HYDN

Smart Contract Security Assessment

October 31, 2022

Project Name:

Swapsicle

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 Swapsicles 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.

About Swapsicle

Swapsicle is a decentralised exchange built on the Avalanche blockchain offering swaps, farming, and staking opportunities. Their goal is to build a comprehensive decentralized trading platform for the future of finance. The $POPS token forms a key part of the Swapsicle ecosystem and can be used to unlock incentivised services for holders.

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 Swapsicle 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

As part of this audit, HYDN focused solely on the Swapsicle Paired Token Staking contract, and nothing further beyond this. This included PairedStakingFixedAPR.sol and the deployment and utility scripts associated with it. Overall the contracts were found to be very well written with only low impact rating issues found.

Project Summary

Platform	Avalanche C-Chain
Language	Solidity
Scope	PairedStakingFixedAPR.sol Deployment and Utility Scripts
Repository	https://github.com/Dev-Legion/swapsicle_paired_token_staking
Initial Commits	8716743ad9408640c6ff19c9d614a7dba20eeb70
Latest Commits	ba460fed929676dc7d1b18abf9a98d37c4532635
Deployment address	0×6aA10ead8531504a8A3B04a9BfCFd18108F2d2c2

Audit Summary

Delivery Date	October 7th 2022
Audit Method	Static Automated Analysis + Manual Review

List of Checks

- Overflows & Underflows
- Reentrancy Vulnerability
- Replay Vulnerability
- Short Address Vulnerability
- Kill-Switch / Self-Destruct
- Denial of Service
- Arithmetic Accuracy
- Uninitialized Storage Pointers
- Time Based Vulnerability
- Loop Vulnerability
- Ownership Takeover
- Access Control
- Oracle Security
- Deployment process
- Unsafe Type Inference
- Visibility Level Explicity
- Centralization Risks
- Deprecated Issuage
- Standard Compliance
- Code Stability
- Best Practices

Breakdown of Findings

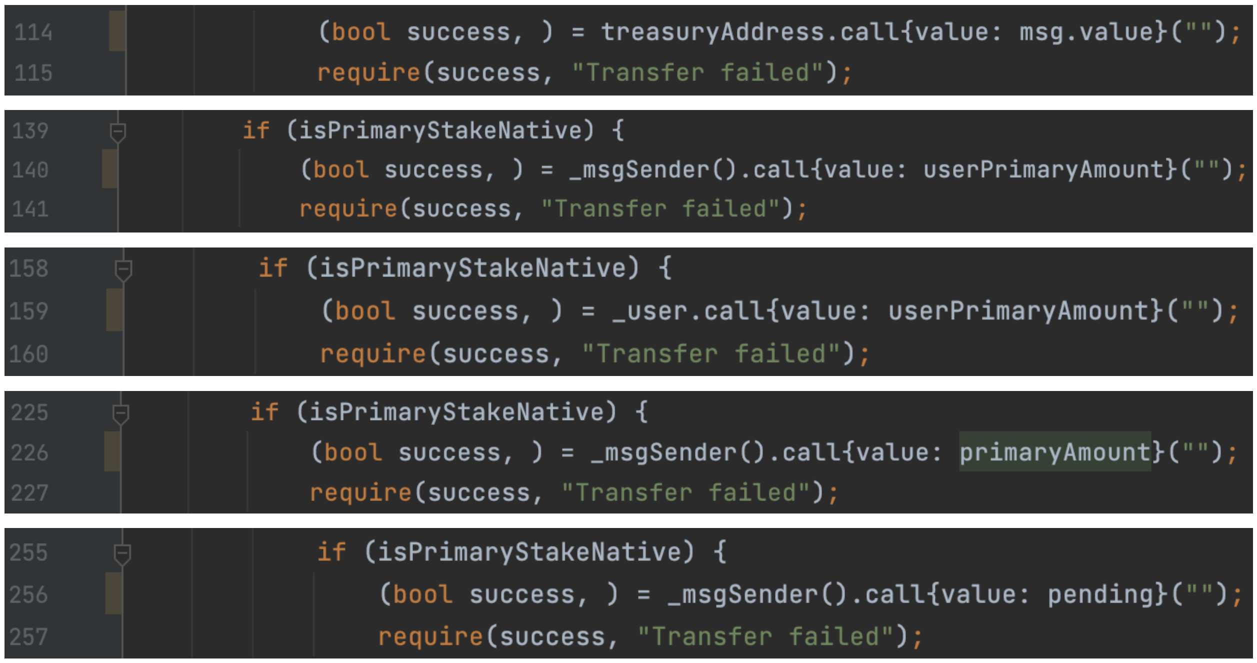
ID	Title	Severity	Status
1	Avoid native call	Low	Resolved
2	Centralization risks	Low	Acknowledged
3	Remove unused storage and error definition	Low	Resolved
4	Gas optimization - flag as immutable constant storage variable	Low	Resolved
5	Outdated Solidity compiler version used	Low	Acknowledged

