EFIP-5 & EFIP-8 Implementation

EtherFi

HALBORN

EFIP-5 & EFIP-8 Implementation - EtherFi

Prepared by: HALBORN

Last Updated 08/01/2024

Date of Engagement by: July 23rd, 2024 - July 25th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	0	3

TABLE OF CONTENTS

1. Introduction

`EtherFi` engaged Halborn to conduct a security assessment on smart contracts beginning on *07/23/2024* and ending on *07/25/2024*. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn dedicated 3 working days for the engagement and assigned one full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, no major issues were found, but Halborn identified minor improvements, which were successfully addressed by the `EtherFi team`. The main recommendations were:

- Implement proper event emissions for important functions to enhance transparency.
- Apply best practices when dealing with loops to improve efficiency.

3. Test Approach And Methodology

Halborn performed a combination of manual, semi-automated and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any vulnerability classes
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Local deployment and testing (`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: smart-contracts

(b) Assessed Commit ID: 4a7c8cc

(c) Items in scope:

- src/EETH.sol
- src/WeETH.sol
- etherfi-protocol/smart-contracts/pull/90

Out-of-Scope:

REMEDIATION COMMIT ID:

- ec7a967ec7a967

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CACHE ARRAY LENGTH OUTSIDE OF LOOP	INFORMATIONAL	SOLVED - 07/26/2024
MISSING EVENTS FOR BLACKLIST AND WHITELIST OPERATIONS	INFORMATIONAL	SOLVED - 07/26/2024
REDUNDANT CONSOLE IMPORT	INFORMATIONAL	SOLVED - 07/26/2024

7. FINDINGS & TECH DETAILS

7.1 CACHE ARRAY LENGTH OUTSIDE OF LOOP

// INFORMATIONAL

Description

In both the EETH and WeETH contracts, the `setWhitelistedSpender` and `setBlacklistedRecipient` functions contain loops that access the length of the input array in each iteration. Specifically:

- In the EETH contract:

```
function setWhitelistedSpender(address[] calldata _spenders, bool _isWhitelisted) external onlyO
    for (uint i = 0; i < _spenders.length; i++) {
        whitelistedSpender[_spenders[i]] = _isWhitelisted;
    }
}

function setBlacklistedRecipient(address[] calldata _recipients, bool _isBlacklisted) external o
    for (uint i = 0; i < _recipients.length; i++) {
        blacklistedRecipient[_recipients[i]] = _isBlacklisted;
    }
}
```

- The WeETH contract has identical implementations of these functions.

If the array length is not cached, the Solidity compiler will read the length of the array during each iteration. While the current implementation uses `calldata` arrays, which have the lowest gas cost (3 gas per iteration) for length access, caching the length is still a best practice.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Here's the recommended modification for both contracts:

- Modify the `setWhitelistedSpender` function:

```
function setWhitelistedSpender(address[] calldata _spenders, bool _isWhitelisted) external onlyO
    uint256 length = _spenders.length;
    for (uint i = 0; i < length; i++) {
        whitelistedSpender[_spenders[i]] = _isWhitelisted;
    }
}
```

- Modify the `setBlacklistedRecipient` function:

```
function setBlacklistedRecipient(address[] calldata _recipients, bool _isBlacklisted) external o
    uint256 length = _recipients.length;
    for (uint i = 0; i < length; i++) {
        blacklistedRecipient[_recipients[i]] = _isBlacklisted;
    }
```

These changes should be applied to both the EETH and WeETH contracts.

Remediation Plan

SOLVED: The suggested mitigation was implemented by the **EtherFi team.**

Remediation Hash

https://github.com/etherfi-protocol/smart-contracts/commit/ec7a967460e9a85673cbc4f85bf0ceed7178a94e

References

etherfi-protocol/smart-contracts/src/WeETH.sol#L129
etherfi-protocol/smart-contracts/src/WeETH.sol#L138
etherfi-protocol/smart-contracts/src/EETH.sol#L202
etherfi-protocol/smart-contracts/src/EETH.sol#L208

7.2 MISSING EVENTS FOR BLACKLIST AND WHITELIST OPERATIONS

// INFORMATIONAL

Description

The EETH and WeETH contracts implement functionality to set blacklisted recipients and whitelisted spenders. However, these critical operations do not emit events when executed. Specifically:

- In both contracts:
  - The `setBlacklistedRecipient` function does not emit an event when addresses are added to or removed from the blacklist.
  - The `setWhitelistedSpender` function does not emit an event when addresses are added to or removed from the whitelist.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Define new events in both EETH and WeETH contracts:

```
event BlacklistStatusChanged(address indexed account, bool isBlacklisted);
event WhitelistStatusChanged(address indexed account, bool isWhitelisted);
```

