Zellic

Matrixdock-STBT

Smart Contract Security Assessment

February 7, 2023

Prepared for:

Ding Yao

Matrixdock

Prepared by:

Varun Verma and Filippo Cremonese

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Matrixdock from January 31st to February 1st. During this engagement, Zellic reviewed Matrixdock-STBT's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are the access controls sufficient to protect against unauthorized issuing?
- Is the proxy pattern implemented safely?
- Could there be any rounding issues in the share-to-value conversion?
- Do the send and receive permissions work as intended?

1.2 Non-Goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. In this case, the limited duration of the assessment prevented us from exhaustively evaluating the protocol's economic performance at a larger scale.

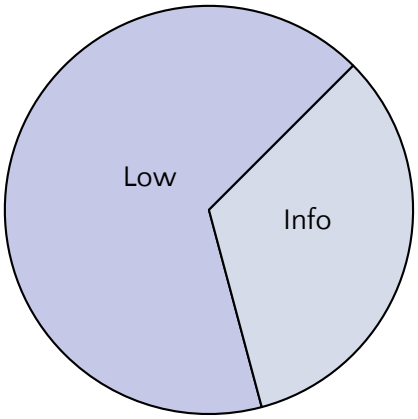
1.3 Results

During our assessment on the scoped Matrixdock-STBT contracts, we discovered three findings. No critical issues were found. Of the three findings, two were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Matrixdock's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	1



2 Introduction

2.1 About Matrixdock-STBT

The Matrixdock STBT (i.e., short-term treasury bill token) is a token backed by the underlying assets of reverse repurchase agreements collateralized by U.S. Treasury securities and U.S. Treasury securities maturing within six months. STBT operates on the Ethereum blockchain under the ERC-1400 standard, which is transferable solely to account holders that have been preapproved via a KYC/AML whitelisting mechanism.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality stan-

dards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Matrixdock-STBT Contracts

Repository	https://github.com/Matrixport-STBT/STBT-contracts
Versions	`ec454eed03ba30cc32e6d9fed87daa7ee0d0e94d`
Programs	• STBT • UpgradeableSTBT • StbtTimelockController
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three person-days. The assessment was conducted over the course of two calendar days.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Varun Verma, Engineer
varun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

January 31, 2023	Start of primary review period
February 1, 2022	End of primary review period

3 Detailed Findings

3.1 Custom proxy architecture

- **Target**: `STBT.sol`
- **Category**: Code Maturity
- **Severity**: Medium
- **Likelihood**: Medium
- **Impact**: Low

Description

STBT will be deployed through a proxy contract, a common design that allows upgrading the code of a contract.

STBT implements a custom proxy, which has strong constraints in the ability to upgrade the contract code and is arguably more error prone than other existing alternatives.

Specifically, the storage layouts of the custom proxy implementation `UpgradeableSTBT` and of the implementation `STBT` are required to not clash. This is implemented by replicating the initial part of the storage layout of the implementation contract in the proxy. The `STBT` contract storage has an `uint[300]` array of `placeholders`, where `UpgradableSTBT` stores the address of the implementation contract.

```
contract UpgradeableSTBT is Proxy {
    // override
    address public owner;
    address public issuer;
    address public controller;
    address public moderator;
    // new state below
    address public implementation;
    // ...
}

contract STBT is Ownable, ISTBT {
    // all the following three roles are contracts of
    governance/TimelockController.sol
    address public issuer;
    address public controller;
    address public moderator;
```

```
    uint[300] public placeholders;
    /// ...
}
```

This coupling of storage layouts is unusual and unnecessary; other proxy implementations move the address of the implementation contract to a different storage location (via inline assembly), in order to not interfere with the implementation storage layout.

Impact

This issue does not describe an exploitable security vulnerability in the code as reviewed and is therefore reported as low severity. However, we believe this design choice introduces a higher risk of errors when upgrading the contract.

Recommendations

We recommend evaluating the adoption of one of the several de facto standard proxy architectures that have been developed and proven effective, such as UUPSUpgradeable.

Remediation

Matrixdock acknowledged the finding and will not remediate at this time.

