Smart Contract Security Assessment

September 27, 2023

Project Name:

Force USD

Prepared by:

HYDN

Contents

Introduction

About HYDN

HYDN is an industry leader in blockchain security and smart contract audits. Founded by Warren Mercer, a world renowned cybersecurity and blockchain expert, who has previously held senior roles at NYSE, Cisco, and Alert Logic. Warren has been a guest speaker at cybersecurity conferences all over the globe, including giving talks on Bitcoin at Microsoft DCC, NorthSec, Kaspersky SAS, VB, and many more.

Having been involved in cryptocurrency for over 10 years, Warren is dedicated to making the blockchain ecosystem as secure as it can be for everyone. Warren serves as the CEO for HYDN and heads up the delivery team to ensure that work is carried out to the highest standard. The HYDN delivery team has over 10 years combined experience in blockchain, having performed smart contract audits and built security systems for a large range of protocols and companies.

HYDN works closely with clients to consider their unique business needs, objectives, and concerns. Our mission is to ensure the blockchain supports secure business operations for all and he has built the team at HYDN from the ground up to meet this very personal objective. To keep up to date with our latest news and announcements, check out our website https://hydnsec.com/ or follow @hydnsecurity on Twitter.

Methodology

When tasked with conducting a security assessment, HYDN works through multiple phases of security auditing to ensure smart contracts are audited thoroughly. To begin the process

automated tests are carried out, before HYDN then moves onto carrying out a detailed manual review of the smart contracts.

HYDN uses a variety of open-source tools and analyzers as and when they are required and alongside this, HYDN primarily focuses on the following classes of security and reliability issues:

Basic Coding Mistakes

One of the most common causes of critical vulnerabilities is basic coding mistakes. Countless projects have experienced hacks and exploits due to simple, surface level mistakes that could have been flagged and addressed by a simple code review. The HYDN automated audit process which includes model checkers, fuzzers, and theorem provers analyses the smart contract for many of these basic coding mistakes. Once the automated audit has taken place, HYDN then performs a manual review of the code to gain familiarity with the contracts.

Business Logic Errors

HYDN reviews the platform or projects design documents before analysing the code to ensure that the team has a deep understanding of the business logic and goals. Following this, HYDN reviews the smart contracts to ensure that the contract logic is in line with the expected functionality.

HYDN also analyses the code for inconsistencies, flaws, or vulnerabilities which could impact business logic such as Tokenomics errors, arbitrage opportunities, and share pricing.

Complex Integration Risks

Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem.
We perform a meticulous review of all of the contract's possible external interactions, and summarise the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code Maturity

HYDN reviews the smart contracts for potential improvements in the codebase. These improvements ensure that the code follows industry best practises and guidelines, or code quality standards. Alongside this, HYDN makes suggestions for code optimization items such as gas optimization, upgradeability weaknesses, centralization risks, and more.

Impact Rankings

Once HYDN has completed the security assessment, each issue is assigned an impact rating. This impact rating is based upon both the severity and likelihood of the issue occurring. Each security assessment is different and the business needs and goals, such as project timelines, and client threat modelling. are taken into account when the impact rankings are created.

HYDN assigns the following impact rankings: Critical, High, Medium, Low (listed by severity).

Disclaimer

This HYDN security assessment does not provide any warranties on finding all possible issues within the scope and the evaluation results do not guarantee the absence of any issues. **HYDN cannot make any guarantees on any further code which is added or altered after the security assessment has taken place**. As a single security assessment can never be considered fully comprehensive, HYDN always recommends multiple independent assessments paired with a bug bounty program. This security assessment report should not be considered as financial or investment advice.

Summary of Findings

Executive Summary

HYDN audited the scoped contracts listed below and discovered 5 findings. One finding was of Critical Severity and two of the findings were of High Severity. The two further findings were both of Low Severity.

