shieldify

Possum Labs

SECURITY REVIEW

Date: 4 November 2023

CONTENTS

1. About Shieldify

We are Shieldify Security – a company on a mission to make web3 protocols more secure, cost-efficient and user-friendly. Our team boasts extensive experience in the web3 space as both smart contract auditors and developers that have worked on top 100 blockchain projects with multi-million dollars in market capitalization.

Book a security review and learn more about us at shieldify.org or @ShieldifySec

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Possum Labs

Possum Labs is a new DeFi primitive that aims to create a vast ecosystem of self-regulating financial products. This is achieved by allowing users to deposit yield-bearing assets and receive their yield up-front. Essentially, depositors can get months' worth of yield immediately after they deposit. Each depositor retains the flexibility to withdraw deposits at any time by paying back part of the upfront yield.

The beating heart of Possum is the Portal smart contract. When a specific Portal is ready to launch, it'll undergo an initial funding phase where funders can deposit PSM tokens up to a certain limit. In return, they're given receipt tokens based on a pre-determined reward rate. These receipt tokens, called **`bTokens`**, can later be redeemed for **`PSM`** – Possum's native token. As the funding pool grows, depositors will accumulate rights to more deposited **`PSM`** tokens. The application is expected to launch on Arbitrum Mainnet.

Take a deep dive into Possum's documentation here.

3.1 Observations

1. The Portal receives a share of the proceeds from the arbitrage mechanism responsible for replenishing the PSM within the internal liquidity pool. Every time there is an influx of PSM, the reward pool for **`bToken`** holders grows.
2. Participation in the funding of new **`Portals`** is open to all.
3. When a **`Portal`** contract is deployed, it is **`inactive`**, indicating that the funding phase is in progress. During this funding phase, anyone holding PSM tokens can invoke a function to contribute PSM to the **`Portal`** and, in return, receive **`bTokens`**. The quantity of **`bTokens`** received is contingent upon the reward rate, which is determined at the time of deployment.
4. Once the funding phase concludes, the Portal transitions into an **`active`** state, signifying that the creation of additional **`bTokens`** and the acceptance of further funding is no longer possible. At this point, the standard **`Portal`** functions, such as staking **`HLP`**, become operational.
5. Staked **`HLP`** generates **`Portal Energy`** (PE) for the user which can be swapped to **`PSM`** via the internal, zero-fee liquidity pool of the **`Portal`**.
6. Funding of the reward pool concludes either when the entire supply of **`bTokens`** becomes zero through burning, or when the final repayment has been executed.

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted two and a half weeks and a total of 544 hours have been spent by four smart contract security researchers from the Shieldify team.

The code is well-written, especially considering Possum's team size. Shieldify's audit report contributed by identifying several Critical and Medium severity issues, residing around improper transfer mechanism, missing validation checks, lack of a deadline check and potential hindrance when minting `bToken` and `portalEnergy`, among other Low findings.

It is important to note that the review commit hash did not include a test suite of any kind, which hindered the audit process to a certain extent. Nevertheless, Possum's fast responsiveness to all of Shieldify's questions, together with the exceptional detail of the provided answers largely mitigated this.

5.1 Protocol Summary

Project Name	**Possum Labs**
Repository	Portals
Type of Project	DeFi, Stake yield-bearing assets and receive instant upfront yield
Audit Timeline	17 days
Review Commit Hash	03b6a8272ac9ff8afe76c8748aa16f970ba1a338
Fixes Review Commit Hash	d58972bac8f0ef205ea877c0a4e4ad7ad68e80c7

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/Portal.sol	448
contracts/MintBurnToken.sol	10
contracts/interfaces/IStaking.sol	3
contracts/interfaces/ICompounder.sol	3
contracts/interfaces/IRewarder.sol	3
Total	**467**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **1**
- **Medium** issues: **3**
- **Low** issues: **4**

