HERE Wallet

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen notdeghost@osec.io

Maher Azzouzi maher@osec.io

Contents

Appendices

01 | Executive Summary

Overview

HERE Wallet engaged OtterSec to perform an assessment of the `storage-contract` program. This assessment of the source code was conducted between December 19th, 2022 and January 6th, 2023 by 2 engineers. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 3 findings total.

In particular, we identified a race condition vulnerability in the `update_total_balance_callback` function that could result in incorrect balances being used (OS-NHW-ADV-00). We also noted issues with storage tracking (OS-NHW-ADV-01).

In addition, we made recommendations around modifying the `ft_resolve_transfer` function to pay dividends to both the sender and the receiver (OS-NHW-SUG-00).

Overall, we commend the HERE Wallet team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/here-wallet/storage-contract. This audit was performed against commit `2b94f77`.

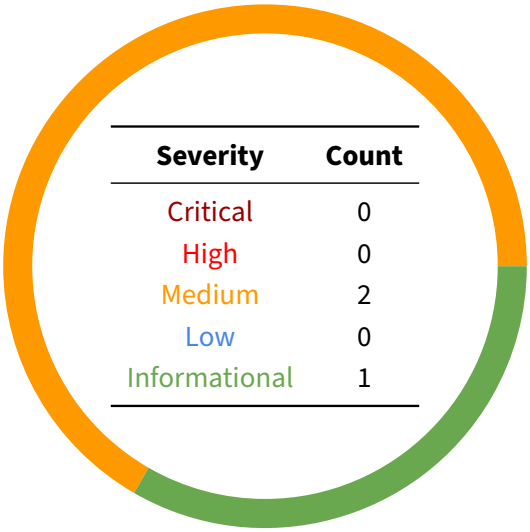
A brief description of the programs is as follows.

Name	Description
HERE Wallet	NEAR storage contract for HERE wallet.

03 | Findings

Overall, we report 3 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	2
Low	0
Informational	1

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-NHW-ADV-00	Medium	Resolved	The `update_total_balance_callback()` doesn't properly account for asynchronous execution.
OS-NHW-ADV-01	Medium	Resolved	Storage staking costs should be properly tracked in the contract

OS-NHW-ADV-00 [med] [resolved] | Update Total Balance Race

Description

The `update_total_balance_callback` function in the smart contract is vulnerable to a race condition issue. If an attacker calls `update_total_balance` and a stake is made before the callback is finished, the total balance may not be correctly updated.

Proof of Concept

1. The attacker calls `update_total_balance` as it is a public function and simultaneously initiates a stake in the contract.
2. The stake is completed before the callback is finished and the contract's balance is changed.
3. When `update_total_balance_callback` is executed, it calculates the total balance at the time it was called, resulting in an incorrect total balance.

Remediation

The `update_total_balance_callback` function should be modified to properly handle the potential for a stake to be made while the function is being executed. This could involve adding additional checks and/or using a mutex to synchronize access to the relevant variables.

Patch

Fixed in 9d3f47e.

OS-NHW-ADV-01 [med] [resolved] | Unaccounted Storage Costs

Description

The `internal_register_account` function does not account for storage staking when creating new users. Eventually, users will no longer be able to create an account due to this mismatch in storage accounting.

Remediation

Charge a nominal storage fee for the creation of a new user account.

Patch

Resolved in #6.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-NHW-SUG-00	The `internal_ft_resolve_transfer` function should pay dividends to the receiver/sender because it can alter token balances.

OS-NHW-SUG-00 [resolved] | Resolve Transfer Accounting

Description

The `ft_resolve_transfer` function should pay dividends to both the sender and the receiver of the transfer. This is because this function can alter user balances, and needs to settle any accrued dividends prior to modifying balances.

Remediation

One way to implement this suggestion is to add calls to the `pay_dividends_to_user` function for both the sender and the receiver in the `ft_resolve_transfer` function. Fixed in 9d3f47e.

contract/src/ft.rs DIFF

```
@@ -56,6 +56,11 @@ impl FungibleTokenResolver for Contract {
         amount: U128,
     ) -> U128 {
         let sender_id: AccountId = sender_id.into();
+        // update min balance
+        self.pay_dividends_to_user(&sender_id.to_string());
+        // update receiver min balance
+        self.pay_dividends_to_user(&receiver_id.to_string());
+
         let (used_amount, _) =
             self.token
                 .internal_ft_resolve_transfer(&sender_id, receiver_id,
    ↪  amount);
```

Patch

Fixed in 9d3f47e.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.