MiloTruck

StakeWise V3

Security Review Report

August, 2023

Table of Contents

Introduction

About MiloTruck

MiloTruck is an independent security researcher who specializes in smart contract audits. Having won multiple audit contests, he is currently one of the top wardens on Code4rena.

For security consulting, reach out to him on Twitter - *@milotruck*

Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

Executive Summary

This review was completed as part of an audit contest on Hats Finance.

About StakeWise

StakeWise is a liquid Ethereum 2.0 staking service that allows anyone to benefit from the yields available on the Beacon Chain.

This codebase consists of contracts for the StakeWise V3 upgrade, which allows any node operator or DApp to launch and manage their own liquid staking solution through custom vaults.

Repository Details

Repository	https://github.com/stakewise/v3-core
Commit Hash	c82fc57d013a19967576f683c5e41900cbdd0e67
Language	Solidity

Issues Found

Severity	Count
High	0
Medium	1
Low	3
Non-Critical	0

Scope

contracts/base/
- ERC20.sol
- ERC20Upgradeable.sol
- Multicall.sol

contracts/keeper/
- Keeper.sol
- KeeperOracles.sol
- KeeperRewards.sol
- KeeperValidators.sol

contracts/libraries/
- Errors.sol
- ExitQueue.sol

contracts/osToken/
- OsToken.sol
- OsTokenConfig.sol
- PriceFeed.sol

contracts/vaults/
- ethereum/
  - EthErc20Vault.sol
  - EthGenesisVault.sol
  - EthPrivErc20Vault.sol
  - EthPrivVault.sol
  - EthVault.sol
  - EthVaultFactory.sol
  - mev/
    - OwnMevEscrow.sol
    - SharedMevEscrow.sol
- modules/
  - VaultAdmin.sol
  - VaultEnterExit.sol
  - VaultEthStaking.sol
  - VaultFee.sol
  - VaultImmutables.sol
  - VaultMev.sol
  - VaultOsToken.sol
  - VaultState.sol
  - VaultToken.sol
  - VaultValidators.sol
  - VaultVersion.sol
  - VaultWhitelist.sol
- VaultsRegistry.sol

Findings

Summary

ID	Description	Severity
M-01	Attacker can leverage flash loans to steal rewards from vaults	Medium
L-01	Users can `migrate()` before the first harvest to gain more shares	Low
L-02	`enterExitQueue()` might be uncallable if the vault experiences a huge loss	Low
L-03	EIP-712 typehash is incorrect in `KeeperRewards.sol` and `KeeperValidators.sol`	Low

Medium Severity Findings

M-01: Attacker can leverage flash loans to steal rewards from vaults

Bug Description

In `KeeperRewards.sol`, the `isHarvestRequired()` function is used to determine if a vault needs to call `harvest()`:

KeeperRewards.sol#L138-L146

```
function isHarvestRequired(address vault) external view override returns (bool) {
  // vault is considered harvested in case it does not have any validators (nonce = 0)
  // or it is up to 1 rewards update behind
  uint256 nonce = rewards[vault].nonce;
  unchecked {
    // cannot overflow as nonce is uint64
    return nonce != 0 && nonce + 1 < rewardsNonce;
  }
}
```

As seen from above, if a vault only has one pending update, `isHarvestRequired()` returns `false`, as vaults are allowed to be one update behind the latest one.

This becomes an issue as `isHarvestRequired()` is used to check if a vault needs to be harvested before a user makes a deposit:

VaultEnterExit.sol#L151-L156

```
function _deposit(
  address to,
  uint256 assets,
  address referrer
) internal virtual returns (uint256 shares) {
  _checkHarvested();
```

VaultImmutables.sol#L41-L43

```
function _checkHarvested() internal view {
  if (IKeeperRewards(_keeper).isHarvestRequired(address(this))) revert Errors.NotHarvested();
}
```

As users are allowed to deposit when the latest update has not been harvested, an attacker can abuse flash loans to deposit a huge amount of assets and steal the profits from the latest update.

