature threshold, the cumulative weight of the actual signers, the `imageHash`, the subdigest of the signed message, and the checkpoint. See the section 4.1 for a more in-depth description of signature validation.

Branches and code coverage

Intended branches:

- Handles signatures of type `LEGACY_TYPE`.
  ☑ Test coverage
- Handles signatures of type `DYNAMIC_TYPE`.
  ☑ Test coverage
- Handles signatures of type `NO_CHAIN_ID_TYPE`.
  ☑ Test coverage
- Handles signatures of type `CHAINED_TYPE`.
  ☑ Test coverage

Negative behavior:

- Invalid signature type.

☑ Negative test

- Rejects invalid nested signatures.
  ☑ Negative test

Inputs

- `_digest`:
  - **Control**: Arbitrary if called directly; hash of TX data when coming from `_execute`.
  - **Constraints**: None.
  - **Impact**: Digest of the message that has to be signed.
- `_signature`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature tree that has to be validated.

Function call analysis

Section 4.1 explains the code paths taken while validating signatures.

Function: `isValidSignature`

Two variants of this function exist, implementing the ERC1271 interface. One variant takes a `bytes` of data, while the other variant takes a `bytes32` hash of the data. The special value `0x20c13b0b` is returned if the given `_signatures` are valid for the given `_data` or `_hash`, respectively. See the section 4.1 for a more in-depth description of signature validation.

Branches and code coverage

Intended branches:

- Handles signatures of type `LEGACY_TYPE`.
  ☑ Test coverage
- Handles signatures of type `DYNAMIC_TYPE`.
  ☑ Test coverage
- Handles signatures of type `NO_CHAIN_ID_TYPE`.
  ☑ Test coverage
- Handles signatures of type `CHAINED_TYPE`.

☑ Test coverage

Negative behavior:

- Invalid signature type.
  ☑ Negative test
- Rejects invalid signatures.
  ☑ Negative test
- Rejects signatures with insufficient weight.
  ☑ Negative test

Inputs (variant 1)

- `_data`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Data that should be signed by the provided signature.
- `_signature`:
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid signature, otherwise zero is returned or the transaction is reverted.
  - **Impact**: Should represent a valid signature for the given data.

Inputs (variant 2)

- `_hash`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Hash of the data that should be signed by the provided signature
- `_signature`:
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid signature, otherwise zero is returned or the transaction is reverted.
  - **Impact**: Should represent a valid signature for the given hash.

Function call analysis

These functions almost exclusively consist in an invocation of `ModuleAuth::_signatureValidation`, which in turn invokes `signatureRecovery`. Section 4.1 explains the code

paths taken while validating signatures.

Function: `updateImageHash`

This function updates the `imageHash` of the signer configuration that is authorized to execute transactions with the wallet.

The function can only be invoked by the wallet itself (through `ModuleCalls::execute`).

Branches and code coverage

Intended branches:

- Updates the contract implementation if needed and updates the `imageHash` in the contract storage.
  ☑ Test coverage

Negative behavior:

- Rejects invalid (zero) `imageHash`.
  ☑ Test coverage
- Caller is not self.
  ☑ Test coverage

Inputs

- `_imageHash`:
  - **Control**: Arbitrary.
  - **Constraints**: Can be any nonzero `bytes32` value.
  - **Impact**: Represents the `imageHash` of the new signer configuration.

Function call analysis

Depending on the current implementation contract, there are two possible code paths.

- `ModuleAuthUpgradable::_updateImageHash()` updates the `imageHash` in the contract storage.
- **What is controllable?** `imageHash`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.

  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert prevents the `imageHash` from being updated. Reentrancy or unusual control flow are not a concern.
- `ModuleAuthFixed::_updateImageHash()` updates the `imageHash` in the contract storage **and** updates the contract implementation to use an instance of `MainModuleUpgradable`.
  - **What is controllable?** `imageHash`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert prevents the `imageHash` from being updated. Reentrancy or unusual control flow are not a concern.

5.3 File: `ModuleAuthConvenience.sol`

Function: `updateImageHashAndIPFS`

This function updates the `imageHash` of the signer configuration that is authorized to execute transactions with the wallet as well as the IPFS root key.

The function can only be invoked by the wallet itself (through `ModuleCalls::execute`).

