Overload Security Review

Pashov Audit Group

Conducted by: dirk_y, Alex Murphy, ast3ros

May 7th 2024 - May 9rd 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our 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 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.

3. Introduction

A time-boxed security review of the **overload-labs/farm** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Overload

`Farm.sol` is a rollout farming contract, a minimal wrapper for ERC-20 tokens, where token addresses are converted into uint256 and accounted for by ERC-6909. The `Router.sol` is a periphery contract and can be updated and re-deployed.

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 - 937907e02072601bf38bfafdfe152f507c036dcd

fixes review commit hash - 457af3301e9d226df7ebaea13ab3ea9cc6fbd6d0

Scope

The following smart contracts were in scope of the audit:

- `Farm`
- `Router`
- `ERC6909`
- `Lock`
- `Payment`
- `TokenId`
- `TransferHelper`

7. Executive Summary

Over the course of the security review, dirk_y, Alex Murphy, ast3ros engaged with Overload to review Overload. In this period of time a total of **4** issues were uncovered.

Protocol Summary

Protocol Name	Overload
Repository	https://github.com/overload-labs/farm
Date	May 7th 2024 - May 9rd 2024
Protocol Type	Yield farming

Findings Count

Severity	**Amount**
Medium	4
Total Findings	**4**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Non-standard tokens not properly accounted for	Medium	Resolved
[M-02]	Issues with unsafe ERC20 operations	Medium	Resolved
[M-03]	Excess ETH from deposits	Medium	Resolved
[M-04]	The withdraw method in Router.sol should not be payable	Medium	Resolved

8. Findings

8.1. Medium Findings

[M-01] Non-standard tokens not properly accounted for

Severity

Impact: High

Likelihood: Low

Description

When depositing into the farm, the shares minted to the depositor are equal to the `amount` specified. However, for fee-on-transfer tokens, the actual amount of tokens that are deposited into the contract will be less than the amount specified in the transfer (since the fee is yet to be taken). As such the depositor receives more shares than they should.

If the protocol wants to support fee-on-transfer tokens then the balance of the transferred token should be checked before and after the `transferFrom` call, and the corresponding number of shares minted (`amountAfter - amountBefore`).

Rebasing tokens dynamically adjust user balances through increases or decreases over time. Within the Farm contract, when users deposit rebasing tokens and receive an equivalent amount of ERC6909 tokens, an issue arises due to the mutable nature of rebasing token balances. If a user's balance of rebasing tokens changes prior to withdrawing, it creates a mismatch between the minted ERC6909 tokens and the actual token quantity available in the contract.

Another exotic example is tokens with maximum transfer handling: Some tokens, like Compound V3's cUSDCV3, interpret `type(uint256).max` as an instruction to transfer the sender's entire balance, not the literal maximum

`uint256` value. This behavior can be exploited to mint more `shares` than the number of tokens actually received by the contract, draining funds.

Recommendations

- Documentation: Clearly document this behavior within the contract's code to avoid potential misuse or misunderstandings by users interacting with tokens that have these special transfer characteristics.

[M-02] Issues with unsafe ERC20 operations

Severity

Impact: High

Likelihood: Low

Description

The farm contract currently utilizes non-safe versions of ERC20 interactions, which could pose a significant risk when interacting with tokens that do not strictly adhere to the ERC20 standard. Specifically, some tokens may return false instead of reverting the transaction when a transfer fails, as observed with tokens like EURS.

This issue could be exploited by malicious users who, despite not having sufficient funds, could initiate a deposit that falsely appears successful, leading to incorrectly minted shares. Subsequently, these users could burn these shares to withdraw actual funds from the contract, effectively draining assets from honest users.

But only using "safe" methods is not enough.

