Sequence Smart Contract Wallet v3

Date	May 2025
Auditors	Heiko Fisch, Martin Ortner

1 Executive Summary

This report presents the results of our engagement with **Sequence** to review their **Smart Contract Wallet v3**.

The review was conducted in **April and May 2025**, with a total effort of 2 x 20 person-days.

Previous security reviews of related Sequence Wallet (documentation) versions include:

- Sequence Wallet v2 (February 2023)
- Arcadeum Wallet (May 2020)

Please note that findings from these earlier engagements—while not reiterated in detail here—may still be relevant to v3 of the wallet.

This report is structured into two main sections:

- Findings: Concrete issues, vulnerabilities, or improvement suggestions identified during the review.
- System Behavior and Configuration Notes: Observations and behaviors that were discussed with the development team, including cases where the team indicated that certain behaviors are intended, accepted, or configurable.

Overall, the wallet's smart contract signature decoding and verification logic allows for significant flexibility in how signatures are encoded. The system generally continues processing on invalid or unexpected data rather than failing early. While this is a deliberate design choice—on the basis that any deviation from the expected format should ultimately result in an invalid wallet configuration root—it does introduce additional degrees of freedom that could, in theory, expand the attack surface. Throughout the review, we examined many such cases and, while they may warrant further scrutiny, we were unable to identify any that resulted in practical exploits. We recommend that the development team carefully review these areas to ensure that all edge cases are properly handled.

Update June 2025: After delivery of the report, the Sequence team has taken our findings into account and provided us with a list of responses, which we included in this updated version of the report. Many of these responses reference PRs, in which the client has addressed the corresponding issue. Neither the responses themselves nor the fixes/changes in the codebase have been reviewed by us.

2 Scope

This review focused on the following repository and code revision:

- 0xsequence/wallet-contracts-v3 (`7773daaf5ec67f2e0634a8db25a870d44c041522`)

The following files are out of Scope:

File	SHA-1 hash	Reason
src/Estimator.sol	`4816cabe9d7cc7e1a050f87f7b4f8423a69d12e7`	Excluded from scope due to overall code increase since initial scoping. Client: Not a production on-chain contract.
src/Simulator.sol	`364a16bcb950a3a4efb1aa1a326d0082c3cb1f59`	Excluded from scope due to overall code increase since initial scoping. Client: Not a production on-chain contract.
src/utils/P256.sol	`e5b705e25d7d8d6479d40ac60a1b331d8d300353`	Excluded from scope due to overall code increase since initial scoping. Client: Copy of Solady Code.
src/utils/WebAuthn.sol	`d2ba38687141bfb848ff9c2c2e092aa10559b4c6`	Excluded from scope due to overall code increase since initial scoping. Client: Copy of Solady Code.
src/utils/Base64.sol	`0dc0cc046b3f5046f312bd911e22e16647cc55e4`	Excluded from scope due to overall code increase since initial scoping. Client: Copy of Solady Code.

The detailed list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the client's team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview

3.1 Wallet Factory Proxy Code Analysis

The `Factory` contract deploys new wallet proxies using the `Wallet` library's minimal proxy bytecode. Each wallet proxy stores the address of its main module and delegates all calls to it, enabling upgradable and modular wallet logic. Deployment uses `CREATE2` for deterministic addresses based on a salt and main module, supporting efficient, predictable wallet creation.

ByteCode

```
library Wallet {
  bytes internal constant creationCode =
    hex"603e600e3d39601e805130553df33d3d34601c57363d3d373d363d30545af43d82803e903d91601c57fd5bf3";
}
```

Disassembly

Step	Loc	Len	Gas	Consumed	Opcode	Instruction	Data
0	0 (0x0)	2	3	3	0x60	PUSH1	0x3e `>`
1	2 (0x2)	2	3	6	0x60	PUSH1	0x0e
2	4 (0x4)	1	2	8	0x3d	RETURNDATASIZE	
3	5 (0x5)	1	3	11	0x39	CODECOPY	
4	6 (0x6)	2	3	14	0x60	PUSH1	0x1e
5	8 (0x8)	1	3	17	0x80	DUP1	
6	9 (0x9)	1	3	20	0x51	MLOAD	
7	10 (0xa)	1	2	22	0x30	ADDRESS	
8	11 (0xb)	1	0	22	0x55	SSTORE	
9	12 (0xc)	1	2	24	0x3d	RETURNDATASIZE	
10	13 (0xd)	1	0	24	0xf3	RETURN	
11	14 (0xe)	1	2	26	0x3d	RETURNDATASIZE	
12	15 (0xf)	1	2	28	0x3d	RETURNDATASIZE	
13	16 (0x10)	1	2	30	0x34	CALLVALUE	
14	17 (0x11)	2	3	33	0x60	PUSH1	0x1c
15	19 (0x13)	1	10	43	0x57	JUMPI	
16	20 (0x14)	1	2	45	0x36	CALLDATASIZE	
17	21 (0x15)	1	2	47	0x3d	RETURNDATASIZE	
18	22 (0x16)	1	2	49	0x3d	RETURNDATASIZE	
19	23 (0x17)	1	3	52	0x37	CALLDATACOPY	
20	24 (0x18)	1	2	54	0x3d	RETURNDATASIZE	
21	25 (0x19)	1	2	56	0x36	CALLDATASIZE	
22	26 (0x1a)	1	2	58	0x3d	RETURNDATASIZE	
23	27 (0x1b)	1	2	60	0x30	ADDRESS	
24	28 (0x1c)	1	200	260	0x54	SLOAD	
25	29 (0x1d)	1	2	262	0x5a	GAS	
26	30 (0x1e)	1	700	962	0xf4	DELEGATECALL	
27	31 (0x1f)	1	2	964	0x3d	RETURNDATASIZE	
28	32 (0x20)	1	3	967	0x82	DUP3	
29	33 (0x21)	1	3	970	0x80	DUP1	
30	34 (0x22)	1	3	973	0x3e	RETURNDATACOPY	
31	35 (0x23)	1	3	976	0x90	SWAP1	
32	36 (0x24)	1	2	978	0x3d	RETURNDATASIZE	
33	37 (0x25)	1	3	981	0x91	SWAP2	
34	38 (0x26)	2	3	984	0x60	PUSH1	0x1c
35	40 (0x28)	1	10	994	0x57	JUMPI	
36	41 (0x29)	1	0	994	0xfd	REVERT	
37	42 (0x2a)	1	1	995	0x5b	JUMPDEST	
38	43 (0x2b)	1	0	995	0xf3	RETURN	

(Includes creation code)

Decompiled

Creation Code:

- The implementation is stored in slot `storage[address(this)]`.

```
storage[this] = (memory[1e]);
return memory[output.length:(output.length+1e)];
```

Runtime Code:

- This is a minimal proxy code.
- The code returns immediately for value transfers (`msg.value > 0`).
- The code `delegatecall`'s into the implementation (at `storage[address(this)]`) for `msg.value == 0`.

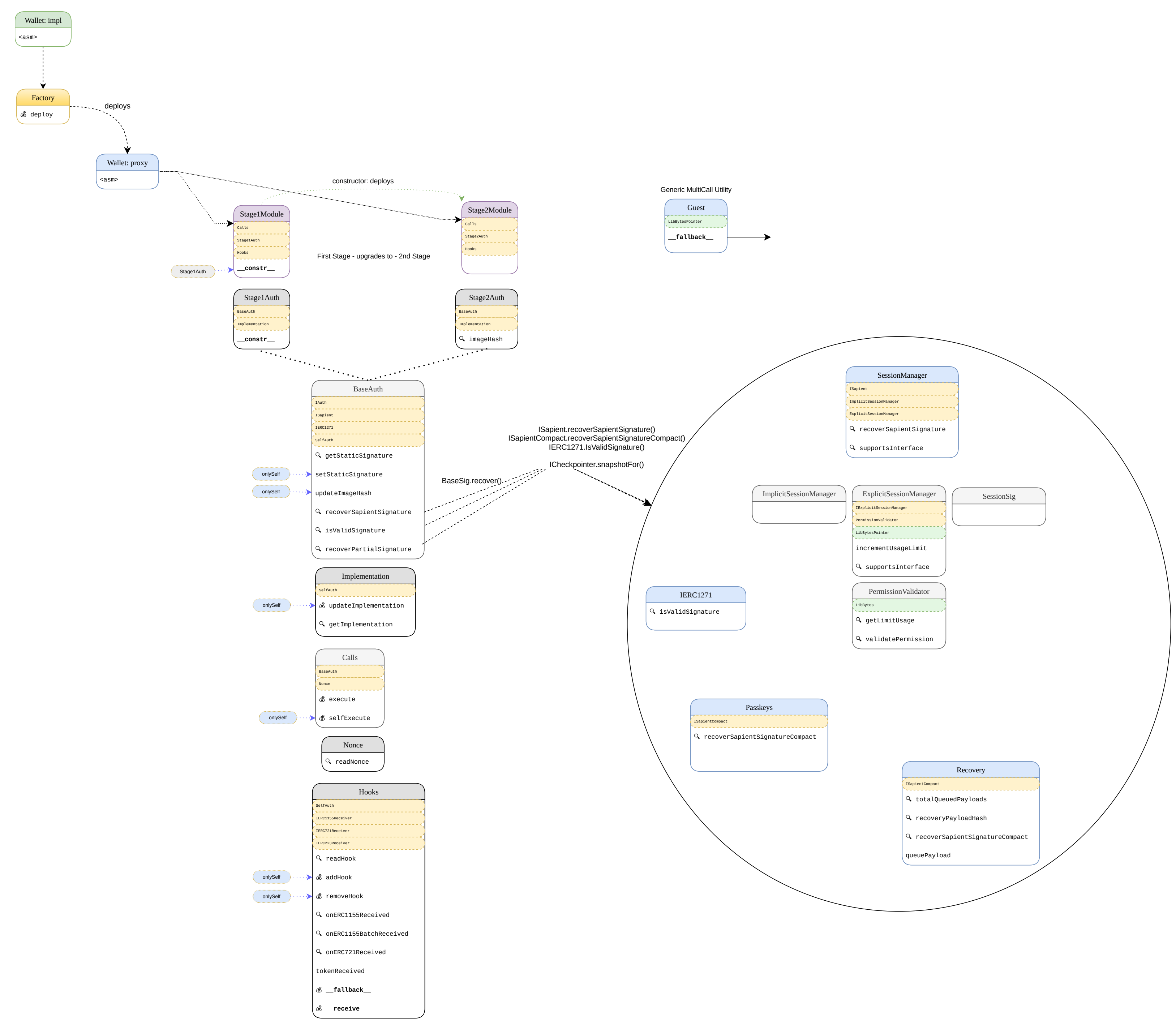
```
if msg.value {
    return memory[output.start:(output.offset+output.length)];
} else {
    if(delegatecall(gasleft(),storage[this],args.offset,msg.data.length,ret.offset,ret.length)) {
        return memory[output.offset:(output.offset+output.length)];
    } else {
        revert(memory[output.offset:(output.offset+output.length)]);
    }
}
```

Note: `payable` functions in the implementation cannot be called. Some functions are decorated `payable` nevertheless. According to the client this is a gas optimization that removes the `msg.value != 0` checks the compiler implicitly embeds for non-payable functions.

3.2 Smart Contract Architecture & Signature Components

This section describes the top-level/deployable contracts, their inheritance structure and interfaces, actors, permissions, and important contract interactions of the system under review.

