Zellic

Hyperliquid

Smart Contract Security Assessment

August 14, 2023

Prepared for:

Hyperliquid Protocol Foundation

Prepared by:

Filippo Cremonese, Kuilin Li, and Daniel Lu

Zellic Inc.

Contents

About Zellic

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

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

1 Executive Summary

Zellic conducted a security assessment for the Hyperliquid contributors from July 10th to July 12th, 2023. During this engagement, Zellic reviewed Hyperliquid's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

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

- Is the interaction between the bridge and L1, as well as the locking logic, implemented correctly and securely?
- Are there any vulnerabilities present in the bridge that could potentially be exploited to steal funds?
- Are there any bugs that result in behavior that deviates from the intended L1 behavior? This includes issues such as misbehaving validators or malfunctioning validator set transitions.

1.2 Non-goals and Limitations

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

- Other smart contracts part of the Hyperliquid system
- Off-chain components, such as validators
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

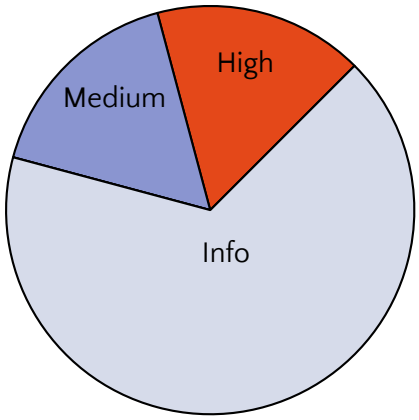
1.3 Results

During our assessment on the scoped Hyperliquid contracts, we discovered six findings. No critical issues were found. One was of high impact, one was of medium impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	4



2 Introduction

2.1 About Hyperliquid

Hyperliquid is an order book perpetual futures DEX. Hyperliquid runs on its own L1, a purpose-built blockchain using the Tendermint SDK that is performant enough to operate the whole platform –– every order, cancel, trade, and liquidation happens transparently on chain with block latency < 1 second.

2.2 Methodology

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

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

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

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

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

Code maturity. We look for potential improvements in the code base in general.

We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

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

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

2.3 Scope

The engagement involved a review of the following targets:

Hyperliquid Contracts

Repository	https://github.com/hyperliquid-dex/contracts
Version	contracts: `43b5267c58778e5e24640c9abac06cb608d63c40`
Programs	• Bridge2 • Signature
Type	Solidity
Platform	Arbitrum

2.4 Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Kuilin Li, Engineer
kuilin@zellic.io

Daniel Lu, Engineer
daniel@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 10, 2023	Kick-off call
July 10, 2023	Start of primary review period
July 12, 2023	End of primary review period
August 8, 2023	Closing call

3 Detailed Findings

3.1 Withdrawal finalization does not work

- **Target**: Bridge2
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The single entry point for finalizing withdrawals is the `batchedFinalizeWithdrawals` function, which iterates over an array of messages and calls `finalizeWithdrawal` on each.

Both functions have the `nonReentrant` modifier.

```
function batchedFinalizeWithdrawals(
  bytes32[] calldata messages
) external nonReentrant whenNotPaused {
  checkFinalizer(msg.sender);

  uint64 end = uint64(messages.length);
  for (uint64 idx; idx < end; idx++) {
    finalizeWithdrawal(messages[idx]);
  }
}

function finalizeWithdrawal(bytes32 message)
    private nonReentrant whenNotPaused {
  require(!finalizedWithdrawals[message], "Withdrawal already finalized");
  Withdrawal memory withdrawal = requestedWithdrawals[message];

  checkDisputePeriod(withdrawal.requestedTime,
    withdrawal.requestedBlockNumber);

  finalizedWithdrawals[message] = true;
  usdcToken.transfer(withdrawal.user, withdrawal.usdc);
  emit FinalizedWithdrawal(
    FinalizedWithdrawalEvent({
      user: withdrawal.user,
```

```
      usdc: withdrawal.usdc,
      nonce: withdrawal.nonce,
      message: withdrawal.message
    })
  );
}
```

Impact

Any finalization attempt will immediately revert because of the `nonReentrant` modifier on `finalizeWithdrawal`, preventing any withdrawal from the bridge from being finalized.