3.2 High rate of failures in test suite

- **Target**: `stbt-test.js`
- **Category**: Code Maturity
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Low

Description

One of the routine steps performed during the evaluation of a codebase is inspection of the accompanying test suite. When running the test suite using the instructions available in the README, we observed a failure rate of 52% (24 out of 46).

Impact

Integrating a comprehensive test suite with a continuous integration service is tremendously important for preventing bugs from being deployed.

For example, for one of the tests within `it("redeem: errors" ...)`, it expects a revert to happen because of `NO_SEND_PERMISSION`, when the revert cause is because the `msg.sender` is not the `issuer`.

```
await expect(stbt.connect(alice).redeem(123, '0x'))
    .to.be.revertedWith("NO_SEND_PERMISSION");
```

Recommendations

We recommend fixing the failing tests and running the tests automatically (e.g., by integrating them with a CI service or in Git hooks).

Remediation

Matrixdock states that this issue was caused by an unsynced test file. The finding was fixed in commit 06c46695 and now all tests pass.

3.3 Lack of sanity checks

- **Target**: `STBT.sol`
- **Category**: Code Maturity
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Informational

Description

Most functions (such as the `UpgradableSTBT` constructor or the setter functions of `STBT`) do not perform any sanity check. Sanity checks are commonly included to prevent human error, for instance by checking that an address is not zero or that a numeric parameter is within a reasonable range.

Impact

While this issue does not describe an exploitable vulnerability, the lack of sanity checks increases the risk of an incorrect interaction with the contract due to human error.

Recommendations

Consider adding sanity checks guarding against trivial user mistakes.

Remediation

Matrixdock acknowledged the finding and will not remediate at this time.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Lack of two-step ownership transfer

Currently, there is a one-step ownership transfer in place. This could threaten a loss of ownership of the contract in the accidental scenario in which a wrong owner is set. A two-step ownership transfer process, in which the new address must claim their ownership, protects against this scenario and could be beneficial from a security standpoint.

4.2 Commented-out code

The codebase includes commented-out code, including entire functions; this makes the codebase less maintainable and is not required since the codebase is maintained through Git.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 File: STBT

Function: `distributeInterests()`

Intended behavior

Distribute interests, increasing the newTotalSupply and keeping track of the last distribution time.

Branches and code coverage

Intended branches:

- Valid interests are distributed and accounted for in the newTotalSupply.
  ☑ Test coverage

Negative behavior:

- MIN_DISTRIBUTE_INTERVAL_VIOLATED
  ☑ Negative test?
- NOT_ISSUER calling function
  ☑ Negative test?
- MAX_DISTRIBUTE_RATIO_EXCEEDED
  ☑ Negative test?

Preconditions

- Issuer has been set.

Inputs

- `interestToTime`:

  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: Any interestToTime can be used; perhaps this should have some input validation. Just used in an event.
- `interestFromTime`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: Any interestFromTime can be used; perhaps this should have some input validation. Just used in an event.
- `_distributedInterest`:
  - **Control**: Less than or equal to oldTotalSupply * maxDistributeRatio.
  - **Authorization**: N/A.
  - **Impact**: Only a valid distributedInterest parameter can be used that does not exceed the desired ratio.

Function: `issue()`

Intended behavior

Mint shares for the supplied tokenHolder.

Branches and code coverage

Intended branches:

- Issue mints the designated value for the tokenHolder.
  ☑ Test coverage

Negative behavior:

- Zero address cannot be receiver.
  ☑ Negative test?
- Non-receive recipient cannot be issued to.
  ☑ Negative test?
- Non-issuer cannot issue.
  ☑ Negative test?

Preconditions

- Issuer is set.

Inputs

- `_data`:
  - **Control**: Any data is permissible.
  - **Authorization**: N/A.
  - **Impact**: Data is only used in event emitting, so no major impact.
- `_value`:
  - **Control**: Any value is acceptable.
  - **Authorization**: N/A.
  - **Impact**: A permissible receiver can be minted any amount of shares.
- `_tokenHolder_`:
  - **Control**: Recipient is non-zero address.
  - **Authorization**: Recipient has receive permissions.
  - **Impact**: Only valid recipients can have shares issued to them.