Contracts are depicted as boxes. Public reachable interface methods are outlined as rows in the box. The 🔍 icon indicates that a method is declared as non-state-changing (view/pure), while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates that that contract is used in a `using for` declaration. Modifiers used as ACL are connected as yellow bubbles in front of methods.



The system has the following components:

- `Wallet.sol` - A minimal delegatecall proxy implementation written in `huff`.
- `Factory.sol` - A factory contract that deploys instances of the proxy via CREATE2.
- `Guest.sol` - A generic MultiCall type utility contract.
- Two-Stage Counterfactual Wallet Implementation - Stage 1 is initially deployed by the factory. Its implementation is `Stage1Module` with the `Stage1Auth` that verifies its `imageHash` by checking that it was deployed by the correct factory. Stage 1 can be upgraded to Stage 2 to implement `Stage2Module` which implements the Merkle Tree based wallet configuration and signature verification via `Stage2Auth`.
- Wallet Auxiliary Modules - Signature programs can call auxiliary modules like ERC-1271 Smart Contract Signature Validation, and `ISapient` interfaces of `SessionManager.sol`, `Passkeys.sol`, and `Recovery.sol`.

The actual signatures can be a complex program of sequential, chained, and recursive instructions. Here's a quick overview on signature types:

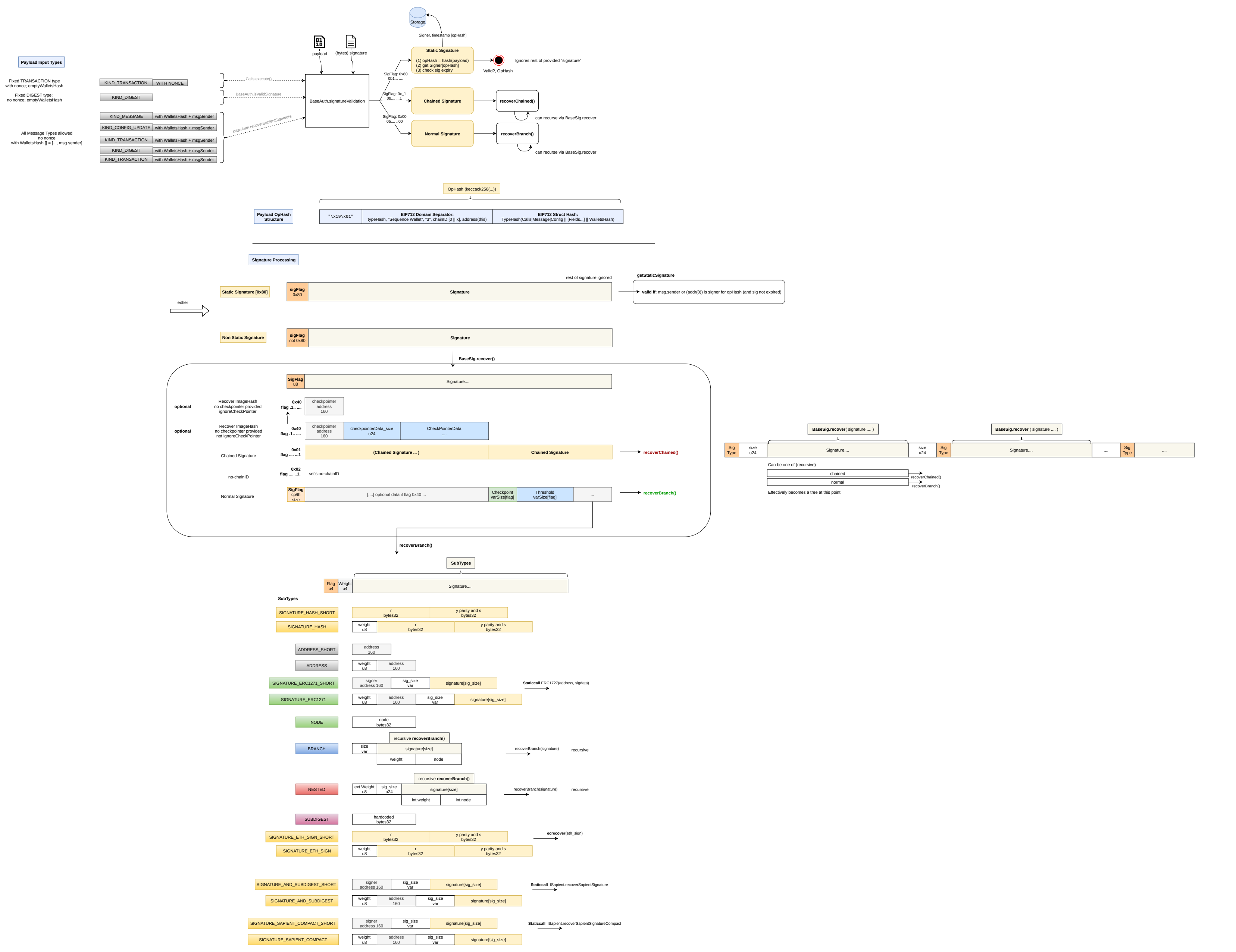
- `SIGNATURE_HASH` - Standard ECDSA signature (ERC-2098 compact). Recovers an address from the signature and adds its weight to the total.
- `ADDRESS` - Includes an address and weight directly, without a signature. Does not add weight to the total (used for Merkle tree structure).
- `ERC-1271` - Calls an external contract implementing ERC-1271 to verify the signature. If valid, adds the specified weight to the total.
- `NODE` - Includes a node hash for Merkle tree structure. Does not add weight.
- `BRANCH` - Recursively processes a branch of the signature Merkle tree, accumulating weight from its contents.
- `NESTED` - Processes a nested branch with its own threshold and external weight. If the internal threshold is met, adds the external weight to the total.
- `SUBDIGEST` - Accepts a hardcoded digest. If it matches the operation hash, sets the weight to the maximum possible value.
- `ETH_SIGN` - Standard Ethereum signed message (`eth_sign`). Recovers an address and adds its weight to the total.
- `SIGNATURE_ANY_ADDRESS_SUBDIGEST` - Accepts a hardcoded digest for any address. If it matches, sets the weight to the maximum possible value.
- `SIGNATURE_SAPIENT` - Calls an external contract implementing `ISapient` to verify a Sapient signature. Adds the specified weight to the total.
- `SIGNATURE_SAPIENT_COMPACT` - Calls an external contract implementing `ISapientCompact` to verify a compact Sapient signature. Adds the specified weight to the total.

The system verifies signatures using a Merkle tree structure, where each signature or signer node contributes a specific weight. To authorize an operation, the total accumulated weight from valid signatures must meet or exceed a defined threshold. Each signature type (e.g., `ECDSA`, `ERC1271`, `Sapient`) is parsed and validated according to its flag, and only valid signatures add their assigned weight. The verification process ensures that the sum of signer weights is at least the required threshold, providing flexible multi-signature and policy enforcement.

There are three main types of signatures:

- **Normal Signature** - A standard signature tree where the provided signatures and their weights are validated against a single threshold. The operation is authorized if the total weight of valid signatures meets or exceeds the threshold. Used for typical multi-signature verification.
- **Chained Signature** - A sequence of signatures, each with its own threshold and checkpoint, processed in order. Each signature in the chain must independently meet its threshold, and checkpoints must decrease monotonically. This allows for off-chain configuration updates that don't have to be manifested on-chain immediately.
- **Static Signature** - A hardcoded digest (subdigest) that, if matched to the operation hash, immediately sets the signature weight to the maximum possible value. This acts as an always-accepted override, bypassing normal weight accumulation and threshold checks. Used for special cases like emergency access or pre-approved operations.

Signature types and signature components are depicted in the following diagram. Starting at the top-left, we see the main validation function in `BaseAuth.signatureValidation` and flows that call this function.



The system supports multiple payload types, each identified by a `kind` field in the `Payload` structure. These types determine how the payload is interpreted and validated during signature verification:

- `KIND_TRANSACTION`
  Represents a standard transaction payload, including all necessary fields for executing on-chain actions. Used for validating signatures on regular contract calls and value transfers.
- `KIND_DIGEST`
  Encapsulates a precomputed digest, typically used for off-chain signature validation or meta-transactions. Allows signatures to be verified against a hash rather than the full transaction data.
- `KIND_MESSAGE`
  Used for arbitrary signed messages, such as user authentication or off-chain approvals. Ensures that signatures are bound to a specific message context, preventing replay in other contexts.
- `KIND_CONFIG_UPDATE`
  Special payload for configuration changes, such as updating signers, thresholds, or other wallet/module settings. Used in chained signatures.

Each payload type enforces context-specific validation rules, ensuring signatures are only valid for their intended operation and reducing the risk of replay or misuse across different contexts.

4 Security Specification

4.1 Actors

The relevant actors are listed below with their respective abilities:

- The *factory contract* creates new instances of the wallet on behalf of users.
- *Wallet owners* can send signed transactions to wallet instances that are executed if the preconfigured threshold is met.

Owners can perform the following actions: execute transactions, change the implementation, add/remove function hooks that delegate execution to external code, and change the configuration (e.g., threshold, owner addresses and weights).

4.2 Trust Model

The Sequence Wallet contracts do not include any mechanisms which would require trust in a centralized administrator type of role. All actions, including code and configuration updates, are performed by the wallet owners via multisig transactions. Hence, the wallet contract system is entirely trustless from the view of the user.

Since users have complete control, they're also fully responsible and must be extremely careful what transactions they sign. While this is trivially and generally true, the following points deserve extra mention:

- Implementation upgrades switch the code that is executed in the context of the wallet instance. Changing the implementation to a flawed contract or wrong address can result in the loss of funds or "brick" the wallet.

- Hooks allow the addition of code to the core functionality of the wallet. Installing a flawed or even malicious hook can put all funds at risk.
- As mentioned above, a wallet instance's configuration is not stored on-chain explicitly; instead, a Merkle tree is built and only its root is stored in the wallet's storage. This is an elegant and efficient design, but it also comes with more risks than the traditional approach of storing the owners explicitly:
  (1) It makes the signature verification process more involved, which means there is a higher risk of bugs in the implementation. This could allow the execution of transactions that, in reality, haven't been authorized, and/or it could prevent the execution of transactions that *have* been authorized.
  (2) Changing the configuration means just storing a new Merkle root. Building the tree and computing its root is an off-chain process, though. If this code has bugs or gets compromised, funds could get stolen or frozen. In theory, users don't have to trust this off-chain code and can verify the Merkle root (or even compute it themselves) before they sign the corresponding transaction. In practice, this is hardly feasible, and many users will lack the time and/or skills to do that, so they will likely opt to trust this code. It should be noted that off-chain code, in particular the code that computes the Merkle root for a given configuration, has been not in scope for this engagement.
  (3) Finally, the "raw data" of the configuration must not be lost or forgotten. Unlike an explicitly stored list of owners that can always be recovered from on-chain data, this is not possible if we store just a hash of the configuration. If – perhaps years after setting a particular configuration – even a small part of it (such as one of the owner addresses) has been forgotten, all funds will be frozen because the witness for the Merkle tree can't be reconstructed. Hence, a good backup strategy for this data is essential.