We classified this issue as high severity due to the fundamental importance of the finalization step for the contract operation.

Recommendations

We recommend removing the `nonReentrant` modifier from the private `finalizeWithdrawal` function and adding test cases to ensure its correct behavior.

Remediation

This issue has been acknowledged by the Hyperliquid contributors, and a fix was implemented in commit e5b7e068.

3.2 Disputed actions are not blocked by validator rotation

- **Target**: Bridge2
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

The bridge implements a two-step mechanism for performing withdrawals and validator set changes. First, a request authorizing the action has to be submitted. The request has to be signed by a two thirds majority of validators. If the request is valid, it is recorded in the contract storage.

The second step, finalization, actually performs the requested action and can only occur after a dispute period has elapsed. The dispute period gives the opportunity to pause the contract in the event of one or more validators being compromised. Unpausing the contract also requires to rotate the validator set, allowing replacement of the compromised validators.

However, the current implementation does not allow to remove pending operations. For example, if a malicious withdrawal was detected and the contract was paused, the operation would stay pending and could be processed when the contract is unpaused.

Impact

If a sufficiently large subset of hot wallets is compromised, the dispute period does not effectively allow malicious withdrawals or validator set updates to be blocked. Even if validators are rotated, pending actions would still be able to be finalized when the contract is unpaused.

Recommendations

We recommend adding a mechanism for invalidating pending messages. For example, this could be implemented in the `emergencyUnlock` function.

Remediation

This issue has been acknowledged by the Hyperliquid contributors, and a fix was implemented in commit 8c4a182a.

3.3 Missing message validation may allow griefing

- **Target**: Bridge2
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The `finalizeWithdrawals` function does not check that the given message corresponds to an existing withdrawal request. Since the uninitialized values of the corresponding withdrawal data will be zero, the call to `checkDisputePeriod` will pass:

```
function checkDisputePeriod(uint64 time, uint64 blockNumber)
    private view {
  require(
    block.timestamp > time + disputePeriodSeconds &&
      (uint64(block.number) - blockNumber) * blockDurationMillis
    > 1000 * disputePeriodSeconds,
    "Still in dispute period"
  );
}
```

Impact

When messages do not correspond to existing withdrawals, they will cause a transfer of zero tokens to the zero address. In the case of USDC on Arbitrum, this will currently result in a revert. However, if this logic is reused for other ERC-20 tokens, there is no guarantee that such a call will be blocked.

Then, although the message does not correspond to an existing withdrawal, it will be marked as finalized, anyway:

```
finalizedWithdrawals[message] = true;
usdcToken.transfer(withdrawal.user, withdrawal.usdc);
```

Thus, any future attempts to finalize that message will fail. If an attacker is able to

1. predict upcoming nonces, or
2. front-run withdrawal requests,

they would be able to block real withdrawals from being finalized.

Recommendations

Consider checking that messages correspond to existing withdrawals during the finalization process. In the case of USDC, this has the additional benefit of improving the error message.

Remediation

This issue has been acknowledged by the Hyperliquid contributors, and a fix was implemented in commit 1c8d3333.

3.4 Signatures may be reused across different contracts

- **Target**: Signature
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

On the Arbitrum side, the bridge operates by allowing users to perform actions approved by validators. For instance, to request a withdrawal, the user needs at least two thirds of the validators (if validator power is equally distributed) to sign off using their in-memory hot keys. The bridge checks these signatures, and if the user is indeed permitted to perform the withdrawal, it transfers them the USDC.

Currently, signatures include a domain separator to prevent reuse across different chains and projects. This is important to ensure that they are specific to the context in which they are used and cannot be maliciously repurposed.

```
function makeDomainSeparator() view returns (bytes32) {
  return
    keccak256(
      abi.encode(
        EIP712_DOMAIN_SEPARATOR,
        keccak256(bytes("Exchange")),
        keccak256(bytes("1")),
        block.chainid,
        VERIFYING_CONTRACT
      )
    );
}
```

However, the signatures do not include the contract or token address.

