33AUDITS

Libree Audit Report

Introduction

A time-boxed security review of the LibreeV3 protocol was done by 33Audit LLC, focusing on the security aspects of the smart contracts.

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 vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits LLC

33Audits LLC is an independent smart contract security researcher and development group. Having conducted over 15 audits with dozens of vulnerabilities found we are experienced in building and auditing smart contract. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check his previous work here or reach out on X @solidityauditor.

About LibreeV3

V3 removes the Superfluid dependencies and introduces the ability to stake in order to subscribe via Aave v3 pools.

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 from a successful attack

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

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

***review commit hash* - 8d58eec1d7cf319ce9cd3c29fa6148ecd66b43a7**

No fixes implemented.

Scope

The following smart contracts were in the scope of the audit:

- `contracts/token/NFTToken.sol`
- `contracts/test/UserEscrowExtended.sol`
- `contracts/test/TestToken.sol`
- `contracts/config/LibreeConfig.sol`
- `contracts/payment/UserEscrow.sol`
- `contracts/payment/UserEscrowFactory.sol`
- `contracts/subscription/FactoringManager.sol`
- `contracts/subscription/base/ManagerUpgradable.sol`
- `contracts/subscription/base/Manager.sol`
- `contracts/registry/Registry.sol`
- `contracts/lib/Errors.sol`
- `contracts/lib/NFTActions.sol`
- `contracts/lib/Constants.sol`
- `contracts/lib/Types.sol`
- `contracts/lib/Staking.sol`
- `contracts/upgradability/BaseUpgradable.sol`
- `contracts/hub/HubAccessControl.sol`
- `contracts/hub/LibreeHub.sol`
- `contracts/vault/VaultFactory.sol`
- `contracts/vault/VaultFactoryYield.sol`
- `contracts/vault/VaultYield.sol`
- `contracts/vault/Vault.sol`
- `contracts/interfaces/IUserEscrowFactory.sol`
- `contracts/interfaces/IERC20Decimals.sol`
- `contracts/interfaces/IRegistry.sol`
- `contracts/interfaces/IManager.sol`
- `contracts/interfaces/IUserEscrow.sol`

Findings Summary

ID	Title	Severity	Status
[H-01]	`unSubscribe` function can be front-run by attacker to grief a user from unsubscribing	High	-
[H-02]	Attacker can drain a users escrow by calling `paySubscription` multiple times	High	
[M-01]	Malicious attacker can cancel a users subscription before it is actually over	Medium	-
[M-02]	A user cannot call `unSubscribe` after subscription period expires	Medium	-
[L-01]	Events emit an event that says the `msg.sender` unsubscribed which is incorrect	Low	-

Detailed Findings

[H-01] - `unSubscribe` function can be front-run by attacker to grief a user from unsubscribing

Context

- UserEscrow.sol

Description

A user subscribes to a subscription. Right afterwards they decide they no longer want to subscribe so they call `unSubscribe`.

```
    function unsubscribe(uint256 _subscriptionId) external onlyUser {
        if (!isSubscribed(_subscriptionId)) {
            revert Errors.SubscriptionNotActive();
        }

        activeSubscriptions[_subscriptionId] = false;

        emit Unsubscribed(msg.sender, _subscriptionId);
    }
```

This function will revert if `isSubscribed` returns false as the ! operator will invert the falsy to true and trigger the error `SubscriptionNotActive`. In this case the user can decided they'll just wait until their subscription is over and unsubscribe then.

Let's say 30 days has passed and they decided to now 'unsubscribe'. Due the the `paySubscription` function being public, an attacker could just call `paySubscription` before the user is able to unsubscribe.

As long as theres tokens in the escrow the subscription would get paid and `lastTimeStamp` would update adding another 30 days before the user is able to `unsubscribe`. The user would then have to wait another 30 days (in the case of our example) to try and unsubscribe because `isSubscribed` checks `lastPayment + subscription.paymentFrecuency > block.timestamp`. The attacker could just call the function again front-running the user and add another 30 days to the `lastPayment`. Continuously griefing them until they either they withdraw funds from the escrow or run out of funds in the contract.

Recommendation

Ideally `paySubscription` should only be callable by the user but since this feature is needed for the protocol to work there's a few things your can do to try to mitigate against this attack vector.

First is verifying that the payment is due, this will stop the attacker from being able to call `paySubscription` before a user can unsubscribe though a small risk still exists that a hacker could watch the mempool for `unsubscribe` call and front-run this transaction by paying more gas than the user and restarting the subscription for another 30 days.

The best solution is since `unSubscribe` is inverting the boolean value returned by `isSubscribed`, then you should instead check if `isActiveSubscription = true`.