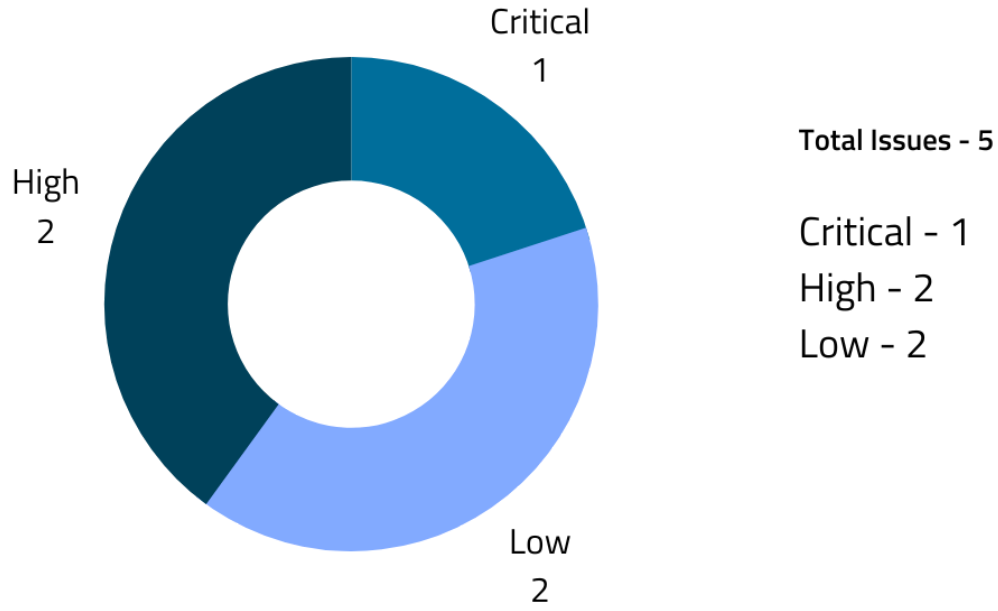
The Critical and High Severity findings require immediate action to remediate as they can result in loss of funds and functionality.

HYDN also recommends reviewing the testing currently in place and to update this based upon best practices for Solidity Unit Tests.

Project Summary

Chain	Telos
Language	Solidity
Scope	FUSDTokenSaleUtils FUSDTokenUtils TokenAdapterUtils
Repository	https://github.com/merchantry/fusd-token
Initial Commits	8bcf768f7e5a2665ce5bcc3757083f105dba1c89

Breakdown of Findings



ID	Title	Severity	Status
1	Unsafe Reliance on Token Symbols	Critical	Resolved
2	Liquidation Denial of Service Risk	High	Resolved
3	Position Storage Denial of Service Risk	High	Resolved
4	Gas Optimization Recommendations	Low	Resolved
5	Outdated Solidity Compiler Version Used	Low	Resolved

Detailed Findings

1 Unsafe Reliance on Token Symbols [Critical]

1.1 Description

The exchange vault's deposit and withdraw operations credit and debit user balances based on a token mapping. This mapping relies on a call to the address provided in the parameters and matches the symbol of the expected contract call result to an internal unique key.

However, there is no assurance that this token symbol will be unique.

Furthermore, it is possible for an attacker to create a malicious contract designed to match the symbol of a legitimate token, potentially leading to security vulnerabilities.

```
11      function _depositToken(
12          address user,
13          address token,
14          uint256 amount
15      ) internal tokenAdapterExists(token) {
16          userTokenBalances[user][ERC20Utils.getTokenKey(token)] += amount;
17          ERC20(token).transferFrom(_msgSender(), address(this), amount);
18      }
```

Extract of ERC20ExchangeVault.sol

```
10      function getTokenKey(address token) internal view returns (bytes32) {
11          return keccak256(bytes(ERC20(token).symbol()));
12      }
13  }
```

Extract of ERC20Utils.sol

1.2 Impact

The attacker can exploit this vulnerability by depositing a malicious contract token with the same symbol as the expected legitimate collateral token. Subsequently, they can open a debt position and generate an infinite amount of FUSD.

In another scenario, after depositing the malicious token, the attacker can withdraw the legitimate collateral token, draining all locked collateral.

In the short term, this would result in a complete drain of the protocol's assets as the attacker:

- Mints an infinite amount of FUSD without providing the necessary collateral.
- Drains all available collateral.

These actions pose a severe threat to the protocol's stability and security.

1.3 Recommendations

Do not rely on token symbols, names, or any potentially ambiguous identifiers for critical operations within the protocol. Instead, it is strongly advised to implement token mapping that is defined and controlled exclusively by an admin account.

This mapping should map addresses to indexes, such as integers, addresses, IDs, or whichever you need.

By adhering to this approach, user input validation becomes more robust, eliminating the need for external calls to resolve addresses. This proactive measure ensures greater security

and mitigates the inherent risks associated with token symbols, names, or identifiers, which may be subject to manipulation or ambiguity

1.4 Remediation Completed

Resolved.

2 Liquidation Denial of Service Risk [High]

2.1 Description

Liquidation is restricted to owner-only access, it then loops through all users to identify those with unhealthy positions. However, due to the limitations of the EVM, this loop is limited by the transaction gas limit and cannot surpass the block gas limit.

