Zellic

Minter

Smart Contract Security Assessment

June 30, 2023

Prepared for:

Ding Yao

Matrixdock

Prepared by:

Aaron Esau

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 June 27th to June 29th, 2023. During this engagement, Zellic reviewed Minter'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 there any logic flaws in the contracts?
- What are the centralization risks?

1.2 Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Plans for deployment

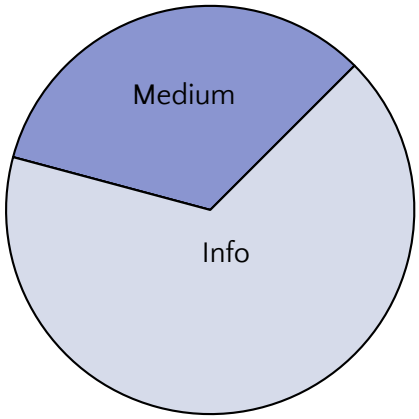
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped Minter contract, we discovered three findings. No critical issues were found. One was of medium impact and the remaining findings were informational in nature.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Minter

Minter offers services that allow users to schedule mint and redeem operations, enabling the conversion between underlying and STBT tokens.

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 contract.

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 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 standards. 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:

Minter Contract

Repository	https://github.com/Matrixdock-STBT/STBT-contracts/
Version	STBT-contracts: `0f5d608eccf8a72c83b9e7e4682fae0cb7f5f828`
Program	Minter
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with one consultant for a total of 12 person-hours. The assessment was conducted over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultant was engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 27, 2023	Start of primary review period
June 29, 2023	End of primary review period

3 Detailed Findings

3.1 Mutable globals introduce centralization risk

- **Target**: Minter
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

The `timeLockContract` variable stores the address of the timelock contract with the "Issuer" role on the STBT contract. Meanwhile, `targetContract` stores the address of the STBT contract with the `permissions` selector and signifies the token to deposit to the `poolAccount`.

Each of these variables is defined to be mutable and has a setter function callable by the owner:

```
address public timeLockContract;
address public targetContract;
address public poolAccount;

    // [...]

function setTimeLockContract(address _timeLockContract)
    onlyOwner external {
    timeLockContract = _timeLockContract;
}

function setTargetContract(address _targetContract) onlyOwner external {
    targetContract = _targetContract;
}

function setPoolAccount(address _poolAccount) onlyOwner external {
    poolAccount = _poolAccount;
}
```

Impact

If the Minter owner multi-sig or governance were compromised, or otherwise performed malicious behavior, the owner could front-run high-value transactions to manipulate them in the following ways:

- **`targetContract`**: A malicious owner could
  - change which users have permissions to send or receive funds.
  - change which token the user pays with in `redeem` (though the transfer must have an approval, if the address is ERC-20 compliant). For example, the owner can make a redeem call without paying STBT by changing the `targetContract` temporarily to a contract whose `transferFrom` makes no state changes other than triggering the owner to change `targetContract` back to STBT. This would result in `targetContract` being "invalid" for the first read from storage (i.e., not charging any token) but being "valid" for the second read from storage:

```
function redeem(uint amount, address token, bytes32 salt,
    bytes calldata extraData) external {
    // [...]
    IERC20(targetContract).transferFrom(msg.sender,
    poolAccount, amount);
    // [...]
    IStbtTimelockController(timeLockContract).schedule(
          targetContract, 0, data, bytes32(""), salt, 0);
    // [...]
}
```

  - prevent `mint` or `redeem` from being callable.
- **`poolAccount`**: A malicious owner could change the destination of the STBT tokens when a user calls `redeem` and thereby steal the tokens.
- **`timeLockContract`**: A malicious owner could prevent the timelock contract from calling `issue` and `redeemFrom` or prevent `mint` or `redeem` from being callable (i.e., without reverting).

Recommendations

Ensure the owner address is trusted, and consider making the owner a governance contract or multi-sig.

Regardless of who the owner is, consider removing the setter functions for `targetContract`, `poolAccount`, and `timeLockContract` and making the addresses immutable. This may hinder upgradability but allows the contract to be safer and more trustless.

Remediation

Matrixdock added that this is by design, and that they do not currently have plans to make changes in response to this finding.

