AQUA PROTOCOL
Security
Audit
Report

BUGBLOW

August 12, 2024

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

BugBlow utilized a widely adopted approach to performing a security audit. Below is a breakout of how our team was able to identify and exploit vulnerabilities.

1.2.1 Architecture review:

- Understand the nature and main objective of the application.
- Read the source code.
- Review the project's architecture.

Stage goals

- Build a good understanding of how the application works.

1.2.2 Threat modeling:

- Understand the project’s critical assets, resources, and security requirements.
- Identify weak spots such as insufficient access control, unvalidated input and output parameters, misconfiguration, etc.
- Check the project against the vulnerability checklist.

Stage goals

- Identify logic and semantic flaws.

1.2.3 Execution:

- Exploit the found weaknesses by performing manual tests and sending unexpected input that would lead to immediate undesired behavior of the application.
- Develop and simulate malicious short- and long-term strategies to compromise the application.
- Document the performed work.

Stage goals

- Simulate both an external and internal threat by trying to break the protocol and gain control of the main assets.

1.2.4. Reporting:

- Discuss the found issues.
- Introduce remediations to mitigate the risks.
- Write an audit report.

Stage goals

- Confirm the relevance of the identified issues and ensure the accuracy of the assigned threat level.
- Deliver the audit report to the Client.

Severity classification

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Vulnerabilities leading to assets theft, fund access locking, or any other loss of funds.
High	Vulnerabilities that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Vulnerabilities that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Vulnerabilities that do not have a significant immediate impact and could be easily fixed.

1.3 Project Overview

The Aqua Protocol is a borrowing protocol on TON that allows minting overcollateralized stablecoins (AquaUSD) against user's collateral. Upon depositing the collateral, a certain amount of AquaUSD is minted and sent to the user's wallet. To maintain the desired collateral ratio (1:1.5) for every borrower, the protocol introduces redemption and liquidation mechanisms. The current prices are received from Storm Oracle.

1.4 Project Dashboard

Project Summary

Title	Description
Client	Aqua
Project name	Aqua Protocol
Timeline	29.07.2024 - 11.08.2024
Number of Auditors	1

Project Last Log

Date	Commit Hash	Note

23.07.2022	d4743fbf86a01c55cc3336791ca7029c0813f562	Merge pull request #8 from aquaprotocolxyz/dev
23.07.2022	663d684aa051c1a7356889e621c4e15c5aef5385	update redeem schema

Project Scope

The audit covered the following files:

File name	Link
contracts/aqua-master.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/aqua-master.func
contracts/aqua-wallet.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/aqua-wallet.func
contracts/collateral-jetton/ jetton-master.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/collateral-jetton/jetton-master.func
contracts/collateral-jetton/ jetton-utils	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/collateral-jetton/jetton-utils.func
contracts/collateral-jetton/jetton-wallet.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/collateral-jetton/jetton-wallet.func
contracts/collateral-jetton/messages.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/collateral-jetton/messages.func
contracts/master/get-methods.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/master/get-methods.func
contracts/master/globals.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/master/globals.func
contracts/master/han	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/

dlers.func	master/handlers.func
contracts/master/messages.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/master/messages.func
contracts/master/packers.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/master/packers.func
contracts/master/storage.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/master/storage.func
contracts/master/utils.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/master/utils.func
contracts/shared/constants.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/constants.func
contracts/shared/error-codes.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/error-codes.func
contracts/shared/extlib.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/extlib.func
contracts/shared/globals.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/globals.func
contracts/shared/jetton-utils.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/jetton-utils.func
contracts/shared/messages.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/messages.func
contracts/shared/op-codes.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/op-codes.func
contracts/shared/packers.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/shared/packers.func
contracts/shared/utils	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/

.func	shared/utils.func
contracts/wallet/get-methods.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/wallet/get-methods.func
contracts/wallet/globals.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/wallet/globals.func
contracts/wallet/handlers.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/wallet/handlers.func
contracts/wallet/packers.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/wallet/packers.func
contracts/wallet/storage.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/wallet/storage.func
contracts/wallet/utils.func	https://github.com/aquaprotocolxyz/contracts/blob/master/contracts/wallet/utils.func