Branches and code coverage

Intended branches:

- Updates the contract implementation if needed and updates the `imageHash` in the contract storage. Also updates the IPFS root key.
  ☑ Test coverage

Negative behavior:

- Rejects invalid (zero) `imageHash`.
  ☑ Test coverage (indirectly)
- Caller is not self.
  ☑ Test coverage

Inputs

- `_imageHash`:
  - **Control**: Arbitrary.

  - **Constraints**: Can be any nonzero `bytes32` value.
  - **Impact**: Represents the `imageHash` of the new signer configuration.
- `_ipfsRoot`:
  - **Control**: Arbitrary.
  - **Constraints**: Can be any `bytes32` value.
  - **Impact**: Represents the new IPFS root key.

Function call analysis

Depending on the current implementation contract, there are two possible code paths taken for updating the `imageHash`.

- `ModuleAuthUpgradable::_updateImageHash()` updates the `imageHash` in the contract storage.
- **What is controllable?** `imageHash`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert prevents the `imageHash` and IPFS root from being updated. Reentrancy or unusual control flow are not a concern.
- `ModuleAuthFixed::_updateImageHash()` updates the `imageHash` in the contract storage **and** updates the contract implementation to use an instance of `MainModuleUpgradable`.
  - **What is controllable?** `imageHash`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert prevents the `imageHash` and IPFS root from being updated. Reentrancy or unusual control flow are not a concern.

The IPFS root key is also updated by invoking `_updateIPFSRoot`.

- `ModuleIPFS::_updateIPFSRoot()`:
  - **What is controllable?** `_hash`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert prevents the `imageHash` and IPFS root from being updated. Reentrancy or unusual control flow are not a concern.

5.4 File: `ModuleCalls.sol`

Function: `execute`

This function is invoked by users to execute transactions with the authority of the wallet. A signature authorizing the transaction must be provided. Refer to section 4.1 for a more in-depth explanation of the signature validation.

Branches and code coverage

Intended branches:

- Validates nonce and signature and executes the transactions.
  ☑ Test coverage

Negative behavior:

- Caller is not EOA or another contract.
  ☑ Test coverage
- Reverts if a transaction marked `revertOnError` fails.
  ☑ Test coverage
- Reverts if nonce is invalid.
  ☑ Test coverage
- Reverts if signature `imageHash` is invalid.
  ☑ Test coverage
- Reverts if signer's weight is insufficient.
  ☑ Test coverage

Inputs

- `_txs`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of transactions to be executed.
- `_nonce`:
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid, unused nonce.
  - **Impact**: Nonce used to prevent replay attacks.
- `_signature`:
  - **Control**: Arbitrary.

- **Constraints**: Must be a valid signature for the given nonce and transactions and match the configured `imageHash`.
- **Impact**: Signature authorizing transaction execution.

Function call analysis

- `ModuleNonce::_validateNonce()` validates the nonce and updates the nonce in storage to prevent signature replay.
- **What is controllable?** `_nonce`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value (reverts if nonce is invalid).
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are issues on purpose to reject invalid nonces. Reentrancy or unusual control flow are not a concern.
- `ModuleAuth::_signatureValidation()` updates the `imageHash` in the contract storage **and** updates the contract implementation to use an instance of `MainModuleUpgradable`.
- **What is controllable?** `_signature`.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are issued in some cases if signature is invalid, preventing unauthorized transaction execution. Reentrancy or unusual control flow are not a concern.
- `ModuleAuth::_execute()` executes the given transactions.
  - **What is controllable?** `_txs`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are issued if a transaction marked `revertOnError` fails. Reentrancy is possible, but it is not deemed problematic.

Function: `selfExecute`

This function can only be invoked by the wallet contract itself. No further authorization checks are performed, since a call coming from the wallet itself is considered to be trusted and coming from an authenticated flow.

Branches and code coverage

This function almost exclusively consists of an invocation of `ModuleCalls::_execute`. The test cases overlap with tests for `ModuleCalls::execute`.

Intended branches:

- Executes a transaction batch.
  ☑ Test coverage

Negative behavior:

- Caller is not self.
  ☑ Test coverage

Inputs

- `_txs`:
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of transactions to be executed.

Function call analysis

- `ModuleCalls::_execute()` executes the given transactions.
  - **What is controllable?** `_txs`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value (reverts if nonce is invalid).
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts might be issued on purpose if a transaction marked `revertOnError` fails. Direct reentrancy is not possible, since only self calls are allowed.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Ethereum or other EVM compatible layer two chains.

During our audit, we discovered one informational finding. Horizon acknowledged the finding, which did not require code changes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.