Security Audit Report

AshSwap Smart Contracts

Delivered: January 11th, 2023

Prepared for AshSwap by

runtime verification

Table of Contents

Executive Summary

AshSwap engaged Runtime Verification Inc. to conduct a security audit of their smart contracts. The objective was to review the contracts' business logic and implementations in Rust and identify any issues that could potentially cause the system to malfunction or to be exploited.

The audit was conducted over the course of 8 calendar weeks (November 2, 2022 through December 28, 2022) focused on analyzing the security of the code related to the contracts of the AshSwap platform, which enables users to stake their assets while receiving liquidity tokens in exchange, and being rewarded with a portion of the fees from the pool. The platform also provides mechanisms for the users to stake their liquidity tokens in exchange of meta Elrond Standard Digital Tokens (ESDTs) that can be locked in a contract, rewarding the depositor with voting power for the governance of the protocol.

The audit led to identifying issues of potential severity for the protocol's health, which have been identified as follows:

- Functional Correctness: A01
- Potential asset security vulnerabilities: A02, A03, A04, A05, A07, A08

In addition, several informative findings and general recommendations also have been made, including:

- Input validations: A02, A05, A06, A07
- Best practices: A04, A08, B01, B02, B03, B04, B05
- Code optimization related particularities: B03
- Blockchain related particularities: B04

The great majority of the potentially critical issues have been addressed, while all the informative findings and general recommendations have been incorporated to the contracts' code. All of the remaining findings have been acknowledged by the client and deemed non-threatening to the integrity of the platform, when not intended by design.

Goal

The goal of the audit is twofold:

1. Review the high-level business logic (protocol design) of AshSwap's system based on the provided documentation;
2. Review the low-level implementation of the system given as smart contracts in Rust.

The audit focuses on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve safety and efficiency of the implementation.

Platform and Contract Features Description

AshSwap is a platform built over the Elrond blockchain that allows users to swap and also stake their stablecoins to obtain liquidity pool tokens, granting the users the benefit of receiving a portion of the fees of operations performed on the protocol's liquidity pool. These liquidity pools follow the stableswap invariant for calculating constants responsible for keeping the values of the pools. The methodology aims to merge the qualities of the constant price market maker invariant ($x + y = const$) and the constant product market maker invariant ($x * y = const$).

With the liquidity tokens obtained by staking, the user is able to enter AshSwap's yield farms, and by participating in these farms, users are rewarded with the platform's DEX ESDTs, the ASH token. These tokens, in turn, can be committed to a voting escrow for periods of up to 4 years, and, by providing their tokens, users are now awarded with voting rights and power in the protocol's governance.

Another benefit for committing their ASH is that users are capable of boosting yields in the platform's yield farm, meaning that the greater the voting power of a user, the higher are the rewards obtained from the platform's yield farm.

To better understand the mechanics of the AshSwap's yield boost mechanism as well as how to reach the maximum 250% boost in the farm contract, reading the following document is recommended: What is Yield Boost and Why is it important?.

Scope

The scope of the audit for is limited to the artifacts available at git-commit-id **bddadd7d0f30879e6e7734c1b317609311c065a0** for the branch **main** of a private repository provided by the client. Additionally, modifications performed up to the git-commit-id **4032b5882a19245d3f5aedfd52a42b5001272140** addressing the findings discovered during this audit engagement were verified.

The comments provided in the code, and a general description of the project, including samples of tests used for interacting with the platform, and also online documentation provided by the client were used as reference material.

The analyzed functionalities consider the handling and exchange of assets between user and contract accounts, staking of assets in exchange of liquidity tokens, staking of liquidity tokens in exchange of ASH tokens and the commitment of the platform tokens into a voting escrow. Those are implemented using six contracts:

1. **Pool contract**: implements all the functionalities for the liquidity pools of the platform, enabling the swapping and staking of assets. These operations are performed by following the stableswap invariant methodology;
2. **Router contract**: implements the manager of pool contracts. The router is responsible for deploying, properly setting and updating new pools, while also collecting administrative fees that are forwarded to the fee distributor contract;
3. **Farm contract**: implements the functionalities for yield farming in the AshSwap protocol, including the deposit and withdrawal of liquidity tokens, burning and minting of the ASH meta ESDT token, and distribution of the farm rewards;
4. **Voting Escrow contract**: implements the functionalities necessary for locking the tokens for a user-defined amount of time. The user's balance, or voting power, is often referred to as the user's veASH;
5. **Fee distributor contract**: implements the necessary functionalities for the distribution of administrative fees among the owners of veASH;
6. **Rewarder contract**: implements the necessary functionalities for the distribution of the aggregated rewards generated by the farm contract.

