shieldify

Multipli

SECURITY REVIEW

Date: 30 May 2025

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 has 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 Multipli

Multipli is a protocol that allows users to bridge tokens between Ethereum/BSC and StarkEx-based Layer 2. The system handles deposits, withdrawals, and yield claims through a set of smart contracts. This contract is expected to be work with all EVM-compatible chains.

Learn more: Docs

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 4 days, dedicated by the Shieldify core team.Add commentMore actions

The audit report identified two Medium-severity issues and two Low-severity vulnerabilities. The vulnerabilities primarily stem from users can lose funds from unsupported tokens, incorrect receipt token minting and fixed gas allocation.

The Multipli team has done a great job with the development and has been highly responsive to the Shieldify research team's inquiries.

5.1 Protocol Summary

Project Name	**Multipli**
Repository	multipli-bridger
Type of Project	DeFi, Bridge
Audit Timeline	4 days
Review Commit Hash	c5b4869fcf7349c9ce9b67ca26f6ae45cc7babce
Fixes Review Commit Hash	4b9f94e4d3d3cccd57099473562822 42c16d8788

5.2 Scope

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

File	**nSLOC**
src/Initializable.sol	34
src/MultipliBridger.sol	119
src/TransferHelper.sol	34
script/deploy/DeployMultipliBridger.s.sol	23
Total	**210**

6. Findings Summary

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

- **Medium** issues: **2**
- **Low** issues: **2**
- **Info** issues: **2**
- **Gas** issues: **2**

ID	Title	Severity	Status
[M-01]	Users Can Lose Funds by Depositing Unsupported Tokens Due to Lack of Whitelisting	Medium	Fixed
[M-02]	Incorrect Receipt Token Minting for Fee on Transfer Tokens	Medium	Fixed
[L-01]	Fixed Gas Allocation in Native Token Transfers Enables DOS	Low	Fixed
[L-02]	Use Two-Step Ownership Transfer for Improved Security	Low	Acknowledged
[I-01]	Unused Functions Increase Contract Size Unnecessarily	Info	Acknowledged
[I-02]	Floating and Outdated Pragma Version	Info	Fixed
[G-01]	Avoid Restoring Values	Gas	Fixed
[G-02]	Cache `address(this)` When Used More Than Once	Gas	Acknowledged

7. Findings

[M-01] Users Can Lose Funds by Depositing Unsupported Tokens Due to Lack of Whitelisting

Severity

Medium Risk

Description

The `MultipliBridger` contract's deposit function accepts any ERC-20 token without validation, as evident in its unrestricted token parameter. This design flaw becomes critical when users deposit tokens that are not supported by the protocol's Layer 2 infrastructure or by the backend system. The vulnerability stems from the contract's assumption that all deposited tokens will have corresponding receipt tokens (xTokens) mintable on StarkEx L2. When unsupported tokens are deposited, they become trapped in the contract since the off-chain systems cannot process them to mint equivalent L2 tokens. The only recovery path requires manual intervention through the removeFunds function by authorized addresses, creating a centralized recovery dependency.

Location of Affected Code

File: src/MultipliBridger.sol#L125

```
function deposit(address token, uint256 amount) external {
  TransferHelper.safeTransferFrom(
    token,
    msg.sender,
    address(this),
    amount
  );
  emit BridgedDeposit(msg.sender, token, amount);
}
```

Impact

Users risk permanent loss of funds when depositing unsupported tokens, as these assets become locked in the contract with no automated recovery mechanism.

Recommendation

The contract should implement a strict token whitelist mechanism managed by the owner.

Team Response

Fixed.

[M-02] Incorrect Receipt Token Minting for Fee on Transfer Tokens

Severity

Medium Risk

Description

The `MultipliBridger` contract's deposit mechanism fails to properly account for fee-on-transfer (FoT) tokens. The vulnerability manifests in the `deposi` function, which invokes `TransferHelper.safeTransferFrom(token, msg.sender, address(this), amount)` to transfer tokens from the user to the contract. This implementation assumes the amount specified will exactly match the tokens received by the contract, which is incorrect for FoT tokens that deduct a transfer fee.

The contract then emits the `BridgedDeposit event` with the full pre-fee amount as the deposited value. Off-chain systems, including the sequencer and bridging back-end that monitor these events, will interpret this logged amount as the actual value received by the contract. This creates a discrepancy where users receive receipt tokens (`xTokens on L2`) based on the pre-fee amount while the contract holds less value than recorded.

Location of Affected Code

File: src/MultipliBridger.sol#L125

```
function deposit(address token, uint256 amount) external {
  TransferHelper.safeTransferFrom(
    token,
    msg.sender,
    address(this),
    amount
  );
  emit BridgedDeposit(msg.sender, token, amount);
}
```

Impact

When FoT tokens are deposited, the protocol will credit users with receipt tokens based on the pre-fee amount while holding less actual token value in the contract. This creates an accounting imbalance that could lead to liquidity shortfalls when processing withdrawals.

Recommendation

The contract should implement FoT token handling by comparing the balance before and after transfers. For the `deposit` function, first record the contract's token balance, then perform the transfer, then verify the new balance to determine the actual received amount.

Team Response

Fixed.

[L-01] Fixed Gas Allocation in Native Token Transfers Enables DOS

Severity

Low Risk

Description

The `removeFundsNative()` function in the MultipliBridger contract uses the `transfer()` method for Ether transfers, which imposes a strict 2300 gas stipend. This becomes problematic when sending funds to smart contract addresses that require more gas for their fallback/receive functions to execute properly. The vulnerability stems from line where to.transfer(amount) is used instead of the recommended `.call{value: amount}("")` pattern. Modern Ethereum best practices discourage using transfer() precisely because of this gas limitation issue, as EIP-1884 increased certain opcode costs making 2300 gas often insufficient.