Detailed Findings

1 Avoid native calls [Low]

1.1 Description

As good practice, the use of low-level calls must be avoided whenever possible.



```
        (bool success, ) = treasuryAddress.call{value: msg.value}("");
        require(success, "Transfer failed");
```

```
    if (isPrimaryStakeNative) {
        (bool success, ) = _msgSender().call{value: userPrimaryAmount}("");
        require(success, "Transfer failed");
```

```
    if (isPrimaryStakeNative) {
        (bool success, ) = _user.call{value: userPrimaryAmount}("");
        require(success, "Transfer failed");
```

```
    if (isPrimaryStakeNative) {
        (bool success, ) = _msgSender().call{value: primaryAmount}("");
        require(success, "Transfer failed");
```

```
    if (isPrimaryStakeNative) {
        (bool success, ) = _msgSender().call{value: pending}("");
        require(success, "Transfer failed");
```

Extract of PairedStakingFixedAPR.sol

1.2 Impact

Using low-level calls may lead to inaccurate results and unexpected behaviours if the receiver is not able to handle the function output. This can then lead to a re-entrancy, which although it does seem safe in this case, is always best prevented.

If the return value of a low-level call is not checked then the execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, then this may cause unexpected behaviours in the subsequent program logic.

1.3 Recommendations

In the case where the expected depositors are standard EOA addresses and there is no need for composability (other contracts interact and react with this one), it's safer to use the high-level transfer() function, which is safer and gas constrained.

1.4 Remediation Completed

Successfully remediated and low-level call has been replaced by transfer().

2 Centralization risks [Low]

2.1 Description

At any time the contract owner is allowed to do the following:

emergencyWithdrawByAdmin: The contract owner can withdraw any amount on behalf of any user.

setBlocksPerDay: The contract owner can increase or reduce the constant value supposed to link the time to the block creation speed. As each chain has its own block time finality and it is not supposed to change, it is safe to only define this parameter into the constructor. Having the ability to update it seems unnecessary.

setEndBlock: The contract owner can increase or reduce the end time of the stacking, without any control, at any time. This means the end time can be set in the past.

setMaxPrimaryStake: The contract owner can increase or reduce the maximum capacity of the pool at any time.

setMaxPrimaryStakePerUser: The contract owner can increase or reduce the maximum capacity of the pool peer address at any time.

setMinPrimaryStakePerUser: The contract owner can increase or reduce the minimum amount to deposit peer addresses at any time.

setRewardPerDayPerPrimary: The contract owner can increase or reduce the yield percentage peer block at any time.

setStakingEnabled: The contract owner can enable or disable deposits at any time.

setTotalReward: The contract owner can increase or decrease the reward pot. This value starts at zero and is increased by calling the receive() fallback method in the case of native. But in the case of the non-native primaryToken it's the only way to define the reward pot amount.

In the case of native, manually defining the value will result in an unexpected state.

In the case of non-native, this function must be called carefully. A safer way would be to have a dedicated function to deposit rewards and increase the totalReward and to remove this manual control.

setTreasuryAddress: The contract owner can change the treasury address. It is only used when a native coin is sent to the contract and reaches the fallback and the primaryToken is not native. Funds are then redirected to the treasury address.

2.2 Impact

As detailed above, the contract owner is able to perform a multitude of actions.

2.3 Recommendations

Ensure that all of the functions are required to satisfy the business logic and implement suggested changes that are detailed beside relevant points in the description.

