shieldify

Harvestflow

SECURITY REVIEW

Date: 13 June 2024

CONTENTS

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 have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

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 Harvest Flow

Harvest flow introduces an innovative approach to lending via exposure to tangible assets that can be monitored in real-time. This is achieved by embedding IoT devices in each asset financed through the platform. These devices provide 24/7 location and usage data, ensuring transparency and security for both lenders and borrowers. The devices are linked to the digital assets with the help of NFTs, turning these agreements into verifiable, secure digital assets on the blockchain. This not only simplifies the management of contracts but also opens new avenues for investors to engage with and monitor their investments.

The goal of representing the lending contracts as digital certificates (NFTs) is to also allow users to claim or redeem rights. The objective of the NFT contract is to enhance transparency and convenience by representing TokTok's lending contracts through NFTs.

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 5 days with a total of 160 hours dedicated to the audit by four researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying an unauthorized access control by claiming/redeeming together with flawed logic by the bonus token functionality. Additionally, there were several differences between the documentation and the code implementation, among other findings of lower severity.

The HarvestFlow team has done a great job with their test suite and provided exceptional support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**Harvest Flow**
Repository	HarvestFlow
Type of Project	NFT Lending Protocol
Audit Timeline	5 days
Review Commit Hash	bfb24f9b9c6fdec935dfc0cb1c99c1285d29daf2
Fixes Review Commit Hash	a073854c96d718a8af3d6f1193e8d9e607006224

5.2 Scope

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

File	nSLOC
src/NftFactory.sol	53
src/TokTokNft.sol	331
Total	**384**

6. Findings Summary

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

- **Critical** and **High** issues: **0**
- **Medium** issues: **4**
- **Low** issues: **4**

ID	Title	Severity	Status
[M-01]	`Claim/Redeem` Functionality Is Executed For The Token Owner Instead For the Current User	Medium	Acknowledged
[M-02]	Bonus Tokens Can Be Claimed Even After The Specified Period End	Medium	Acknowledged
[M-03]	Discrepancy Between the Documentation and Actual Implementation	Medium	Fixed
[M-04]	Bonus Token Functionality Will Not Work For Some ERC-20 Tokens	Medium	Fixed
[L-01]	Minter Could Front-Run the setPublicPrice() Admins' Transaction and Execute a Public Mint at the Old Price	Low	Acknowledged
[L-02]	Use `Call` Instead of `Transfer`	Low	Fixed
[L-03]	Use `safeMint()` Instead of `mint()`	Low	Acknowledged
[L-04]	Incomplete Validation	Low	Fixed

7. Findings

[M-01] `Claim/Redeem` Functionality Is Executed For The Token Owner Instead For the Current User

Severity

Medium Risk

Description

In the `TokTokNft` contract, the `claim()`, `claimAll()`, `redeem()`, `redeemAll()` and `claimToken()` functions can be executed by any user, not just the owner of a specific NFT. This is problematic because these functions transfer the accrued interest and principal, depending on the function, which decreases the contract's balance of the `payableToken`.

As a result, a malicious user could exploit this by reducing the payableToken balance on behalf of NFT owners without their permission. This vulnerability could severely impact the NFT contract or its owners, especially considering the calculation of claimable interest in `claim()`, `claimAll()`, and the principal refunded in `redeem()` and `redeemAll()`. Such exploitation could lead to insolvency for the protocol.