The audit is limited in scope to the artifacts listed above. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are not in the scope of this engagement.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our Disclaimer, we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic. To this end, we carefully analyzed all the proposed features of the platform and actors involved in the lifetime of a deployed contract.

Second, we thoroughly reviewed the contract source code to detect any unexpected (and possibly exploitable) behaviors. To facilitate our understanding of the platform's behavior, several higher-level representations of the Rust codebase were created, where the most comprehensive were:

1. A high-level relationship model to describe and correlate the operations performed by the individual contracts of the AshSwap platform, which could be used for the analysis of the separate entities of the protocol,
2. Modeled sequences of mathematical operations as equations and, considering the limitations of size and types of variables in the WASM VM, checking if all desired properties hold for any possible input value.

This approach enabled us to systematically check consistency between the logic and the provided Rust implementation of the contracts.

Finally, we performed rounds of internal discussion with Elrond experts over the code and platform design, aiming to verify possible exploitation vectors and to identify improvements for the analyzed contracts.

Additionally, given the nascent Elrond's Rust development and auditing community, we reviewed this list of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to Elrond's smart contracts; if they apply, we checked whether the code is vulnerable to them.

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

All findings have a severity level ranging and an execution difficulty level, ranging from low to high, as well as categories in which it fits.

A01: Possibility of precision loss when calculating constant value

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

Description

The AshSwap protocol implements the stableswap methodology to handle the asset rates in pools, in which the joint characteristics of the constant price market maker and the constant product market maker methodologies result in the following equation for calculating the asset prices:

$$\chi D^{n-1} \sum x_i + \prod x_i = \chi D^n + \left(\frac{D}{n}\right)^n .$$

Where χ is the leverage controlling how much the constant product section of the equation will weigh over the constant price section; *x* is the amount of an asset of index *i*, *n* is the number of assets in the pool, and *D* is a value that must remain constant whenever swap operations are performed.

When calculating the constant *D* of the automated market maker, the Newton-Raphson method is used for obtaining an approximated value. This is a popular and battle-tested solution for solving for the Stableswap invariant, as it was implemented by Curve protocol and subsequently used in a number of protocols with shared purposes.

Here we highlight a slight precision issue in a certain computation involved in the implementation of the Newton-Raphson algorithm.

See the following lines of code:

Calculation of constant value D (amm.rs module)

```
let mut d_prod = d.clone();
for token in tokens.into_iter() {
    d_prod = &d_prod * &d / (&token.amount * &n_coins);
}
```

Concretely, the loop containing the snippet of code presented above calculates the value of $\frac{D^n}{Product\ token\ amounts \cdot n^n}$. This is done by multiplying and dividing one value at a time. While this approach can yield benefits in a protocol where the magnitude of intermediate results can be troublesome, it is not necessary on a platform benefiting from big integer operations, such as Elrond. Furthermore, because this approach performs several intermediate divisions, the result will be slightly imprecise compared to the natural alternative.

For this specific calculation, the precision loss is expected to be 10^{-18} at most, which, given the range of values that *D* will be assigned, can be considered insignificant.

Recommendation

To avoid the precision loss originated by the approximation of the root values of the mentioned equation, we suggest that the given formula is implemented directly: compute the numerator and denominator independently and divide them at the end, resulting in a single division.

Status

This issue has been acknowledged by the client and not addressed given the significance of the precision loss elaborated above.

A02: Lack of validations in the initialization of pools

[Severity: Medium | Difficulty: Low | Category: Asset Security, Input Validations]

Description

Given the nature of smart contracts of being publicly exposed in the blockchain, all necessary validations of the values received in interactions with the platform should be performed in the contracts themselves. In other words, there should never be assumptions that the received parameters and transfers are formatted in the way they are intended to be.