The `safeTransferFrom` and `safeTransfer` functions, as implemented in the `TransferHelper` library, perform low-level calls to the token address and check for successful execution. However, these functions currently fail to verify whether the token address is an actual contract. This oversight could lead to severe security vulnerabilities, allowing an attacker to exploit the system by interacting with a non-contract address. If a token address lacks

executable code, the low-level call will not revert, potentially enabling unauthorized token acquisition or draining of funds.

```
function safeTransferFrom(
        address token,
        address from,
        address to,
        uint256 value
    ) internal {
        (bool success, bytes memory data) =
            token.call(abi.encodeWithSelector
              (IERC20.transferFrom.selector, from, to, value));
        require(success && (data.length == 0 || abi.decode(data,
        //(bool))), "STF"); // @audit do not check if target is a contract
    }

    /// @notice Transfers tokens from msg.sender to a recipient
    /// @dev Errors with ST if transfer fails
    /// @param token The contract address of the token which will be transferred
    /// @param to The recipient of the transfer
    /// @param value The value of the transfer
    function safeTransfer(
        address token,
        address to,
        uint256 value
    ) internal {
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector
          (IERC20.transfer.selector, to, value));
        require(success && (data.length == 0 || abi.decode(data,
        //(bool))), "ST"); // @audit do not check if target is a contract
    }
```

Let's consider the following scenario:

- The Farm contract opts to support USDT, so it replaces the `transferFrom` function with the `safeTransferFrom` function from the `TransferHelper` library.
- An attacker predicts and uses a non-existent token address (by front-running token deployment transactions or exploiting CREATE2 predictability or using the same token address in the different chain if the token is in multichain) to interact with the deposit function.
- This interaction fails to revert due to the non-contract nature of the address, allowing the attacker to erroneously receive ERC6909 tokens.
- Legitimate users subsequently `deposit` the actual tokens into the Farm contract, which the attacker can then drain using the previously acquired ERC6909 tokens.

POC:

- Alice predicts a future token address and uses it to interact with the deposit function.
- Alice receives ERC6909 tokens as if she had deposited real tokens.
- Once the token is legitimately deployed and Bob deposits into the same contract, Alice uses her tokens to drain the funds, leaving Bob unable to withdraw.

Setup:

- Create a file: test/ExploitSafeTransferFrom.t.sol
- Execute tests: forge test -vvvvv --match-path test/ExploitSafeTransferFrom.t.sol --match-test testPOC

Test: link

Recommendations

1. implement `safeTransferFrom`, `safeTransfer`, and `forceApprove` functions.
2. implement an additional verification step to confirm that the token address is indeed a contract. This can be achieved by adopting a method similar to the OpenZeppelin's approach in their `safeTransfer` and `safeTransferFrom` implementations:

```
function verifyCallResultFromTarget(
        address target,
        bool success,
        bytes memory returndata
    ) internal view returns (bytes memory) {
        if (!success) {
            _revert(returndata);
        } else {
            // only check if target is a contract if the call was successful and
            // the return data is empty
            // otherwise we already know that it was a contract
            if
            //(returndata.length == 0 && target.code.length == 0) { // @audit verify t
                revert AddressEmptyCode(target);
            }
            return returndata;
        }
    }
```

link

[M-03] Excess ETH from deposits

Severity

Impact: High

Likelihood: Low

Description

A user can deposit WETH into the farm by calling `deposit` in `Router.sol` with `msg.value > 0` and `token = WETH`; the ether provided with the call is wrapped to WETH. The amount that is wrapped and deposited in the farm is controlled by the user input `amount`.

However, there is no validation that the ether provided by the user is actually equal to the `amount` argument that is used to control the deposit. As a result, it is possible for a user to provide more ether than `amount`, yet only `amount` of WETH will be deposited into the farm.

The excess ETH can be stolen by the next caller by calling `deposit` with `amount = [excess ETH]`. The caller uses the excess ETH provided by the previous caller and deposits it into the farm under their address.

Similarly, a user can call this method with `msg.value > 0` when intending to deposit a different token and their provided ETH will be happily accepted and can be stolen by the next user.

Recommendations

If the `token` argument to the `Router.sol:deposit` function is WETH, then validate that `amount == msg.value`. Otherwise, enforce that `msg.value == 0`.

[M-04] The `withdraw` method in `Router.sol` should not be payable

Severity

Impact: High

Likelihood: Low

Description

The `withdraw` method in `Router.sol` is payable which means callers can provide ETH with their function call. However, there is no reason for this method to be payable, since any ETH provided will simply be gladly accepted by the contract and not returned to the user.

Other users can steal any accidentally provided ETH by calling `deposit` with `amount == [excess ETH] && token == WETH`.

Recommendations

Remove the `payable` modifier from `withdraw`.