ID	Title	Severity
[C-01]	Users Can Not Withdraw Their Deposits Using `forceUnstakeAll()` Function	Critical
[M-01]	Missing Deadline Checks Allow Pending Transactions To Be Maliciously Executed For `convert()`, `buyPortalEnergy()` and `sellPortalEnergy()` Functions	Medium
[M-02]	Insufficient Input Validation	Medium
[M-03]	The `bToken` and `portalEnergy` Tokens Cannot be Minted if the `Portal.sol` Contract Owner is Not Set	Meidum
[L-01]	Access Control Implementation Logic Flaw After Funds Unstake	Low
[L-02]	Redundant `_updateAccount()` Function Used in Multiple Functions	Low
[L-03]	Multiple Methods Does Not Follow The `Checks-Effects-Interactions` Pattern	Low
[L-04]	Hardcoded Addresses	Low

7. Findings

[C-01] Users Can Not Withdraw Their Deposits Using `forceUnstakeAll()` Function

Severity

Critical Risk

Description

The function **`forceUnstakeAll()`** is meant to send the whole stake balance to the user but instead, it attempts to transfer the funds from the user's account to the Portal contract. In the best case, this means that the user cannot withdraw their stake via the **`forceUnstakeAll()`** method and the transaction will fail in most cases.

In the worst case, however, the user can first set an allowance for the Portal that equals or exceeds their staked position and subsequently call **`forceUnstakeAll()`**. If the user has a sufficient amount in their wallet, the function will then transfer additional funds, corresponding to the staked amount, from the user's wallet to the Portal contract and, at the same time, reduce the **`totalPrincipalStaked`** by the staked amount so that **`totalPrincipalStaked`** will no longer reflect the correct value of the total principal staked in the portal.

This sequence of actions not only poses a risk of unintentional loss of funds for regular users but also provides a direct path for a malicious actor to manipulate the internal accounting of the portal.

Location of Affected Code

File: contracts/Portal.sol#L298

```
function forceUnstakeAll() external nonReentrant {;
    .
    .
    .
    /// @dev Send the user´s staked balance to the user
    IERC20(principalToken).safeTransferFrom(
        msg.sender,
        address(this),
        balance
    );

    /// @dev Update the global tracker of staked principal
    totalPrincipalStaked -= balance;

    /// @dev Emit an event with the updated stake information
    emit StakePositionUpdated(
        msg.sender,
        accounts[msg.sender].lastUpdateTime,
        accounts[msg.sender].stakedBalance,
        accounts[msg.sender].maxStakeDebt,
        accounts[msg.sender].portalEnergy,
        accounts[msg.sender].availableToWithdraw
    );
}
```

Recommendations

`safeTransfer()` should be used instead of `safeTransferFrom()` as follows:

```
- IERC20(principalToken).safeTransferFrom(msg.sender, address(this),
   balance);
+ IERC20(principalToken).safeTransfer(msg.sender, balance);
```

Team Response

Fixed.

[M-01] Missing Deadline Checks Allow Pending Transactions To Be Maliciously Executed For `convert()`, `buyPortalEnergy()` and `sellPortalEnergy()` Functions

Severity

Medium Risk

Description

The `Portal.sol` contract does not allow users to submit a deadline for `convert()` action. This missing feature enables pending transactions to be maliciously executed at a later point.

The following scenario can happen:

1. Alice wants to convert `1,000,000` PSM tokens for `100` X tokens. She signs the transaction calling `Portal.convert()` with `_token = X token address` and `_minReceived = 99 X tokens` to allow for some slippage.
2. The transaction is submitted to the Mempool, however, Alice chose a transaction fee that is too low for miners to be interested in including her transaction in a block. The transaction stays pending in the Mempool for extended periods, which could be hours, days, weeks, or even longer.
3. When the average gas fee drops far enough for Alice's transaction to become interesting again for miners to include it, her conversation will be executed. In the meantime, the price of `X token` could have drastically changed. She will still at least get `99 X tokens` due to `_minReceived`, but the `X token` value of that output might be significantly lower. She has unknowingly performed a bad conversion due to the pending transaction she forgot about.