Impact

The fact that the domain separator does not by default include any contract-specific data introduces some maintenance risk: the protocol must ensure that signatures cannot be reused across contracts on the same chain.

For instance, if the exact same contract were used for a different ERC-20 token, an attacker may be able to steal funds by replaying withdrawal messages.

Recommendations

We recommend including either the contract address or the token address in signatures (either the domain separator or in the message itself) to increase robustness and avoid future issues.

Remediation

This issue has been acknowledged by the Hyperliquid contributors, and a fix was implemented in commit 97225667.

3.5 Withdrawal and validator update signatures include no action

- **Target**: Bridge2
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

To include important parameters in signatures, the bridge packs them together and hashes them. This data is stored in the `connectionId` slot of the `Agent` struct, which has an associated function `hash` for creating the actual signed message.

```
struct Agent {
  string source;
  bytes32 connectionId;
}
```

In some functions, the hashed data in `connectionId` includes the name of an action:

```
Agent memory agent = Agent("a", keccak256(abi.encode("modifyLocker",
    locker, isLocker, nonce)));
```

However, the `connectionIds` used in the `requestWithdrawal` and `updateValidatorSet` `Agent`'s do not. Instead, they rely on the arguments being different to prevent valid signatures from being used in the wrong function. From `requestWithdrawal` and `updateValidatorSet`:

```
Agent memory agent = Agent("a", keccak256(abi.encode(msg.sender, usdc,
    nonce)));
Agent memory agent = Agent(
  "a",
  keccak256(
    abi.encode(
      newValidatorSet.epoch,
      newValidatorSet.hotAddresses,
      newValidatorSet.coldAddresses,
      newValidatorSet.powers
    )
  )
```

```
);
```

Impact

This introduces some maintenance risk: updating these signature arguments may have the unintended consequence of allowing confusion between the two types. That might allow users to use withdrawal signatures to maliciously update validators.

Recommendations

We recommend consistently prefixing all messages with the action to guarantee that changes in arguments do not cause bugs.

Remediation

This issue has been acknowledged by the Hyperliquid contributors, and a fix was implemented in commit b198269c.

3.6 Unchecked `transferFrom` return value in `deposit`

- **Target**: Bridge2
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The bridge handles deposits by transferring an amount of USDC from the sender and emitting a corresponding event.

```
// An external function anyone can call to deposit usdc into the bridge.
// A deposit event will be emitted crediting the L1 with the usdc.
function deposit(uint64 usdc) external whenNotPaused nonReentrant {
  address user = msg.sender;
  emit Deposit(DepositEvent({ user: user, usdc: usdc }));
  usdcToken.transferFrom(user, address(this), usdc);
}
```

The return value of the `transferFrom` call is not checked. Although USDC on Arbitrum will currently revert when the transfer is not permitted (e.g., if the approval amount is insufficient), this behavior is not required by the ERC-20 specification. Instead, `transferFrom` must return a boolean indicating whether it was successful.

Impact

If this logic were reused for other token types, it could potentially result in a loss of funds. Users would be able to arbitrarily call `deposit` and cause the corresponding event to be emitted.

Recommendations

Consider checking the return value as well, or use the SafeERC20 OpenZeppelin library and its `safeTransferFrom` function.

Remediation

This issue has been acknowledged by the Hyperliquid contributors, and a fix was implemented in commit 57df1c10.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Signature checks do not require arrays of signers

The `requestWithdrawal`, `modifyLocker`, and `emergencyUnlock` functions accept a `signers` array. This is passed into `checkValidatorSignatures` to verify that the action is authorized:

```
function checkValidatorSignatures(
  bytes32 message,
  ValidatorSet memory activeValidatorSet, // Active set of all L1
    validators
  address[] memory signers, // Subsequence of the active L1 validators
    that signed the message
  Signature[] memory signatures,
  bytes32 validatorSetHash
) private view {
```

The array of signers is not necessary for performing the validator check. Instead, `checkValidatorSignatures` could simply recover the array of signers and check that it is a sufficiently weighted subsequence of the chosen validator set. Omitting this argument from these external functions would simplify the interface and save gas.

