Zellic

February 12, 2024

**Prepared for**
**Kaso Qian**
**Alex Valaitis**
**Hao Jün Tan**
Chateau Capital

**Prepared by**
**Ulrich Myhre**
**Vlad Toie**
Zellic

# Chateau

Smart Contract Security Assessment

## Contents

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

# 1. Overview

## 1.1. Executive Summary

Zellic conducted a security assessment for Chateau Capital from February 9th to February 11th, 2024. During this engagement, Zellic reviewed Chateau's code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.2. 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:

- What is the process for performing staking and unstaking operations in the StakingPool?
- Is there a risk of system congestion due to an excessive influx of issue information?
- Can malicious users exploit vulnerabilities to drain the pools?
- How are swap operations conducted, and what assets are expected to be swapped out?

## 1.3. 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

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

During this assessment, the limited amount of time available prevented us from comprehensively assessing the potential implications of each of the possible tokens that are to be used and/or traded within the pools. Issues that may arise here could be related to the decimals of the tokens, the potential of miscalculating the amount of tokens to be swapped, rounding errors, and so on. Additionally, some weird ERC-20 tokens ↗ may do unexpected things like reenter upon `safeTransferFrom` and have the potential of causing further issues.

The prices of tokens are based on real-world assets (RWA), with prices that cannot be manipulated by on-chain methods. This essentially removes arbitrage (and similar market manipulation tactics) from the equation during swaps, and those are out of scope for this particular use-case.

Regarding the last point, Chateau Capital states:

Chateau Real World Asset Tokens are issued against underlying instruments held in a portfolio, fund or entity. The price of these tokens are issued or redeemed at Net Asset Value, tracking the price of their underlying instruments stated in monthly or quarterly admin reports (see asset offering documents for more details). For more information on the team and the latest financial reports on each token, visit the corresponding token page.

## 1.4. Results

During our assessment on the scoped Chateau contracts, we discovered six findings. Three critical issues were found. One was of medium impact and the remaining findings were informational in nature.

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

### Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| 🟥 Critical | 3 |
| 🟧 High | 0 |
| 🟨 Medium | 1 |
| 🟩 Low | 0 |
| ⬜ Informational | 2 |

After the completion of this audit, Chateau Capital acknowledged all identified issues and successfully implemented the necessary fixes.

# 2. Introduction

## 2.1. About Chateau

Chateau Capital is a DeFi protocol for real world equity, debt, and derivatives.

## 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 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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

## 2.3. Scope

The engagement involved a review of the following targets:

### Chateau Contracts

| | |
|---|---|
| **Repository** | https://github.com/chateau-capital/ca ↗ |
| **Version** | ca: `16b086152d95a69553eabe7204ef45835da6ca9f` |
| **Programs** | • Share.sol<br>• NotAmerica.sol<br>• Factory.sol<br>• StakingPool.sol<br>• VaultPool.sol |
| **Type** | Solidity |
| **Platform** | EVM-compatible |

## 2.4. Project Overview

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

## Contact Information

The following project manager was associated with the engagement:

**Chad McDonald**
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

**Ulrich Myhre**
Engineer
ulrich@zellic.io ↗

**Vlad Toie**
Engineer
vlad@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **February 9, 2024** | Start of primary review period |
| **February 11, 2024** | End of primary review period |

# 3. Detailed Findings

## 3.1. Potential DOS during swap

| | | | |
|---|---|---|---|
| **Target** | StakingPool | | |
| **Category** | Business Logic | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

The `swap` function iterates through the entire array of `issues`. These `issues` are created upon performing a `stake()` operation, and they essentially represent a user deposit.