With that in mind, the lack of validations in the protocol's liquidity pools include the following findings:

1. In the contracts that represent liquidity pools, there are no validations that guarantee that the assets held by the pool are distinct. One example would be a pool that is instantiated to hold two assets: USDC and USDC. Furthermore, malicious users could try to set up a pool with such configurations and, with different exchange rates for each token in the pool, try to trick other users into depositing their liquidity in this same pool, only to later withdraw their LPs to obtain a potential profit. The severity of such a scenario is aggravated by the fact that only one payment would be needed for adding liquidity for both assets which, in fact, are the same. The profit of the malicious user of course would depend on the amounts deposited by each legitimate user;
2. There are no guarantees that prevent a user from trying to create a liquidity pool with a single token;
3. There are no validations for the rate of the assets. This enables users to, in the first deposit of a newly created pool, use distorted rates for the assets deposited there. It is expected that arbitrageurs normalize the rate of assets in the pool shortly after its deployment in such a scenario;
4. There are no limitations regarding the maximum amount of assets that a pool might hold. This might imply the existence of unusable pools due to the gas amount needed to perform operations in pools holding several different tokens.

Recommendation

For the four topics described above we suggest the following modifications to AshSwap pools:

1. During its instantiation, a pool contract should check if the ESDTs that it should hold, as informed by the caller, are different from each other;
2. Require a minimum amount of tokens to be supplied by the user when instantiating a pool (i.e., the list of ESDTs that this pool should hold is greater than or equals to 2);
3. For the first deposit of the pool, a rate for the deposited assets should be respected. Since these are stable swap pools, one possibility is to assert that all deposits have the same amount for the first “add liquidity” operation;
4. Require a maximum amount of tokens to be supplied by the user when instantiating a pool (i.e., the list of ESDTs that this pool should hold is smaller than x, where x is, for example, 6).

Status

This issue has been fully addressed.

A03: Initial issuance of tokens are kept in the router can be withdrawn to the fee_distributor address

[Severity: Medium | Difficulty: High | Category: Asset Security]

Description

In the router contract, which is responsible for the management of liquidity pools, after the creation of a pool, the router contract is supposed to create the liquidity token that will be minted by the pool, as well as to grant it the rights for administrative operations regarding the ESDT.

When the initial token is issued in the router contract, 1000 * 10^{-18} LP tokens are created as the initial supply, but are not burnt or transferred to any address. The amount of LP ESDTs contains the product by 10^{-18} as the token is programmatically deployed with 18 decimals, meaning that a request to issue 1000 tokens will actually mint 0.000000000000001000 tokens. This amount of LP ESDTs is negligible in contrast to the expected amount of liquidity tokens in circulation for this protocol. Initially this is harmless, but the router contract has the capability to withdraw tokens to another address, which is the fee distributor contract.

In case the owner or the router contract itself becomes compromised, a malicious user could add the issued LP tokens to the fee_tokens list, calling the convert_fee_to_lp function in order to withdraw the LP tokens to the fee distributor address, possibly even modifying this address to an contract deployed and controlled by the malicious user.

While the chances of the owner or router to become compromised are considerably low and even lower is the possibility of the accidental execution of the sequence of operations necessary for this scenario to happen, users who are more technically knowledgeable may be able to identify this flow of actions into the contract. Such an occurrence can become the source of trust issues between the user base and the AshSwap protocol.

Recommendation

A simple solution to this event is to programmatically burn the tokens that are held in the router after the ESDT instantiation.

Status

This issue has been fully addressed by, in the updated version of the contract, minting zero tokens during the first token issuance operation.

A04: “set_lp_token_identifier” should not be able to be called by the emergency wallet/owner address

[Severity: High | Difficulty: High | Category: Asset Security, Best Practices]

Description

Pools store the addresses of both its deployer and, if its deployer is a contract, the address that deployed its deployer. This is necessary as to remember who deployed the pool, which, according to the protocol’s design, should be the router, and the deployer of the router, which is supposed to be the administrator account of the AshSwap’s protocol.

These entities are recorded in the pool contract to manage the authorization to perform certain actions in the liquidity pools. One specific action is the one executed by the “setLpTokenIdentifier” endpoint, where the ESDT identifier that represents the liquidity pool token is modified.

When setting the LP token identifier pool, the contract validates if the caller has permission to perform such an operation by the require_permissions function, where, in the intended design, both the router and an owner/admin address may have permissions to perform authentication-required tasks. In case of a manual execution of this operation or if the owner/admin account is compromised, it is possible to invalidate the liquidity tokens held by the legitimate users of the protocol and, for instance, use an identifier of a ESDT minted by a malicious user. This would enable an attacker to withdraw all the tokens held by the liquidity pool.

Recommendation

As the call for the “setLpTokenIdentifier” endpoint is only made once and programmatically by the router contract, it is advised to remove the capability of the owner to call this endpoint, avoiding the risks elaborated above.

