shieldify

H4SHFund

SECURITY REVIEW

Date: 8 March 2025

CONTENTS

1. **About Shieldify** 3
2. **Disclaimer** 3
3. **About H4SHFund** 3
4. **Risk classification** 3
   - 4.1 Impact 3
   - 4.2 Likelihood 4
5. **Security Review Summary** 4
   - 5.1 Protocol Summary 4
   - 5.2 Scope 4
6. **Findings Summary** 5
7. **Findings** 5

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

Learn more about us at shieldify.org.

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 H4SHFund

H4SHFund is Avalanche's #1 Memecoin printer, the best place to create and launch tokens on the red-chain.

- Simple and easy to use: Launch a token in a few clicks in <10 seconds and for just a few cents
- Fully customizable: H4SHFund allows you to customize your launch parameters however you wish
- Integrated vesting: Add token vesting to presales to prevent dumping

Learn more here.

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 6 days with a total of 144 hours dedicated by 3 researchers from the Shieldify team.

Overall, the code is well-written. The audit report identified five Medium-severity issues, and four Low-severity issues. The vulnerabilities primarily stem from a potential denial of service through drop creation, an incorrect DAO treasury owner setting, two malicious maximum limit bypasses, and a discrepancy with the documentation.

The H4SHFund team has been highly responsive to the Shieldify research team's inquiries and promptly implemented all recommendations.

5.1 Protocol Summary

Project Name	**H4SHFund**
Repository	leap-contracts
Type of Project	Tokens launchpad
Audit Timeline	6 days
Review Commit Hash	1ba23a2cd5c4c0d5a406c9876e4dd713e03b3f64
Fixes Review Commit Hash	d8b59cef10c3512fa82b8a0196d257d42218ea1a

5.2 Scope

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

File	nSLOC
src/abstracts/LeapBase.sol	281
src/types/DataTypes.sol	37
src/Leap.sol	128
src/LeapDaoManager.sol	58
src/LeapERC20.sol	11
src/LeapFactory.sol	88
Total	**603**

6. Findings Summary

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

- **Medium** issues: **5**
- **Low** issues: **4**

ID	Title	Severity	Status
[M-01]	The `createDrop()` Function Is Vulnerable to DoS	Medium	Fixed
[M-02]	The Owner of `daoTreasury` Is Set Incorrectly	Medium	Fixed
[M-03]	The Attacker Can Prevent The Deposit From Reaching `maxRaiseAmount`	Medium	Fixed
[M-04]	Users Can Prevent Maximum Deposit Cap from Being Reached	Medium	Fixed
[M-05]	Improper Initialization of `LaunchToken` Can Cause Campaign Finalization Failure	Medium	Fixed
[L-01]	Max Dao Manager Length in The Code Does Not Match The Definition in The Documentation	Low	Fixed
[L-02]	The `issuerTimelock` Setting Is Not Used	Low	Fixed
[L-03]	Unnecessary Check Inside `removeDaoManager()`	Low	Fixed
[L-04]	Single-step Ownership Transfer Pattern Can Be Dangerous	Low	Fixed

7. Findings

[M-01] The `createDrop()` Function Is Vulnerable to DoS

Severity

Medium Risk

Description

In `LeapFactory.createDrop()`, the salt used for creating `Leap` is directly taken from the input parameters. This allows an attacker to extract the salt from the mempool and front-run the `createDrop()`, causing the victim's `createDrop()` to revert when checking if the `predictedAddress` already exists.

```
function createDrop(
    LEAPDataTypes.LaunchToken memory _launchToken,
    LEAPDataTypes.PhaseTimeStamps memory _phaseTimeStamps,
    LEAPDataTypes.LaunchParams memory _launchParams,
    bytes32 deploymentSalt
) external returns (address) {
    address predictedAddress = Clones.predictDeterministicAddress(
        tokenImplementation,
        deploymentSalt,
        address(this)
    );
    if (predictedAddress.code.length > 0) revert DropAlreadyExists();

    /// @dev note that all other validation is done in the LeapBase
       contract for the initialize function

    ILeap newLeap = ILeap(
        //@audit-issue salt without msg.sender
        Clones.cloneDeterministic(tokenImplementation, deploymentSalt)
    );

    newLeap.initialize(
        _launchToken,
        _phaseTimeStamps,
        _launchParams,
        msg.sender
    );

    emit DropCreated(address(newLeap), msg.sender, deploymentSalt);
    return address(newLeap);
}
```

Similar vulnerability: https://github.com/code-423n4/2023-04-caviar-findings/issues/419

Location of Affected Code

File: src/LeapFactory.sol#L123-L151

Impact

The attacker can prevent normal users from executing `createDrop()`.