Proof of Concept

```
function test_claim_canBeCalledByAnyone(uint256 amount) public {
    address alice = _getRandomActor(1);
    address maliciousUser = _getRandomActor(2);

    amount = bound(amount, 1, cap);
    vm.startPrank(alice);
    toktok.publicMint(amount);
```

```
    vm.startPrank(toktok.owner());
    toktok.activate();

    uint256 payableTokenContractBalanceBefore = payableToken.balanceOf(
        address(toktok)
    );
    uint256 payableTokenNftOwnerBalanceBefore = payableToken.balanceOf(
        alice
    );
    uint256 payableTokenNftMaliciousUserBalanceBefore = payableToken.
       balanceOf(
        maliciousUser
    );

    uint256 expectedClaimAmount = (price * toktok.yield()) / 1e18;

// Check if  lice is a owner of tokenId 1 and that maliciousUser balance
   if 0
    assertEq(toktok.ownerOf(1), alice);
    assertEq(toktok.balanceOf(maliciousUser), 0);

    vm.warp(toktok.maturity());
    vm.startPrank(maliciousUser);
// Expect a revert as the malicious user isn't the owner of tokenId 1
    vm.expectRevert(abi.encodeWithSelector(TokTokNft.NotAuthorized.
       selector, 1, alice, maliciousUser));
    toktok.claim(1);

// Check if Alice's deposit is the same as before the claim
    assertEq(
        payableToken.balanceOf(alice),
        payableTokenNftOwnerBalanceBefore
    );
}
```

Location of Affected Code

File: contracts/src/TokTokNft.sol#L266

File: contracts/src/TokTokNft.sol#L282

File: contracts/src/TokTokNft.sol#L314

File: contracts/src/TokTokNft.sol#L337

File: contracts/src/TokTokNft.sol#L430

Recommendation

Implement the following validation for all functions to ensure that the user who is claiming/redeeming is the owner of the specified NFT(s):

```
+ error NotAuthorized(uint256 tokenId, address owner, address msgSender);

+ if(msg.sender != owner) revert NotAuthorized(tokenId, owner, msg.sender
   );
```

Team Response

Acknowledged.

[M-02] Bonus Tokens Can Be Claimed Even After The Specified Period Ends

Severity

Medium Risk

Description

The `claimToken()` function can be executed after the `bonusToken` valid period has passed and the user will still be able to claim successfully.

Proof of Concept

When the bonus period is over a revert is expected:

```
function test_claimToken_afterBonusPeriod() public {
// add a second participant to check for proper token ownership
   proportion
  address secondActor = _getRandomActor(1);
  vm.prank(secondActor);
  toktok.publicMint(3);

  address receiver = toktok.owner();
  vm.startPrank(receiver);
  payableToken.approve(address(toktok), type(uint256).max);
  uint256 tokenId = toktok.nextTokenId();
  toktok.publicMint(1);
  toktok.activate();
```

```
    uint256 bonusTokenAmount = 100 ether;
    MockERC20 bonusToken = new MockERC20("MockERC20", "MOCK",
        bonusTokenAmount);
    bonusToken.approve(address(toktok), type(uint256).max);
    uint256 l = 1;
    uint256 r = 1 + 365 days;
    toktok.addBonusToken(address(bonusToken), bonusTokenAmount, l, r);

// warp after the bonus period end so the claim should revert
    vm.warp(l + r + 1);
    uint256 bonusTokenBalanceContractBefore = bonusToken.balanceOf(address(
        toktok));
    uint256 bonusTokenBalanceReceiverBefore = bonusToken.balanceOf(receiver
        );

    vm.expectRevert(abi.encodeWithSelector(TokTokNft.BonusTokenPeriodPassed
        .selector, r, block.timestamp));
    toktok.claimToken(address(bonusToken), tokenId);
}
```

Location of Affected Code

File: contracts/src/TokTokNft.sol#L414

Recommendation

Consider implementing the following check that will ensure that the user will be able to claim only before the period has concluded.

```
+ error BonusTokenPeriodNotPassed(uint256 periodEnd, uint256
    currentTimestamp);

+ if (block.timestamp > settings.r) {
+     revert BonusTokenPeriodNotPassed(settings.r, block.timestamp);
+ }
```

Team Response

Acknowledged.

[M-03] Discrepancy Between the Documentation and Actual Implementation

Severity

Medium Risk

Description

The project implementation deviates from its documentation for the following reasons:

- The `_mintCheck()` function does not verify that the user has enough funds to cover the minting cost.
- The `withdraw()` function does not check if the contract holds a sufficient amount of tokens/eth.
- The `removeBonusToken()` function does not delete the `claimedToken[token]` value.