3.2 Concerns in `redeemSettle` and `rescue` functions

- **Target**: Minter
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following `rescue` function allows users to withdraw funds accidentally transferred into the contract:

```
function rescue(address token, address receiver, uint amount)
    onlyOwner external {
    require(virtualCountOfRedeemSettled == nonceForRedeem, "MINTER:
    PENDING_REDEEM");
    IERC20(token).transfer(receiver, amount);
}
```

The `redeemSettle` function allows the owner to refund `redeem` requests:

```
function redeemSettle(address token, uint amount, uint64 nonce,
    bytes32 redeemTxId,
              uint redeemServiceFeeRate, uint executionPrice)
    onlyOwner external {
    virtualCountOfRedeemSettled++;
    address target = redeemTargetMap[nonce];
    require(target != address(0), "MINTER: NULL_TARGET");
    IERC20(token).transfer(target, amount);
    emit Settle(target, token, nonce, amount, redeemTxId,
    redeemServiceFeeRate, executionPrice);
    delete redeemTargetMap[nonce];
}
```

There are a few concerns with the `redeemSettle` function:

- The owner may arbitrarily choose the `token` address.
- The owner may arbitrarily choose the `amount` value.
- The owner can reenter through the `IERC20(token).transfer( ... )` call before the `delete redeemTargetMap[nonce];` to provide multiple refunds for one nonce and `redeemSettle` a nonce that had `redeemFrom` called on STBT successfully already.

This could cause one `redeemTargetMap` to increment `virtualCountOfRedeemSettled` an arbitrary number of times, allowing the owner to bypass the `require` statement at the start of the `rescue` function.

Impact

Despite how powerful the `redeemSettle` and `rescue` functions appear —– and the fact that one `require` check is bypassable —– the functions do not pose a centralization risk because the contract does not necessarily hold funds not transferred in by the owner.

Recommendations

Ensure the owner address is trusted, and consider making the owner a governance contract or multi-sig.

Consider moving the `redeemTargetMap` key deletion earlier in the function (in a checks-effects-interactions-type pattern):

```
function redeemSettle(address token, uint amount, uint64 nonce,
    bytes32 redeemTxId,
              uint redeemServiceFeeRate, uint executionPrice)
    onlyOwner external {
    virtualCountOfRedeemSettled++;
    address target = redeemTargetMap[nonce];
    require(target != address(0), "MINTER: NULL_TARGET");
    delete redeemTargetMap[nonce];
    IERC20(token).transfer(target, amount);
    emit Settle(target, token, nonce, amount, redeemTxId,
    redeemServiceFeeRate, executionPrice);
    delete redeemTargetMap[nonce];
}
```

Remediation

This issue has been acknowledged by Matrixdock, and a fix was implemented in commit 205050b3.

3.3 Missing limits on `feeRate`

- **Target**: Minter
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following code in the `mint` function extracts the `feeRate` from the `purchaseInfoMap` then subtracts it from `UNIT` (defined as $10 * 10$).

```
uint receiverAndRate = purchaseInfoMap.get(token);
require(receiverAndRate ≠ 0, "MINTER: TOKEN_NOT_SUPPORTED");
address receiver = address(uint160(receiverAndRate>>96));
uint feeRate = uint96(receiverAndRate);
DepositConfig memory config = depositConfigMap[token];
require(config.minimumDepositAmount ≠ 0 &&
    depositAmount ≥ config.minimumDepositAmount, "MINTER:
    DEPOSIT_AMOUNT_TOO_SMALL");
uint proposeAmount = depositAmount*(UNIT-feeRate)/UNIT;
```

In 96 bits, the maximum value that can be stored is 79228162514264337593543950336, which is much greater than `UNIT`. This means it is possible for `UNIT-feeRate` to underflow.

The owner can change the `feeRate` on demand using the following function, which updates `purchaseInfoMap`:

```
function setCoinInfo(address token, uint receiverAndRate)
    onlyOwner external {
    purchaseInfoMap.set(token, receiverAndRate);
}
```

Note that we do not consider the ability of the admin to change the fees on demand itself to be a centralization risk with security impact, because the `mint` function has a `minProposedAmount` parameter that ensures the caller will receive at least the amount they expect:

```
require(proposeAmount ≥ minProposedAmount, "MINTER:
    PROPOSE_AMOUNT_TOO_SMALL");
```

Impact

Underflow reversions would block execution of this function. So, the owner can permanently block execution of `mint` by setting `feeRate` to a value greater than `UNIT`.

Additionally, the presence of this issue may hinder future formal verification of the `mint` function.

Recommendations

Ensure the owner address is trusted, and consider making the owner a governance contract or multi-sig.

Additionally, consider adding the following requirement to `setCoinInfo`:

```
function setCoinInfo(address token, uint receiverAndRate)
    onlyOwner external {
    require(uint96(receiverAndRate) < UNIT, "MINTER: FEE_RATE_TOO_
        LARGE");
    purchaseInfoMap.set(token, receiverAndRate);
}
```

Remediation

This issue has been acknowledged by Matrixdock, and a fix was implemented in commit 205050b3.

4 Audit Results

At the time of our audit, the audited code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Minter contract, we discovered three findings. No critical issues were found. One was of medium impact and the remaining findings were informational in nature. Matrixdock acknowledged all findings and implemented fixes.

4.1 Disclaimer

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 code added to the 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 samples 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.