CD SECURITY

AUDIT REPORT

Chateau Capital
March 2024

Introduction

A time-boxed security review of the **Chateau Capital** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Chateau Capital**

Chateau Capital allows users to use ERC20 tokens and gain exposure to real world assets (RWA). For each RWA, the `Factory` creates new set of `StakingPool` and `VaultPool`. A part of the protocol is centralized and the below process is implemented:

1. Users stake an ERC20 token (usually stablecoin) which is set during the deployment of the contracts for the specific RWA.
2. Once a month the owners withdraws the balance of the `StakingPool` and issue shares to all the users who have staked based on the latest NAV price.
3. Users receive shares which can be redeemed back for ERC20 tokens in the `VaultShare`.

Important to note is that the NAV calculation and deciding how much shares each staker should receive happens off-chain and is not in-scope for this audit.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 78a3b3d6b3616d770897e2496e15e6a7930cc9bd**

Scope

The following smart contracts were in scope of the audit:

- `Share.sol`
- `Factory.sol`
- `StakingPool.sol`
- `VaultPool.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 2 issues
- Low: 2 issues
- Informational: 4 issues

Findings Summary

ID	Title	Severity
[M-01]	Iterating over unbounded array can run out of gas	Medium
[M-02]	`whenNotPaused` should not be used while redeem	Medium
[L-01]	Lack of two-step role transfer	Low
[L-02]	Check if array arguments have the same length	Low
[I-01]	Solidity safe pragma best practices are not used	Informational
[I-02]	Prefer Solidity Custom Errors over `require` statements	Informational
[I-03]	Prefer battle-tested code	Informational
[I-04]	Missing event emission in `setVault()`	Informational

Detailed Findings

[M-01] Iterating over unbounded array can run out of gas

Severity

Impact: High

Likelihood: Low

Description

If an user wants to `unstake` from the `StakingPool`, a `for-loop` will iterate over all of his stake positions to unstake the relevant indexes. However, this loop will go over all of the past position as well, and in long period of time, if the user interacts with the protocol many times, this array can grow too big and the call can exceed the block gas limit which will DoS the `StakingPool`.

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

Recommendations

Consider popping the unstaked positions from the `userIssueIndex[]`.

[M-02] `whenNotPaused` should not be used while redeem

Severity

Impact: High

Likelihood: Low

Description

Pausing functionality is a great feature in times of emergency. But you should be careful about which functionalities you want to pause and which not. The redeem function should be free of any Pausing constraints, otherwise if owner decides to never unpause the contract then user funds will get stuck in the contract.

Recommendations

Consider removing the `whenNotPaused` modifier from `redeem()`.

[L-01] Lack of two-step role transfer

All of the contracts in scope have imported the `Ownable.sol` contract forked from OZ which means they lack two-step role transfer. The ownership transfer should be done with great care and two-step role transfer should be preferable.

Use Ownable2Step by OpenZeppelin.

[L-02] Check if array arguments have the same length

In `Share::mint()` you have two array-type arguments. Validate that the arguments have the same length so you do not get unexpected errors if they don't.

[I-01] Solidity safe pragma best practices are not used

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. All of the contracts are currently using a floatable version.

[I-02] Prefer Solidity Custom Errors over `require` statements

Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well. Consider replacing the `require` statements with custom errors.

[I-03] Prefer battle-tested code

Always prefer battle-tested code over reimplementing common patterns. Instead of reimplementing your own `reentrant` modifier, use the `ReentrancyGuard` provided by OpenZeppelin, since it is well tested and optimized.

[I-04] Missing event emission in `setVault()`

Setters should emit an event so that Dapps can detect important changes to storage. Add an event emission in function `setVault()` in `Share.sol`.