Location of Affected Code

File: src/MultipliBridger.sol#L206)

```
function removeFundsNative(
  address payable to,
  uint256 amount
) public _isAuthorized {
  require(address(this).balance >= amount, "INSUFFICIENT_BALANCE");
// @audit should use .call
  to.transfer(amount);
}
```

Impact

When attempting to withdraw native tokens to smart contract addresses, the transfers will fail if the recipient's fallback function requires more than 2300 gas.

Recommendation

Replace the `transfer()` call with a low-level `.call()` pattern that: (1) Forwards all remaining gas, and (2) Includes proper security checks.

Team Response

Fixed.

[L-02] Use Two-Step Ownership Transfer for Improved Security

Severity

Low Risk

Description

The `Ownable2Step` pattern is an improvement over the traditional `Ownable` pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Location of Affected Code

File: src/MultipliBridger.sol#L19

```
abstract contract OwnableUpgradeable is Initializable, ContextUpgradeable
    {}
```

Impact

Without the `Ownable2Step` pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the `Ownable2Step` pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

Recommendation

It is recommended to use `Ownable2StepUpgradeable`.

Team Response

Acknowledged.

[I-01] Unused Functions Increase Contract Size Unnecessarily

Severity

Informational

Description

The contract inherits from `ContextUpgradeable` but never use it:

- `__Context_init()` - Empty initialization function In `TransferHelper.sol` these two functions also doesn't get used anywhere.
- `safeApprove()`
- `safeTransferETH()`

Location of Affected Code

File: src/MultipliBridger.sol

```
function __Context_init() internal initializer {
    __Context_init_unchained();
}

function safeTransferETH(address to, uint256 value) internal {
    (bool success, ) = to.call{value: value}(new bytes(0));
    require(success, "TransferHelper: ETH_TRANSFER_FAILED");
}

function safeApprove(
    address token,
    address to,
    uint256 value
) internal {
// bytes4(keccak256(bytes('approve(address,uint256)')));
    (bool success, bytes memory data) = token.call(
        abi.encodeWithSelector(0x095ea7b3, to, value)
    );
    require(
        success && (data.length == 0 || abi.decode(data, (bool))),
        "TransferHelper: APPROVE_FAILED"
    );
}
```

Impact

Increases deployment costs unnecessarily

Recommendation

Consider removing the unused functions.

Team Response

Acknowledged.

[I-02] Floating and Outdated Pragma Version

Severity

Informational

Description

The contract employs a deprecated Solidity compiler version `pragma solidity >=0.4.22 <0.9.0` in its pragma declaration. The Solidity language undergoes continuous enhancement, with each iteration delivering critical security updates, defect resolutions, and gas optimizations. Relying on obsolete compiler versions leaves the code susceptible to documented vulnerabilities that subsequent

releases have remedied. Modern compiler iterations also incorporate advanced syntax capabilities and architectural refinements that bolster contract robustness while reducing execution costs.

Location of Affected Code

File: src/MultipliBridger.sol#L19

File: src/Initializable.sol

Impact

The use of an outdated Solidity compiler version can have significant negative impacts. Security vulnerabilities that have been identified and patched in newer versions remain exploitable in the deployed contract.

Recommendation

It is suggested to use the `0.8.29` pragma version.

Team Response

Fixed.

[G-01] Avoid Restoring Values

Severity

Gas Optimization

Description

The function is found to be allowing restoring the value in the contract's state variable even when the old value is equal to the new value. This practice results in unnecessary gas consumption due to the Gsreset operation (2900 gas), which could be avoided. If the old value and the new value are the same, not updating the storage would avoid this cost and could instead incur a Gcoldsload (2100 gas) or a Gwarmaccess (100 gas), potentially saving gas.

Location of Affected Code

File: src/MultipliBridger.sol

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(
        newOwner != address(0),
        "Ownable: new owner is the zero address"
    );
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

File: src/MultipliBridger.sol

```
function authorize(address user, bool value) external onlyOwner {
    authorized[user] = value;
}
```

Recommendation

To optimize gas usage, add a check to compare the old value with the new value before updating the storage. Only perform the storage update if the new value is different from the old value. This approach will prevent unnecessary storage writes and reduce gas consumption.

Team Response

Fixed.

[G-02] Cache `address(this)` When Used More Than Once

Severity

Gas Optimization

Description

The repeated usage of `address(this)` within the contract could result in increased gas costs due to multiple executions of the same computation, potentially impacting efficiency and overall transaction expenses.

Location of Affected Code

File: src/MultipliBridger.sol

```
function deposit(address token, uint256 amount) external {
    TransferHelper.safeTransferFrom(
        token,
        msg.sender,
        address(this),
        amount
    );
    emit BridgedDeposit(msg.sender, token, amount);
}
```

File: src/MultipliBridger.sol

```
function addFunds(address token, uint256 amount) external _isAuthorized {
    TransferHelper.safeTransferFrom(
        token,
        msg.sender,
        address(this),
        amount
    );
}
```

File: src/MultipliBridger.sol

```
function removeFundsNative(
    address payable to,
    uint256 amount
) public _isAuthorized {
    require(address(this).balance >= amount, "INSUFFICIENT_BALANCE");
    to.transfer(amount);
}
```

Recommendation

Optimize gas usage by caching the value of address(this) and reusing it throughout the contract, reducing redundant computations and thereby enhancing efficiency.

Team Response

Acknowledged.

shieldify

Thank you!