Status

This issue has been fully addressed.

A05: User has no voting power until his deposited amount is greater than MAXTIME variable

[Severity: Low | Difficulty: Low | Category: Asset Security, Input Validation]

Description

In the voting escrow contract, a user is capable of locking his ASH tokens for up to four years. The management of this maximum time is made through the use of a variable referenced as MAXTIME.

If the user has a total deposited balance in the voting escrow that is smaller than MAXTIME (4 * 365 * 86400 = 126144000), the division that happens when creating a checkpoint for after a deposit, specifically when the slopes are being calculated, is subject to precision loss.

This happens in two different points of the contract, where the calculation is performed as the example shown below:

Calculation of slopes in the voting escrow (lib.rs module)

```
u_old.slope = BigInt::from_biguint(Sign::NoSign,
&old_locked.amount / &max_time);
```

Since the variables are handled as big integers, and the result of the division is also converted to a big integer, the fractional part of the number will be lost as the value is truncated. This may lead to a user that deposits an amount smaller than MAXTIME having the slope and bias used for calculating their voting power equals zero.

Given that the tokens are supposed to have 18 decimal places, any amount smaller than MAXTIME is considered a small amount of assets, but this also will depend on the value of the ASH token received when farming.

Recommendation

Initially, two solutions are available for handling this issue:

- Explicitly requiring users to deposit amounts greater than MAXTIME;

- or scaling the values for the slopes to always have more decimals than the MAXTIME value, although a certain degree of precision loss will still affect the calculations mentioned above.

Status

This issue has been fully addressed.

A06: No validation of tokens in functions and views used for calculating amounts

[Severity: Low | Difficulty: Low | Category: Input Validation]

Description

In the liquidity pool contract, when attempting to fetch the amount of LPs received for adding liquidity to a pool using the “estimateAddLiquidity” view, or when attempting to fetch the amount out of a swap operation using the “estimateAmountOut” view, whoever calls this function is responsible for providing as parameters the tokens deposited or the token sent and received for both views, respectively.

Given this is a publicly exposed view which will be used directly by users, it is comprehensible that the parameters received by the handler function should be properly validated. For both of these functions, there is no validation over the tokens that are being handled.

A user can attempt to fetch the amount LPs received on a pool by supplying a token that is not held by that pool, or even supplying the same token as input and output of the operations, which does not stop the contract from attempting to yield a value.

Analogously, if the user attempts to fetch the output amount for a swap with tokens that are not held by the pool, the execution of the logic for the function will be executed by the pool smart contract.

Recommendation

In the “estimateAddLiquidity” and “estimateAmountOut”, validate if the tokens provided as parameters are different and exist in the queried liquidity pool.

Status

This issue has been fully addressed.

A07: It is possible to override the locked token identifier in the voting escrow

[Severity: High | Difficulty: High | Category: Asset Security, Input Validation]

Description

Considering the init function in Elrond’s Rust smart contracts, two situations allow this function to be called: on the contract deployment and contract update.

In the voting escrow contract, the first operation performed during the init function is the setting of the locked token identifier, and there is no validation to prevent it from being overridden in update operations, unlike the validation of “point_history” that happens just after that.

Similar to the finding A04, consequences of overriding one of the token identifiers can have a significant impact on the ability of users to retrieve or even deposit more assets in the pool.

Recommendation

It is recommended to verify if a value is already set before attempting to set it, unless the value override is intended.

Status

This issue has been fully addressed.

A08: No validations over the output token when using convert_to_fee_tokens

[Severity: Low | Difficulty: Medium | Category: Best practices]

Description

In the fee collector module of the router contract, the intent of the “convert_to_fee_tokens” function is to modify all tokens received as fees by the liquidity pools to one of the tokens accepted as fee. These tokens are stored in the “fee_tokens” VecMapper. This conversion is performed through a sequence of swaps using “burners” or addresses of liquidity pools in which the tokens can be swapped.

Still, the function has no guarantees that the tokens that come as an output of the sequence of swaps through burners are actually in the fee_tokens list. Furthermore, there are no validations with regards to the amount of tokens that are received in the end as well. Considering a scenario where one of the burners is an unhealthy pool, this could affect the amount of tokens that will be received at the end of the conversion, therefore it would be advisable to validate if the amount of tokens received are of at least a percentage of the input amount. This is feasible especially given that the pools are of stablecoins.