1.5 Summary of findings

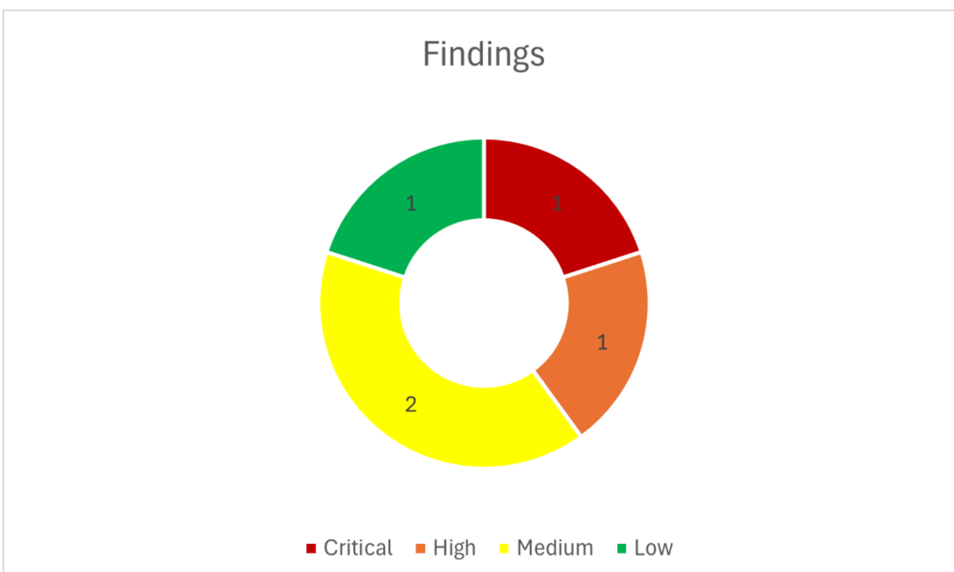

Figure 1. Findings chart

Severity	# of Findings
CRITICAL	1
HIGH	1
MEDIUM	2
LOW	1

2. FINDINGS REPORT

2.1 Critical

2.1.1 Price manipulation with a Replay Attack

Description

A malicious actor can copy past Oracle prices data and start sending it directly to the contract over and over within 120 seconds. If the real price changes within these 120 seconds, the attacker can trick the contract into thinking that the price is still the same. This way, if the actual price of the tokens, the attacker wants to use as collateral, goes down, the attacker can artificially inflate the collateral price and mint more AquaUSD tokens. On a massive scale, with enough money, the attacker can automate this attack and even drain the whole contract.

The problem arises from the fact that the contract doesn't check if the same data has already been sent. It only has a loose check using timestamp whether the data is valid, i.e. "if the signed data has been sent later than 120 seconds ago - disregard it". However, with the volatile market, anything can happen within these 120 seconds. So, the need for an additional security check is present.

Remediation

Request a nonce for each signed data and store it along with the data itself for the period of TTL. If an attacker tries to send the same data, check whether you have already received it in the past. When it is past the TTL period, clean the data from the storage [1]. For example:

```
(int) parse_storm_oracle(cell signed_ref, cell keys_ref) inline_ref {
. . .
;; Load and check the sequence number
int sequence_number = data_slice~load_uint(32);
throw_unless(error::replay_attack_detected, !
;; Mark the sequence number as used
is_sequence_number_used(sequence_number));  mark_sequence_number_used(sequence_number);
. . .
```

2.2 High

2.2.1 Single point of trust

Description

The contract fully trusts the data from a single Oracle source, as long as data has been signed by the Oracle's private key.

```
(int) verify_signature(int data_hash, cell signature_ref, cell public_keys_ref, int public_keys_count) inline_ref {
. . .
slice public_keys = public_keys_ref.begin_parse();


while (public_keys_count) {
   int public_key = public_keys~load_uint(256);
   int valid = check_signature(data_hash, signature, public_key);
   if (valid) {
      return true;
   }
. . .
```

If the Oracle gets compromised or a malicious insider [2] decides to use the private key to compromise Aqua Protocol, they can fully control the collateral price and mint an infinite amount of AquaUSD thus disbalancing the protocol.