5 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- Major 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.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

5.1 Session Signers Can Spend the Allowed Amount of the Native Asset Repeatedly Major

Resolution

Sequence team's response:

We recognize the issue and we implemented the proposed fix. The PR for addressing the issue can be found here: 0xsequence/wallet-contracts-v3#41

Description

The `SessionPermissions` struct has a member `valueLimit` that defines how much of the native asset the `signer` is allowed to spend.

src/extensions/sessions/explicit/IExplicitSessionManager.sol:L6-L16

```
/// @notice Permissions configuration for a specific session signer
struct SessionPermissions {
  /// @notice Address of the session signer these permissions apply to
  address signer;
  /// @notice Maximum native token value this signer can send
  uint256 valueLimit;
  /// @notice Deadline for the session. (0 = no deadline)
  uint256 deadline;
  /// @notice Array of encoded permissions granted to this signer
  Permission[] permissions;
}
```

That value is to be understood *cumulatively*, a concept that is also used for values in parameter rules. Here, it means the signer should not be able to spend this amount over and over again, in different transactions; instead, this is the *total* amount that the signer can spend.

In order to achieve this, the value already spent is tracked when a sequence of calls is processed

src/extensions/sessions/explicit/ExplicitSessionManager.sol:L98-L101

```
// Increment the total value used
if (call.value > 0) {
  sessionUsageLimits.totalValueUsed += call.value;
}
```

and the `_validateLimitUsageIncrement` function finally ensures that the last call in the array writes the total amount spent to storage.

This value should be picked up the next time to initialize the tracking value, but this doesn't happen; the tracking value is initialized to zero:

src/extensions/sessions/SessionManager.sol:L67-L70

```
// Initialize new session usage limits
limits.signer = callSignature.sessionSigner;
limits.limits = new UsageLimit[](0);
limits.totalValueUsed = 0;
```

As a result, the `valueLimit` can be spent repeatedly by the signer, not only once as intended.

Recommendation

In the last line of the last code excerpt above, the initialization value for `limits.totalValueUsed` should be read from storage.

Moreover, a test for the described scenario should be added, i.e., trying to spend more than `valueLimit` of the native asset over the course of two transactions should fail.

5.2 Mind Resource Limitations for Signature Verification Medium

Resolution

Sequence team's response:

We implemented tests to determine the limit depth configuration that a Sequence wallet contract can safely handle. We determined that a Sequence wallet can handle a tree of 54 layers before it fails due to stack constraints.

This limit is more than enough for the balanced trees that are used in production.

But to be safe, we implemented sanity checks at the SDK level. These sanity checks compute the highest depth of any configuration and throw when a wallet is attempted to be created with such an impossible-to-verify configuration.

The PR with the added tests to determine the depth limits is: 0xsequence/wallet-contracts-v3#52

The PR that adds the sanity checks on the SDK is: 0xsequence/sequence.js#773

Description

Verifying a top-level signature for a Sequence v3 wallet is a complicated endeavor; in fact, most of the codebase that is in scope for this review is dedicated to this task. Essentially, a *configuration* for a wallet is a Merkle tree. The wallet stores only the root of this tree, and a signature is valid if the derived root equals the stored one. Leaves can store signer addresses that each contribute a certain weight to the signature. There are several bells and whistles on top of that multisig functionality. For instance, some leaves of this tree can themselves be roots of nested Merkle trees, i.e., the entire nested tree contributes some weight to the main tree, allowing for multisig-in-multisig functionality. Another feature are session signatures, which form an entire sub-configuration that also has a tree structure, whose root contributes a leaf with some weight to the main tree. For simplicity, we'll only talk about nested trees in the following, but the situation is the same for other kinds of trees that could be located at the leaf of another tree.

While the height of a balanced binary tree is logarithmic in the number of its leaves, the verification of a signature can be still be a resource-intensive task for a sufficiently big configuration, especially since there is, in general, not only a single Merkle branch involved, but we may have to combine and prove several branches that contribute weight. In the worst case, the entire tree could have to be reconstructed during a signature verification, altough that is certainly not a typical scenario. For this computation, recursion is utilized, and this happens in two different ways: First of all, *inside* a tree, whenever a right subtree has to be explored, this is achieved via recursion. Secondly, recursion is also used when a leaf of a tree is itself a tree.

It is important to keep in mind that signature verification can, at least in theory, fail due to limited resources. This may happen in selected, individual verification paths, but in the worst case, it could affect *every* path and "brick" the wallet. The two resources to consider in this context are:

1. Gas. It is well known that transactions may only consume a limited amount of gas. In particular, a transaction can't be included if it would use more gas than the block gas limit.
2. Stack size. The EVM's stack size is 1024 slots. There's generally less awareness for this limitation than for gas, and indeed, it rarely has to be taken into account. The reason is that for contracts without recursion, the maximum attainable stack depth is independent of the input size, i.e., it depends only on the code and not on the input for the code. So in practice, this is usually not a concern. With recursion, however, and the "right" input, it is possible to exhaust the stack, so in our case, that is another important resource limitation to be kept in mind.

Recommendation

This is an off-chain topic; when the stored Merkle root is updated, the contract has no way of knowing the size(s) of the tree(s) and the amount of gas that is required in the worst case to arrive at this root. Hence, it is an off-chain duty to keep the trees and configurations sufficiently small such that signature verification is not prevented by the mentioned resource limitations. We recommend extensive experiments and erring on the side of caution in that regard.

A more practical side remark: It might be beneficial to craft the configurations deliberately in a way that keeps the stack depth small. For instance, as discussed above, only *right* subtrees are visited recursively. It would therefore make sense to place nested trees in leaves that minimize the number of "right turns" we have to take to reach the corresponding leaf from the root.

For example, assume the main tree is a full binary tree of height 10, and assume we have one nested tree, also of height 10. We can place this nested tree at whatever leaf of the main tree we want. If we use the rightmost leaf for the nested tree, then the maximum call depth is 20. However, if we place the nested tree at the leftmost leaf, we won't recurse at all to reach this leaf, so the maximum call depth is only 11 – without any loss or of functionality.

Remarks

1. As mentioned above, only *right* subtrees are explored recursively. That is not only a convention but even enforced by the contracts; recursing into a *left* subtree is not possible in `BaseSig` and `Recovery`. In `SessionSig`, however, it *is* possible. Since this is unlikely to be used in practice, this could be streamlined, and the code could follow the same pattern as in `BaseSig` and `Recovery`.
2. This finding is more a word of caution than an issue with the contracts.

5.3 Missing Possibility to Reissue a Cumulative Permission Medium

Resolution

Sequence team's response:

We acknowledge this issue and we propose generating a new session key if the need to replenish a permission arises. Future iterations of the smart sessions extension will improve the handling of this scenario.

Description

Assume Alice issues a permission that allows Bob to spend 100 USDC cumulatively. When Bob has spent it all, Alice would like to grant him the same permission again, i.e., Bob should be able to spend another 100 USDC. Currently, that's not possible – at least not without jumping through some hoops. The reason is that this new permission would be the exact same as the original one and therefore, the hash value would be the same too. Hence, the already accumulated spending would be picked up again for the accounting, and Bob wouldn't be able to spend more than he already has.

There are some hacky workarounds for this situation: For instance, Alice could make an unsubstantial modification to the permission like allowing Bob to spend only 99.99 USDC instead of 100. Or she could include an additional rule that's always fulfilled (e.g., `mask` and `value` are `0`, `operation` is `EQUAL`). Or she could duplicate the rule for the selector, etc. All these measures would change the permission and therefore its hash value – with the result that a "new cumulation" starts. But such workarounds would be inelegant and cumbersome.

Recommendation

A clean solution would be to include some sort of nonce (in the sense of a "distinguisher of last resort") that should be included when hashing. That would allow Alice to just use a different nonce for the new permission. It would probably be best to make this nonce a member of the `SessionPermissions` struct (vs. the `Permission` struct) and also include it for the `usageHash` of the native asset (although there are other options).

5.4 SessionManager - Session Usage Limits Entry Created for Zero Address on Invalid Signature Medium

Resolution

Sequence team's response:

We consider this issue minor, as there is no valid reason for a smart session configuration to include the `address(0)` as a session key.

That being said, we still decided to address this issue directly by checking that `ecrecover` does not fail (does not return `address(0)`).

The PR adding the additional check is: [0xsequence/wallet-contracts-v3#39]

Description

In `SessionManager`, when processing session usage limits, if a session signature verification fails (i.e., `ecrecover` returns `address(0)`), the code still initializes a session usage limit entry with the zero address. While this could theoretically result in all invalid signatures sharing the same usage tracking entry, this situation will later revert in `_validateExplicitCall`, which checks that the session signer is valid and not the zero address.

Example

Here, `limits.signer = callSignature.sessionSigner` would be `address(0)`.

src/extensions/sessions/SessionSig.sol:L149-L157

```
{
  bytes32 r;
  bytes32 s;
  uint8 v;
  (r, s, v, pointer) = encodedSignature.readRSVCompact(pointer);

  bytes32 callHash = hashCallWithReplayProtection(payload.calls[i], payload);
  callSignature.sessionSigner = ecrecover(callHash, v, r, s);
}
```

src/extensions/sessions/SessionManager.sol:L65-L76

```
for (limitsIdx = 0; limitsIdx < sessionUsageLimits.length; limitsIdx++) {
  if (sessionUsageLimits[limitsIdx].signer == address(0)) {
    // Initialize new session usage limits
    limits.signer = callSignature.sessionSigner;
    limits.limits = new UsageLimit[](0);
    limits.totalValueUsed = 0;
    break;
  }
  if (sessionUsageLimits[limitsIdx].signer == callSignature.sessionSigner) {
    limits = sessionUsageLimits[limitsIdx];
    break;
  }
```

This will later revert downstream in `_validateExplicitCall` which is not immediately obvious.

src/extensions/sessions/explicit/ExplicitSessionManager.sol:L42-L61

```
function _validateExplicitCall(
  Payload.Decoded calldata payload,
  uint256 callIdx,
  address wallet,
  address sessionSigner,
  SessionPermissions[] memory allSessionPermissions,
  uint8 permissionIdx,
  SessionUsageLimits memory sessionUsageLimits
) internal view returns (SessionUsageLimits memory newSessionUsageLimits) {
  // Find the permissions for the given session signer
  SessionPermissions memory sessionPermissions;
  for (uint256 i = 0; i < allSessionPermissions.length; i++) {
    if (allSessionPermissions[i].signer == sessionSigner) {
      sessionPermissions = allSessionPermissions[i];
      break;
    }
  }
  if (sessionPermissions.signer == address(0)) {
    revert SessionErrors.InvalidSessionSigner(sessionSigner);
  }
```

Recommendation

Relying on this pattern of not reverting on signature recovery immediately is not ideal. Using or initializing state for an error-recovered address (such as `address(0)`) should be avoided entirely, as it can lead to subtle bugs, unnecessary gas usage, and complicates reasoning about the code.

Add explicit zero address validation immediately after signature recovery to prevent reliance on downstream validations. Add validation to reject zero address session signers before initializing usage limits.

5.5 Hashing an Array of Calls Incurs Quadratic Memory Consumption Medium

Resolution

Sequence team's response:

We acknowledge the issue, but we consider it a minor issue, since the number of transactions within a bundle tends to be a rather low number. It is unlikely that the quadratic memory consumption will cause gas usage to spike too much.