Note: this issue was addressed in commit d355798d; the signers array is no longer required to be provided by the user, and the signer's address recovered from the individual signatures is used instead.

4.2 Possible improvements for contract transparency

Currently, the address of the bridged token is held as a private variable in contract storage. We encourage the Hyperliquid contributors to consider making it public, which would let users more easily verify the asset bridged.

Additionally, `nValidators` is currently exposed for convenience, but it may inadvertently misrepresent the number of validators. For example, if multiple members of

the validator set have the same address, the power of the validator is the sum of each occurrence's power. However, `nValidators` would count each occurrence separately.

Note: the visibility of the `usdcToken` variable containing the address of the bridged token was changed to public in commit 7a2f1a82.

5 Threat Model

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

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

> Please note that our threat model was based on commit 43b5267c, which represents a specific snapshot of the codebase. Therefore, it's important to understand that the absence of certain tests in our report may not reflect the current state of the test suite.
>
> During the remediation phase, the Hyperliquid contributors took proactive steps to address the findings by adding unit test cases for `batchedFinalizeWithdrawals` in commit 6a1ddbc8. This demonstrates their dedication to enhancing the code quality and overall reliability of the system, which is commendable.

5.1 Module: Bridge2.sol

Function: `batchedFinalizeWithdrawals(byte[32][] messages)`

This function can be used to finalize a batch of pending withdrawals, transferring the owed USDC amounts.

Inputs

- `messages`
  - **Control**: Arbitrary.
  - **Constraints**: Each message must not be already finalized and correspond to a withdrawal for which the dispute period has elapsed.
  - **Impact**: Hashes identifying the withdrawals to be finalized.

Branches and code coverage (including function calls)

Intended branches

- For each withdrawal message, it checks the dispute period and ensures the

withdrawal was not already processed, then transfers the tokens.
  - ☐ Test coverage

Negative behavior

- Reverts if the withdrawal was already processed.
  - ☐ Negative test
- Reverts if the dispute period has not elapsed.
  - ☐ Negative test

Function call analysis

- `rootFunction → finalizeWithdrawal(messages[idx])`
  - **What is controllable?** `messages[idx]`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not a concern (USDC makes no external calls).
- `finalizeWithdrawal → checkDisputePeriod(withdrawal.requestedTime, withdrawal.requestedBlockNumber)`
  - **What is controllable?** Nothing directly.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `finalizeWithdrawal → usdcToken.transfer(withdrawal.user, withdrawal.usdc)`
  - **What is controllable?** Nothing directly.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not a concern (USDC makes no external calls).

Function: `changeBlockDurationMillis(uint64 newBlockDurationMillis, uint64 nonce, ValidatorSet activeColdValidatorSet, address[] signers, Signature[] signatures)`

This function can be used to change the block duration.

Inputs

- `newBlockDurationMillis`
  - **Control**: Arbitrary.

    - **Constraints**: None.
    - **Impact**: New block duration.
- `nonce`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Nonce used as part of the signed hash.
- `activeColdValidatorSet`
    - **Control**: Arbitrary.
    - **Constraints**: Hash must match the stored validator set hash.
    - **Impact**: Active set of validators — used to validate signatures.
- `signers`
    - **Control**: Arbitrary.
    - **Constraints**: Length must match `signatures`.
    - **Impact**: Addresses of the signers of the action.
- `signatures`
    - **Control**: Arbitrary.
    - **Constraints**: Each element must be a valid signature for the corresponding address in `signers`.
    - **Impact**: Signatures authorizing the action.

Branches and code coverage (including function calls)

Intended branches

- Checks that the same message has not been used, checks the validity of the validator signatures, and updates the block duration.
    - ☑ Test coverage

Negative behavior

- Reverts if the same message has already been used.
    - ☐ Negative test
- Reverts if the hash of the provided validator set does not match the stored one.
    - ☐ Negative test
- Reverts if a signature does not correspond with the signer.
    - ☐ Negative test
- Reverts if the signers' cumulative voting power is insufficient.
    - ☐ Negative test
- Reverts if the length of the signers and signatures do not match.
    - ☐ Negative test

Function call analysis

- `rootFunction → hash(agent)`
  - **What is controllable?** `agent`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable, used to identify the transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts and reentrancy cannot happen.
- `rootFunction → checkMessageNotUsed(message)`
  - **What is controllable?** `message`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `rootFunction → checkValidatorSignatures( … )`
  - **What is controllable?** `message` (some parts of the hash), `activeColdValidatorSet`, `signers`, and `signatures`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).