- Modify the `setBlacklistedRecipient` function in both contracts:

```
function setBlacklistedRecipient(address[] calldata _recipients, bool _isBlacklisted) external o
    for (uint i = 0; i < _recipients.length; i++) {
        if (blacklistedRecipient[_recipients[i]] != _isBlacklisted) {
            blacklistedRecipient[_recipients[i]] = _isBlacklisted;
            emit BlacklistStatusChanged(_recipients[i], _isBlacklisted);
        }
    }
}
```

- Modify the `setWhitelistedSpender` function in both contracts:

```
function setWhitelistedSpender(address[] calldata _spenders, bool _isWhitelisted) external onlyO
    for (uint i = 0; i < _spenders.length; i++) {
        if (whitelistedSpender[_spenders[i]] != _isWhitelisted) {
            whitelistedSpender[_spenders[i]] = _isWhitelisted;
            emit WhitelistStatusChanged(_spenders[i], _isWhitelisted);
        }
    }
}
```

Remediation Plan

SOLVED: The suggested mitigation was implemented by the **EtherFi team.**

Remediation Hash

https://github.com/etherfi-protocol/smart-contracts/commit/ec7a967460e9a85673cbc4f85bf0ceed7178a94e

References

7.3 REDUNDANT CONSOLE IMPORT

// INFORMATIONAL

Description

The `WeETH` contract currently imports `forge-std/console.sol`, which is a debugging tool typically used during development and testing. This import is redundant in a production environment.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove the following import statement from the `WeETH` contract:

```
import "forge-std/console.sol";
```

Remediation Plan

SOLVED: The suggested mitigation was implemented by the **EtherFi team.**

Remediation Hash

https://github.com/etherfi-protocol/smart-contracts/commit/ec7a967460e9a85673cbc4f85bf0ceed7178a94e

8. AUTOMATED TESTING

Introduction

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
INFO:Detectors:
WeETH.wrap(uint256) (src/WeETH.sol#55-61) ignores return value by eETH.transferFrom(msg.sender,address(this),_eETHAmount) (src/WeETH.sol#59)
WeETH.unwrap(uint256) (src/WeETH.sol#77-83) ignores return value by eETH.transfer(msg.sender,eETHAmount) (src/WeETH.sol#81)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
```

```
WeETH.permit(address,address,uint256,uint256,uint8,bytes32,bytes32).owner (src/WeETH.sol#87) shadows:
        - OwnableUpgradeable.owner() (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#48-50) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Reentrancy in WeETH.wrapWithPermit(uint256,ILiquidityPool.PermitInput) (src/WeETH.sol#66-72):
        External calls:
        - eETH.permit(msg.sender,address(this),_permit.value,_permit.deadline,_permit.v,_permit.r,_permit.s) (src/WeETH.sol#70)
        - wrap(_eETHAmount) (src/WeETH.sol#71)
                - eETH.transferFrom(msg.sender,address(this),_eETHAmount) (src/WeETH.sol#59)
        Event emitted after the call(s):
        - Transfer(address(0),account,amount) (lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/ERC20Upgradeable.sol#274)
                - wrap(_eETHAmount) (src/WeETH.sol#71)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
```

```
INFO:Detectors:
EETH.allowance(address,address)._owner (src/EETH.sol#98) shadows:
        - OwnableUpgradeable._owner (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#22) (state variable)
EETH.increaseAllowance(address,uint256).owner (src/EETH.sol#108) shadows:
        - OwnableUpgradeable.owner() (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#48-50) (function)
EETH.decreaseAllowance(address,uint256).owner (src/EETH.sol#115) shadows:
        - OwnableUpgradeable.owner() (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#48-50) (function)
EETH.permit(address,address,uint256,uint256,uint8,bytes32,bytes32).owner (src/EETH.sol#135) shadows:
        - OwnableUpgradeable.owner() (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#48-50) (function)
EETH._approve(address,address,uint256)._owner (src/EETH.sol#168) shadows:
        - OwnableUpgradeable._owner (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#22) (state variable)
EETH._useNonce(address).owner (src/EETH.sol#191) shadows:
        - OwnableUpgradeable.owner() (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#48-50) (function)
EETH.nonces(address).owner (src/EETH.sol#233) shadows:
        - OwnableUpgradeable.owner() (lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol#48-50) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
EETH.permit(address,address,uint256,uint256,uint8,bytes32,bytes32) (src/EETH.sol#134-154) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp <= deadline,ERC20Permit: expired deadline) (src/EETH.sol#144)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

The security team assessed all findings identified by the Slither software, most of which were informational and non-critical:

- The issue related to reentrancy in the `WeETH.sol` is a false positive.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.