That being said, we still decided to address this issue by following the recommended approach: 0xsequence/wallet-contracts-v3#37

Description

The `hashCalls` function in the `Payload` library hashes an array of calls (where `Call` is a struct type) in the way mandated by EIP-712, which means each individual struct is hashed first, and then the concatenation of the resulting `bytes32` values is hashed again:

src/modules/Payload.sol:L194-L205

```
function hashCalls(
  Call[] memory calls
) internal pure returns (bytes32) {
  // In EIP712, an array is often hashed as the keccak256 of the concatenated
  // hashes of each item. So we hash each Call, pack them, and hash again.
  bytes memory encoded;
  for (uint256 i = 0; i < calls.length; i++) {
    bytes32 callHash = hashCall(calls[i]);
    encoded = abi.encodePacked(encoded, callHash);
  }
  return keccak256(encoded);
}
```

This implementation is correct, but its memory consumption is quadratic in the number of calls. Specifically, the line

src/modules/Payload.sol:L202

```
encoded = abi.encodePacked(encoded, callHash);
```

in the loop first *copies* the current `encoded` array, appends the new `bytes32` value, and then takes this array as the new value for `encoded`. In each iteration, the copied array is 32 bytes larger than in the previous round. More precisely, in iteration i, the `abi.encodePacked` call creates a new array of length 32 * (i + 1) bytes. Summing the lengths of all these arrays up, we arrive at 16 * (n^2 + n) bytes, where n is the length of the array of calls. Here, we've employed the well-known Gauss summation formula, which says that the sum of the natural numbers from 1 to n is n * (n+1) / 2. Hence, the memory consumption of this function is (at least) quadratic. (Note that for simplicity, we have ignored the length fields of these arrays, which add an additional 32 * n bytes, but that doesn't change the big picture – which is quadratic growth.)

Moreover, the gas cost of memory usage is itself a quadratic function. While the constants in the formula ensure that reasonable amounts of memory remain cheap, *quadratic* use of memory quickly catapults us out of this range.

In practice, it might be the case that the number of calls we want to hash is almost always *very* small (less than 10, maybe, so the effects of quadratic memory consumption might not become dramatic or even noticeable. Nevertheless – and especially as other gas optimization opportunities have been seized – a reimplementation of `hashCalls` that uses only a linear amount of memory would be clearly superior even for a moderate number of calls and allow hashing a higher number without becoming prohibitively expensive.

Recommendation

We recommend an implementation of `hashCalls` with linear memory consumption:

1. Allocate a `bytes32` memory array `callHashes` of length `calls.length`.
2. Populate the array with the hashes of the calls.
3. As final step, return `keccak256(abi.encodePacked(callHashes))`.

The `abi.encodePacked` in the last step will make a copy of the `callHashes` array, but the total memory consumption is still linear. Alternatively, step 3 could utilize assembly and hash the `callHashes` array directly, thus avoiding the copy.

Remark

The severity of this finding depends on what number of calls can be reasonably expected in practice.

5.6 Wrong Magic Value for ERC-1271's `isValidSignature` Medium

Resolution

Sequence team's response:

We acknowledge the issue and we consider its assigned severity of `medium` an understatement, as if left unnoticed this issue may lead to wallets being bricked in scenarios where the Sequence wallet is being used as a nested signer.

We fixed the contracts in the following PR: 0xsequence/wallet-contracts-v3#40

Description

The well-known ERC-1271 standard proposes a method for contracts to signal whether they deem a signature valid or not. More specifically, contracts adhering to this standard have to implement a function `isValidSignature`, which takes a hash and a signature and, if the signature is considered valid, returns a magic value. This magic value is the selector of said function: `bytes4(keccak256("isValidSignature(bytes32,bytes)"))` – which evaluates to `0x1626ba7e`.

However, the codebase uses the wrong magic value `0x20c13b0b`, which happens to be the selector of a similar function that takes as input the full data (`bytes`) instead of its hash (`bytes32`). The wrong magic value appears in two files: `IERC1271.sol` and `BaseAuth.sol`:

src/modules/interfaces/IERC1271.sol:L4-L19

```
bytes4 constant IERC1271_MAGIC_VALUE = 0x20c13b0b;

interface IERC1271 {

  /**
   * @notice Verifies whether the provided signature is valid with respect to the provided hash
   * @dev MUST return the correct magic value if the signature provided is valid for the provided hash
   *   > The bytes4 magic value to return when signature is valid is 0x20c13b0b : bytes4(keccak256("isValidSignature(bytes,bytes)")
   *   > This function MAY modify Ethereum's state
   * @param _hash       keccak256 hash that was signed
   * @param _signature  Signature byte array associated with _data
   * @return magicValue Magic value 0x20c13b0b if the signature is valid and 0x0 otherwise
   */
  function isValidSignature(bytes32 _hash, bytes calldata _signature) external view returns (bytes4 magicValue);

}
```

Note that the constant as well as the NatSpec annotation for `isValidSignature` is wrong. The constant defined here is never actally used, though. Instead, `BaseAuth` hardcodes this constant too:

src/modules/auth/BaseAuth.sol:L121-L130

```
function isValidSignature(bytes32 _hash, bytes calldata _signature) external view returns (bytes4) {
  Payload.Decoded memory payload = Payload.fromDigest(_hash);

  (bool isValid,) = signatureValidation(payload, _signature);
  if (!isValid) {
    return bytes4(0);
  }

  return bytes4(0x20c13b0b);
}
```

In contrast, the documentation mentions the correct magic:

docs/SIGNATURE.md:L193-L200

```
### 5.3 **Signature ERC-1271** (`flag = 2`)

- The free nibble bits are used as:
  - The bottom two bits are the weight (with the same "0 => dynamic read, else 1..3" logic).
  - The next two bits define the size of the "signature size" field: 0..3 means we read 0..3 bytes to get the dynamic length of the ı
- Then we read 20 bytes for the contract address, read that dynamic-size signature, and call `IERC1271(addr).isValidSignature(_opHasl
- Weight is added if valid.
```

Recommendation

We recommend the following steps:

1. In `IERC1271.sol`, change the constant `0x20c13b0b` to `0x1626ba7e`. Correct the NatSpec annotation for `isValidSignature` as well. (Note that the statement `This function MAY modify Ethereum's state` is not true either, since the function is `view`.)
2. In `BaseAuth`, don't hardcode a magic value at all, and use the (fixed) constant from the (already inherited) `IERC1271` interface instead. Instead of hardcoding the return value `bytes4(0)` for an invalid signature, consider introducing a second symbolic constant for the failure case in `IERC1271`.

5.7 Masking-Related and Various Minor Points in `LibBytes*` Medium

Resolution

Sequence team's response:

We acknowledge the issue and we implemented the series of recommended fixes in the following PR: 0xsequence/wallet-contracts-v3#43

Description

The libraries `LibBytesPointer` and `LibBytes` provide various low-level functions to read raw bytes from calldata (or, occasionally, memory) and make this data accessible for further processing via regular typed values. A typical pattern for these functions is to read one or more words from calldata, shift as appropriate, and mask if necessary.

In one case, the masking is missing:

src/utils/LibBytesPointer.sol:L109-L114

```
function readBytes4(bytes calldata _data, uint256 _pointer) internal pure returns (bytes4 a, uint256 newPointer) {
  assembly {
    a := calldataload(add(_pointer, _data.offset))
    newPointer := add(_pointer, 4)
  }
}
```

The 28 lower-order bytes of `a` can be dirty and should be cleared.

In several other functions in `LibBytesPointer`, a mask is applied needlessly because it won't change the value:

- `readAddress`
- `readUint16`
- `readUint24`
- `readUint64`
- `readUint160`

In these functions, the bits that are masked off to zero just came in through a `shr` and, therefore, are zero already.

Recommendation

Masking should be added to `LibBytesPointer.readBytes4`, as discussed above. While the needlessly applied masks don't hurt and are cheap, they are no-ops and can be removed. Moreover, very similar functions in `LibBytesPointer` (e.g., `readUint8`) don't apply a mask, so removing the unnecessary masks would also improve readability by making the code more consistent. Note that the mask in `readUint8Address` has to be kept, since it clears a byte that wasn't shifted in.

There are several stylistic points in `LibBytesPointer` and `LibBytes` that could be addressed too:

1. The functions in `LibBytesPointer` have no NatSpec annotations; `LibBytes` is only partially annotated.

2. The ordering of the functions in both libraries follows no discernible pattern; rearranging the functions would make the files better readable and navigatable.
3. The `readRSVCompact` functions in both libraries as well as `readMRSVCompact` in `LibBytes` each have two unncessary explicit conversions of `yParityAndS` to `uint256`; these can be removed for better readability.

5.8 Guest - Unsafe Fallback Pattern Medium

Resolution

Sequence team's response:

The `Guest` contract holds no privileges on a Sequence wallet system. It serves the exclusive purpose of facilitating batching of calls.

Based on this context, we decided to make the `Guest` contract payable to reduce the gas usage of validating these conditions. We are ignoring any complex reentrancy scenarios, as the `Guest` contract is not meant to handle or hold balances in production.

The PR that makes `Guest` payable is: 0xsequence/wallet-contracts-v3#53

Description

The `Guest` contract is basically a MultiCall-type utility contract that can be used by anyone for any purpose. According to the developers it does not hold any special privileges in the system and is merely a utility contract.

The contract uses it's fallback function to handle bundled calls which introduces several concerns:

1. The fallback is non-payable, causing any value transfers in the call bundles to revert. The encoded calls allow to specify a `call.value` which will revert if the value is non-zero.
2. If made payable, complex reentrancy scenarios could occur:
   - External calls could re-enter the dispatch logic during execution
   - Malicious contracts could exploit this to drain funds if the contract holds a balance
3. There's no clear separation between handling regular fund transfers and bundle execution. A value transfer might trigger the fallback which execute the dispatcher (although it might just return due to empty calldata resolving to no calls for the for loop). `Payload.fromPackedCalls(msg.data)` likely decodes just fine because it is reading from calldata, returning an empty bundle struct.
4. No mechanism exists to verify if all transferred funds are properly handled by the end of bundle execution

Example

- All fallback invocations are dispatched

src/Guest.sol:L17-L28

```
fallback() external {
  Payload.Decoded memory decoded = Payload.fromPackedCalls(msg.data);
  bytes32 opHash = Payload.hash(decoded);
  _dispatchGuest(decoded, opHash);
}

function _dispatchGuest(Payload.Decoded memory _decoded, bytes32 _opHash) internal {
  bool errorFlag = false;

  uint256 numCalls = _decoded.calls.length;
  for (uint256 i = 0; i < numCalls; i++) {
    Payload.Call memory call = _decoded.calls[i];
```

- `call.value` but contract cannot hold `balance`

src/Guest.sol:L49-L52

```
bool success = LibOptim.call(call.to, call.value, gasLimit == 0 ? gasleft() : gasLimit, call.data);
if (!success) {
  if (call.behaviorOnError == Payload.BEHAVIOR_IGNORE_ERROR) {
    errorFlag = true;
```

Recommendation

There are two ways forward:

1. Disallow value transfers: `call.value != 0` will lead to reverts. The utility cannot be used in calls that return native token `ETH`. Reentrancy (direct or from a callee) must be considered. Generally less complexity, safer to use.
2. Allow value transfers: `call.value != 0` may succeed if enough funds are available in the contract. More complex reentrancy issues (caller of a bundle might reenter; bundle might reenter directly). More complex accounting of funds. The contract should ensure that at the end of the bundle no funds stay left in the contract as they can easily be stolen by anyone.

The recommendations highly depend on how this contract is used in the system. Lack of specification makes it non-trivial to suggest good improvements. In general we recommend the less complex use-case w/o allowing value transfers.

1. Replace fallback with explicit function for bundle execution similar to the typical MultiCall utilities.
2. Add separate receive/fallback functions to handle plain ETH transfers if you want to allow value transfers

3. Implement reentrancy protection for the MultiCall function unless you explicitly want to allow this (complex from a security perspective, can be problematic)
4. Add accounting to track funds through bundle execution
5. Consider adding explicit access control for bundle execution

5.9 Proxy Doesn’t Pass Calls With Non-Zero Value and Non-Empty Calldata to Implementation Minor

Resolution

Sequence team’s response:

We acknowledge that this is a serious limitation of the proxy contract, and that this limitation can’t be “corrected” by a future upgrade, since the limitation resides in the proxy contract itself.

