Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

testnet3

Veridise Inc.

August 23, 2024

► **Prepared For:**

Alphaswap
https://alphaswap.pro/

► **Prepared By:**

Mark Anthony
Nicholas Brown

► **Contact Us:**

contact@veridise.com

► **Version History:**

Aug. 20, 2024 V1
Aug. 23, 2024 V2

Contents

1 Executive Summary

From Aug. 5, 2024 to Aug. 6, 2024, Alphaswap engaged Veridise to review the security of their testnet3. The review covered changes to a protocol for wrapping and exchanging ARC20 tokens, intended to be deployed on the Aleo testnet. Veridise conducted the assessment over 4 person-days, with 2 engineers reviewing code over 2 days on commit `a12c3e2`. The auditing strategy involved an extensive manual review of the source code performed by Veridise engineers.

Code assessment. The testnet3 developers provided the source code of the testnet3 contracts for review. The source code appears to be mostly original code written by the testnet3 developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code did not contain a test suite.

Summary of issues detected. The audit uncovered 2 issues, both of which are assessed to be of informational severity by the Veridise auditors. Specifically, the token faucet allows users to withdraw an arbitrary amount of tokens (V-AST-VUL-002). And the use of hard coded numerical constants in lieu of Leo constants (V-AST-VUL-001). Among the 2 issues, 1 issue has been acknowledged and fixed by Alphaswap and 1 issue has been determined to be intended behavior after discussions with Alphaswap. This issue has also been included in the report by the Veridise auditors, so that readers are aware of expected behaviour.

Recommendations. After auditing the protocol, the auditors had a suggestion to improve the testnet3. The auditors recommend that the testnet3 developers add a mechanism that ensures that a user cannot claim tokens from the faucet multiple times in a given time period. This will protect the faucet from being drained very quickly by a single user.

Currently, testnet3 relies on the difficulty of obtaining testnet aleo credits. But, if in the future it is possible to get enough testnet credits to spam transactions then the faucet could still be vulnerable to a DoS attack.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
testnet3	`a12c3e2`	Aleo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 5–Aug. 6, 2024	Manual & Tools	2	4 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	0	0	0
Warning-Severity Issues	0	0	0
Informational-Severity Issues	2	1	1
TOTAL	2	1	1

Table 2.4: Category Breakdown.

Name	Number
Maintainability	1
Logic Error	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of testnet3's Aleo programs. In our audit, we sought to answer questions such as:

- Do the changes to this Aleo program impact the functionality?
- If the changes to a function impact its functionality, does it still behave correctly?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved an extensive manual analysis by human experts.

Scope. The scope of this audit is limited to the `main.leo` files in the following folders of the source code provided by the testnet3 developers, which contain the Aleo program implementation of the testnet3: `aleo_wrapper`, `arc20_token_u128`, `arc20_token_u64`, `arc20_wrapper_u128`, `arc20_wrapper_u64`, `swap_config`, `swap_core`, `swap_router`, and `token_faucet` .

Methodology. Veridise auditors reviewed the reports of previous audits for testnet3 and read the testnet3 documentation. They then began a manual review of the code. During the audit, the Veridise auditors met with the testnet3 developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-AST-VUL-001	Unnecessary Hardcoded Constants	Info	Fixed
V-AST-VUL-002	Faucet Allows Users to Claim All Program . . .	Info	Intended Behavior

4.1 Detailed Description of Issues

4.1.1 V-AST-VUL-001: Unnecessary Hardcoded Constants

Severity	Info	Commit	a12c3e2
Type	Maintainability	Status	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	ced04e9		

In several places in the codebase, there are hardcoded numerical constants used. These values can be managed more easily as Leo constants, and if they are managed in multiple places, using a Leo constant will ensure that the values are the same throughout the codebase.

Hardcoded numerical constants are used in the following places:

- `swap_config.aleo`
  - `finalize_init()` lines 29-33
    - The initial `swap_fee` and `base_portion` of the `protocol_fee`
  - `set_fee()` line 47
    - The `fee` is checked to be less than a hardcoded max value
- `swap_core.aleo`
  - `finalize_create_pair()` line 481
    - The `decimals` field of the `TokenInfo` object
  - `finalize_swap_exact_tokens_for_tokens()` lines 896-897
  - `finalize_swap_tokens_for_exact_tokens()` line 1020
  - `finalize_swap_private_for_exact_private()` line 1086
    - `10000` is used based on the units of the `swap_fee`

Impact If any of these values are changed, the values in the code could become out of sync if any remaining values aren't updated accordingly. This could lead to major issues in the math. Using Leo constants also makes the code more readable and could prevent future errors due to developer misunderstandings.

For the hardcoded value `10000`, since it is used in so many places, using a Leo constant instead could prevent a typo (like an extra 0) from occurring in one of the locations in which it is used.

Recommendation Use Leo constants instead of hardcoded numerical constants.

4.1.2 V-AST-VUL-002: Faucet Allows Users to Claim All Program Funds

Severity	Info	**Commit**	a12c3e2
Type	Logic Error	**Status**	Intended Behavior
File(s)	`token_faucet/src/main.leo`		
Location(s)	claim()		
Confirmed Fix At	N/A		

The `token_faucet` allows a user to call the `claim()` function which will transfer tokens to that user. The `claim()` function will limit the amount of funds a user can withdraw in a given call to prevent a user from withdrawing large amounts of funds at once. However, the `claim()` function will not track a user's withdrawals, so a user can immediately withdraw additional times after the first.

```
1  // Obtain some tokens from the token's faucet for free.
2      async transition claim(public token_id: field, public amount: u128) -> Future {
3          assert_eq(self.caller, self.signer);
4          let f1: Future = alphaswap.aleo/transfer_public(token_id, self.caller, amount
       );
5          return finalize_claim(token_id, amount, f1);
6      }
7
8      async function finalize_claim(public token_id: field, public amount: u128, public
        f1: Future) {
9          f1.await();
10         // Can't claim more than the faucet setting.
11         assert(amount <= faucets.get(token_id));
12     }
```

Snippet 4.1: Snippet from `claim()`

Impact Any user can drain the protocol of funds by repetitively calling `claim()` . This is a protocol breaking issue, but we have left the severity at medium because the developers have indicated that this program will only be deployed on testnet, so no funds will actually be stolen.

Recommendation Track a user's most recent withdrawal in a mapping and enforce a wait period in blocks before the user can withdraw again.

Developer Response This contract is only used to claim test tokens on the testnet, and we intentionally designed it without simple address and frequency restrictions because such restrictions would be meaningless.

Although the current implementation appears to have no restrictions, it is actually constrained. Firstly, we can set the single-claim amount for each test token, and users must spend aleo credits as a network fee for each claim. Therefore, our faucet claim limitation is essentially the same as the claim limitation on the official aleo faucet.

In general, test tokens have no value, and are only intended for testing purpose.