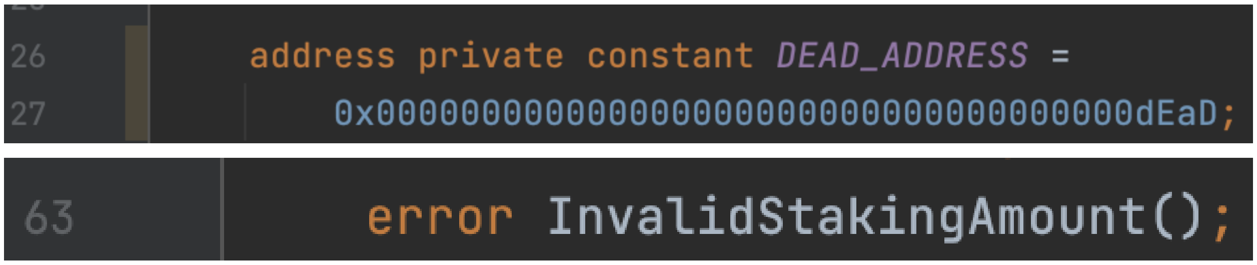
2.4 Remediation Completed

The issue has been acknowledged by the client, but the decision was made to make no further changes.

3 Remove unused storage and error definition [Low]

3.1 Description

The contract currently includes a storage slot that is never read or called.



```
    address private constant DEAD_ADDRESS =
        0x000000000000000000000000000000000000dEaD;
```

```
        error InvalidStakingAmount();
```

Extract of PairedStakingFixedAPR.sol

3.2 Impact

The impact of this is that unused storage variables lead to misrepresentation and extra deployment gas costs.

3.3 Recommendations

To remediate this issue, simply remove the unused storage.

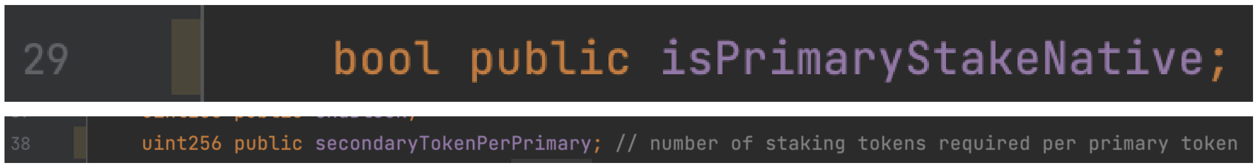
3.4 Remediation Completed

Successfully remediated and all unused storage has been removed.

4 Gas optimization - flag as immutable constant storage variable [Low]

4.1 Description

Storages that are initialized in the constructor and never updated later can be flagged as immutable to reduce deployment gas costs. In the contract, some of them have already been properly defined, but some are missing.



```
bool public isPrimaryStakeNative;
```

```
uint256 public secondaryTokenPerPrimary; // number of staking tokens required per primary token
```

Extract of PairedStakingFixedAPR.sol

4.2 Impact

The impact of this issue is that it can lead to extract gas costs during deployment and during execution.

4.3 Recommendations

Add an immutable flag on these storage variables.

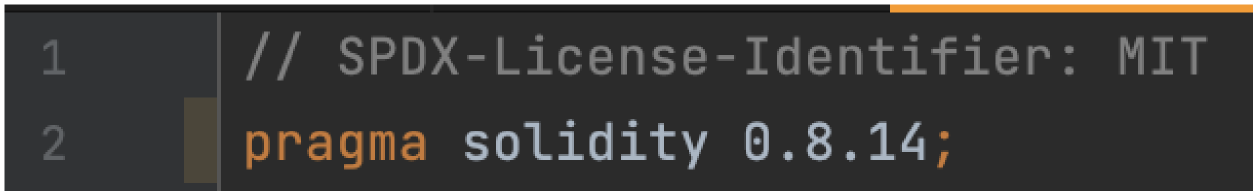
4.4 Remediation Completed

Successfully remediated, all flagged as immutable.

5 Outdated Solidity compiler version used [Low]

5.1 Description

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



```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.14;
```

Extract of PairedStakingFixedAPR.sol

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

5.4 Remediation Completed

The issue has been acknowledged by the client, but the decision was made to make no further changes.

Remediations Summary

Successfully remediated

- Avoid native calls [Low]
- Remove unused storage and error definition [Low]
- Gas optimization - flag as immutable constant storage variable [Low]

Acknowledged but no further action taken

- Outdated Solidity compiler version used [Low]
- Centralization risks [Low]

HYDN Seal

Each HYDN Seal is an ERC-1155 token that contains key metadata about the completed audit, including the hash of the contract bytecode, date, and link to the report.

As each HYDN Seal is stored on-chain, it allows for real-time monitoring of a project's status. The image on the Seal is served dynamically. We will notify you if the contract changes so you can validate if it has been changed by nefarious activity.

The HYDN Seal Token ID assigned to the contracts covered by this report:
431140000001

The token factory contract can be found at:
0×5AFE007958D169abE98299bf645b4C5890901dCB on the same network (Avalanche C-Chain ID: 43114).