Function: `changeDisputePeriodSeconds(uint64 newDisputePeriodSeconds, uint64 nonce, ValidatorSet activeColdValidatorSet, address[] signers, Signature[] signatures)`

This function can be used to change the dispute period.

Inputs

- `newDisputePeriodSeconds`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New dispute period.
- `nonce`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Nonce used as part of the signed hash.
- `activeColdValidatorSet`
  - **Control**: Arbitrary.
  - **Constraints**: Hash must match the current cold validator set hash.
  - **Impact**: Current active validator set — used to validate the signatures.

- `signers`
  - **Control**: Arbitrary.
  - **Constraints**: Length must match `signatures`.
  - **Impact**: Addresses of the signers of the action.
- `signatures`
  - **Control**: Arbitrary.
  - **Constraints**: Each element must be a valid signature for the corresponding address in `signers`.
  - **Impact**: Signatures authorizing the action.

Branches and code coverage (including function calls)

Intended branches

- Checks that the same message has not already been used and that the signature is valid, then changes the dispute period.
  ☑ Test coverage

Negative behavior

- Reverts if the same message has already been used.
  ☐ Negative test
- Reverts if the hash of the provided validator set does not match the stored one.
  ☐ Negative test
- Reverts if a signature does not correspond with the signer.
  ☐ Negative test
- Reverts if the signers' cumulative voting power is insufficient.
  ☐ Negative test
- Reverts if the length of the signers and signatures do not match.
  ☐ Negative test

Function call analysis

- `rootFunction → hash(agent)`
  - **What is controllable?** `agent`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable — used to identify the transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts and reentrancy cannot happen.
- `rootFunction → checkMessageNotUsed(message)`
  - **What is controllable?** `message`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** N/A.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `rootFunction → checkValidatorSignatures( … )`
  - **What is controllable?** `message` (some parts of the hash), `activeColdValidatorSet`, `signers`, and `signatures`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).

Function: `deposit(uint64 usdc)`

This function can be used to deposit USDC into the bridge.

Inputs

- `usdc`
  - **Control**: Arbitrary.
  - **Constraints**: None (apart from the caller having sufficient balance).
  - **Impact**: Amount to be deposited.

Branches and code coverage (including function calls)

Intended branches

- Deposits USDC into the contract and emits a corresponding event.
  ☑ Test coverage

Negative behavior

- Reverts if user balance is insufficient or USDC transfer fails.
  ☐ Negative test

Function call analysis

- `rootFunction → usdcToken.transferFrom(user, address(this), usdc)`
  - **What is controllable?** `usdc`.
  - **If return value controllable, how is it used and how can it go wrong?** Not controlled nor used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (USDC makes no external calls).

Function: `emergencyLock()`

This function can be used to lock the contract.

Branches and code coverage (including function calls)

Intended branches

- Ensures the caller is authorized and locks the contract.
  ☑ Test coverage

Negative behavior

- Reverts if the caller is not authorized.
  ☑ Negative test

Function: `emergencyUnlock(ValidatorSetUpdateRequest newValidatorSet, ValidatorSet activeColdValidatorSet, address[] signers, Signature[] signatures, uint64 nonce)`

This function can be used to change the validator set and unlock the contract.