Because there is no minimal cap as to the amount of tokens to `stake()` in the pool, a malicious user could perform `stake()` with one unit of tokens multiple times, which would essentially clog the `issues` array with a lot of entries, severely impairing the for loop performed in the `swap` function:

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        if (amountB > 0) {
            if (amountB >= issueInfo.issueAmount) {
                uint amountA = (issueInfo.issueAmount * 10000) / rate;
                amountB -= issueInfo.issueAmount;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.isStaking = false;
            } else {
                uint amountA = (amountB * 10000) / rate;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.issueAmount -= amountB;
                amountB = 0;
            }
        }
    }
}
```

### Impact

The lack of adequate checks in `stake()` could potentially lead to a denial of service (DOS) for the StakingPool contract, as a malicious user could clog the system with a lot of small `stake` operations, which would make the `swap` function perform a lot of unnecessary iterations, eventually reaching the gas limit and rendering the pool unusable.

### Recommendations

We recommend implementing a minimum cap for the amount of tokens that can be staked in the pool. This would limit the amount of `issues` that could be created and would prevent a potential DOS attack. Additionally, we recommend reevaluating the `swap` function to extract the necessary liquidity in various ways, such as using a more efficient algorithm or by using a more dynamic approach to determining which `issues` should be considered for the swap.

On top of the aforementioned mitigation directions, we recommend implementing a break condition should `amountB` be 0 to prevent unnecessary iterations.

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        if (amountB > 0) {
            // ...
        } else {
            break;
        }
    }
}
```

### Remediation

This issue has been acknowledged by Chateau Capital, and a fix was implemented in commit 8184d6c5 ↗.

## 3.2. Transparent pool ordering allows for preferential swaps

| **Target** | StakingPool | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Critical |
| **Likelihood** | Low | **Impact** | Critical |

### Description

The way swaps are performed in the pool allows for anyone to anticipate whose liquidity will be used for the next swap. This transparent queue allows for front-running and other types of attacks that could be used to manipulate the pool's liquidity to an attacker's advantage, should the market conditions be favorable for such an attack.

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        if (amountB > 0) {
        // ...
}
```

Note that the `swap` function starts iterating through the `issues` array from the last index to the first, which means that the last user to `stake` will be the first to have their liquidity used for the next swap.

### Impact

Anyone can anticipate whose liquidity will be used for the next swap, which potentially allows for front-running or even denying the service to other users by staking a large amount of tokens and then withdrawing them, effectively clogging the pool's liquidity and rendering the entire pool unusable. As the stakes are essentially a "last in, first out" (LIFO) stack, the newest stake will be always be considered first for a swap. Someone who wants to hamper swapping could potentially front-run an incoming swap, making sure that their own stake is considered first for the swap. The same situation could arise whenever market conditions are right, exacerbated by the fact that the rate is static.

### Recommendations

We recommend redesigning the `swap` function to use a more dynamic approach in determining which `issues` should be considered for the swap and to prevent any potential front-running attacks. This could be achieved by using a more efficient aggregation algorithm, rather than a simple last-in, first-out approach.

### Remediation

This issue has been acknowledged by Chateau Capital, and a fix was implemented in commit `849a8a63` ↗.

Chateau Capital removed the `swap` function and states that user will instead use Uniswap or similar exchanges.

## 3.3. Withdraw leads to loss of stake

| **Target** | StakingPool | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

Whenever the owner of the StakingPool contract calls `withdraw()`, the `indexStar` variable is set to be equal to `indexEnd`.

```
function withdraw() public onlyOwner {
    // ...
    indexStar = indexEnd;
    // ...
}
```

These two variables represent the cut-off index for withdrawn stakes and the next index to assign when someone stakes; `indexStar` is initialized to `0` and `indexEnd` starts at `1`. During a stake, the `indexEnd` value is used, then increased. The first stake will be put into the `issues` mapping at index `1`, and `indexEnd++` will be `2`.

```
function stake(uint256 amount) public NOT_AMERICAN {
    // ...
    issues[indexEnd] = Issue(msg.sender, amount, block.timestamp, true);
    userIssueIndex[msg.sender].push(indexEnd);
    indexEnd++;
    // ...
    }
```

During unstake, there is a check that the index fetched from `userIssueIndex` is **larger than** `indexStar`.