Location of Affected Code

File: contracts/Portal.sol

```
function convert(
    address _token,
    uint256 _minReceived
) external nonReentrant {

function buyPortalEnergy(
    uint256 _amountInput,
    uint256 _minReceived
) external nonReentrant {

function sellPortalEnergy(
    uint256 _amountInput,
    uint256 _minReceived
) external nonReentrant {
```

Recommendations

Introduce a **deadline** parameter to the mentioned functions.

```
function X(
+   uint256 deadline
) external nonReentrant {
+   if (deadline < block.timestamp) revert DeadlineExpired();
    .
    .
    .
}
```

Team Response

Fixed.

[M-02] Insufficient Input Validation

Severity

Medium Risk

Description

1. The following state variables are not validated in any regard: **fundingPhaseDuration**, **fundingExchangeRatio**, **fundingRewardRate**, **terminalMaxLockDuration** and **amountToConvert** in the constructor at initializing phase. This can lead to disfunction of the Portal due to the fact that all of these variables are **immutable** so they can not be changed after deployment time.

Example scenarios are:

- If the deployer accidentally sets **amountToConvert** to zero. When the Portal has accrued some yield balance any user will be able to buy the yield with PSM tokens literally for free.
- If the deployer accidentally sets **fundingRewardRate** to zero, zero bTokens will be minted to the user when depositing PSM to provide an initial upfront yield.

- If `fundingExchangeRatio` is zero, then `constantProduct` will also be zero because of the internal calculations and `buyPortalEnergy()` function will most probably revert depending on the user-supplied `_minReceived` parameter or in the worst case scenario the portalEnergy of the user will remain the same after selling some PSM tokens.
- If `fundingRewardRate` rate is set to a large value, a massive amount of bTokens will be minted to the user which can later be burned to receive PSM and the protocol can be entirely drained.
- `fundingPhaseDuration` is an important value that determines when can the Portal be activated. The input value in the constructor should validate that it is within certain boundaries, for example, between several hours and 1 week (or any period that the developers agree on). Otherwise, the funding phase might be super short, not allowing anyone to fund the pool (e.g. 10 seconds), or it might last an exorbitant amount of time (e.g. 5 years).

2. Missing zero value check for the input amount in `stake()`, `buyPortalEnergy()`, `sellPortalEnergy()`, `mintPortalEnergyToken()`, `burnPortalEnergyToken()` and `convert()` functions.

All of these functions lack zero value validation on important parameters. Setting invalid parameters in the best case will result in the waste of gas for their execution with zero amount as an input parameter. In the worst case in `convert()` function, for example, it can result in a large loss of funds for the user.

3. Missing zero address validation checks in `convert()` function and in the constructor.

Despite the fact that it is expected to revert due to other validation checks, it is still the best practice to add zero address checks for all address input parameters in these functions. Nevertheless, it is helpful to add zero address validation checks to be consistent and ensure high availability of the protocol with resistance to accidental misconfigurations.

Location of Affected Code

File: contracts/Portal.sol#L27

```
constructor(
    uint256 _fundingPhaseDuration,
    uint256 _fundingExchangeRatio,
    uint256 _fundingRewardRate,
    address _principalToken,
    address _bToken,
    address _portalEnergyToken,
    address _tokenToAcquire,
    uint256 _terminalMaxLockDuration,
    uint256 _amountToConvert
) {
    fundingPhaseDuration = _fundingPhaseDuration;
    fundingExchangeRatio = _fundingExchangeRatio;
    fundingRewardRate = _fundingRewardRate;
    principalToken = _principalToken;
    bToken = _bToken;
    portalEnergyToken = _portalEnergyToken;
    tokenToAcquire = _tokenToAcquire;
    terminalMaxLockDuration = _terminalMaxLockDuration;
    amountToConvert = _amountToConvert;
    creationTime = block.timestamp;
}
```