Inputs

- `newValidatorSet`
  - **Control**: Arbitrary.
  - **Constraints**: `hotAddresses`, `coldAddresses`, and `powers` array lengths must match; `newValidatorSet.epoch > activeHotValidatorSet.epoch`; and cumulative power must be greater than zero.
  - **Impact**: New validator set.
- `activeColdValidatorSet`
  - **Control**: Arbitrary.
  - **Constraints**: Hash must match the stored active cold validator set hash.
  - **Impact**: Set of active cold validators — used to validate signatures.
- `signers`
  - **Control**: Arbitrary.
  - **Constraints**: Length must match `signatures`.
  - **Impact**: List of signers for the unlock action.
- `signatures`
  - **Control**: Arbitrary.
  - **Constraints**: Each element must be a valid signature for the corresponding address in `signers`.
  - **Impact**: Signatures authorizing the action.

- `nonce`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Nonce used as part of the signed message.

Branches and code coverage (including function calls)

Intended branches

- Checks that the same action has not already been performed and that the signatures are valid, updates the validator set, and unlocks the contract.
  ☑ Test coverage

Negative behavior

- Reverts if the same message has already been used.
  ☐ Negative test
- Reverts if the hash of the provided validator set does not match the stored one.
  ☐ Negative test
- Reverts if a signature does not correspond with the signer.
  ☐ Negative test
- Reverts if the signers' cumulative voting power is insufficient.
  ☐ Negative test
- Reverts if the length of the signers and signatures do not match.
  ☐ Negative test

Function call analysis

- `rootFunction` → `checkMessageNotUsed(message)`
  - **What is controllable?** `message`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `rootFunction` → `updateValidatorSetInner( ... )`
  - **What is controllable?** `newValidatorSet`, `activeColdValidatorSet`, `signers`, `signatures`, and `message` (indirectly, it is a hash).
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `updateValidatorSetInner` → `checkNewValidatorPowers(newValidatorSet.powers)`
  - **What is controllable?** `newValidatorSet.powers`.

  - **If return value controllable, how is it used and how can it go wrong?** Used as the sum of the voting powers.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `updateValidatorSetInner` → `checkValidatorSignatures(message, activeValidatorSet, signers, signatures, validatorSetHash)`
  - **What is controllable?** `message`, `activeValidatorSet`, `signers`, and `signatures`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `checkValidatorSignatures` → `makeValidatorSetHash(activeValidatorSet)`
  - **What is controllable?** `activeValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Not meaningfully controllable, compared against the expected validator set hash.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `checkValidatorSignatures` → `recoverSigner(message, signatures[signerIdx], domainSeparator)`
  - **What is controllable?** `message` and `signatures[signerIdx]`.
  - **If return value controllable, how is it used and how can it go wrong?** Not meaningfully controllable, compared against the expected signer; it is not possible to forge a signer's address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `updateValidatorSetInner` → `makeValidatorSetHash(newHotValidatorSet)`
  - **What is controllable?** `newHotValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the hash of the new hot validator set.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `updateValidatorSetInner` → `makeValidatorSetHash(newColdValidatorSet)`
  - **What is controllable?** `newHotValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the hash of the new cold validator set.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `rootFunction` → `finalizeValidatorSetUpdateInner()`
  - **What is controllable?** N/A.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `rootFunction → _unpause()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts or reentrancy are not possible.

Function: `finalizeValidatorSetUpdate()`

This function can be called to finalize a validator set update, making it effective.

Branches and code coverage (including function calls)

Intended branches

- Check that an update is pending and that the dispute period has elapsed, then finalizes the update by updating various storage variables.
  ☑ Test coverage

Negative behavior

- Reverts if no update is pending.
  ☐ Negative test
- Reverts if the dispute period has not elapsed.
  ☐ Negative test

Function call analysis

- `rootFunction → checkDisputePeriod(pendingValidatorSetUpdate.updateTime, pendingValidatorSetUpdate.updateBlockNumber)`
  - **What is controllable?** Nothing directly.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `rootFunction → finalizeValidatorSetUpdateInner()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).

Function: `modifyLocker(address locker, bool isLocker, uint64 nonce, ValidatorSet activeColdValidatorSet, address[] signers, Signature[] signatures)`

This function can be used to grant or revoke authorization for the locker role, which grants the ability to pause the contract.