In cases involving a large number of users and/or a significant number of unhealthy positions, attempting to execute a transaction to liquidate all of them on-chain would become impossible.

```
82      function liquidateAllDebtorsBelowLiquidationThreshold() public onlyOwner {
83          address[] memory debtorsBelowLiquidationThreshold = getDebtorsBelowLiquidationThreshold();
84          for (uint256 i = 0; i < debtorsBelowLiquidationThreshold.length; i++) {
85              liquidateUser(debtorsBelowLiquidationThreshold[i]);
86          }
87      }
```

Extract of LiquidatingUserAssetsBelowLiquidationThreshold.sol

```
60      function getDebtorsBelowLiquidationThreshold() public view returns (address[] memory) {
61          address[] memory allDebtors = getAllDebtors();
62          address[] memory debtorsBelowLiquidationThreshold = new address[](allDebtors.length);
63          uint256 debtorsBelowLiquidationThresholdCount = 0;
64          for (uint256 i = 0; i < allDebtors.length; i++) {
65              address debtor = allDebtors[i];
66              if (isDebtorBelowLiquidationThreshold(debtor)) {
67                  debtorsBelowLiquidationThreshold[debtorsBelowLiquidationThresholdCount] = debtor;
68                  debtorsBelowLiquidationThresholdCount++;
69              }
70          }
```

Extract of LiquidatingUserAssetsBelowLiquidationThreshold.sol

2.2 Impact

The inability to liquidate unhealthy positions would result in the accumulation of bad debt within the protocol. Over time, this could lead to the liquidation function becoming non-functional, effectively rendering liquidation impossible.

This situation poses a significant long-term risk to the protocol's financial health and stability.

2.3 Recommendations

Always consider limitations for loops. In addressing this particular issue, consider introducing an additional public function that allows for the liquidation of a single user at a time.

Additionally, it's advisable to review all for loops to ensure that they do not exceed reasonable gas execution limits, especially when the protocol is expected to operate at a larger scale.

2.4 Remediation Completed

Resolved.

3 Position Storage Denial of Service Risk [High]

3.1 Description

Similar to the problem covered in Issue 2 above, a user might open a significant number of positions, which could result in the inability to repay their own debt due to too large of a loop over all positions.

```
82      function payOffDebt(uint256 amount) public {
83          address user = _msgSender();
84          uint256 totalDebt = getTotalDebt(user);
85
86          require(amount <= totalDebt, "FUSDTokenSale: amount exceeds total debt");
87          _addRepayment(user, amount, time());
88          address withdrawable = getERC20WithdrawableAddress();
89          transferFUSD(user, withdrawable, amount);
90      }
```

Extract of FUSDTokenSale.sol

```
50      function calculateBaseDebtAndInterest(address user) public view returns (uint256, uint256) {
51          DebtChange[] memory changes = debtChanges[user];
52          uint256 baseDebt = 0;
53          uint256 totalInterest = 0;
54          uint256 lastChangeAt;
55
56          for (uint256 i = 0; i < changes.length; i++) {
57              DebtChange memory change = changes[i];
```

Extract of DebtHandler.sol

3.2 Impact

When a user accumulates a large number of positions, they may find themselves in a situation where they are unable to add more collateral due to the collateral ratio calculations exceeding loop limits, as indicated by `collateralRatioSafe() -> getTotalDebt() -> calculateBaseDebtAndInterest()` logic.

Combined with the problem raised in Issue 2, it creates a scenario where a single user's actions can effectively render the entire protocol incapable of liquidating any positions, presenting a significant risk to the protocol's overall functionality and financial health.

3.3 Recommendations

Similar to the problem described in Issue 2, concerns about loops are raised. However, in this case, finding a straightforward remedy is challenging.

One potential solution could be to impose limits on the number of open positions allowed per user, but such restrictions might not align with business requirements and user expectations.

A more comprehensive approach involves a fundamental redesign of the storage logic related to user positions / mantissa. This redesign should prioritise efficiency and scalability, ensuring that the protocol can accommodate a large number of positions while maintaining performance and stability. This type of restructuring can address the underlying issues and provide a more robust foundation for the protocol's long-term sustainability.

3.4 Remediation Completed

Resolved.

4 Gas Optimization Recommendations [Low]

4.1 Description

Functions not called internally and externally can be flagged as public or external.

4.2 Impact

This will result in unnecessary deployment and execution costs.

4.3 Recommendations

In order to reduce deployment and execution costs, functions called exclusively from external EOA/contract must be flagged as external in place of public.

4.4 Remediation Completed

Resolved.

5 Outdated Solidity Compiler Version Used [Low]

5.1 Description

Currently an outdated Solidity version has been used on the contracts.

5.2 Impact

Compiler issues can be caused by using outdated versions of Solidity. Alongside this, using different compiler versions will produce arbitrary outputs and lead to unexpected issues.

5.3 Recommendations

To remediate this issue, it is recommended to use the latest version of the Solidity compiler, and use a fixed version of the compiler to avoid any unexpected issues.

5.4 Remediation Completed

Resolved.