File: contracts/Portal.sol

```
function stake(uint256 _amount) external nonReentrant {
function buyPortalEnergy(uint256 _amountInput, uint256 _minReceived)
    external nonReentrant {
function sellPortalEnergy(uint256 _amountInput, uint256 _minReceived)
    external nonReentrant {
function mintPortalEnergyToken(address _recipient, uint256 _amount)
    external nonReentrant {
function burnPortalEnergyToken(address _recipient, uint256 _amount)
    external nonReentrant {
function convert(address _token, uint256 _minReceived) external
    nonReentrant {
```

Recommendations

Implement the corresponding validation checks and revert with custom errors if they are not met.

Team Response

Fixed.

[M-03] The `bToken` and `portalEnergy` Tokens Cannot be Minted if the `Portal.sol` Contract Owner is Not Set

Severity

Medium Risk

Description

Both **`bToken`** and **`portalEnergy`** will use the basic **`MintBurnToken.sol`** contract. They inherit the basic **`Ownable`** contract from OpenZeppelin, which makes the deployer of the contracts their owner by default. Since the **`contributeFunding()`** and **`mintPortalEnergyToken()`** functions call **`mint()`** on the corresponding tokens, it will not be possible for the tokens to be minted, as only their owner (deployer) could mint them.

Location of Affected Code

File: contracts/Portal.sol

```
/// @dev Mint bTokens to the user
MintBurnToken(bToken).mint(msg.sender, mintableAmount);

/// @dev Mint portal energy tokens to the recipient's wallet
MintBurnToken(portalEnergyToken).mint(_recipient, _amount);
```

Recommendations

Make the **`Portal.sol`** contract an owner (deployer) of the **`bToken`** and **`portalEnergy`** tokens, so that it could successfully call their **`mint()`** functions. This can be achieved by deploying the contracts in the

`Portal.sol`'s constructor. Additionally, consider changing the name of the `MintBurnToken` contract to a more appropriate one, to avoid confusion.

```
+ MintBurnToken bToken;
+ MintBurnToken portalEnergyToken;

constructor(
    uint256 _fundingPhaseDuration,
    uint256 _fundingExchangeRatio,
    uint256 _fundingRewardRate,
    address _principalToken,
    address _tokenToAcquire,
    uint256 _terminalMaxLockDuration,
    uint256 _amountToConvert
) {
+   bToken = new MintBurnToken();
+   portalEnergyToken = new MintBurnToken();
    .
    .
    .
}
```

Team Response

Fixed.

[L-01] Access Control Implementation Logic Flaw After Funds Unstake

Severity

Low Risk

Description

The functions `unstake()` and `forceUnstake()` are just subtracting the user's balance and the check for the account existence in the depositors mapping continues to pass but actually, the user may have withdrawn his whole stake. In addition, some of the functions insist the user be a staker so they call it but a user with zero staked amount must not be able to call them when their whole deposit is withdrawn.

Therefore, the transaction is expected to revert from the user's balance check but it is a better design approach to add an appropriate access modifier and disable users that have withdrawn their entire deposits to call functions which require the user to have a staked amount.

Location of Affected Code

File: contracts/Portal.sol

```
require(accounts[msg.sender].isExist == true,"User has no stake");
function unstake(uint256 _amount) external nonReentrant {
function forceUnstakeAll() external nonReentrant {
```

Recommendations

It is advisable to delete the user from the accounts mapping since he has withdrawn his whole balance.

For example, functions `buyPortalEnergy()`, `sellPortalEnergy()` have a requirement that the user must exist in the accounts mapping but if he has zero staked amount he should not be able to call this function.

Team Response

Acknowledged.