Recommendation

Add `msg.sender` to `deploymentSalt`, for example:

```
Clones.cloneDeterministic(tokenImplementation, keccak256(abi.encode(msg.
    sender, deploymentSalt)));
```

Team Response

Fixed.

[M-02] The Owner of `daoTreasury` Is Set Incorrectly

Severity

Medium Risk

Description

In `LeapBase.initialize()`, if `daoTreasury` is not set, a `LeapDaoManager` contract will be automatically created, and the `owner` will be set to `msg.sender`.

```
/// Deploy a new treasury contract if one was not provided
/// @dev this is required for AI powered DAOs in future
launchParams.daoTreasury = launchParams.daoTreasury == address(0)
    ? address(new LeapDaoManager(msg.sender)) //@audit-issue should be
       _deployer
    : launchParams.daoTreasury;
```

The issue here is that `LeapBase.initialize()` is called by `LeapFactory`, meaning `LeapDaoManager`'s owner is set to `LeapFactory`. However, `LeapFactory` does not and should not have the functionality to manage the DAO. The correct `owner` should be the `deployer`.

Location of Affected Code

File: src/abstracts/LeapBase.sol#L163-L165

Impact

The `owner` of `LeapDaoManager` is incorrectly set to `LeapFactory` instead of the `deployer`.

Recommendation

Consider passing the `deployer` address:

```
- address(new LeapDaoManager(msg.sender))
+ address(new LeapDaoManager(_deployer))
```

Team Response

Fixed.

[M-03] The Attacker Can Prevent The Deposit From Reaching `maxRaiseAmount`

Severity

Medium Risk

Description

In `Leap.depositEth()`, if the current deposit exceeds `maxRaiseAmount`, it will revert:

```
//uncapped if set to 0 initially.
if (launchParams.maxRaiseAmount != 0) {
    if (totalRaised + _value > launchParams.maxRaiseAmount)
        //cannot raise more than the cap.
        revert Errors.ERR_MaximumDepositReached();
}
```

During the `PHASE_DEPOSIT`, deposit and withdraw do not require fees and lock time:

```
//users can make a deposit and withdraw at any time with 0% fee
PHASE_DEPOSIT,
```

Therefore, the attacker can first deposit up to `maxRaiseAmount` and then withdraw, effectively blocking normal users from depositing at a low cost.

Here's an example:
1. Assume `totalRaised=90`, `maxAllocation=10`, and `maxRaiseAmount=100`.
2. Alice wants to `deposit 10`, which should normally succeed.
3. Bob monitors Alice's request, deposits `10` before her deposit, and withdraws `10` after her deposit.
4. The final transaction execution order is:
- Bob `deposits 10`: success
- Alice `deposits 10`: revert, cannot raise more than the cap.
- Bob `withdraw 10`: success
5. In the end, Bob prevents Alice's deposit at a very low cost (gas + create bundle), thereby preventing the total raised amount from reaching `maxRaiseAmount`.
6. Although each user is limited by `maxAllocation`, the attacker can use multiple accounts to carry out the attack simultaneously. The more accounts used, the larger the deposit amount that can be blocked.

Location of Affected Code

File: src/Leap.sol#L101

Impact

The attacker can prevent the deposit from reaching `maxRaiseAmount`, the more accounts used by the attacker, the larger the deposit amount that can be blocked.

Recommendation

Add a lock-up period during `PHASE_DEPOSIT` to increase the attacker's capital cost.

Team Response

Fixed, by also allowing deposits in `PHASE_ONE`.

[M-04] Users Can Prevent Maximum Deposit Cap from Being Reached

Severity

Medium Risk

Description

The vulnerability arises in the `depositEth()` function within the contract, specifically when dealing with the `launchParams.maxRaiseAmount` and `launchParams.minAllocation` parameters. The contract imposes a minimum allocation requirement for deposits, ensuring that each deposit is at least a certain amount (`minAllocation`) and that the total amount raised does not exceed the maximum cap (`maxRaiseAmount`).

However, a logical flaw exists in the way the contract handles situations where the total raised amount (`totalRaised`) is close to the maximum raise amount (`maxRaiseAmount`), but the remaining capacity for deposits is less than the minimum deposit amount (`minAllocation`). In this case, users cannot deposit any amount that is lower than the `minAllocation`, even if the remaining capacity is sufficient to accept smaller deposits. This prevents the contract from fully reaching its maximum raise amount and effectively "locks" the remaining space, causing it to remain unused.

For example, if `launchParams.maxRaiseAmount` is set to 100 and `launchParams.minAllocation` is set to 10, and the current `totalRaised` is 95, any deposit attempt below 10 (e.g., a deposit of 5) would be rejected, even though the remaining capacity is exactly 5. As a result, the contract would never reach its maximum raise amount of 100, even though there is still available capacity for a smaller deposit.