```
function unstake() public NOT_AMERICAN {
    uint[] memory userIssueIndexs = userIssueIndex[msg.sender];

    uint unstakeAmount;
    for (uint i; i < userIssueIndexs.length; i++) {
        uint index = userIssueIndexs[i];
        if (index > indexStar) {
            // Unstake
        }
    }
    // ...
}
```

The point of this is that all issues should be invalidated when a withdraw happens, but because `indexEnd` points to the **next** issue index, it will also invalidate the next stake, which has not happened yet. A similar check happens in `swap`, where it will count down from the end of the issues, **starting at an invalid index** and counting down to the value above `indexStar`.

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        // ...
    }
    // ...
}
```

## Impact

The first stake that happens after a withdraw cannot be unstaked and will not be considered for swap. These funds are essentially lost unless the owner calls withdraw again and manually gives the stake back to the user. But doing this will trigger the bug again for the next stake.

The swap will also read one invalid issue each time, as `issues[indexEnd]` is never actually assigned, except for a brief moment in `stake` until `indexEnd++` is called right after.

## Recommendations

Rename `indexEnd` to something like `nextIndex` to reflect what it actually is, and make sure it is always strictly larger than `indexStar`. In the withdraw function, set `indexStar` to `nextIndex - 1` or similar. Change any loops to not start at this exact index, as it is not yet valid.

Write test cases for more complicated scenarios that involve multiple users, doing multiple stakes and unstakes, both before and after the admin calls withdraw.

## Remediation

This issue has been acknowledged by Chateau Capital, and fixes were implemented in the following commits:

- `c296913a` ↗
- `8a221f45` ↗

## 3.4. Centralization risks

| | | | |
|---|---|---|---|
| **Target** | StakingPool, VaultPool | | |
| **Category** | Protocol Risks | **Severity** | High |
| **Likelihood** | Low | **Impact** | Medium |

### Description

The StakingPool contract has a few centralization risks that could potentially lead to a single point of failure or to a single entity having too much control over the pool's operations. For example, the `withdraw` function allows the owner to withdraw all the liquidity from the pool, and the `setRate` function allows the owner to change the rate at which `redeemToken` is swapped for `issueToken`.

In StakingPool:

```
function withdraw() public onlyOwner {
    uint balance = issueToken.balanceOf(address(this));
    issueToken.safeTransfer(msg.sender, balance);
    indexStar = indexEnd;
    pendingLiquidation = 0;
    emit AdminWithdraw(msg.sender, balance);
}

function setRate(uint _rate) public onlyOwner {
    rate = _rate;
    emit RateChange(_rate);
}
```

In VaultPool:

```
function withdraw() public onlyOwner {
    uint balance = issueToken.balanceOf(address(this));
    issueToken.safeTransfer(msg.sender, balance);
    _pause();
    emit AdminWithdraw(msg.sender, balance);
}
```

### Impact

The owner of the StakingPool and VaultPool contracts has the ability to withdraw all the liquidity from the pool and to change the rate at which `redeemToken` is swapped for `issueToken`. In the case where a malicious actor gets control of the owner's account, this could potentially lead to a rug pull or to the owner manipulating the pool's swap rates to their advantage.

### Recommendations

We recommend reevaluating the `withdraw` and `setRate` functions to ensure that the owner's control over the pool's operations is limited and that the pool's liquidity is not at risk of being withdrawn by the owner. This could be achieved by removing the `withdraw` function or enforcing an in-contract timelock. Additionally, we recommend implementing a more timelocked approach to changing the rate at which `redeemToken` is swapped for `issueToken` to prevent the owner from manipulating the pool's swap rates to their advantage very quickly.

### Remediation

This issue has been acknowledged by Chateau Capital.

Chateau Capital states that the centralization aspect is required in order to withdraw the stakes and trade them into real-world assets. They will also use a multi-sig account to control the owner account.

Chateau Capital provided the following list of trust assumptions their users should consider when purchasing tokens.