[L-02] Redundant `_updateAccount()` Function Used in Multiple Functions

Severity

Low Risk

Description

In `stake()`, `unstake()`, `forceUnstakeAll()`, `buyPortalEnergy()`, `sellPortalEnergy()` functions `_updateAccount()` is being called. It updates the whole data of the user but for most of the functions it is only needed some of the user data to be updated not everything as these functions handle the updating of what's needed. The following makes `_updateAccount()` unnecessary in most of the cases because this makes the gas paid for execution a lot more expensive than needed and also makes the code dusty.

For example:

- In `unstake()` and `forceUnstakeAll()` what we only need from `_updateAccount()` to update is `lastUpdateTime` and `portalEnergyEarned` before taking some actions in the `unstake()` and `forceUnstakeAll()`, but instead it calculates `stakedBalance`, `maxStakeDebt` without being needed, which leads to unnecessary computation. Both functions take care of these three parameters, as it is calculating them in final the stage of the functions.

Location of Affected Code

File: contracts/Portal.sol#L139

```
function _updateAccount(address _user, uint256 _amount) private {
```

Recommendations

Consider implementing separate functions for the account data update, since it is currently called in two different ways with the actual amount in `stake()` and with zero amount in all other functions.

Team Response

Acknowledged.

[L-03] Multiple Methods Does Not Follow The `Checks-Effects-Interactions` Pattern

Severity

Low Risk

Description

The `stake()`, `forceUnstakeAll()`, `buyPortalEnergy()` and `burnPortalEnergyToken()` functions do not follow the `Checks-Effects-Interactions (CEI)` pattern. It is recommended to always first change the state before doing external calls - while the code is not vulnerable right now due to the fact that all of these functions have `nonReentrant` modifier and therefore no potential financial loss. However, it is still a best practice to be followed mainly because it is possible that the code changes with time and new functionalities might be added when the protocol is deployed on new chains.

Location of Affected Code

File: contracts/Portal.sol

```
function stake(uint256 _amount) external nonReentrant {
function forceUnstakeAll() external nonReentrant {
function buyPortalEnergy(uint256 _amountInput, uint256 _minReceived)
    external nonReentrant {
function burnPortalEnergyToken(address _recipient, uint256 _amount)
    external nonReentrant {
```

Recommendations

Consider following the CEI(Checks-Effects-Interactions) pattern in `stake()`, `forceUnstakeAll()`, `buyPortalEnergy()` and `burnPortalEnergyToken()` functions.

Team Response

Acknowledged and mostly fixed.

[L-04] Hardcoded Addresses

Severity

Low Risk

Description

The existence of hardcoded addresses could lead to detrimental consequences for the protocol if the smart contracts, deployed at these addresses are exploited in some way.

Location of Affected Code

File: contracts/Portal.sol#L67-L77

```
address payable private constant compounderAddress = payable (0
    x8E5D083BA7A46f13afccC27BFB7da372E9dFEF22);
address payable private constant HLPstakingAddress = payable (0
    xbE8f8AF5953869222eA8D39F1Be9d03766010B1C);
address private constant HLPprotocolRewarder = 0
    x665099B3e59367f02E5f9e039C3450E31c338788;
address private constant HLPemissionsRewarder = 0
    x6D2c18B559C5343CB0703bB55AADB5f22152cC32;
address private constant HMXstakingAddress = 0
    x92E586B8D4Bf59f4001604209A292621c716539a;
address private constant HMXprotocolRewarder = 0
    xB698829C4C187C85859AD2085B24f308fC1195D3;
address private constant HMXemissionsRewarder = 0
    x94c22459b145F012F1c6791F2D729F7a22c44764;
address private constant HMXdragonPointsRewarder = 0
    xbEDd351c62111FB7216683C2A26319743a06F273;
```

Recommendations

Consider adding a setter function for all the hardcoded addresses, callable only by the owner of the `Portal.sol` contract.

Team Response

Acknowledged.

shieldify

Thank you!