Location of Affected Code

File: src/Leap.sol

```
function depositEth()
    external
    payable
    atPhase(LEAPDataTypes.Phase.PHASE_DEPOSIT)
    nonReentrant
{
    uint256 _value = msg.value;

    //if the whitelist is enabled and the user is not whitelisted, revert
    if (launchParams.isWhitelistEnabled && !whitelist[msg.sender])
        revert Errors.ERR_WhitelistRequired(msg.sender);

    //uncapped if set to 0 initially.
    // @audit min amount of tokens would be remaining to deposit
    if (launchParams.maxRaiseAmount != 0) {
        if (totalRaised + _value > launchParams.maxRaiseAmount)
            //cannot raise more than the cap.
            revert Errors.ERR_MaximumDepositReached();
    }

    //perform required checks to make sure deposit is valid
    if (_value == 0) revert Errors.ERR_NoEthDeposited();
    if (_value < launchParams.minAllocation)
        revert Errors.ERR_RequiresMinimumAmount();
    if (_value > launchParams.maxAllocation)
        revert Errors.ERR_MaxAllocationAmount();

    //update the total raised amount and mint the receipt tokens
    totalRaised += _value;
    _mint(msg.sender, _value);

    //emit an event to show that the eth has been deposited
    emit EthDeposited(msg.sender, _value);
}
```

Impact

This vulnerability has the potential to lead to a denial of service (DoS) situation, where the remaining deposit capacity cannot be used effectively. As a result, the contract will not reach its intended fundraising goal.

Recommendation

To resolve this issue, the `depositEth()` function should be modified to allow deposits that are smaller than the `minAllocation` when the remaining capacity in the contract is less than the `minAllocation`.

```
function depositEth()
  external
  payable
  atPhase(LEAPDataTypes.Phase.PHASE_DEPOSIT)
  nonReentrant
{
  uint256 _value = msg.value;

  // if the whitelist is enabled and the user is not whitelisted, revert
  if (launchParams.isWhitelistEnabled && !whitelist[msg.sender])
      revert Errors.ERR_WhitelistRequired(msg.sender);

  // uncapped if set to 0 initially
  if (launchParams.maxRaiseAmount != 0) {
      if (totalRaised + _value > launchParams.maxRaiseAmount)
          revert Errors.ERR_MaximumDepositReached();
  }

  // Perform required checks to make sure deposit is valid
  if (_value == 0) revert Errors.ERR_NoEthDeposited();
- if (_value < launchParams.minAllocation)
-         revert Errors.ERR_RequiresMinimumAmount();
+ if (_value < launchParams.minAllocation) {
+      uint256 remainingCapacity = launchParams.maxRaiseAmount -
    totalRaised;

+     // Allow deposits that are smaller than minAllocation if they fit in
     the remaining capacity
+       if (_value < launchParams.minAllocation && _value !=
    remainingCapacity) {
+           revert Errors.ERR_RequiresMinimumAmount();
+       }
+ }
}
```

Team Response

Fixed.

[M-05] Improper Initialization of `LaunchToken` Can Cause Campaign Finalization Failure

Severity

Medium Risk

Description

The vulnerability lies in the `initialize()` function, where there is no validation of the `tokenAddress` inside the `LaunchToken` struct. During the initialization, the contract does

not ensure that the `tokenAddress` is set to `address(0)`, which is necessary for the proper deployment and setup of the launch token. This issue arises because, if a non-zero address is provided for the `tokenAddress` during initialization, the contract will not check or prevent this scenario. Consequently, during the execution of the `finalizeCampaign()` function (which is responsible for finalizing the fund-raising process), a function like `_createTokenAndMintSupply()` could fail due to the presence of a pre-set non-zero `tokenAddress`. This results in a revert during the campaign finalization process.

```
function _createTokenAndMintSupply(
    address _treasury,
    bytes32 tokenSalt
) internal {
    //check that the token has not already been created
    // @audit this check should be there during initialization or it
       would revert here
    require(
        launchToken.tokenAddress == address(0),
        "!Token already created"
    );
    // code
}
```

This also prevents users from withdrawing their deposited ETH due to launch event being in `PHASE_THREE` and completing the campaign, potentially freezing the funds and creating a situation where only the owner can intervene to withdraw the funds and distribute them manually.

Location of Affected Code

File: src/abstracts/LeapBase.sol

```
function _createTokenAndMintSupply(
    address _treasury,
    bytes32 tokenSalt
) internal {
    //check that the token has not already been created
    // @audit this check should be there during initialization or it
       would revert here
    require(
        launchToken.tokenAddress == address(0),
        "!Token already created"
    );
    // code
}
```

Impact

The potential impact of this vulnerability is severe, as it blocks the normal operation of the fund-raising campaign and prevents users from withdrawing their ETH. Since the contract fails during the finalization of the campaign due to the invalid tokenAddress, all users who have deposited ETH will be stuck in the contract without the ability to reclaim their funds. Moreover, this flaw also places a heavy

burden on the contract owner or administrator, who would need to manually intervene, withdraw the funds, and redistribute them to users.