> Tokens issued by Chateau Protocol represent shares in an issuing entity, or fund. Investors who participate in token issuances on chateau.capital ↗ are placing trust in the integrity of the token offering as stipulated in the offering document, and have full legal recourse against the issuing entity, which may be Chateau Capital Corp. or other parties using Chateau's platform.
>
> To mitigate smart contract risk, Chateau utilizes a 2/3 Multisig for $CHAD.D, and strive to move stablecoins out of the contract as soon as possible during issuance windows. Custody of the underlying instruments are held with the issuing fund, or professional custody firms, with the issuer providing ongoing updates to token holders. Token investors can also contact the Issuer at any time for up to date audits of the underlying assets.
>
> Chateau will also apply for Virtual Asset licenses in several jurisdictions to enhance compliance.

## 3.5. Lack of documentation

| | | | |
|---|---|---|---|
| **Target** | Project Wide | | |
| **Category** | Code Maturity | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

No code documentation has been provided as of the time of the audited commit. Additional reaffirmation of the mechanisms would help with comprehensibility as well as security sanity checks. For example, the following snippet from StakingPool.sol's `swap` function:

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        if (amountB > 0) {
            if (amountB >= issueInfo.issueAmount) {
                uint amountA = (issueInfo.issueAmount * 10000) / rate;
                amountB -= issueInfo.issueAmount;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.isStaking = false;
            } else {
                uint amountA = (amountB * 10000) / rate;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.issueAmount -= amountB;
                amountB = 0;
            }
        }
    }
}
```

For the nontechnical user, the above code is not very readable. It is important to add comments to the code to explain the operations and the expected behavior. For example, the following comments could be added to the code:

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];

    // Check if the issue is staking
    if (issueInfo.isStaking) {
        // Check if there are any tokens to redeem
        if (amountB > 0) {

            // Check if the amount of tokens to redeem is greater than
                the issue amount
            if (amountB >= issueInfo.issueAmount) {

                // Calculate the amount of tokens to redeem
                uint amountA = (issueInfo.issueAmount * 10000) / rate;

                // Transfer the tokens to the user
                amountB -= issueInfo.issueAmount;
                redeemToekn.safeTransfer(issueInfo.user, amountA);

                // Mark the issue as not staking
                issueInfo.isStaking = false;
            } else {
                // Calculate the amount of tokens to redeem
                uint amountA = (amountB * 10000) / rate;

                // Transfer the tokens to the user
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.issueAmount -= amountB;

                // Set the amount of tokens to redeem to 0
                amountB = 0;
            }
        }
    }
}
```

## Impact

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified later on.

In general, a lack of documentation impedes the auditors' and external developers' ability to read, understand, and extend the code. The problem is also carried over if the code is ever forked or reused.

## Recommendations

We recommend adding more comments to the code in order to explain the operations and the expected behavior. This would help with comprehensibility and would make the code more readable for any users, regardless of their technical background.

### Remediation

This issue has been acknowledged by Chateau Capital, and a fix was implemented in commit `0cca694b` ↗.

## 3.6. Lack of reentrancy protection

| | | | |
|---|---|---|---|
| **Target** | StakingPool | | |
| **Category** | Code Maturity | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

There are currently a few scenarios that do not sufficiently protect against reentrancy attacks. In order to execute such attacks, it is necessary to interact with a token that gives control back to a caller during execution. This currently happens for certain tokens that offer hook functions when tokens are transferred or that require fees to be paid for each transfer.

