Florence Finance Security Review

Pashov Audit Group

Conducted by: pashov

May 18th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I 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.

3. Introduction

A time-boxed security review of the **Florence Finance** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Florence Finance

Florence Finance is a protocol that allows users to fund real-world loans with crypto assets. It works by letting users deposit stablecoins to the protocol in return for receipt tokens ($FLR or Loan Vault Tokens). Then the protocol will provide fiat currency (euros) to real-world lending platforms, who in turn lend the fiat currency to businesses. When loans are repaid, interest and principal flow back to the protocol to be distributed back to users. The protocol does not guarantee liquidity and is similar to peer-2-peer lending.

Users of Florence Finance can use the platform in two ways:

1. Use your stablecoins in a Loan Vault to earn yield from a real-world loan
2. Stake $FLR and earn Medici Token ($MDC)

In both ways the user should have $FLR, which he can get by swapping his stablecoins on a DEX for $FLR, which is backed 1:1 to EUR by the principal amount in the real-world loans. Then he can deposit them in a Loan Vault and receive Loan Vault Tokens (the vaults are following ERC4626) with which he immediately starts earning interest and can later withdraw his $FLR, receiving the principal plus earned yield, by burning the Loan Vault Tokens. If the user chooses to earn $MDC rewards instead, he can stake his $FLR in the staking pool and then withdraw or claim rewards any time.

More docs

Observations

The protocol is integrated to work with EURS which does not follow best practices for ERC20 implementation.

The protocol's Loan Vaults are using the ERC4646 standard, which is error-prone. There is also a mechanism for the primary funding of the protocol, which requires a funder to be whitelisted (should provide KYC & AML-related documentation).

FLR is not guaranteed to be pegged 1:1 to EUR on-chain. It is backed by real-world loans.

Funding tokens in a Vault have to be whitelisted. Funders have to be whitelisted as well.

Threat Model

Privileged Roles

- Delegate - can create a new Funding Request, then manage the loan (repay, write down, write up, default)
- Primary(whitelisted) Funder - can create a Funding Attempt to request to fulfill a Funding Request in a Vault
- Fund Approver - can approve a Funding Attempt and execute it simultaneously
- Vault Owner - deployed the vault and can set all other roles and configure the vault
- Florin Treasury Owner - can pause/unpause the treasury contract and also transfer $FLR token ownership
- Loan Vault Registry Owner - can whitelist Loan Vaults in the registry

Actors

- Chainlink Price feed - provides the price of an asset on-chain
- Staker - stakes $FLR to receive $MDC rewards
- Vault depositor - deposits funding tokens as a loan and receives vault shares

Security Interview

Q: What in the protocol has value in the market?

A: The $FLR token, the `LoanVault` shares and the `EURS` balance of the treasury.

Q: In what case can the protocol/users lose money?

A: If a loan defaults or if they can't redeem their deposits/stakes.

Q: What are some ways that an attacker achieves his goals?

A: Force the exit methods to revert or exploit staking/vault shares calculation math to steal tokens from the protocol/users.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

***review commit hash* - ef3f96a7c1c5948b1da20af3e1c161c029711484**

Scope

The following smart contracts were in scope of the audit:

- `Errors`
- `FlorinStaking`
- `FlorinToken`
- `FlorinTreasury`
- `LoanVault`
- `LoanVaultRegistry`
- `Util`

7. Executive Summary

Over the course of the security review, pashov engaged with Florence Finance to review Florence Finance. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Florence Finance
Date	May 18th, 2023