To better future-proof the wallet addresses, we have added an additional condition to the proxy contract: if a call to it has value and data, then the call is still forwarded to the implementation.

The changes can be found in the following PR: 0xsequence/wallet-contracts-v3#51

Description

The proxy contract used for the wallets directly accepts calls with non-zero value – without delegatecalling to the implementation contract. The intention here is to avoid problems with contracts that use Solidity’s `transfer` function to send the native asset. However, even if calldata is non-empty (which can’t happen with `transfer`), the proxy contract returns immediately, without forwarding the call to the implementation contract.

Currently, there is no function on the implementation contracts that has to be payable, i.e., expects a non-zero amount of the native asset. (To be clear, some functions are marked as `payable`, but that is merely a gas optimization, because – as a consequence of the preceding discussion - verifying that `msg.value` is zero is pointless on the implementation.) Moreover, we are not able to come up with a plausible scenario where a truly payable function would be needed. Nevertheless, not seeing a usecase now might be deemed insufficient for a commitment to never wanting to including such a function in an implementation contract upgrade.

Recommendation

Reassess whether you’re comfortable with the proxy returning immediately for calls with non-zero value even if calldata is non-empty. If you’re not, the proxy code should be changed to return only if value is non-zero *and* calldata empty. This would add a few more opcodes to the proxy contract and make deployments slightly more expensive.

5.10 ERC-1271 Signatures May Fail Unexpectedly and Revert the Transaction Minor

Resolution

Sequence team’s response:

We acknowledge the inconsistency between how ECDSA and ERC-1271 signers are being handled, but we do not consider this to be an issue, as any failing signature due to an invalid ERC-1271 signer can be re-encoded with such a signer, assuming the signature can reach the threshold by other means.

Description

A top-level signature for a Sequence v3 wallet can be comprised of several elementary signatures, such as an ECDSA signature from an EOA or an ERC-1271 signature to be verified by a contract. The validity of an ECDSA signature doesn’t change; in particular, it is independent of the blockchain state. ERC-1271 signatures, however, have to be verified by a contract, and this contract’s behavior may very well take its or other contracts' state into account, as well as the overall blockchain state such as blocktime or -number. Therefore, it is unpredictable whether the verification of an ERC-1271 signature will succeed. It should also be noted that a state change that invalidates such a signature could happen via front-running.

If an invalid ERC-1271 signature is encountered, the transaction reverts:

src/modules/auth/BaseSig.sol:L284-L287

```
// Call the ERC1271 contract to check if the signature is valid
if (IERC1271(addr).isValidSignature(_opHash, _signature[rindex:nrindex]) != IERC1271_MAGIC_VALUE) {
  revert InvalidERC1271Signature(_opHash, addr, _signature[rindex:nrindex]);
}
```

To assess the potential problems of this situation, let’s assume that we have a multisig setup with a certain threshold and several EOA and contract signers. Let’s also assume that we include Eve’s ERC-1271 signature – which we have checked to be valid. But Eve will invalidate the signature via front-running, so the transaction reverts. We now have to find a threshold-sized subset of signers without Eve. If we can’t find such a set, well, then we were never going to succeed anyway, and Eve could have just not signed in the first place, instead of revoking her signature later. So the main problem is the surprise element; the top-level signature – and hence the transaction – fails *unexpectedly*, causing delays and possibly confusion. This is particularly bad if time is of the essence, which can be the case in some situations.

Recommendation

Apart from peferring EOA signatures over contract signers whenever possible, the following approach could be a mitigation: If an ERC-1271 signature fails, don't make the entire top-level signature fail, but just don't count the weight of the failed signature. This allows the inclusion of "extra weight" (i.e., more signatures) in a top-level signature, such that a single or even a few rogue contract signers can't make the entire transaction fail.

Two other points have to be considered for this to work:

1. The `isValidSignature` call on the ERC-1271-compatible contract could revert itself. Such a revert would have to be caught in `BaseSig`.
2. The ERC-1271-compatible contract could just consume all gas it was given. Hence, the amount of gas forwarded to the contract signer would have to be limited to a relatively small amount that would allow the computation in `BaseSig` to proceed normally even if used up completely in the `isValidSignature` call. This is not an ideal or even a very clean solution. In fact, the ERC specification itself mentions that limiting the amount of gas might cause some signature validations to fail due to insufficient gas, even though the signature would be considered valid normally.

So this approach could mitigate the problem, but it's not a perfect solution. If no changes in the code will be made with respect to this finding, it is – at the very least – something to be aware of and to keep in mind, especially in time-critical situations.

5.11 ERC-165-related Shortcomings Minor

Resolution

Sequence team's response:

We have decided to remove any support for ERC-165 from the contracts. 0xsequence/wallet-contracts-v3#44

Description

The codebase doesn't deal a lot with ERC-165. In fact, only two contracts implement a `supportsInterface` function: `ExplicitSessionManager` and `SessionManager`.

src/extensions/sessions/explicit/ExplicitSessionManager.sol:L160-L164

```
function supportsInterface(
  bytes4 interfaceId
) public pure virtual returns (bool) {
  return interfaceId == type(ISapient).interfaceId || interfaceId == type(IExplicitSessionManager).interfaceId;
}
```

src/extensions/sessions/SessionManager.sol:L117-L121

```
function supportsInterface(
  bytes4 interfaceId
) public pure virtual override returns (bool) {
  return interfaceId == type(ISapient).interfaceId || super.supportsInterface(interfaceId);
}
```

Notably, `SessionManager` inherits from `ExplicitSessionManager`:

src/extensions/sessions/SessionManager.sol:L21

```
contract SessionManager is ISapient, ImplicitSessionManager, ExplicitSessionManager {
```

There are two shortcomings in the code above:

A. The `ISapient` interface contains only a single function:

src/modules/interfaces/ISapient.sol:L12-L20

```
interface ISapient {

  /// @notice Recovers the root hash of a given signature
  function recoverSapientSignature(
    Payload.Decoded calldata _payload,
    bytes calldata _signature
  ) external view returns (bytes32);

}
```

This function is implemented in `SessionManager` but not in `ExplicitSessionManager`. Yet, as can be seen above, `ExplicitSessionManager.supportsInterface` answers that it implements this interface. With the current state of the codebase, there shouldn't be any practical consequences, since `ExplicitSessionManager` is an abstract contract and `SessionManager` – which does implement the interface and whose `supportsInterface` function answers accordingly – is the only contract that inherits from it. Nevertheless, having `ExplicitSessionManager` answer that it implements `ISapient` is technically wrong and could lead to problems when code is changed.

B. If a contract wants to know whether some other contract C implements a certain interface I, it is a common pattern to first query whether C implements the ERC-165 interface – and only if this is answered in the affirmative, send another query to find out whether I is supported. This is actually a suggestion in the ERC-165 specification itself, and this approach is also implemented e.g. in OpenZeppelin's `ERC165Checker`, which is frequently used for the task at hand. The contracts above, however, would answer that they don't support ERC-165, which could mislead the caller to believe that they don't implement `ISapient` either.

Recommendation

It is not entirely clear how/whether the two `supportsInterface` functions are used. At the very least, they are not called from within the current codebase. If they're not useful, they can be removed entirely, but assuming they should be kept, we recommend the following:

1. Add an `ERC165` contract with a `supportsInterface` function which answers that it supports (only) the ERC-165 interface.
2. Other contracts that implement a `supportsInterface` function should inherit from `ERC165`, and their `supportsInterface` implementation should also check `super.supportsInterface`. This is the pattern that is already correctly employed in `SessionManager`.
3. As discussed in A above, the `ISapient` part should be removed from `ExplicitSessionManager.supportsInterface` (but the `super` part mentioned in 2 should be added).

ERC-165 is used sparingly in the codebase. Check to what extent you want/have to support it. If it's not useful, consider removing the two `supportsInterface` functions completely. Otherwise, check whether you support ERC-165 in all the places you want. For instance, `ISapient.sol` contains a second interface, `ISapientCompact`, which is similar to `ISapient` and is factually implemented by some contracts, but none of these have a `supportsInterface` function.

5.12 Missing Events for State-Changing Functions Minor

Resolution

Sequence team's response:

We decided to implement the missing events in the smart sessions extension, yet we will retain the current implementation of the Factory.

The Factory is a contract that must be as efficient as possible, thus any addition to it must justify its increment on gas usage. We do not believe the usefulness of an event justifies the additional gas usage.

The PR adding the events to the smart sessions extension is: [0xsequence/wallet-contracts-v3#45]()

Description

The lack of event emission makes it difficult to track wallet deployments off-chain and could complicate user interface integration or system monitoring. Events for contract creation are especially important in upgradeable systems to maintain a complete audit trail of deployed instances.

Examples

- `Factory` – deployment should emit an event

src/Factory.sol:L19-L27

```
function deploy(address _mainModule, bytes32 _salt) public payable returns (address _contract) {
  bytes memory code = abi.encodePacked(Wallet.creationCode, uint256(uint160(_mainModule)));
  assembly {
    _contract := create2(callvalue(), add(code, 32), mload(code), _salt)
  }
  if (_contract == address(0)) {
    revert DeployFailed(_mainModule, _salt);
  }
}
```

- `PermissionValidator` – `setLimitUsage` should emit an event

src/extensions/sessions/explicit/PermissionValidator.sol:L27-L29

```
function setLimitUsage(address wallet, bytes32 usageHash, uint256 usageAmount) internal {
  limitUsage[wallet][usageHash] = usageAmount;
}
```

Recommendation

Implement an event to log crucial state-changing operations like the deployment of new contracts and updates of usage amounts.

5.13 Implementation Contracts Signal Ability to Handle NFTs Minor

Resolution

Sequence team's response:

We acknowledge this behavior, but we do not consider it is worth correcting. The likelihood of anyone sending an NFT to an implementation contract is low, and adding an explicit check for `onlyProxy` would increase gas usage on every real wallet in turn.

Description

Since the wallet can and is supposed to handle tokens – including NFTs – transferred to it, it signals the corresponding abilities by implementing the `onERC1155Received`, `onERC1155BatchReceived`, `onERC721Received`, and `tokenReceived` functions (where the latter is less familiar and defined in ERC-223). However, these functions exhibit the same behavior when called directly on the implementation contract as when they're called on a proxy, although the tokens would be stuck on the implementation contract:

src/modules/Hooks.sol:L75-L93

```
function onERC1155Received(address, address, uint256, uint256, bytes calldata) external pure returns (bytes4) {
  return Hooks.onERC1155Received.selector;
}

function onERC1155BatchReceived(
  address,
  address,
  uint256[] calldata,
  uint256[] calldata,
  bytes calldata
) external pure returns (bytes4) {
  return Hooks.onERC1155BatchReceived.selector;
}

function onERC721Received(address, address, uint256, bytes calldata) external pure returns (bytes4) {
  return Hooks.onERC721Received.selector;
}

function tokenReceived(address, uint256, bytes calldata) external { }
```

Recommendation

The point of these functions is to assure the caller that the tokens aren't accidentally being sent to a contract that is not prepared or willing to receive them. Since only proxies – but not the implementation contracts – should ever receive tokens, the latter should not falsely signal their ability to handle tokens and thereby defeat the purpose of the mechanism.