Inputs

- `locker`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the locker.
- `isLocker`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: If true, the permission will be granted; otherwise, it will be revoked.
- `nonce`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Nonce used as part of the signed action.
- `activeColdValidatorSet`
  - **Control**: Arbitrary.
  - **Constraints**: The hash must match the stored cold validator set hash.
  - **Impact**: The currently active set of cold validators.
- `signers`
  - **Control**: Arbitrary.
  - **Constraints**: Length must match `signatures`.
  - **Impact**: Addresses of the signers for the request.
- `signatures`
  - **Control**: Arbitrary.
  - **Constraints**: Each element must be a valid signature for the corresponding `signers` entry.
  - **Impact**: Signatures authorizing the action.

Branches and code coverage (including function calls)

Intended branches

- Checks for signature reuse, checks the signatures' validity, and grants/revokes

permissions.
  ☑ Test coverage

Negative behavior

- Reverts if the same signature was already used.
  ☐ Negative test
- Reverts if the validator set does not match the expected one.
  ☐ Negative test
- Reverts if the sum of the signers' voting power is insufficient.
  ☐ Negative test
- Reverts if the length of the signers and signatures do not match.
  ☐ Negative test

Function call analysis

- `rootFunction → hash(agent)`
  - **What is controllable?** `agent`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable — used to identify the transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts and reentrancy cannot happen.
- `rootFunction → checkMessageNotUsed(message)`
  - **What is controllable?** `message`, indirectly (some parts of the hash).
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `rootFunction → checkValidatorSignatures( … )`
  - **What is controllable?** `message` (some parts of the hash), `activeColdValidatorSet`, `signers`, and `signatures`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).

Function: `requestWithdrawal(uint64 usdc, uint64 nonce, ValidatorSet hot ValidatorSet, address[] signers, Signature[] signatures)`

This function can be used to request a withdrawal from the bridge.

Inputs

- `usdc`

  - **Control**: Arbitrary.
  - **Constraints**: None directly (must match the signature).
  - **Impact**: Amount to be withdrawn.
- `nonce`
  - **Control**: Arbitrary.
  - **Constraints**: None directly (must match the signature).
  - **Impact**: Nonce used to deduplicate signatures.
- `hotValidatorSet`
  - **Control**: Arbitrary.
  - **Constraints**: The hash must match the current hot validator set hash.
  - **Impact**: Validator set.
- `signers`
  - **Control**: Arbitrary.
  - **Constraints**: Elements must match the corresponding element in `signatures`.
  - **Impact**: Array of addresses that signed the request.
- `signatures`
  - **Control**: Array.
  - **Constraints**: Must be valid signatures for the hash of the withdrawal request determined by the other parameters.
  - **Impact**: Signatures authorizing the transfer.

Branches and code coverage (including function calls)

Intended branches

- Validates the validator set and the provided signature, and it records the pending withdrawal.
  ☑ Test coverage

Negative behavior

- Reverts if the provided validator set does not match the recorded validator set hash.
  ☐ Negative test
- Reverts if a signature does not match the corresponding signer.
  ☐ Negative test
- Reverts if the cumulative signing power is insufficient.
  ☑ Negative test
- Reverts if the same withdrawal was already requested.
  ☐ Negative test

Function call analysis

- `rootFunction → hash(agent)`
  - **What is controllable?** `agent`, indirectly (some parts of the hash)
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable, used to identify the transfer
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts and reentrancy can't happen
- `rootFunction → checkValidatorSignatures(message, hotValidatorSet, signers, signatures, hotValidatorSetHash)`
  - **What is controllable?** `message`, `hotValidatorSet`, `signers`, and `signatures`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `checkValidatorSignatures → makeValidatorSetHash(activeValidatorSet)`
  - **What is controllable?** `activeValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Not meaningfully controllable, compared against the expected validator set hash.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `checkValidatorSignatures → recoverSigner(message, signatures[signerIdx], domainSeparator)`
  - **What is controllable?** `message` and `signatures[signerIdx]`.
  - **If return value controllable, how is it used and how can it go wrong?** Not meaningfully controllable, compared against the expected signer; it is not possible to forge a signer's address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).