One such example is in the `swap` function,

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        if (amountB > 0) {
            if (amountB >= issueInfo.issueAmount) {
                uint amountA = (issueInfo.issueAmount * 10000) / rate;
                amountB -= issueInfo.issueAmount;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.isStaking = false;
            } else {
                uint amountA = (amountB * 10000) / rate;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.issueAmount -= amountB;
                amountB = 0;
            }
        }
    }
}
```

where the `isStaking` flag is set to `false` *after* the transfer has happened.

### Impact

If the `swap` function above interacts with a token that gives some interactability during `safeTransfer`, it is possible to reenter the function, pay more redeem tokens, and then the contract will pick the same issue over again.

### Recommendations

Make sure that all tokens are carefully vetted to ensure that there are no ways to trigger reentrancy during transfer. Additionally, consider adding reentrancy protections using OpenZeppelin's `nonReentrant` modifier ↗ and refactor function logic to follow the checks-effects-interaction pattern ↗.

### Remediation

This issue has been acknowledged by Chateau Capital, and a fix was implemented in commit `910de03d` ↗.

# 4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

## 4.1. Multiple variables have typos in their names

Across the codebase, multiple variables have typos in their names. This can lead to confusion and potential bugs. For example —

In StakingPool,

- `redeemToekn` should be `redeemToken`.
- `indexStar` should be `indexStart`.

In VaultPool,

- `sharetoekn` should be `shareToken`.

## 4.2. Additional checks could be performed across the codebase

Some additional checks could be performed across the codebase to ensure the robustness of the system.

For example —

- At the end of the `swap` function, revert if `amountB > 0`. This edge case could occur when there have not been enough stakers' deposits to pay for the swap.

```
function(uint256 amount) external {
    for (uint i = indexEnd; i > indexStar; i--) {
        // ...
    }
    require(amountB == 0, "Not enough stakers to pay for the swap");
}
```

- Ensure that there are zero `redeemTokens` left after the swap.

```
function(uint256 amount) external {
    for (uint i = indexEnd; i > indexStar; i--) {
        // ...
    }
   require(redeemToken.balanceOf(address(this)) == 0, "Not all
       redeemTokens were swapped");
```

## 4.3. Some states are not updated

Across the codebase, some states are not updated. For example, in StakingPool.sol —

- In the `unstake()` function:

```
function unstake() public NOT_AMERICAN {
    uint[] memory userIssueIndexs = userIssueIndex[msg.sender];

    uint unstakeAmount;
    for (uint i; i < userIssueIndexs.length; i++) {
        uint index = userIssueIndexs[i];
        if (index > indexStar) {
            Issue storage issueInfo = issues[index];
            if (issueInfo.isStaking) {
                unstakeAmount += issueInfo.issueAmount;
              issueInfo.issueAmount = 0;
                issueInfo.isStaking = false;
            }
        }
    }
```

- In the `swap()` function:

```
for (uint i = indexEnd; i > indexStar; i--) {
    Issue storage issueInfo = issues[i];
    if (issueInfo.isStaking) {
        if (amountB > 0) {
            if (amountB >= issueInfo.issueAmount) {
                uint amountA = (issueInfo.issueAmount * 10000) / rate;
                amountB -= issueInfo.issueAmount;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
              issueInfo.issueAmount = 0;
                issueInfo.isStaking = false;
            } else {
                uint amountA = (amountB * 10000) / rate;
                redeemToekn.safeTransfer(issueInfo.user, amountA);
                issueInfo.issueAmount -= amountB;
                amountB = 0;
                }
            }
        }
    }
```

## 4.4. Quadrata usage not comprehensive

This codebase uses the Quadrata protocol to determine whether users are KYC-ed or not. For that reason, in the `constructor` of `NotAmerica`, the following code is used to determine the address of the `reader` contract, a component of the Quadrata protocol:

```
constructor() {
    uint chainId = _chainID();
    if (chainId == 11155111)
        reader = IQuadReader(0x49CF5d391B223E9196A7f5927A44D57fec1244C8); //
    Sepolia
    if (chainId == 10)
        reader = IQuadReader(0xFEB98861425C6d2819c0d0Ee70E45AbcF71b43Da); //
    Optimistic
    if (chainId == 42161)
        reader = IQuadReader(0x49CF5d391B223E9196A7f5927A44D57fec1244C8); //
    Arbitrum One
}
```

If the `chainId` is not one of the above, the `reader` will be left to `0x0`. This means any calls to `reader` will revert. This is an especially important consideration, as calls to `reader` are made in the `modifier NOT_AMERICAN`, which is called in all of the most important functions of the contract.

Therefore, ensuring that the `reader` is set to a valid address is crucial, and the contract deployment should revert if that is not the case.

## 4.5. Additional concerning areas

As noted in the non-goals section, due to the limited time available, we were unable to comprehensively assess the potential implications of each of the possible tokens that are to be used and/or traded within the pools.

Additionally, other concerning areas were identified during the assessment, including but not limited to

- The potential for slippage-related issues in the `swap` function of StakingPool.

- The `redeem` function in VaultPool. This function assumes that the `issueToken` has been sent beforehand, otherwise it will not function properly.

Note that the `swap` function was removed in commit 849a8a63 ↗.

## 5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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. Module: Share.sol

#### Function: `mint(address[] tos, uint256[] amounts)`

Mints more `amounts` shares to the addresses given in the corresponding array `tos`. Can only be called by the owner.

#### Inputs

- `tos`
    - **Control**: Fully controllable.
    - **Constraints**: None.
    - **Impact**: The addresses to mint to.
- `amounts`
    - **Control**: Fully controllable.
    - **Constraints**: None, but must be at least equal to the length of `tos`, though this is not checked.
    - **Impact**: The amounts to mint.

#### Branches and code coverage

**Intended branches**

- Owner mints to single address.
    - [x] Test coverage
- Owner mints to multiple address.
    - [ ] Test coverage

**Negative behavior**

- Caller is not the owner.
    - [ ] Negative test
- The length of `tos` and `amounts` mismatches.
    - [ ] Negative test

### 5.2. Module: StakingPool.sol

#### Function: `stake(uint256 amount)`

Users can pledge funds in a contract to be used as proof of subscription.

#### Inputs

- `amount`
    - **Control**: Full control.
    - **Constraints**: Nonzero.
    - **Impact**: Number of tokens applied for. This amount is transferred from the caller to this pool, and an `issue` object is generated and assigned to the caller.

#### Branches and code coverage

**Intended branches**

- User is able to stake successfully.
    - [x] Test coverage
- User is able to stake successfully multiple times.
    - [x] Test coverage

**Negative behavior**

- User is not able to stake zero tokens.
    - [ ] Negative test
- American user is not able to stake.
    - [ ] Negative test
- User is not able to stake with too little funds.
    - [ ] Negative test

#### Function: `swap(uint256 amount)`

NATdoc describes this function:

> Users of historical batches can sell their RWA shares to those who want to buy them at the moment by using this function, which can be executed if there are enough funds for the current round of subscriptions and they have not been withdrawn.

The function will iterate over all active issues and do either a full or a partial swap depending on the remaining amount to pay out. If there is not enough balance of either share or issue tokens on either side, it should fail.

Vulnerable to reentrancy issues if used with tokens that offer callbacks, as the stakes are invalidated *after* being transferred.

### Inputs

- `amount`
    - **Control**: Fully controllable.
    - **Constraints**: Nonzero.
    - **Impact**: The amount of redeem tokens to trade in. Depending on the globally set rate, this decides how many share tokens that will be paid out.

### Branches and code coverage

**Intended branches**

- Swap when there is a single issue with the exact amount.
    - ☑ Test coverage
- Swap when there are multiple issues that sum up to the exact amount.
    - ☐ Test coverage
- Swap so that there is at least one partial swap.
    - ☐ Test coverage

**Negative behavior**

- Swap with too little user balance of redeem tokens.
    - ☐ Negative test
- Swap with too little contract balance of share (issue) tokens.
    - ☐ Negative test

### Function: `unstake()`

User fully withdraws all their stakes if called before the end of the collection cycle. All issue indexes that are connected to the caller are fetched from memory, and then each issue is fetched from storage. All stores that have the `isStaking` flag set are summed up, and the flag is set to `false`.