Function call analysis

- `getAmountByShares()`
  - **What is controllable?** The sharesDelta, controllable by the Issuer.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is only used in event emitting.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert unless totalShares is zero, an improbable scenario and no risk of reentrancy.
- `_mintSharesWithCheck()`
  - **What is controllable?** Issuer can control the tokenHolder and sharesDelta.
  - **If return value controllable, how is it used and how can it go wrong?** N/A. No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Shares are not minted under revert scenario, which can only happen if the receiver does not have receive permissions or is address(0) and there are no external calls to risk a reentrancy.
- `getSharesByAmount()`
  - **What is controllable?** The value passed in.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the shared for the tokenHolder, only controllable by a valid issuer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Shares are not minted under revert scenario, no external calls happening to create a reentrancy threat or unusual control flow.

Function: `redeemFrom()`

Intended behavior

Redeem for another tokenHolder address given that the allowance is sufficient.

Branches and code coverage

Intended branches:

- tokenHolder is redeemed.
  - ☑ Test coverage

Negative behavior:

- Non-issuer cannot call this function.
  - ☐ Negative test?
- tokenHolder can have send permissions.
  - ☑ Negative test?
- Redeem exceeds allowance.
  - ☑ Negative test?

Preconditions

- Total supply is not zero.
- Issuer has been set.

Inputs

- `_data_`:
  - **Control**: User has full control.
  - **Authorization**: None.
  - **Impact**: Only used in event emitting, so no large impact.
- `_value`:
  - **Control**: User has full control, granted it is less than or equal to the current allowance.
  - **Authorization**: N/A.
  - **Impact**: Can only redeem for an authorized value.
- `_tokenHolder`:
  - **Control**: User has full control.
  - **Authorization**: Has send permissions and sufficient allowance for the specific msg.sender.
  - **Impact**: Redeeming for a particular tokenHolder must pass allowance checks.

Function call analysis

- `burnSharesWithCheck()`
  - **What is controllable?** tokenHolder and sharesDelta.
  - **If return value controllable, how is it used and how can it go wrong?** N/A. No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can only revert if send permissions are not valid for that tokenHolder or the amount of shares to burn is greater than the user currently holds. No external calls for reentrancy or unusual control flow.
- `getSharesByAmountRoundUp()`
  - **What is controllable?** The value.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is controllable to the extent that the tokenHolder actually has that amount of shares or more.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can never revert - the edge case of a totalSupply of zero is accounted for.

5.2 File: StbtTimelockController

Function: `schedule()`

Intended behavior

Schedules a function call that can be executed after a preconfigured time delay has passed.

Branches and code coverage

Intended branches:

- Gets the delay for the selector and schedules the function call.
  ☑ Test coverage

Negative behavior:

- The selector is not one of the allowed ones.
  ☑ Negative test?
- Caller is unauthorized.
  ☑ Negative test?

Preconditions

- Caller has been authorized.
- Selector delay has been configured.

Inputs

- `target`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: The contract that will be invoked.
- `value`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: The ETH value to be used for the call.
- `data`:
  - **Control**: First four bytes must be an approved selector.
  - **Authorization**: N/A.
  - **Impact**: Calldata provided in the call.
- `predecessor`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: Identifier of the invocation that must have been executed first to allow execution of this call.
- `salt`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: Salt included in the computation of the ID of the call (hash of all parameters).

5.3 File: UpgradeableSTBT

Function: `resetImplementation()`

Intended behavior

Changes the address of the implementation contract invoked by the proxy.

Branches and code coverage

Intended branches:

- Changes the address of the implementation contract.
  ☑ Test coverage

Negative behavior:

- Caller is unauthorized.
  ☑ Negative test?

Preconditions

- Caller has been authorized.

Inputs

- `_impl`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: The address of the new implementation contract.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered three findings. Of these, two were low risk and one was informational in nature. Matrixdock acknowledged all findings and implemented fixes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.