Attack Scenario

- `updateRewards()` is called to update rewards for all vaults.
- An attacker sees that a certain vault has made a huge profit.
- To steal most of the profit from that vault, he does the following:
  - Borrow 1000 ether using a flash loan.
  - Call `deposit()` to deposit 1000 ether as assets into the vault.
  - Call `updateState()`, which adds profits from the latest update to the vault's assets.
  - Call `redeem()` to withdraw all his shares for ETH.
  - Repay his flashloan.
- Due to the attacker's large deposit, he held most of the shares in the vault when `updateState()` was called.
- Therefore, after `updateState()` is called, most of the vault's profits from the latest rewards update is accrued to his shares, allowing him to withdraw more than 1000 ether when calling `redeem()`.

In the scenario above, the attacker has managed to gain most of the vault's profits at no risk, which is a theft of yield from other stakers.

Impact

As users can make deposits while the vault is one rewards update behind, an attacker can leverage flash loans to steal most of the yield from the latest rewards update. This can be done without any risk as attackers can call `redeem()` in the same transaction as `deposit()`.

Recommended Mitigation

Consider only allowing deposits and other functions to be called when all pending updates have been harvested. This can be achieved by making `isHarvestRequired()` return `true` when the vault is 1 rewards update behind:

KeeperRewards.sol#L138-L146

```
  function isHarvestRequired(address vault) external view override returns (bool) {
    // vault is considered harvested in case it does not have any validators (nonce = 0)
    // or it is up to 1 rewards update behind
    uint256 nonce = rewards[vault].nonce;
    unchecked {
      // cannot overflow as nonce is uint64
-     return nonce != 0 && nonce + 1 < rewardsNonce;
+     return nonce != 0 && nonce < rewardsNonce;
    }
  }
```

Team Response

Fixed in commit `bee6eef` and commit `66fd138`.

Low Severity Findings

L-01: Users can `migrate()` before the first harvest to gain more shares

Bug Description

In `EthGenesisVault.sol`, on the first harvest, the total rewards accumulated in the legacy pool is deducted:

EthGenesisVault.sol#L107-L110

```
    if (!isCollateralized) {
      // it's the first harvest, deduct rewards accumulated so far in legacy pool
      totalAssetsDelta -= SafeCast.toInt256(_rewardEthToken.totalRewards());
    }
```

Since almost all assets will still be in the legacy pool, most of the deduction penalty will be passed on to V2's `RewardETHToken` contract by calling `updateTotalRewards()`:

EthGenesisVault.sol#L115-L125

```
    // calculate total principal
    uint256 totalPrincipal = _totalAssets + legacyPrincipal;
    if (totalAssetsDelta < 0) {
      // calculate and update penalty for legacy pool
      int256 legacyPenalty = SafeCast.toInt256(
        Math.mulDiv(uint256(-totalAssetsDelta), legacyPrincipal, totalPrincipal)
      );
      _rewardEthToken.updateTotalRewards(-legacyPenalty);
      // deduct penalty from total assets delta
      totalAssetsDelta += legacyPenalty;
    } else {
```

In the `RewardEthToken` contract, the penalty will be added to `totalPenalty`:

RewardEthToken.sol#L246-L251

```
        } else if (rewardsDelta < 0) {
            uint256 _totalPenalty = totalPenalty; // gas savings
            _totalPenalty = _totalPenalty.add(uint256(- rewardsDelta));
            require(_totalPenalty <= totalAssets(), "RewardEthToken: invalid penalty amount");
            totalPenalty = _totalPenalty;
        }
```

Therefore, after the first harvest, users from V2 that are migrating to V3 will receive shares based on only their assets in `StakedEthToken`, and will not receive shares for accrued rewards, due to `totalPenalty`:

RewardEthToken.sol#L323-L330

```
        uint256 _totalPenalty = totalPenalty; // gas savings
        if (_totalPenalty > 0) {
            uint256 _totalAssets = totalAssets(); // gas savings
            // apply penalty to assets
            uint256 assetsAfterPenalty = assets.mul(_totalAssets.sub(_totalPenalty)).div(_totalAssets);
            totalPenalty = _totalPenalty.add(assetsAfterPenalty).sub(assets);
            assets = assetsAfterPenalty;
        }
```

However, there is no check in `migrate()` that ensures the vault is collateralized, which means users can call `migrate()` before `updateState()` has been called once.