Remediation

1. Use at least 2 different sources of Oracles
2. Limit the deviation of the collateral price

2.3 Medium

2.3.1 DOS (Denial of Service) via vaults

Description

TON has a limit for the number of cells in a single contract. In this code, the vaults represent a dictionary that are capable of storing an unlimited amount of data. If the external source such as backend or Keeper is compromised, the can overwhelm the protocol and fill up the storage.

```
() save_data() impure inline_ref { public_keys_count) inline_ref {
  . . .
  save_vault();
  cell wallet_data = begin_cell()
    .store_coins(ctx::wallet_balance)
    .store_dict(ctx::vaults) ;; VAULTS
    .end_cell();

  set_data(
    begin_cell()
      .store_slice(ctx::owner_address)
      .store_slice(ctx::master_address)
      .store_ref(wallet_data)
      .end_cell()
  );
```

```
}
```

Remediation

1. Introduce a limit on how many vaults the contract can store.
2. Implement self-destruction mode when the number of vaults hit a critical number

2.3.2 Borrow rate manipulation

Description

The borrowing rate calculation relies on timestamp "**now()**" that can be slightly manipulated by miners [3]. And the **update_fraction** function does not check whether the result of the calculation is negative. This can lead to the borrowing rate change to the attacker's advantage.

```
() update_fraction() impure inline_ref {
 . . .
  int growth = mulp(divp(now() - ctx::fraction_last_update, YEAR), borrowing_rate);
  ctx::current_borrowing_fraction += growth;
  ctx::fraction_last_update = now();
}
```

Remediations

1. Introduce a check whether **growth** is negative and throw an exception if so.

2.4 Low

2.4.1 Error confusion

Description

2 errors result in the same error code.

```
const int error::unauthorized_incoming_transfer = 707;
const int error::unauthorized_mint = 707;
```

Although unlikely, If two distinct errors share the same error code, an external entity can mistakenly handle one error as if it were the other. This can lead to a situation where critical checks or operations are bypassed, or where incorrect fallback logic is executed.

Remediation

1. Use a different error code for one of the errors

CONSCLUSION

In this audit, we identified several vulnerabilities that could impact the security and operability of the contract:

- Critical: Price Manipulation with a Replay Attack: a critical vulnerability exists where an attacker can repeatedly send stale Oracle data within a 120-second window, allowing them to manipulate collateral prices and mint excessive AquaUSD tokens. This can lead to the contract being drained if not addressed.

- High: Single Point of Trust: the contract relies entirely on a single Oracle for price data, making it vulnerable to manipulation if the Oracle is compromised. This could result in the uncontrolled minting of AquaUSD and destabilization of the protocol.

- Medium: Denial of Service (DoS) via Vaults: the vault system allows for potentially unlimited storage, which could be exploited by an attacker to overwhelm the contract and exhaust its resources, leading to a denial of service.

- Medium: Borrow Rate Manipulation: the borrowing rate calculation is susceptible to manipulation via the now() timestamp, which could be adjusted by miners. Additionally, the lack of a check for negative growth values could be exploited to alter the borrowing rate to the attacker's advantage.

- Low: Error Code Confusion: two distinct errors in the contract share the same error code, which could lead to incorrect handling of error conditions, potentially causing unintended behavior or bypassing critical checks.

Each issue has been documented with appropriate remediation steps. Addressing these issues will enhance the security and robustness of the Aqua protocol.

The Client is advised to implement the recommended changes and conduct a follow-up audit to verify the effectiveness of this remediation.

REFERENCES

[1] https://crypto.stackexchange.com/questions/41170/what-advantage-is-there-for-using-a-nonce-and-a-timestamp

[2] https://securityintelligence.com/news/insider-hacks-exfiltrate-fives-times-more-records/

[3] https://medium.com/coinmonks/smart-contract-security-block-timestamp-manipulation-baec1b95c921