This could be achieved by implementing an `onlyProxy` modifier that compares `address(this)` to an immutable variable that is set to `address(this)` in the implementation contract's constructor and rejects if the two are the same – meaning the call is happening on the implementation contract and not the proxy. The downside of this approach is slightly increased gas cost even for legitimate calls on the proxy.

5.14 Insufficient NatSpec Documentation Minor

Resolution

Sequence team's response:

We acknowledge the lack of NatSpec documentation. The following PR aims to increase the coverage of the documentation: 0xsequence/wallet-contracts-v3#50

Description

Many contracts lack NatSpec annotations, both for the contract itself as well as its functions and state variables.

Recommendation

We recommend adding a comprehensive set of NatSpec annotations to the codebase in order to facilitate a better and more explicit understanding for developers, testers, and auditors alike. Comprehensive documentation will help clearly describe function inputs, outputs, and behaviors, as well as specify edge cases and intended usage, making the code easier to understand and review.

5.15 Hooks - Unreachable `receive()` Function Due to Wallet Proxy Implementation Rejecting Value Transfers Minor

Resolution

Sequence team's response:

We acknowledge the unreachability of the `receive()` function when using the wallet proxy included in the project.

We consider it is worth leaving the functionality on the `Hooks` contract, since this contract could also be used alongside a proxy that does not short-circuit transfers, and unreachability has no negative consequences.

Description

The `receive()` function is marked as payable but is effectively unreachable. The proxy implementation directly returns for all calls with non-zero `msg.value`, making the receive function redundant and potentially misleading for developers integrating with the contract.

Example

src/modules/Hooks.sol:L108-L108

```
receive() external payable { }
```

Wallet Code returns immediately if `msg.value > 0` not forwarding the call to the implementation implementing `Hooks` :

```
ifmsg.value {
    return memory[output.start:(output.offset+output.length)];
} else {
    if(delegatecall(gasleft(),storage[this],args.offset,msg.data.length,ret.offset,ret.length)) {
        return memory[output.offset:(output.offset+output.length)];
    } else {
        revert(memory[output.offset:(output.offset+output.length)]);
    }
}
```

Recommendation

Remove the unreachable `receive()` function to improve code clarity. Document the proxy's behavior regarding value transfers and add comments if needed.

5.16 Unused Imports Minor

Resolution

Sequence team's response:

We acknowledge the issue and we have fixed it in the following PR: 0xsequence/wallet-contracts-v3#50

Description

Some files contain unused imports, which add unnecessary complexity to the codebase.

Examples

- `Guest.sol`

src/Guest.sol:L7-L8

```
import { IAuth } from "./modules/interfaces/IAuth.sol";
import { LibBytesPointer } from "./utils/LibBytesPointer.sol";
```

`IAuth` interface is imported but never used, and the `LibBytesPointer` library is imported and declared with a `using` statement, but none of its methods are utilized in the contract.

- `Calls.sol`

src/modules/Calls.sol:L8-L9

```
import { SelfAuth } from "./auth/SelfAuth.sol";
import { IDelegatedExtension } from "./interfaces/IDelegatedExtension.sol";
```

The `SelfAuth` contract is imported but not explicitly used anywhere. The contract inherits `SelfAuth` from `BaseAuth` .

- `IAuth.sol`

src/modules/interfaces/IAuth.sol:L4

```
import { Payload } from "../Payload.sol";
```

The `Payload` library is imported but never used.

Recommendation

Remove unused imports to improve code clarity and reduce unnecessary dependencies.

5.17 Unnecessary Assignments

Resolution

Sequence team's response:

We acknowledge the issue and we have fixed it in the following PRs: 0xsequence/wallet-contracts-v3#29, 0xsequence/wallet-contracts-v3#46

Description

A. Consider the following excerpt from `BaseSig` :

src/modules/auth/BaseSig.sol:L455-L463

```
// Call the ERC1271 contract to check if the signature is valid
bytes32 sapientImageHash = ISapient(addr).recoverSapientSignature(_payload, _signature[rindex:nrindex]);
rindex = nrindex;

// Add the weight and compute the merkle root
weight += addrWeight;
bytes32 node = _leafForSapient(addr, addrWeight, sapientImageHash);
root = root != bytes32(0) ? LibOptim.fkeccak256(root, node) : node;
rindex = nrindex;
```

The assignment `rindex = nrindex;` occurs two times – without any of the two variables changing their value in-between. Hence, the second assignment is redundant and can be removed.

Note also that the comment is copy-pasted from the `FLAG_SIGNATURE_ERC1271` logic and doesn't make sense here.

B. A similar situation occurs later a second time:

src/modules/auth/BaseSig.sol:L490-L498

```
// Call the Sapient contract to check if the signature is valid
bytes32 sapientImageHash =
  ISapientCompact(addr).recoverSapientSignatureCompact(_opHash, _signature[rindex:nrindex]);
rindex = nrindex;
// Add the weight and compute the merkle root
weight += addrWeight;
bytes32 node = _leafForSapient(addr, addrWeight, sapientImageHash);
root = root != bytes32(0) ? LibOptim.fkeccak256(root, node) : node;
rindex = nrindex;
```

Again, the second assignment is redundant and can be removed. The comment is correct here.

C. Finally, there's a third occurrence of this pattern in `Recovery`:

src/extensions/recovery/Recovery.sol:L113-L118

```
rindex = nrindex;

verified = verified || nverified;
root = LibOptim.fkeccak256(root, nroot);

rindex = nrindex;
```

D. In the following code snippet, the last assignment is unnecessary and can be removed:

src/extensions/sessions/SessionSig.sol:L331-L336

```
// Update the permissions array length to the actual count
SessionPermissions[] memory permissions = sig.sessionPermissions;
assembly {
  mstore(permissions, permissionsCount)
}
sig.sessionPermissions = permissions;
```

The `mstore` doesn't change the value of `permissions`; it only changes memory at the location that is stored in `permissions`.

Recommendation

For A, B, and C, remove the second, redundant assignment. In A, also fix the comment. For D, the last line can be removed.

5.18 Duplicated Low-Level Code

Resolution

Sequence team's response:

We acknowledge the issue and we have fixed it in the following PR: 0xsequence/wallet-contracts-v3#47

Description

ERC-2098 specifies a compact format for ECDSA signatures that requires only two words; essentially, the MSB of the `s` parameter is used for the parity bit. This format is used throughout the codebase.

Extracting the r, s, and v values to be fed into `ecrecover` from such a compact signature requires low-level bit operations such as shifting and masking. Consequently, the libraries `LibBytes` and `LibBytesPointer` each provide a `readRSVCompact` function that takes care of the low-level work and returns the extracted parameters with their correct type. (`LibBytesPointer.readRSVCompact` also returns a pointer to the calldata location after the compact signature, which is a general pattern in this library and is useful for parsing the calldata). However, despite the presence of these library functions, there are several places in the codebase where some form of low-level code is implemented directly to extract the r, s, and v values from a compact signature.

This makes the code harder to read and introduces the risk of mistakes; instead, the functions provided by the aforementioned libraries should be utilized.

Examples

There are two occurrences of this in `BaseSig` (which are even slightly different):

src/modules/auth/BaseSig.sol:L222-L229

```
bytes32 r;
bytes32 yParityAndS;
(r, rindex) = _signature.readBytes32(rindex);
(yParityAndS, rindex) = _signature.readBytes32(rindex);

uint256 yParity = uint256(yParityAndS >> 255);
bytes32 s = bytes32(uint256(yParityAndS) & 0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
uint8 v = uint8(yParity) + 27;
```

src/modules/auth/BaseSig.sol:L394-L401

```
bytes32 r;
bytes32 yParityAndS;
(r, rindex) = _signature.readBytes32(rindex);
(yParityAndS, rindex) = _signature.readBytes32(rindex);

uint256 yParity = uint256(yParityAndS >> 255);
bytes32 s = bytes32(uint256(yParityAndS) & ((1 << 255) - 1));
uint8 v = uint8(yParity) + 27;
```

and one in `Recovery`:

src/extensions/recovery/Recovery.sol:L179-L182

```
(bytes32 r, bytes32 yParityAndS) = abi.decode(_signature, (bytes32, bytes32));
uint256 yParity = uint256(yParityAndS >> 255);
bytes32 s = bytes32(uint256(yParityAndS) & 0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
uint8 v = uint8(yParity) + 27;
```

It might also be worth mentioning that the `LibBytes` library defines a function named `readMRSVCompact` (note the "M"; it reads the compact signature from memory instead of calldata) that isn't used anywhere in the codebase.

Recommendation

These library functions should be utilized consistently in order to avoid duplicating the low-level code. Unused functionality can be removed from the libraries (and the codebase in general).

5.19 Specification/Documentation Inconsistencies

Resolution

Sequence team's response:

We acknowledge the issue and we have fixed it in the following PR: 0xsequence/wallet-contracts-v3#50

Description

This is a non-exhaustive list of examples where the documentation/specification deviates from the actual implementation.

Undocumented Static Signature Bit

According to the specification, Bit 7 in the top-level signature format is reserved, while in the current system, it is used to indicate usage of static signatures. Static signature requirements are not documented.

docs/SIGNATURE.md:L24-L27

```
## **2. Top-level Signature Format**

When `recover` is first invoked, it reads the **first byte** of the signature as `signatureFlag`. That byte is bit-packed as follows
```

docs/SIGNATURE.md:L29-L35

```
 ┌───────────────────── Bit 7 (0x80) : Reserved
 │ ┌─────────────────── Bit 6 (0x40) : Checkpointer usage flag
 │ │ ┌───────────────── Bit 5 (0x20) : Threshold size indicator (0 => 1 byte, 1 => 2 bytes)
 │ │ │  ┌────────────── Bits 4..2 (0x1C) : Checkpoint size (encoded as an integer 0..7)
 │ │ │  │  ┌─────────── Bit 1 (0x02) : "no chain id" signature type
 │ │ │  │  │ ┌──────── Bit 0 (0x01) : "chained" signature type
[X 6 5 432 1 0]
```

src/modules/auth/BaseAuth.sol:L61-L70

```
function signatureValidation(
  Payload.Decoded memory _payload,
  bytes calldata _signature
) internal view virtual returns (bool isValid, bytes32 opHash) {
  // Read first bit to determine if static signature is used
  bytes1 signatureFlag = _signature[0];

  if (signatureFlag & 0x80 == 0x80) {
    opHash = _payload.hash();
```

Unspecified Signature Type 0x11 is accepted

According to the inline documentation there are 3 distinct types of signatures:

- `00` normal
- `01` chained
- `10` no chain id
- `11` UNSPECIFIED

src/modules/auth/BaseSig.sol:L68-L74

```
// The possible flags are:
// - 0000 00XX (bits [1..0]): signature type (00 = normal, 01 = chained, 10 = no chain id)
// - 000X XX00 (bits [4..2]): checkpoint size (00 = 0 bytes, 001 = 1 byte, 010 = 2 bytes...)
// - 00X0 0000 (bit [5]): threshold size (0 = 1 byte, 1 = 2 bytes)
// - 0X00 0000 (bit [6]): set if imageHash checkpointer is used
// - X000 0000 (bit [7]): reserved by base-auth
```

The code tests logical AND `0x01` which only checks 1 bit of information. This is true for signature types `0x01` AND `0x11` which is unspecified. The inline comment mentions that if type is `01` a chained signature is assumed, however, only the last bit is tested. The type `0x11` is not specified and should therefore not be allowed, as there is no chained signature with `no_chainid`.