Location of Affected Code

File: contracts/src/TokTokNft.sol

Recommendation

It is crucial for the documentation and implementation to completely align.

Team Response

Fixed.

[M-04] Bonus Token Functionality Will Not Work For Some ERC-20 Tokens

Severity

Medium Risk

Description

Tokens not compliant with the ERC20 specification could return `false` from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Check the return value, which is a requirement, as written in the EIP-20 specification.

In the `TokTokNft` contract, DAI tokens are intended to be used as the `payable_token` in most functions without any issues. However, in the `addBonusToken()`, `removeBonusToken()`, and `claimToken()` functions, different eligible tokens may be utilized. It's important to note that not all ERC20 token implementations behave consistently. Some tokens, such as USDT on the mainnet and ZRX, would not revert on failure. Instead, they might return `false` or, in some cases, not return any value at all.

Although the DAI token will currently serve as the primary payment token, this may change in the future. Therefore, it is advisable to apply the following recommendations not only to the bonus token functionality but also to the main functions listed.

Location of Affected Code

File: contracts/src/TokTokNft.sol#L368-L433

```
/// @notice Add `amount` of bonus `token` to the contract, with yielding
    period from `l` to `r`. Can be executed only by the owner.
/// @param token Address of the token to add
/// @param amount Amount of the token to add
/// @param l Start time of the yielding period
/// @param r End time of the yielding period
function addBonusToken(address token, uint256 amount, uint256 l, uint256
    r) public onlyOwner whenNotPaused {
     if (bonusToken[token].amount > 0) {
         revert BonusTokenAlreadySet(token);
     }
     if (l < block.timestamp || l > r) {
         revert InvalidInput(l, r);
     }

     ERC20(token).transferFrom(msg.sender, address(this), amount);

     bonusToken[token] = BonusTokenSettings(amount, l, r);
     bonusTokenList.push(token);

     emit BonusTokenAdded(token, amount, l, r);
}

/// @notice Remove bonus `token` from the contract, transferring the
    remaining balance to `receiver`. Can be executed only by the owner.
/// @param token Address of the token to remove
/// @param receiver Recipient of the remaining balance
function removeBonusToken(address token, address receiver) public
    onlyOwner whenNotPaused {
     if (bonusToken[token].amount == 0) {
         revert BonusTokenNotSet(token);
     }
     delete bonusToken[token];

     for (uint256 i = 0; i < bonusTokenList.length; i++) {
         if (bonusTokenList[i] == token) {
             bonusTokenList[i] = bonusTokenList[bonusTokenList.length -
                 1];
             bonusTokenList.pop();
             break;
         }
     }

     ERC20(token).transfer(receiver, ERC20(token).balanceOf(address(this))
         );

     emit BonusTokenRemoved(token);
}
```

```
/// @notice Claim bonus `token` for `tokenId`.
/// @param token Address of the bonus token to claim
/// @param tokenId Token ID to claim bonus for
function claimToken(address token, uint256 tokenId) public whenNotPaused
   {
    if (!_isActive) revert NotActive();
    BonusTokenSettings memory settings = bonusToken[token];
    if (block.timestamp < settings.l) {
        revert BonusTokenStartPeriodNotReached(settings.l, block.
           timestamp);
    }

// Proportion of time interval, scaled to the 1e18
    uint256 proportionOfIntervalScaled =
        ((Math.min(settings.r, block.timestamp) - settings.l) * 1e18) / (
           settings.r - settings.l);

// Amount of token to be sent
    uint256 sendAmount =
        ((((settings.amount * proportionOfIntervalScaled) / 1e18) /
           totalSupply()) - claimedToken[token][tokenId]);

    claimedToken[token][tokenId] += sendAmount;
    ERC20(token).transfer(ownerOf(tokenId), sendAmount);

    emit BonusTokenClaimed(token, tokenId, sendAmount);
}
```

File: contracts/src/TokTokNft.sol

```
function publicMint(uint256 mintAmount) public whenNotPaused returns (
   uint256) {

function preMint(uint256 mintAmount, uint256 maxMintAmount, bytes memory
   signature) public whenNotPaused returns (uint256) {

function claim(uint256 tokenId) public whenNotPaused {

function claimAll(uint256[] memory tokenIds) public whenNotPaused {

function redeem(uint256 tokenId) public whenNotPaused {

function redeemAll(uint256[] memory tokenIds) public whenNotPaused {

function withdraw(address token, uint256 amount, address receiver) public
    onlyOwner whenNotPaused {
```

Recommendation

Consider using OpenZeppelin's `SafeERC20` library and use the `safeTransfer()` function instead of `transfer()` and adding the `nonReentrant` modifier to all of these functions.