The summed-up amount is transferred to the caller and also removed from the global `pendingLiquidation` counter.

### Branches and code coverage

**Intended branches**

- User can stake and unstake.
    - ☑ Test coverage

- Unstaking multiple times has no effect.
  - ☐ Test coverage

**Negative behavior**

- American users cannot unstake.
  - ☐ Negative test
- Stakes older than `indexStar` cannot be unstaked.
  - ☐ Negative test

### Function: `withdraw()`

Extracts all the `issueToken` tokens from the contract. Only callable by the owner. All stakes are invalidated by setting the cut-off between valid and invalid stakes (`indexStar`) to the current `indexEnd`. Currently, this is off by one, as `indexEnd` is supposed to always be one more than `indexStar`.

### Branches and code coverage

**Intended branches**

- Admin is able to withdraw.
  - ☑ Test coverage
- User is able to stake and unstake before a withdraw but not unstake after.
  - ☐ Test coverage
- User is able to stake and unstake new issues after a withdraw.
  - ☐ Test coverage

**Negative behavior**

- Non-admin cannot withdraw.
  - ☐ Negative test
- User cannot withdraw old stakes.
  - ☐ Negative test

## 5.3. Module: USDT.sol

### Function: `mint(address to, uint256 amount)`

This is a mock version of the ERC-20 token USDT only used internally for testing. This function allows for the minting of USDT tokens, which increases the total supply.

## 5.4. Module: VaultPool.sol

### Function: `reedem(uint256 amount)`

Redeem stablecoins with RWA Assets. Calling this method will give the caller stablecoin and burn RWA tokens. Can only be called by users that are KYC-ed and are not American.

### Inputs

- `amount`
    - **Control**: Full control.
    - **Constraints**: Nonzero.
    - **Impact**: The amount of shares to redeem.

### Branches and code coverage

**Intended branches**

- Caller successfully redeems.
    - [x] Test coverage
- Include function calls.
    - [ ] Test coverage
- End sentences with periods.
    - [ ] Test coverage

**Negative behavior**

- Caller tries to redeem zero share tokens.
    - [ ] Negative test
- Caller tries to redeem during pause.
    - [ ] Negative test
- Caller tries to redeem while being American or not KYC.
    - [ ] Negative test
- Caller tries to redeem when `issueTotal` is zero.
    - [ ] Negative test
- Caller tries to redeem when amount is nonzero but the calculated `withdrawAmount` is zero.
    - [ ] Negative test

### Function call analysis

- `this.shareToekn.totalSupply()`
    - **What is controllable?** Not controllable.

    - **If the return value is controllable, how is it used and how can it go wrong?** Not directly controllable.
    - **What happens if it reverts, reenters or does other unusual control flow?** If total supply is extremely high, `shareTotal + 1` will overflow and revert.
- `this.issueToken.balanceOf(address(this))`
    - **What is controllable?** Not controllable.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable.
    - **What happens if it reverts, reenters or does other unusual control flow?** If balance is zero, `issueTotal - 1` will underflow and revert.
- `SafeERC20.safeTransferFrom(this.shareToekn, msg.sender, address(this), amount)`
    - **What is controllable?** Amount is controllable and must be nonzero.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable.
    - **What happens if it reverts, reenters or does other unusual control flow?** Should revert if user has too low of a balance.
- `this.shareToekn.burn(amount)`
    - **What is controllable?** Amount is controllable but has been transferred from the caller before it gets this far.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SafeERC20.safeTransfer(this.issueToken, msg.sender, withdrawAmount)`
    - **What is controllable?** The `withdrawAmount` is indirectly controllable but is done after burning.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

### Function: `withdraw()`

Owner withdraws the stablecoins for the next round of RWA requisitions. Empties the entire contract, sending everything to the sender and produces an `AdminWithdraw` event.

### Branches and code coverage

**Intended branches**

- Owner withdraws a pool with issue tokens.
    - ☑ Test coverage
- Owner withdraws a pool without issue tokens.
    - ☐ Test coverage

**Negative behavior**

- Caller is not an owner.
  - ☐ Negative test

## 6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Chateau contracts, we discovered six findings. Three critical issues were found. One was of medium impact and the remaining findings were informational in nature. Chateau Capital acknowledged all findings and implemented fixes.

### 6.1. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.