Prepared for
Hyperliquid Protocol Foundation
The Hyperliquid contributors

Prepared by
Filippo Cremonese
Yuhang Wu
Zellic

November 27, 2023

Hyperliquid

Smart Contract Patch Review

Contents

About Zellic 3

1. Executive Summary 3

1.1. Goals of the Assessment 4

1.2. Non-goals and Limitations 4

1.3. Results 5

2. Introduction 5

2.1. About Hyperliquid 6

2.2. Methodology 6

2.3. Scope 8

2.4. Project Overview 8

2.5. Project Timeline 9

3. Discussion 9

3.1. Changes since previous assessment 10

3.2. Error handling in `depositWithPermit` 11

4. Assessment Results 11

4.1. Disclaimer 12

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 November 23rd to November 24th, 2023. During this engagement, Zellic reviewed Hyperliquid's code for security vulnerabilities, design issues, and general weaknesses in security posture.

The engagement was exclusively focused on the changes made since the previous security assessment, specifically in commits `204f017e` ↗, `2ad53ac9` ↗, and `9189a7dd` ↗.

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.

The engagement was exclusively limited to the changes made in commits `204f017e` ↗, `2ad53ac9` ↗, and `9189a7dd` ↗.

1.3. Results

During our assessment on the scoped Hyperliquid contracts, there were no security vulnerabilities discovered.

Zellic recorded its notes and observations from the assessment for The Hyperliquid contributors's benefit in the Discussion section (3. ↗).

Breakdown of Finding Impacts

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

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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (3. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Hyperliquid Contracts

Repository	https://github.com/hyperliquid-dex/contracts ↗

Versions	`204f017e52954d9c6814f1427bb66aa78bc6c07c`
	`2ad53ac96bd6103b4194ace23041fc20fedd96a9`
	`9189a7ddeb410e835548c1190414949f04311bfa`

Programs	• Bridge2.sol • Signature.sol
Type	Solidity
Platform	Arbitrum

2.4. Project Overview

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

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

Yuhang Wu
Engineer
yuhang@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 23, 2023	Start of primary review period
November 24, 2023	End of primary review period

3. 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.

3.1. Changes since previous assessment

The following changes were observed in the in-scope commits:

Withdrawals support specifying destination

Withdrawals now support specifying a destination, instead of assuming the destination is `msg.sender`. All the information regarding the withdrawal, including user, destination, and amount, still require a valid signature from a supermajority of the validators to take effect.

Batch functions

A batch variant of `requestWithdrawal` was introduced. This change required to move the `nonReentrant` modifier to the public batch versions. This does not introduce a security issue because the nonbatch variant has been restricted to internal visibility. We note that the `whenNotPaused` modifier could be removed in the internal function for a minor gas-efficiency gain.

All nonbatch variants are now not directly externally reachable; the batch variants call the nonbatch variants to process each operation. Because of this, the nonbatch variants were modified to log an event and return early instead of reverting if an error occurs. This is required to prevent a single error from reverting an entire batch of operations. We note that this is only partially effective, as reverts originating from external calls are not caught and will therefore revert the entire batch.

Support deposits using `ERC20Permit`

Two new functions, `batchedDepositWithPermit` and `depositWithPermit`, were introduced to support depositing on behalf of a third party using a signature via the `ERC20Permit` interface.

Configurable locker threshold required to lock the bridge

Locking (pausing) the bridge now requires votes from a configurable number of lockers.

Changing the threshold requires signatures from a two-thirds supermajority of the cold validator set.

Lockers can vote to pause and can also revoke their vote, but they cannot unpause the contract, which requires a two-thirds supermajority of the cold validators.

Adding and removing lockers still require a supermajority of the hot validators.

Minor change to the required validator power

The `checkValidatorSignatures` function was changed to require the cumulative power of the signatures being processed to be greater than two thirds of the total voting power of all the val-

idators.

Previously, the function required the cumulative power to be greater than or equal to the two-thirds threshold.

Other minor changes

This includes the following:

- The `changeDisputePeriodSeconds` does not require the contract to not be paused.
- The `invalidateWithdrawals` does not require the contract to not be paused, and it emits one event per invalidated withdrawal instead of a single event with all invalidated withdrawals.
- The `changeBlockDurationMillis` does not require the contract to be paused.

Usage of correct source for block number

Arbitrum `block.number` lags behind the actual `block.number` on mainnet Ethereum. The equivalent of the Ethereum Mainnet block number is obtained using facilities specifically provided by Arbitrum.

3.2. Error handling in `depositWithPermit`

The function `depositWithPermit` allows a user to deposit USDC into the contract with a permit signature, bypassing the need for a separate approval transaction. The function's current implementation includes a try-catch block surrounding the `permit` call but not the subsequent `safeTransferFrom` call.

Ideally, `safeTransferFrom` could not fail unexpectedly, but it is better to put the entire function in a try-catch block to catch any unexpected errors. This will make the function more clear and robust, and also the user will be able to see the error message.

4. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Arbitrum Mainnet.

During our assessment on the scoped Hyperliquid contracts, there were no security vulnerabilities discovered.

4.1. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.