Team Response

Fixed.

[L-01] Minter Could Front-Run the `setPublicPrice()` Admins' Transaction and Execute a Public Mint at the Old Price

Severity

Low Risk

Description

The `setPublicPrice()` method sets the public sale price of a token, and only the owner can call it. If a user monitors the mempool and spots a transaction that raises the `publicPrice`, they can front-run this transaction and call `publicMint()` at the old, lower price to save money.

The same applies to the `preMint()` function as well but it can be called only by whitelisted addresses so the likelihood is lower.

Location of Affected Code

File: contracts/src/TokTokNft.sol#L236

Recommendation

Consider using a timelock in `setPublicPrice()` to avoid instant changes to the `publicPrice` parameters. Additionally, the transaction can be propagated through private mempool services, to prevent frontrunning.

Team Response

Acknowledged.

[L-02] Use `Call` Instead of `Transfer`

Severity

Low Risk

Description

The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase in the SLOAD instruction.

Location of Affected Code

File: contracts/src/TokTokNft.sol#L360

Scenario

Additionally, the use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

- The withdraw smart contract does not implement a payable function.
- The withdraw smart contract does implement a payable fallback which uses more than 2300 gas units.
- The withdraw smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.
- Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Recommendation

Use `call()` instead of `transfer()`, but be sure to respect the CEI pattern and/or add re-entrance guards, as several hacks already happened in the past due to this recommendation not being fully understood.

Team Response

Fixed.

[L-03] Use `safeMint()` Instead of `mint()`

Severity

Low Risk

Description

Using `ERC721::_mint()` can mint ERC721 tokens to addresses which do not support ERC721 tokens, while `ERC721::_safeMint()` ensures that ERC721 tokens are only minted to addresses which support them.

Both `publicMint()` and `preMint()` methods in `TokTokNft` use the `_mint()` method which is missing a check if the recipient is a smart contract that can actually handle ERC721 tokens. If the case is that the recipient can not handle ERC721 tokens then they will be stuck forever.

Location of Affected Code

File: contracts/src/TokTokNft.sol

Recommendation

Consider using `_safeMint()` instead of `_mint()` for ERC721 tokens, but do this very carefully, because this opens up a reentrancy attack vector, therefore add a `nonReentrant` modifier in the method that is calling `_safeMint()` because of this.

Team Response

Acknowledged.

[L-04] Incomplete Validation

Severity

Low Risk

Description

In the `NftFactory.sol` contract the following validation checks are missing:

- `constructor()` missing zero-address check for `_nftImplementation` parameter.

The `InitializationParams` struct input, is the parameter of the `deploy()` function. The following validation checks are missing:

- The `cap`, `price`, `lendingAt`, `yield` and `lendingPeriod` parameters are missing zero-value checks.
- The `payableToken`, `owner` and `signerAddress` parameters are missing zero-address checks.

In the `TokTokNft.sol` contract the following validation checks are missing:

- `withdraw()` function missing zero-address check for `receiver` parameter.
- `withdraw()` function missing zero-value check for `amount` parameter.
- `addBonusToken()` function missing zero-address check for `token` parameter.
- `removeBonusToken()` function missing zero-address check for `token` and `receiver` parameters.
- `claimToken()` function missing zero-address check for `token` parameter.

Location of Affected Code

File: contracts/src/NftFactory.sol

File: contracts/src/TokTokNft.sol

Recommendation

Consider implementing the validation checks from above.

Team Response

Fixed.

shieldify

Thank you!