Findings Count

Severity	**Amount**
Critical	1
High	1
Medium	6
Low	3
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Anyone can move EURS tokens that the user allowed FlorinTreasury to spend	Critical	Resolved
[H-01]	Stakers/vault depositors can be front-run and lose their funds	High	Resolved
[M-01]	The Chainlink price feed's input is not properly validated	Medium	Resolved
[M-02]	The ERC4626 standard is not followed correctly	Medium	Resolved
[M-03]	User exit/claim methods should not have a whenNotPaused modifier	Medium	Resolved
[M-04]	The apr and fundingFee percentage values are not constrained	Medium	Resolved
[M-05]	The protocol uses _msgSender() extensively, but not everywhere	Medium	Resolved
[M-06]	Multiple centralization attack vectors are present in the protocol	Medium	Resolved
[L-01]	If too many funding tokens are whitelisted then removal might become impossible	Low	Resolved
[L-02]	Number of registered loan vaults should be capped	Low	Resolved
[L-03]	Allowance check gives a false sense of security	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Anyone can move `EURS` tokens that the user allowed `FlorinTreasury` to spend

Severity

Impact: High, as funds will be moved from a user's wallet unwillingly

Likelihood: High, as it requires no preconditions and is a common attack vector

Description

The `depositEUR` method in `FlorinTreasury` looks like this:

```
function depositEUR(address from, uint256 eurTokens) external whenNotPaused {
    eurTokens = Util.convertDecimals(eurTokens, 18, Util.getERC20Decimals
      (eurToken));
    SafeERC20Upgradeable.safeTransferFrom(florinToken, from, address
      (this), eurTokens);
    emit DepositEUR(_msgSender(), from, eurTokens);
}
```

The problem is that the `from` argument is user controlled, so anyone can check who has allowed the `FlorinTreasury` contract to spend his tokens and then pass that address as the `from` argument of the method. This will move `eurTokens` amount of `EURS` tokens from the exploited user to the `FlorinTreasury` contract, even though the user did not do this himself. The `depositEUR` method is expected to be called by `LoanVault::repayLoan` or `LoanVault::depositRewards`, where the user should first approve the `FlorinTreasury` contract to spend his `EURS` tokens. This is especially problematic if the user set `type(uint256).max` as the allowance of the contract, because in such case all of his `EURS` balance can be drained.

Recommendations

Use `msg.sender` instead of a user-supplied `from` argument, so tokens can only be moved from the caller's account.

Discussion

pashov: Resolved.

8.2. High Findings

[H-01] Stakers/vault depositors can be front-run and lose their funds

Severity

Impact: High, as it results in a theft of user assets

Likelihood: Medium, as it works only if the attacker is the first staker

Description

Let's look at the following example:

1. The `FlorinStaking` contract has been deployed, unpaused and has 0 staked $FLR in it
2. Alice sends a transaction calling the `stake` method with `florinTokens == 10e18`
3. Bob is a malicious user/bot and sees the transaction in the mempool and front-runs it by depositing 1 wei of $FLR, receiving 1 wei of shares
4. Bob also front-runs Alice's transaction with a direct `ERC20::transfer` of 10e18 $FLR to the `FlorinStaking` contract
5. Now in Alice's transaction, the code calculates Alice's shares as `shares = florinTokens.mulDiv(totalShares_, getTotalStakedFlorinTokens(), MathUpgradeable.Rounding.Down);`, where `getTotalStakedFlorinTokens` returns `florinToken.balanceOf(address(this))`, so now `shares` rounds down to 0
6. Alice gets minted 0 shares, even though she deposited 10e18 worth of $FLR
7. Now Bob back-runs Alice's transaction with a call to `unstake` where `requestedFlorinTokens` is the contract's balance of $FLR, allowing him to burn his 1 share and withdraw his deposit + Alice's whole deposit

This can be replayed multiple times until the depositors notice the problem.

Note: This absolute same problem is present with the ERC4626 logic in `LoanVault`, as it is a common vulnerability related to vault shares calculations.

OpenZeppelin has introduced a way for mitigation in version 4.8.0 which is the used version by this protocol.

Recommendations

UniswapV2 fixed this with two types of protection:

First, on the first `mint` it actually mints the first 1000 shares to the zero-address

Second, it requires that the minted shares are not 0

Implementing them both will resolve this vulnerability.