src/modules/auth/BaseSig.sol:L100-L103

```
// If signature type is 01 we do a chained signature
if (signatureFlag & 0x01 == 0x01) {
  return recoverChained(_payload, _checkpointer, snapshot, _signature[rindex:]);
}
```

Wrong information regarding the weight for FLAG_ADDRESS

The table in SIGNATURE.md says that the weight for the `FLAG_ADDRESS` type is added.

docs/SIGNATURE.md:L142

```
| `FLAG_ADDRESS`                          | 1               | Just an address "leaf" (adds that address's weight, but no actual ECDSA
```

In the code, the weight is not added – which is the correct behavior:

src/modules/auth/BaseSig.sol:L254-L257

```
// Compute the merkle root WITHOUT adding the weight
bytes32 node = _leafForAddressAndWeight(addr, addrWeight);
root = root != bytes32(0) ? LibOptim.fkeccak256(root, node) : node;
continue;
```

Inconsistent Example

The example decoding suggests that the input is `0110 1100` and the decoding bits are not listed in order (see Bit 4..2 vs 5). The actual example that yields the desired decoded output is `0b1111000` and `4..2` should be listed after `5`.

docs/SIGNATURE.md:L320-L326

```
1. **signatureFlag** = `0x6C` => in binary `0110 1100`
   - Bit 6 => `1`, so we have a checkpointer
   - Bits 4..2 => `110` => checkpoint size = 6 bytes
   - Bit 5 => `1`, threshold uses 2 bytes
   - Bit 1 => `0`, normal chain id usage
   - Bit 0 => `0`, not chained
```

Cumulative bit in parameter rules

`SESSIONS.md` states that bit 7 (i.e., the highest-order bit) is the cumulative bit in the parameter rule encoding:

docs/SESSIONS.md?plain=1:L103-L113

```
Parameter Rule Encoding:

┌──────────────────────────────────────────────────────────────────────────┐
│ Operation & Cumulative Flag (1 byte)                                     │
│ ┌──────────────────────────────────────────────────────────────────────┐ │
│ │ Bit 7 (0x80): Cumulative flag (1 = cumulative)                       │ │
│ │ Bits 6..0  (0x7F): Operation (e.g., 0 = EQUAL, etc.)                 │ │
│ └──────────────────────────────────────────────────────────────────────┘ │
│ Value (bytes32)                                                          │
│ Offset (uint256)                                                         │
│ Mask (bytes32)                                                           │
└──────────────────────────────────────────────────────────────────────────┘
```

But the code treats bit 0 (i.e., the lowest-order bit) as the cumulative bit:

src/extensions/sessions/explicit/Permission.sol:L53-L58

```
uint8 operationCumulative;
(operationCumulative, pointer) = encoded.readUint8(pointer);
// 000X: cumulative
permission.rules[i].cumulative = operationCumulative & 1 != 0;
// XXX0: operation
permission.rules[i].operation = ParameterOperation(operationCumulative >> 1);
```

Blacklist count encoding

`SESSIONS.md` states that the number of blacklist entries is a `uint24`:

docs/SESSIONS.md?plain=1:L162-L166

```
Blacklist Node Layout:

┌─────────────────────────────────────────────┐
│ Blacklist Count (uint24, with overflow check)│
│ Blacklisted Addresses (sorted array)         │
└─────────────────────────────────────────────┘
```

However, in the code the blacklist count is encoded in the lower nibble of the first byte for 0..14, and for bigger values, it's the next two bytes.

src/extensions/sessions/SessionSig.sol:L290-L295

```
// Read the blacklist count from the first byte's lower 4 bits
uint256 blacklistCount = uint256(firstByte & 0x0f);
if (blacklistCount == 0x0f) {
  // If it's max nibble, read the next 2 bytes for the actual size
  (blacklistCount, pointer) = encoded.readUint16(pointer);
}
```

Recommendation

Verify the inconsistencies. Update the documentation to reflect the current feature set of the system. Ensure that off-chain code doesn't simply follow the specification but is consistent with the contracts.

5.20 Payload - Redundant Implementation of `hash()` With `hashFor()`

Resolution

Sequence team's response:

We acknowledge the duplicated code, yet we consider the amount of code duplication is minimal and does not justify reuse.

Description

`Payload.hashFor()` implements the exact same logic as `Payload.hash()`. The only difference is that `hash()` hardcodes `wallet` to `address(this)`

Examples

src/modules/Payload.sol:L232-L244

```
function hash(
  Decoded memory _decoded
) internal view returns (bytes32) {
  bytes32 domain = domainSeparator(_decoded.noChainId, address(this));
  bytes32 structHash = toEIP712(_decoded);
  return keccak256(abi.encodePacked("\x19\x01", domain, structHash));
}

function hashFor(Decoded memory _decoded, address _wallet) internal view returns (bytes32) {
  bytes32 domain = domainSeparator(_decoded.noChainId, _wallet);
  bytes32 structHash = toEIP712(_decoded);
  return keccak256(abi.encodePacked("\x19\x01", domain, structHash));
}
```

Recommendation

Instead of duplicating the code, consider calling `hashFor` from `hash` instead, providing `address(this)`.

5.21 Payload - Unused Internal Functions

Resolution

Sequence team's response:

We acknowledge the unused internal functions, yet we consider the functions should be maintained for the sake of correctness of the Payload library, even if unused in this particular instance.

Description

The internal functions `fromMessage` and `fromConfigUpdate` in the Payload contract are never used within the codebase.

Example

src/modules/Payload.sol:L79-L91

```
function fromMessage(
  bytes memory message
) internal pure returns (Decoded memory _decoded) {
  _decoded.kind = KIND_MESSAGE;
  _decoded.message = message;
}

function fromConfigUpdate(
  bytes32 imageHash
) internal pure returns (Decoded memory _decoded) {
  _decoded.kind = KIND_CONFIG_UPDATE;
  _decoded.imageHash = imageHash;
}
```

Recommendation

Remove unused functions or document their intended future use with appropriate comments. If the functions are meant for future extensibility, clearly document this intention in NatSpec comments.

5.22 (Sub)Module Contracts Should Be Declared `abstract`

Resolution

Sequence team's response:

We acknowledge the issue and we have fixed it in the following PR: 0xsequence/wallet-contracts-v3#48

Description

In Solidity, the keyword `abstract` is used for contracts when at least one of their functions is not implemented. Contracts may be marked as abstract even though all functions are implemented. As a side-effect, `abstract` contracts cannot be deployed directly.

Most of the (Sub)Modules provide a base set of functions, intended to be used by the inheriting main module contracts. Therefore, they do not need to be deployable themselves and should be marked `abstract` to clearly signal this.

Examples

src/modules/auth/SelfAuth.sol:L4-L15

```
contract SelfAuth {

  error OnlySelf(address _sender);

  modifier onlySelf() {
    if (msg.sender != address(this)) {
      revert OnlySelf(msg.sender);
    }
    _;
  }

}
```

Recommendation

Contracts that are not meant to be deployable directly should be marked abstract.

6 Intended or Accepted System Behavior

6.1 Missing Bounds Checking and Arithmetic Under/Overflows in Low-Level Operations

Description and Recommendation

Various libraries implements calldata reading operations using low-level `calldataload` assembly operations without performing bounds checking. If a read operation attempts to access data beyond the calldata bounds, instead of reverting, it will silently return zero values. This can lead to security issues where invalid, incomplete, or malicious input data is treated as valid zero values rather than causing a revert, potentially leading to unexpected contract behavior or security vulnerabilities in dependent contracts.

We would like to point out that we have not found a scenario where this would introduce a concrete security risk right now, hence, we tagged this a recommendation. However, since this behavior is not documented in-code at all we would recommend adding inline documentation to make sure future developers and consumers of the functions are aware of this side-effect.

Example

- `LibBytePointer` (one example) - Silently returning zero values (calldata)

src/utils/LibBytesPointer.sol:L52-L59

```
function readAddress(bytes calldata _data, uint256 _index) internal pure returns (address a, uint256 newPointer) {
  assembly {
    let word := calldataload(add(_index, _data.offset))
    a := and(shr(96, word), 0xffffffffffffffffffffffffffffffffffffffff)
    newPointer := add(_index, 20)
  }
}
```

- `LibBytes` (one example) - Silently returning zero values if not enough calldata pas provided

src/utils/LibBytes.sol:L20-L24

```
function readBytes32(bytes calldata data, uint256 index) internal pure returns (bytes32 a) {
  assembly {
    a := calldataload(add(data.offset, index))
  }
}
```

- `LibBytePointer` (one example) - Theoretical Over/Underflow for large values of `length`

src/utils/LibBytesPointer.sol:L80-L91

```
function readUintX(
  bytes calldata _data,
  uint256 _index,
  uint256 _length
) internal pure returns (uint256 a, uint256 newPointer) {
  assembly {
    let word := calldataload(add(_index, _data.offset))
    let shift := sub(256, mul(_length, 8))
    a := and(shr(shift, word), sub(shl(mul(8, _length), 1), 1))
    newPointer := add(_index, _length)
  }
}
```

It is recommended, to add explicit bounds checking before performing calldata reads.

In discussion with the development team they acknowledged this as intended to optimize for gas usage and avoid unnecessary checks.

6.2 Implementation - Address Existence Not Validated During Upgrade

Description and Recommendation

The `updateImplementation` function does not verify if the provided implementation address contains code before updating the proxy, allowing setting an EOA as implementation. When a proxy delegates calls to an EOA, the calls will always succeed if the function doesn't expect a return value, as EOAs return empty data which is interpreted as a successful execution. This can lead to a completely broken proxy that appears to work but performs no actual logic.

src/modules/Implementation.sol:L10-L14

```
function updateImplementation(
  address _implementation
) external payable virtual onlySelf {
  _updateImplementation(_implementation);
}
```

It is recommended, to add validation to ensure the new implementation contains code before updating.

In discussion with the development team they acknowledged this as a configurative issue.

6.3 Factory - Minimalistic Deploy Function

Description

Most likely with the intention to make wallet deployments as cheap as possible, the `deploy` function in the `Factory` contract is very minimalistic:

src/Factory.sol:L19-L27

```
function deploy(address _mainModule, bytes32 _salt) public payable returns (address _contract) {
  bytes memory code = abi.encodePacked(Wallet.creationCode, uint256(uint160(_mainModule)));
  assembly {
    _contract := create2(callvalue(), add(code, 32), mload(code), _salt)
  }
  if (_contract == address(0)) {
    revert DeployFailed(_mainModule, _salt);
  }
}
```

For instance, it does not emit an event for a successful deployment, which would allow to keep track of the wallets that have been deployed.

Moreover, there is no check whether the given `_mainModule` address – which will become the implementation address the wallet delegatecalls to – really has code. If it didn't, calls to the proxy would generally succeed but not have the desired effect, and there is some risk that this might go unnoticed for a while. (However, the client has pointed out that wallet addresses are even used counterfactually before deployment, so discovering the mistake here would already be too late in a sense. And in case of a wrong `_mainModule` argument for the `deploy` call but a correct counterfactual address, the error might very well be caught through the resulting address mismatch, but these considerations are out of scope for this smart contract review.)

Recommendation

Reassess whether you're comfortable with this minimalistic design or whether adding an event for successful deployments and/or a check that the given `_mainModule` has indeed non-empty codesize should be added. If you decide to not do the latter, make sure you have appropriate offchain checks and processes in place to mitigate the risk discussed above.

Remark

These points have already been discussed in the previous report (5.3, 5.10), and the client has indicated that they don't want to make any changes in this regard.