Recommendation

Validate if the output token obtained in “convert_to_fee_tokens” is actually the desired fee token, while also validating if the amount received is greater than a minimum specified parameter amount.

Status

This issue has been fully addressed.

Informative Findings

Findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need some external support or where the code deviates from best practices. We have also included information on potential code size reductions and remarks on the operational perspective of the contract.

B01: Different reasons for failure with the same error code

[Severity: - | Difficulty: - | Category: Best Practices]

Description

On the different points of the contract, consecutive requirement assertions similar to the one presented below can be seen:

Requirement statements in the fee module of pool contracts

```
require!(swap_fee_percent < 100_000, ERROR_BAD_FEE_PERCENT);
require!(admin_fee_percent < 100_000, ERROR_BAD_FEE_PERCENT);
```

Both reasons for failure have the same error code, even though two different properties can cause the failure. This validation, present in the liquidity pool smart contract, is also performed in the router contract (which shares the same error message for the same two different reasons).

The same validation is performed in the create and upgrade endpoints for pools in the router contract, with the exact same error code.

In the farm contract, the following requirement assertions can be seen:

Requirement statements in the fee module of pool contracts

```
require!(self.is_accept_farming_payment(&storage_cache.farming_token_id, &farming_token_payment), ERROR_BAD_PAYMENTS);
require!(self.is_accept_farm_payment(&storage_cache.farm_token_id, &storage_cache.payments), ERROR_BAD_PAYMENTS);
```

Similarly, as these requirements are used to validate the payments received in the farm, there is no indication of which payment is causing the error.

Status

The issue has been fully addressed.

B02: Nomenclature of the old_attribute variable

[Severity: - | Difficulty: - | Category: Best practices]

Description

See the following lines of code:

Requirement statements in the fee module of pool contracts

```
let mut attribute = token.attribute.clone();
let old_attribute = old_balances.get(index).attribute.reserve;
```

The first line uses a descriptive name to the actual purpose of the variable being instantiated, as it receives a token attribute and serves as a representation of one. The second one, although it receives the reserves of an asset as a BigUInt, its name does not reflect such purpose.

Recommendation

For the instances of where the mentioned variable is instantiated and used, either “old_attribute” should receive “old_balances.get(index).attribute”, and be used as “old_attribute.reserve” or should be renamed to “old_reserves”.

Status

The issue has been fully addressed.

B03: The same validation is performed twice when claiming in the fee distributor

[Severity: - | Difficulty: - | Category: Code Optimization, Best Practices]

Description

See the following lines of code:

Conditionals in the claim endpoint of the fee distributor contract

```
272   if amount > BigUint::zero() {
273       self.user_epoch_of(addr).set(&user_epoch);
274       self.time_cursor_of(addr).set(&week_cursor);
275   }
276
277   if amount != 0 {
278       let token = self.token().get();
279       self.send_tokens_non_zero(addr, &token, 0, &amount);
280       self.token_last_balance().set(&(&self.token_last_balance(
          ).get() - &amount));
281   }
```

The condition that checks if the amount to be claimed is greater than zero encapsulates the next conditional. Since “amount” is a BigUInt and cannot have a negative value, the contents from line 279 to 281 can be merged into the first conditional block.

Status

This issue has been fully addressed.

B04: Payable tag on rewarder endpoint is not needed

[Severity: - | Difficulty: - | Category: Blockchain Related Particularities, Best Practices]

Description

In the rewarder contract, the only endpoint in the which is not reserved for the owner is named “mintToken”, and it is also payable. However, the only pre-defined interaction for this endpoint is specified in the farm contract, where no payments are given in the proxy contract arguments when calling the rewarder.

To avoid unpredictable or undesired actions, it is advisable to remove the “payable” annotation from this endpoint.

Status

The issue has been fully addressed.

B05: Nomenclature of the calculate_per_block_rewards function

[Severity: - | Difficulty: - | Category: Best Practices]

Description

In the farm contract, the “calculate_per_block_rewards” function returns the amount of rewards that is expected to be distributed in an interval between two blocks; that is can be expressed in the following pseudo algorithm:

```
Logic of the “calculate_per_block_rewards” function

reward per block * (current block nonce - block nonce of last reward
distribution).
```

Considering this, we reach the conclusion that what is returned is not the per block rewards amount, and a more fitting name for this function would be “calculate_rewards_for_time_interval”.

Status

This issue has been fully addressed.