Therefore, users can migrate from V2 to V3 before the first harvest, which means `totalPenalty` would not have been initialized yet. As such, both their assets and rewards in V2 will be migrated, allowing them to unfairly gain more shares in the Genesis Vault compared to users who migrate after the first harvest.

Impact

Users can migrate from V2 before the first harvest to unfairly gain more shares in the Genesis Vault, as compared to users who migrate after the first harvest is made.

This results in a direct loss of funds for users who migrate after the first harvest, as they will gain less shares for migrating the same amount of assets and rewards from V2.

Recommended Mitigation

Consider allowing `migrate()` to only be called after the Genesis Vault has been collateralized:

EthGenesisVault.sol#L146-L149

```
  function migrate(address receiver, uint256 assets) external override returns (uint256 shares) {
    if (msg.sender != address(_rewardEthToken)) revert Errors.AccessDenied();

    _checkHarvested();
+   _checkCollateralized();
```

Team Response

Fixed in commit 1e181e3.

L-02: `enterExitQueue()` might be uncallable if the vault experiences a huge loss

Bug Description

In `VaultEnterExit.sol`, the `enterExitQueue()` is called by users to enter the exit queue. This adds their shares to `queuedShares`:

VaultEnterExit.sol#L77-L80

```
    unchecked {
      // cannot overflow as it is capped with _totalShares
      queuedShares = SafeCast.toUint96(_queuedShares + shares);
    }
```

Where:

- `shares` is the number of shares the caller wishes to withdraw.

As seen from above, `shares` is downcast to a `uint96` since `queuedShares` has the type `uint96`.

This could potentially be a problem if the vault's shares to assets ratio is extremely high, as users will need to specify a large amount of shares to withdraw a substantial amount of assets. For example:

Assume the following:

- A vault currently has a 1 to 1 shares to assets ratio, which means that:
  - `totalShares = 32e18`
  - `totalAssets = 32 ether`
- The vault stakes its 32 ETH to register a validator.
- After a period of time, the validator is slashed with an extremely large penalty, causing it to lose most of its stake:
  - We assume that most of the validator's balance is slashed, leaving only `1e8` ETH remaining.
  - As a result, `totalAssets` is reduced to `1e8` for the vault.
- Now, if a user calls `deposit()` to deposit 1 ETH, he will get `32e28` shares:

```
shares = depositAmount * totalShares / totalAssets = 1e18 * 32e18 / 1e8 = 32e28
```

- After some time, the user wants to withdraw his ETH. As such, he calls `enterExitQueue()` with `shares = 32e28`.
- However, as `32e28` is larger than `type(uint96).max`, the downcast shown above will revert.

In the scenario above, the `enterExitQueue()` function will always revert for any substantial amount of assets, making it permanently DOSed.

Impact

As `queuedShares` is stored as a `uint96`, `enterExitQueue()` could potentially be permanently DOSed if the vault's share to assets ratio is extremely high.

Such a scenario is possible if a vault only has one validator, and nearly all of its effective balance is slashed. This could occur due to the correlation penalty, as seen here:

> *The maximum slash is the full effective balance of all slashed validators (i.e. if there are lots of validators being slashed they could lose their entire stake).*

Recommended Mitigation

Consider storing `queuedShares` in a `uint128` instead:

ExitQueue.sol#L20-L23

```
  struct Checkpoint {
    uint160 totalTickets;
-   uint96 exitedAssets;
+   uint128 exitedAssets;
  }
```

Team Response

Fixed in commit `0693bea`.