Recommendation

To address this issue, the initialization function should be updated to validate that the `tokenAddress` in the `LaunchToken` struct is set to `address(0)` during initialization.

Team Response

Fixed.

[L-01] Max Dao Manager Length in The Code Does Not Match The Definition in The Documentation

Severity

Low Risk

Description

In `LeapDaoManager.addDaoManager()`, the required length should be less than or equal to five, but the `push` operation is after the `require` statement, which means the maximum length can be 5+1=6. This is inconsistent with the comment that states - `Max 5 managers`.

```
function addDaoManager(address manager) external onlyDaoManager {
    require(!isDaoManager(manager), "Already a manager");
    // @audit-issue incorrect check
    require(daoManagers.length <= 5, "Max 5 managers");
    daoManagers.push(manager);
}
```

Location of Affected Code

File: src/LeapDaoManager.sol#L89-L95

Impact

Max dao manager length in the code does not match the definition in the documentation.

Recommendation

Consider changing the check as follows:

```
- require(daoManagers.length <= 5, "Max 5 managers");
+ require(daoManagers.length < 5, "Max 5 managers");
```

Team Response

Fixed.

[L-02] The `issuerTimelock` Setting Is Not Used

Severity

Low Risk

Description

The `issuerTimelock` in `LaunchParams` is not used in other code.

```
struct LaunchParams {
    ...
    /// @notice Timelock duration post phase x when issuer can withdraw
       their LP tokens
    uint256 issuerTimelock; // Slot 7
    ...
}
```

```
# pwd leap_contracts-main/src
grep -ri 'issuerTimelock' .
./types/DataTypes.sol:        uint256 issuerTimelock; // Slot 7
```

In the `currentPhase()` function, the lockup functionality is implemented using `phaseThreeEnd`, and the `issuerTimelock` parameter is not utilized.

```
if (currentTime < phaseTimeStamps.phaseThreeEnd) {
    return LEAPDataTypes.Phase.PHASE_THREE;
}

return LEAPDataTypes.Phase.DEPLETED;
```

Location of Affected Code

File: src/types/DataTypes.sol#L83

Impact

The `issuerTimelock` in `LaunchParams` is not used in other code.

Recommendation

Remove this parameter or add the corresponding functionality.

Team Response

Fixed.

[L-03] Unnecessary Check Inside `removeDaoManager()`

Severity

Low Risk

Description

The `daoManagers.length > 1`, check in `removeDaoManager()` is unnecessary. The intent behind this check is to prevent a situation where the DAO has no managers left. However, this check is redundant.

It has already prevented a manager from removing themselves, ensuring that they cannot remove themselves from the list of DAO managers. Therefore, even if there is only one DAO manager remaining, the manager in question would not be able to remove themselves because of this check. The require(`daoManagers.length > 1`) condition effectively becomes unnecessary because it cannot be satisfied if there is only one manager, and the self-removal restriction (`require(manager != msg.sender)`) will prevent the last manager from removing themselves.

Location of Affected Code

File: src/LeapDaoManager.sol

```
function removeDaoManager(address manager) external onlyDaoManager {
      require(isDaoManager(manager), "Not a manager");
      require(manager != msg.sender, "Cannot remove yourself as manager")
         ;
      require(daoManagers.length > 1, "Cannot remove last manager");

     // code
}
```

Impact

Unnecessary check inside `removeDaoManager()` which is of no use and should be removed.

Recommendation

Consider removing this check as it is of no use:

```
require(daoManagers.length > 1, "Cannot remove last manager");
```

Team Response

Fixed.

[L-04] Single-step Ownership Transfer Pattern Can Be Dangerous

Severity

Low Risk

Description

Inheriting from OpenZeppelin's `Ownable` contract means you are using a single-step ownership transfer pattern. If an admin provides an incorrect address for the new owner, this will result in none of the `onlyOwner` marked methods being callable again. The better way to do this is to use a two-step ownership transfer approach, where the new owner should first claim its new rights before they are transferred.

Location of Affected Code

File: src/abstracts/LeapBase.sol

File: src/LeapFactory.sol

Impact

If an admin provides an incorrect address for the new owner, this will result in none of the `onlyOwner` marked methods being callable again.

Recommendation

Consider using the OpenZeppelin's `Ownable2Step` instead of `Ownable`.

Team Response

Fixed.

shieldify

Thank you!