Resolution

[H-02] - Attacker can drain a users escrow by calling `paySubscription` multiple times

Context

- UserEscrow.sol

Description

`paySubscription` is callable by anyone which poses a problem considering it doesn't check when the last payment was due. A user can subscribe to a subscription and an attacker can then call `paySubscription` as many times as they want draining the users escrow completely, and if done correctly they can do it right after a user has just subscribed making `lastSubscriptionPayment[_subscriptionId]` the same value that the user set when they first subscribed. This is because the following function does not check that a subscription payment is due.

```
    function paySubscription(uint256 _subscriptionId) external {
        if (isSubscribed(_subscriptionId)) {
            revert Errors.SubscriptionIsActive();
        }

        if (lastSubscriptionPayment[_subscriptionId] == 0) {
            revert Errors.SubscriptionNotActive();
        }

        Types.Subscription memory subscription = IRegistry(
            config.getRegistryAddress()
        ).getSubscription(_subscriptionId);

        uint256 userBalance = IERC20(subscription.paymentToken).balanceOf(
            address(this)
        );

        if (userBalance < subscription.paymentAmount) {
            activeSubscriptions[_subscriptionId] = false;
            emit Unsubscribed(msg.sender, _subscriptionId);
            return;
        }

        IERC20(subscription.paymentToken).safeTransfer(
            subscription.vault,
            subscription.paymentAmount
        );

        lastSubscriptionPayment[_subscriptionId] = block.timestamp;
        emit SubscriptionPaid(msg.sender, _subscriptionId);
    }
```

Consider the follow example.

User subscribes for a subscription. They now have a service for 30 days, and have paid 10 USDC for the subscription. They still have 100 USDC in their escrow in case they want to pay for other subscriptions.

An attack can maliciously call the `paySubscription` function 10 times and drain the users escrow while their ongoing subscription is still active. This is because the following modifier checks `lastPayment + subscription.paymentFrecuency > block.timestamp` which would return false if only 1 day has passed and `lastSubscriptionPayment[_subscriptionId] == 0` would return false as well. This would not cause any of the checks to revert and would allow the escrow to be drained.

```
if (isSubscribed(_subscriptionId)) {
            revert Errors.SubscriptionIsActive();
        }
```

Recommendation

`paySubscription` should be checking if the `paymentDue == true` if so it will continue to execute. In combination with the fix above it should also check that the subscription is active as to not allow an attacker to pay for subscription that is inactive and grief the user.

Resolution

[M-01] - Malicious attacker can cancel a users subscription before it is actually over

Context

- UserEscrow.sol

Description

Consider the following example. A user can create a subscription with 100 USDC tokens for a month. Lets say there's a user that wants to subscribe to this and has 100USDC in their escrow. They pay for the first month and then afterwards don't add funds to their escrow. An attacker can then call `paySubscription` and cause a users `activeSubscription` boolean to flip to false prematurely.

```
    uint256 userBalance = IERC20(subscription.paymentToken).balanceOf(
            address(this)
        );

        if (userBalance < subscription.paymentAmount) {
            activeSubscriptions[_subscriptionId] = false;
            emit Unsubscribed(msg.sender, _subscriptionId);
            return;
        }
```

The the impact on users funds is negligent unless this variable is used on the front end for verification purposed. Of course it may be used at some point in the future to actually verify if a subscription is valid at which point this attack vector could be exploited at risk to the user.

Recommendation

Ideally `paySubscription` is only callable by the user themselves or if needed to be called by Libree this address is set in the user escrow and a modifier checks that either the user or Libree is the caller of the function. Since we discussed this already in the chat it may not be needed to actually fix however I found it important to note here that the issue still exists.

Resolution

[M-02] - A user cannot call `unSubscribe` after a subscription period expires

Context

- UserEscrow.sol

Description

Consider the following example. A user subscribes for a subscription and pays for the first "month". They then choose to unsubscribe after 31 days because they don't like the service. However if they call `unSubscribe` it would revert because `lastPayment + subscription.paymentFrecuency > block.timestamp` would return false and due to the code below a user would not be able to `unsubscribe` post their subscription period.

In decimals we would get the following.

lastPayment + frequency > block.timestamp 1 + 30 > 32 = false

```
function unsubscribe(uint256 _subscriptionId) external onlyUser {
        if (!isSubscribed(_subscriptionId)) {
            revert Errors.SubscriptionNotActive();
        }

        activeSubscriptions[_subscriptionId] = false;

        emit Unsubscribed(msg.sender, _subscriptionId);
    }
```

In this case a user would have to submit another payment for a subscription in order to unsubscribe.

Recommendation

`unSubscribe` should just check if a subscription is active. If it is then it should flip the bool to falsy.

Resolution

[L-01] - Events emit an event that says the `msg.sender` unsubscribed which is incorrect

Context

- UserEscrow.sol

Description

The events in `paySubscription` log `msg.sender` as the address that has unsubscribed and the address who paid the subscription when this should actually be the user/owner of the escrow.

Recommendation

Change these to be the user of the escrow.

Resolution