Function: `updateValidatorSet(ValidatorSetUpdateRequest newValidatorSet, ValidatorSet activeHotValidatorSet, address[] signers, Signature[] signatures)`

This function can be called to initiate an update to the validator set.

Inputs

- `newValidatorSet`
  - **Control**: Arbitrary.
  - **Constraints**: `hotAddresses`, `coldAddresses`, and `powers` array lengths must

match; `newValidatorSet.epoch > activeHotValidatorSet.epoch`; and cumulative power must be greater than zero.
    - **Impact**: New validator set.
- `activeHotValidatorSet`
    - **Control**: Arbitrary.
    - **Constraints**: `makeValidatorSetHash(activeHotValidatorSet) == hotValidatorSetHash`.
    - **Impact**: Active validator set — used to validate the request.
- `signers`
    - **Control**: Arbitrary.
    - **Constraints**: Length must match `signatures`.
    - **Impact**: Addresses of the signers of the request.
- `signatures`
    - **Control**: Arbitrary.
    - **Constraints**: Length must match `signers`, and signatures must correspond with `signers` entries.
    - **Impact**: Signatures authorizing the validator set update.

Branches and code coverage (including function calls)

Intended branches

- Validates the new set, checks the signature from the current set, and stores the pending validator set update.
  ☑ Test coverage

Negative behavior

- Reverts if the provided active validator hash set does not match.
  ☑ Negative test
- Reverts if the lengths of the arrays in the new validator set do not match.
  ☐ Negative test
- Reverts if the epoch of the new validator set is not greater than the epoch of the active validator set.
  ☑ Negative test
- Reverts if a signature does not match the corresponding signer.
  ☑ Negative test
- Reverts if a signature is invalid.
  ☐ Negative test
- Reverts if the signers' total voting power is insufficient.
  ☑ Negative test

- Reverts if the new total voting power is zero.
  - ☐ Negative test

Function call analysis

- `rootFunction → makeValidatorSetHash(activeHotValidatorSet)`
  - **What is controllable?** `activeHotValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Compared against the expected hot validator set hash.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `rootFunction → hash(agent)`
  - **What is controllable?** `agent` (indirectly).
  - **If return value controllable, how is it used and how can it go wrong?** Hash representing the action being performed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `rootFunction → updateValidatorSetInner( ... )`
  - **What is controllable?** `newValidatorSet`, `activeHotValidatorSet`, `signers`, `signatures`, and `message` (indirectly, it is a hash).
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `updateValidatorSetInner → checkNewValidatorPowers(newValidatorSet.powers)`
  - **What is controllable?** `newValidatorSet.powers`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the sum of the voting powers.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `updateValidatorSetInner → checkValidatorSignatures(message, activeValidatorSet, signers, signatures, validatorSetHash)`
  - **What is controllable?** `message`, `activeValidatorSet`, `signers`, and `signatures`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `checkValidatorSignatures → makeValidatorSetHash(activeValidatorSet)`
  - **What is controllable?** `activeValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Not

meaningfully controllable, compared against the expected validator set hash.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `checkValidatorSignatures` → `recoverSigner(message, signatures[signerIdx], domainSeparator)`
  - **What is controllable?** `message` and `signatures[signerIdx]`.
  - **If return value controllable, how is it used and how can it go wrong?** Not meaningfully controllable, compared against the expected signer; it is not possible to forge a signer's address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy cannot happen (no external calls).
- `updateValidatorSetInner` → `makeValidatorSetHash(newHotValidatorSet)`
  - **What is controllable?** `newHotValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the hash of the new hot validator set.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).
- `updateValidatorSetInner` → `makeValidatorSetHash(newColdValidatorSet)`
  - **What is controllable?** `newHotValidatorSet`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the hash of the new cold validator set.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (no external calls).

6 Audit Results

At the time of our audit, the audited code was not deployed to the Arbitrum Mainnet.

During our assessment on the scoped Hyperliquid contracts, we discovered six findings. No critical issues were found. One was of high impact, one was of medium impact, and the remaining findings were informational in nature. The Hyperliquid contributors acknowledged all findings and implemented fixes.

6.1 Disclaimer

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

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

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