Discussion

pashov: Resolved.

8.3. Medium Findings

[M-01] The Chainlink price feed's input is not properly validated

Severity

Impact: High, as it can result in the application working with an incorrect asset price

Likelihood: Low, as Chainlink oracles are mostly reliable, but there has been occurrences of this issue before

Description

The code in `LoanVault::getFundingTokenExchangeRate` uses a Chainlink price oracle in the following way:

```
(, int256 exchangeRate, , , ) =
    fundingTokenChainLinkFeeds[fundingToken].latestRoundData();

if (exchangeRate == 0) {
    revert Errors.ZeroExchangeRate();
}
```

This has some validation but it does not check if the answer (or price) received was actually a stale one. Reasons for a price feed to stop updating are listed here. Using a stale price in the application can result in wrong calculations in the vault shares math which can lead to an exploit from a bad actor.

Recommendations

Change the code in the following way:

```
-
- (, int256 exchangeRate, , , ) = fundingTokenChainLinkFeeds[fundingToken].latestRound
+ (, int256 exchangeRate, , uint256 updatedAt , ) =
+ fundingTokenChainLinkFeeds[fundingToken].latestRoundData();

- if (exchangeRate == 0) {
+ if (exchangeRate <= 0) {
+     revert Errors.ZeroExchangeRate();
+ }

+ if (updatedAt < block.timestamp - 60 * 60 /* 1 hour */) {
+     pause();
+ }
```

This way you will also check for negative price (as it is of type `int256`) and also for stale price. To implement the pausing mechanism I proposed some other changes will be needed as well, another option is to just revert there.

Discussion

pashov: Resolved.

[M-02] The ERC4626 standard is not followed correctly

Severity

Impact: Medium, as functionality is not working as expected but without a value loss

Likelihood: Medium, as multiple methods are not compliant with the standard

Description

As per EIP-4626, the `maxDeposit` method "MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0.". This is not the case currently, as even if the contract is paused, the `maxDeposit` method will still return what it usually does.

When it comes to the `decimals` method, the EIP says: "Although the convertTo functions should eliminate the need for any use of an EIP-4626 Vault's decimals variable, it is still strongly recommended to mirror the underlying token's decimals if at all possible, to eliminate possible sources of confusion and simplify integration across front-ends and for other off-chain

users." The `LoanVault` contract has hardcoded the value of 18 to be returned when `decimals` are called, but it should be the decimals of the underlying token (it might not be 18 in some case maybe).

Recommendations

Go through the standard and follow it for all methods that `override` methods from the inherited ERC4626 implementation.

Discussion

pashov: Resolved.

[M-03] User exit/claim methods should not have a `whenNotPaused` modifier

Severity

Impact: High, as user funds can be left stuck in the contract

Likelihood: Low, as it requires a malicious or a compromised owner

Description

The `unstake` and `claim` methods in `FlorinStaking` have a `whenNotPaused` modifier and the same is true for the `redeem` and `_withdraw` methods in `LoanVault`. This opens up an attack vector, where the protocol owner can decide if the users are able to withdraw/claim any funds from it. There is also the possibility that an admin pauses the contracts and renounces ownership, which will leave the funds stuck in the contract forever.

Recommendations

Remove the `whenNotPaused` modifier from user exit/claim methods in the protocol or reconsider the `Pausable` integration in the protocol altogether.

Discussion

pashov: Acknowledged.

[M-04] The `apr` and `fundingFee` percentage values are not constrained

Severity

Impact: High, as this can result in the contract being in a state of DoS or in 0 rewards for users

Likelihood: Low, as it requires a malicious or a compromised owner, or a big mistake on the owner side

Description

Neither the `setApr` nor the `setFundingFee` methods have input validations, checking if the percentage value arguments are too big or too small. A malicious/compromised owner, or one that does a "fat-finger", can input a huge number as those methods' argument, which will result in a state of DoS for the contract. Also the values of 0 or 100 (percentage) are valid as well, but shouldn't be - they will result in either 0 rewards for users or high fees (100% fees are not possible because of the slippage check in `approveFundingAttempt`).