6.4 Interfaces Unexpectedly Implement Code

Description and Recommendation

This finding is similar to 5.8 from a review of a previous version.

According to their naming pattern (`I<module_name>`) and location in the `interfaces` subdirectory, `IAuth` should be interface declarations. However, the source units is an `abstract` contracts instead of `interface` and unexpectedly contain concrete *implementations* of functions (even though the return value is hardcoded).

Based on the contract name and filename (prefix `I`) as well as the filesystem location (`./interfaces/...`), one would assume that the source units contain external interface declarations only. Finding concrete implementations or internal abstract methods is highly unexpected.

Here, `IAuth.sol` - hints `interface` but is `abstract` and implements code.

src/modules/interfaces/IAuth.sol:L6-L12

```
abstract contract IAuth {

  function _isValidImage(
    bytes32
  ) internal view virtual returns (bool) {
    return false;
  }
```

It is recommended to refactor the `abstract` 'interface' contracts, moving the internal declarations and concrete function implementations to the respective implementation source unit. The `interfaces` subfolder should only contain external interface declarations (and types, etc.) without executable code or internal function declarations.

6.5 Hooks - Check for Contract Existence Before Executing Hook

Description and Recommendation

This finding is similar to 5.2 from a review of a previous version.

The `Hooks` functionality allows the wallet to set hooks on function signatures handled in the `fallback` function. However, the second-level function dispatcher does not ensure that the `delegatecall` target contract exists before the call is made. This might lead to the hook being executed on an address that does not contain code – resulting in the `delegatecall` always returning success and hiding that the target contract did not exist and no code was executed.

src/modules/Hooks.sol:L95-L106

```
fallback() external payable {
  if (msg.data.length >= 4) {
    address target = _readHook(bytes4(msg.data));
    if (target != address(0)) {
      (bool success, bytes memory result) = target.delegatecall(msg.data);
      assembly {
        if iszero(success) { revert(add(result, 32), mload(result)) }
        return(add(result, 32), mload(result))
      }
    }
  }
}
```

Before the `delegatecall` is made, check for target contract existence to ensure the hook target address is a contract. Consider making the same check when the hook is added to catch mistakes early.

6.6 BaseSig - Pot. Integer Overflow in Signature Weight Accumulation

Description

The signature verification logic in `recoverBranch` can overflow the accumulated weight when, for example, combining a `FLAG_SUBDIGEST` node (sets `weight = type(uint256).max`) with additional branches (e.g., `FLAG_BRANCH`). This results in a weight wraparound, but such cases are considered incorrectly encoded signatures and are allowed to be rejected by the system.

Here is the relevant code parts, encoding a subdigest proof and a signature part together, may cause a weight overflow.

src/modules/auth/BaseSig.sol:L308-L328

```
if (flag == FLAG_BRANCH) {
  // Free bits layout:
  // - XXXX : Size size (0000 = 0 byte, 0001 = 1 byte, 0010 = 2 bytes, ...)

  // Read size
  uint256 sizeSize = uint8(firstByte & 0x0f);
  uint256 size;
  (size, rindex) = _signature.readUintX(rindex, sizeSize);

  // Enter a branch of the signature merkle tree
  uint256 nrindex = rindex + size;

  (uint256 nweight, bytes32 node) = recoverBranch(_payload, _opHash, _signature[rindex:nrindex]);
  rindex = nrindex;

  weight += nweight;
  root = LibOptim.fkeccak256(root, node);

  rindex = nrindex;
  continue;
}
```

src/modules/auth/BaseSig.sol:L365-L380

```
// Subdigest (0x05)
if (flag == FLAG_SUBDIGEST) {
  // Free bits left unused

  // A hardcoded always accepted digest
  // it pushes the weight to the maximum
  bytes32 hardcoded;
  (hardcoded, rindex) = _signature.readBytes32(rindex);
  if (hardcoded == _opHash) {
    weight = type(uint256).max;
  }

  bytes32 node = _leafForHardcodedSubdigest(hardcoded);
  root = root != bytes32(0) ? LibOptim.fkeccak256(root, node) : node;
  continue;
}
```

This finding has been discussed with the development team which is aware of the pot. integer wrapping. They provided the following statement:

> encoding a subdigest proof + a signature part (and causes overflow) it is just an incorrectly encoded signature, it is rejected. We wouldn't mind it to be accepted tho, it 'should' be valid, yet we rather consider it a failure to encode it and leave it undefined, rather than paying for the branching needed to normalize the case.

6.7 Parameter Rules Have to Enforce Strict Encoding for Dynamic-Type Parameters

Description and Recommendation

An important concept for explicit sessions are *permissions*, which allow the wallet to define restrictions on what calls a session signer can execute on behalf of the wallet. A permission includes an address to which calls may be made and a list of parameter rules. Parameter rules apply to a call's calldata on the bytes level. More specifically, a parameter rule contains an `offset` that specifies which 32-bytes word in the calldata this rule should target. The corresponding word is then read from the calldata, a `mask` is applied, and the result is compared to the `value` in the rule. There are a few more details, but they are not needed for the following discussion; the important point is that we read 32 bytes at an `offset` defined by the rule and check (with some bells and whistles) whether this value is allowed.

This was the technical perspective. On a conceptual level, we're thinking of calldata in terms of function arguments. So the task of the parameter rule creator is to translate function parameters into calldata offsets. And here, a subtle but important point comes into play, related to how Solidity encodes and decodes data. While there is a standard encoding, called strict encoding, the Solidity ABI-decoder, which is responsible in a contract to decode the calldata into function arguments, accepts not only this strict encoding but is more lenient. We recommend studying the Contract ABI Specification for the details, but the gist is that the argument for a dynamic-type parameter can't be found at a fixed place in the calldata; instead, it is found by following a "pointer." And conversely, reading from a fixed place in the calldata does not necessarily give us what later becomes the argument in the function.

Hence, special care must be taken with the parameter rules for functions that have not exclusively static-type parameters. Essentially, in such cases, the rules also have to enforce strict encoding. If this is not done, the rules that are supposed to cover the dynamic-type parameters can be *completely bypassed*! Again, we refer to the Contract ABI Specification in the Solidity documentation for the details.

6.8 Hooks - Functions Marked Payable

Description and Recommendation

The `addHook` and `removeHook` functions in Hooks.sol are marked as `payable`. This may create confusion about the intended functionality because there is no operational reason for hook management functions to accept value.

src/modules/Hooks.sol:L48-L62

```
function addHook(bytes4 signature, address implementation) external payable onlySelf {
  if (_readHook(signature) != address(0)) {
    revert HookAlreadyExists(signature);
  }
  _writeHook(signature, implementation);
}

function removeHook(
  bytes4 signature
) external payable onlySelf {
  if (_readHook(signature) == address(0)) {
    revert HookDoesNotExist(signature);
  }
  _writeHook(signature, address(0));
}
```

During discussions with the developers, they indicated that functions are intentionally marked `payable` to achieve gas optimizations.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
src/Factory.sol	`9d158fad6d9b0247689c1ae1b7d826b0769d7e88`
src/Guest.sol	`4d88a6d8c0a3d868d1a075a9d72431334a6d3f2f`
src/Stage1Module.sol	`f1e7df6cb69094a3fcbd8057debc692c0186a6f8`
src/Stage2Module.sol	`14457e4444641ace1370e8058f35716bbc6c250e`
src/Wallet.huff	`20f47df8b26ff24ab14d479ae83833183a854262`
src/Wallet.sol	`5f38ee5be174e07f484f5d93e187ac57714cafd5`
src/extensions/passkeys/Passkeys.sol	`0582ad8366c9f1634eda08d275c04e8bcd8e5f4d`
src/extensions/recovery/Recovery.sol	`ec75a88a096afaa919f973d9be3e1df8076642d3`
src/extensions/sessions/SessionErrors.sol	`d74ac5cfd6934eab2698047bb25e1de6900f193c`
src/extensions/sessions/SessionManager.sol	`d4551aac843dca0f7669bac9aebbbe1f79a0e8e3`
src/extensions/sessions/SessionSig.sol	`05aa775f18546ce344d6412976b44de85a91c39f`
src/extensions/sessions/explicit/ExplicitSessionManager.sol	`daf4abce662a1e909097737960fc3dd7dc17bf63`
src/extensions/sessions/explicit/IExplicitSessionManager.sol	`07fd50152372166fc64c2022f4089b1233a1e8d0`
src/extensions/sessions/explicit/Permission.sol	`fd136590eaf0350204d047066cacdb34d16e6b7d`
src/extensions/sessions/explicit/PermissionValidator.sol	`7dbc3b1a2f374a1ec261471e72d36866b7f71e6b`
src/extensions/sessions/implicit/Attestation.sol	`bd3cb0c0664c8b34d661d54b9768b342807c18c6`
src/extensions/sessions/implicit/ISignalsImplicitMode.sol	`6c0efecf2b2a17688c2a4ced0ca58d9d30599170`
src/extensions/sessions/implicit/ImplicitSessionManager.sol	`21299b501a48059eee6a873a482fc6e21f90a307`
src/modules/Calls.sol	`5cee5f6f2af255443592977d24b97fbe2dc9b843`
src/modules/Hooks.sol	`b6f51d751061d4b999388c99f9d0a95ad156fc4a`
src/modules/Implementation.sol	`6fffe294e68048385812be00b7e00422fc159769`
src/modules/Nonce.sol	`05166a267f966a56025aa55f0034c78cf57fe457`
src/modules/Payload.sol	`615f067ad744cd8221dff995b05e55197baab6a7`
src/modules/Storage.sol	`0c40b988b4e8cf08cbe29fb16faf7662841415ca`
src/modules/auth/BaseAuth.sol	`87854fd692eea6fd082140abaf240ff2692ed24b`
src/modules/auth/BaseSig.sol	`2e4931a6640eefd3a9433990b40eb88d606b6860`
src/modules/auth/SelfAuth.sol	`1ca62ca7061bc61a57a8ca38e2ecac523909dab0`
src/modules/auth/Stage1Auth.sol	`de8a2298011f42d62200a34129b6f796fc582f7c`
src/modules/auth/Stage2Auth.sol	`a1f38ee010eecaa23b8e5012099e1d155dad3496`
src/modules/interfaces/IAuth.sol	`0f3bdba3297bd3714d73ac28f104e9a8e1e1cbec`
src/modules/interfaces/ICheckpointer.sol	`541108ca330c6fa4d491ddcfb8ec74726a2bd6ed`
src/modules/interfaces/IDelegatedExtension.sol	`3c27aab5aca8e8e0ba103a3e2b1694487a293f70`
src/modules/interfaces/IERC1271.sol	`29834c83b0dafdb5616248d7e357f2c95d8237cc`

File	SHA-1 hash
src/modules/interfaces/IPartialAuth.sol	`bde66c6f406aab3d20a489c61a549e4bf23e6903`
src/modules/interfaces/ISapient.sol	`3204ca666f2ef79c02e730572c09f11c6d97caf4`
src/utils/LibBytes.sol	`188fe463c483f64ca236409901f888ca06c8adaf`
src/utils/LibBytesPointer.sol	`52325d02dc050fe38caf4a882ac6116f4ab9f8ab`
src/utils/LibOptim.sol	`135e850551084def804c86f8bed1302476b74ee4`

Appendix 2 - Document Change Log

Version	Date	Description
1.0	2025-05-26	Initial report
2.0	2025-06-09	Added Sequence team's responses

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 bug-free 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.

A.3.1 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 code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as 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 specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.3.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.3.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.