L-03: EIP-712 typehash is incorrect in `KeeperRewards.sol` and `KeeperValidators.sol`

Bug Description

In `KeeperRewards.sol`, `updateRewards()` verifies signatures according to the EIP-712 standard:

KeeperRewards.sol#L92-L106

```
    // verify rewards update signatures
    _verifySignatures(
      rewardsMinOracles,
      keccak256(
        abi.encode(
          _rewardsUpdateTypeHash,
          params.rewardsRoot,
          keccak256(bytes(params.rewardsIpfsHash)),
          params.avgRewardPerSecond,
          params.updateTimestamp,
          nonce
        )
      ),
      params.signatures
    );
```

`params.rewardsIpfsHash` is a `string` in the `RewardsUpdateParams` struct:

IKeeperRewards.sol#L79-L85

```
  struct RewardsUpdateParams {
    bytes32 rewardsRoot;
    uint256 avgRewardPerSecond;
    uint64 updateTimestamp;
    string rewardsIpfsHash;
    bytes signatures;
  }
```

However, `_rewardsUpdateTypeHash`, which is the function's EIP-712 typehash, incorrectly declares `rewardsIpfsHash` as `bytes32` instead:

KeeperRewards.sol#L19-L22

```
  bytes32 private constant _rewardsUpdateTypeHash =
    keccak256(
      'KeeperRewards(bytes32 rewardsRoot,bytes32 rewardsIpfsHash,uint256
avgRewardPerSecond,uint64 updateTimestamp,uint64 nonce)'
    );
```

Similarly, in `KeeperValidators.sol`, `approveValidators()` uses EIP-712 to verify signatures as well:

KeeperValidators.sol#L56-L69

```
    // verify oracles approved registration
    _verifySignatures(
      validatorsMinOracles,
      keccak256(
        abi.encode(
          _registerValidatorsTypeHash,
          params.validatorsRegistryRoot,
          msg.sender,
          keccak256(params.validators),
          keccak256(bytes(params.exitSignaturesIpfsHash))
        )
      ),
      params.signatures
    );
```

As seen below, in the `ApprovalParams` struct, `params.validators` is declared as `bytes` and `params.exitSignaturesIpfsHash` is a `string`:

IKeeperValidators.sol#L56-L61

```
  struct ApprovalParams {
    bytes32 validatorsRegistryRoot;
    bytes validators;
    bytes signatures;
    string exitSignaturesIpfsHash;
  }
```

However, in `_registerValidatorsTypeHash`, `validators` and `exitSignaturesIpfsHash` are incorrectly declared as bytes32:

KeeperValidators.sol#L17-L20

```
  bytes32 private constant _registerValidatorsTypeHash =
    keccak256(
      'KeeperValidators(bytes32 validatorsRegistryRoot,address vault,bytes32
validators,bytes32 exitSignaturesIpfsHash)'
    );
```

Impact

Due to the use of incorrect typehashes, the signature verification in the functions listed above is not EIP-712 compliant.

Contracts or dapps/backends that use "correct" typehashes with correct types for the parameters of these functions will end up generating different signatures, causing them to revert when called.

Recommended Mitigation

Amend the typehashes shown above to have matching parameters with their respective functions:

`_rewardsUpdateTypeHash`:

```
  bytes32 private constant _rewardsUpdateTypeHash =
    keccak256(
      'KeeperRewards(bytes32 rewardsRoot,string rewardsIpfsHash,uint256
avgRewardPerSecond,uint64 updateTimestamp,uint64 nonce)'
    );
```

`_registerValidatorsTypeHash`:

```
  bytes32 private constant _registerValidatorsTypeHash =
    keccak256(
      'KeeperValidators(bytes32 validatorsRegistryRoot,address vault,bytes
validators,string exitSignaturesIpfsHash)'
    );
```

Team Response

Fixed in commit 7de5a68.