Recommendations

Add a min and max value checks in both the `setApr` and `setFundingFee` methods in `LoanVault`.

Discussion

pashov: Resolved.

[M-05] The protocol uses `_msgSender()` extensively, but not everywhere

Severity

Impact: Low, because protocol will still function normally, but an expectedly desired types of transactions won't work

Likelihood: High, because it is certain that he issue will occur as code is

Description

The code is using OpenZeppelin's `Context` contract which is intended to allow meta-transactions. It works by using doing a call to `_msgSender()` instead of querying `msg.sender` directly, because the method allows those special transactions. The problem is that the `onlyDelegate` and `onlyFundApprover` modifiers in `LoanVault` use `msg.sender` directly instead of `_msgSender()`, which breaks this intent and will not allow meta-transactions at all in the methods that have those modifiers, which are one of the important ones in the `LoanVault` contract.

Recommendations

Change the code in the `onlyDelegate` and `onlyFundApprover` modifiers to use `_msgSender()` instead of `msg.sender`.

Discussion

pashov: Resolved.

[M-06] Multiple centralization attack vectors are present in the protocol

Severity

Impact: High, as it can result in a rug from the protocol owner

Likelihood: Low, as it requires a compromised or a malicious owner

Description

The protocol owner has privileges to control the funds in the protocol or the flow of them.

The `mint` function in `FlorinToken` is callable by the contract owner, which is `FlorinTreasury`, but `FloriNTreasury` has the `transferFlorinTokenOwnership` method. This makes it possible that the `FlorinTreasury` deployer to mint as many `FlorinToken` tokens to himself as he wants, on demand.

The `withdraw` method in `FlorinStaking` works so that the owner can move all of the staked `florinToken` tokens to any wallet, including his.

The `setMDCperFLRperSecond` method in `FlorinStaking` works so that the owner can stop the rewards at any time or unintentionally distribute them in an instant.

The method `setFundingTokenChainLinkFeed` allows the owner to set any address as the new Chainlink feed, so he can use an address that he controls and returns different prices based on rules he decided.

Recommendations

Consider removing some owner privileges or put them behind a Timelock contract or governance.

Discussion

pashov: Acknowledged.

8.4. Low Findings

[L-01] If too many funding tokens are whitelisted then removal might become impossible

The `setFundingToken` method in `LoanVault` allows the owner to whitelist as many funding tokens as he wants, pushing them to an unbounded array. The problem is that if he whitelists too many tokens, then the array will grow too big and removing a value (for example the last one) from the array might cost too much gas, even more than the block gas limit, resulting in impossibility of removing some funding tokens from the whitelist. To fix this you should put an upper limit on the `_fundingTokens` array size, for example 50.

Discussion

pashov: Resolved.

[L-02] Number of registered loan vaults should be capped

The owner of the `LoanVaultRegistry` might call `registerLoanVault` too many times, which will make the `loanVaultIds` array very big. Calling `getLoanVaultIds` on-chain might result in a revert if the gas required for the call is too much (more than the block gas limit for example). It is recommended that you limit the number of loan vaults that can be registered, for example a maximum of 200.

Discussion

pashov: Resolved.

[L-03] Allowance check gives a false sense of security

The `LoanVault::createFundingAttempt` method contains the following allowance check:

```
if (fundingToken.allowance(_msgSender(), address
  (this)) < fillableFundingTokenAmount) {
    revert Errors.InsufficientAllowance();
}
```

But it does not do a `transferFrom` for the `fundingToken` by itself. This means that the call to the method can be back-ran with an allowance revoke transaction. This will invalidate the check and later revert when the funding attempt is executed, so you are better off removing the check.

Discussion

pashov: Acknowledged.