HALBORN

FortunaFi - Tokenized Asset Protocol (TAP)

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **October 1st, 2023 - October 9th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	10/02/2023
0.2	Draft Version	10/09/2023
0.3	Draft Review	10/09/2023
0.4	Draft Review	10/11/2023
1.0	Remediation Plan	10/12/2023
1.1	Remediation Plan Update	10/17/2023
1.2	Remediation Plan Review	10/18/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

FortunaFi team's Tokenized Asset Protocol (TAP) (codenamed OffchainFund) enables the automation of the operations of a traditional investment fund via a smart contract.

FortunaFi engaged Halborn to conduct a security assessment on their smart contracts beginning on October 1st, 2023 and ending on October 9th, 2023. The security assessment was scoped to the smart contracts provided in the fortunafi/offchain-fund GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Per Halborn's recommendation, based on an initial scope assessment, one week and one security engineer were dedicated to the engagement to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks, none of which exceeded medium severity, which were addressed and accepted by the FortunaFi team. The main one was the following:

- Change the contract's logic to prevent griefing attacks.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to 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 the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry, Brownie).

2. 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.

2.1 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:

Exploitability 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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

2.3 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.

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

2.4 SCOPE

Code repositories:

1. Tokenized Asset Protocol (TAP)

   - Repository: fortunafi/offchain-fund
   - Initial Commit ID: c2c2a9b
   - Smart contracts in scope:
     - src/OffchainFund.sol
   - Fix commit ID (Final): d812e57
   - SHA-256 hash of the flattened smart contract (d812e57): 32490287f8113ad01c7212b56ffef646e9becd83cc0f73cee27f34535bbf658a

Out-of-scope

- Third-party libraries and dependencies
- Economic attacks
- Gas optimization recommendations

Note that the flattened version was created with the forge flatten command.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	4	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) GRIEFING ATTACK	Medium (5.0)	SOLVED - 10/11/2023
(HAL-02) SHARE PRICE IS NOT VALIDATED	Low (2.0)	RISK ACCEPTED
(HAL-03) LACK OF ROLE-BASED ACCESS CONTROL	Low (2.0)	RISK ACCEPTED
(HAL-04) SINGLE STEP OWNERSHIP TRANSFER PROCESS	Low (2.0)	RISK ACCEPTED
(HAL-05) OWNER CAN RENOUNCE OWNERSHIP	Low (2.0)	RISK ACCEPTED
(HAL-06) INCOMPATIBILITY WITH NON-STANDARD TOKENS	Informational (1.2)	ACKNOWLEDGED
(HAL-07) MISSING EVENTS FOR CONTRACT OPERATIONS	Informational (0.8)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) GRIEFING ATTACK - MEDIUM (5.0)

Description:

The OffchainFund contract allows updating the share value and progress to the next epoch only after all current deposit orders were processed using either the processDeposit() or batchProcessDeposit() functions by the FortunaFi team or by the users. These functions are also designed to revert if there are no associated orders. A malicious user can exploit this behavior to delay the share value update process by continuously reverting the batch function call by front-running the Owner and processing an element from the batch.

Suppose the FortunaFi team does not transfer funds to the contract before calling the update() function. In this case, a malicious user can transfer a small amount of USDC and call the batchProcessRedeem() function before the FortunaFi team refills the contract with USDC. This would result in a small number of shares being burned without transferring any funds to the users due to the rounding of the calculation. It would also delay the redemption of the affected users until the next epoch.

Proof of Concept:

The process functions revert if there is no associated order:

```
>>> offchainFund.processDeposit(alice, {'from': deployer})
Transaction sent: 0xb75b8d8deac5736624d164e21d4147f26d81883fe4a6974b996f9ef2f830a7d0
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 16
  OffchainFund.processDeposit confirmed (account has no mint order)   Block: 18311108   Gas used: 23264 (0.08%)

<Transaction '0xb75b8d8deac5736624d164e21d4147f26d81883fe4a6974b996f9ef2f830a7d0'>
>>> offchainFund.processRedeem(carl,{'from': deployer})
Transaction sent: 0xdf1efc1771cb647651db70e713013b93069da27427de2768977c67b1163cd0b7
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 17
  OffchainFund.processRedeem confirmed (account has no redeem order)   Block: 18311109   Gas used: 23279 (0.08%)

<Transaction '0xdf1efc1771cb647651db70e713013b93069da27427de2768977c67b1163cd0b7'>
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to modify the batch process functions, so they do not revert if there are no associated orders.

It is also recommended to change the OffchainFund contract's logic to prevent processing redeems until the contract is refilled with sufficient USDC.

Remediation Plan:

SOLVED: The FortunaFi team solved the issue in commits f8713d5 and 3e57942 by changing the batchProcessDeposit() and batchProcessRedeem() functions, so they do not revert if there are no associated orders and allowing only the Owner to execute the processRedeem() and batchProcessRedeem() functions.

4.2 (HAL-02) SHARE PRICE IS NOT VALIDATED - LOW (2.0)

Description:

The OffchainFund contract is inherited from the Ownable contract. The Owner is responsible for directly setting the value of the shares in each epoch using the update() function. In the current implementation of the protocol, there is no check that the price is valid or realistic, nor that the previously set value has not expired yet. Without these checks, an invalid price might be configured or used in the protocol.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

It is recommended to modify the protocol to verify the validity of the share price on-chain and prevent using an expired price. For example, it is considered to be good practice to use a price oracle instead of directly setting the share value by the function caller. Another solution could be to provide cryptographic proof as a parameter in the update function to enable verifying the share price.

Remediation Plan:

RISK ACCEPTED: The FortunaFi team made a business decision to accept the risk of this finding and not alter the contracts. The team stated that pricing updates will require approval from multiple administrators, which will be enforced through a change management policy and fine-grained role-based access control, and will be monitored through reports and alerts. All of which are supported natively on-chain via their MPC wallet.

4.3 (HAL-03) LACK OF ROLE-BASED ACCESS CONTROL - LOW (2.0)

Description:

The OffchainFund contract is inherited from the Ownable contract. The Owner can use the emergency token and Ether recovery functions and configure the contract's settings. However, the Owner is also used to drain the deposited USDC from the contract and update the share price. These operations are performed in each epoch. The risk of using the Owner for regular tasks increases the risk of account compromise. With both the recover and drain functions, the Owner can extract all funds to their wallet from the contract.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

It is recommended to create low-privileged roles to execute daily tasks in the protocol to limit the impact of possible compromise of such accounts.

It is also recommended to update the drain() function and, instead of transferring the funds to the caller, transfer it to a dedicated address configured by the Owner.

Remediation Plan:

RISK ACCEPTED: The FortunaFi team made a business decision to accept the risk of this finding and not alter the contracts. The team stated that fine-grained role-based access control, policy-based change management as well as reporting and alerting will be enforced through their MPC wallet.

4.4 (HAL-04) SINGLE STEP OWNERSHIP TRANSFER PROCESS - LOW (2.0)

Description:

The ownership of the contracts can be lost as the OffchainFund contract is inherited from the Ownable contract and their ownership can be transferred in a single-step process. The address the ownership is changed to should be verified to be active or willing to act as the owner.

Code Location:

Listing 1: openzeppelin-contracts/contracts/access/Ownable.sol

```
69 function transferOwnership(address newOwner) public virtual
↳ onlyOwner {
70     require(newOwner != address(0), "Ownable: new owner is the
↳ zero address");
71     _transferOwnership(newOwner);
72 }
```

Listing 2: openzeppelin-contracts/contracts/access/Ownable.sol

```
78 function _transferOwnership(address newOwner) internal virtual {
79     address oldOwner = _owner;
80     _owner = newOwner;
81     emit OwnershipTransferred(oldOwner, newOwner);
82 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

Consider using the Ownable2Step library over the Ownable library or implementing similar two-step ownership transfer logic into the contract.

Remediation Plan:

RISK ACCEPTED: The FortunaFi team made a business decision to accept the risk of this finding and not alter the contracts. The team stated the ownership transfer will require approval from multiple administrators, which will be enforced through a change management policy and fine-grained role-based access control, and monitored through reports and alerts. All of which are supported natively on-chain via their MPC wallet.

4.5 (HAL-05) OWNER CAN RENOUNCE OWNERSHIP - LOW (2.0)

Description:

The OffchainFund contract is inherited from the Ownable contract. The Owner of the contract is usually the account that deploys the contract. As a result, the Owner can perform some privileged functions. In the Ownable contracts, the renounceOwnership() function is used to renounce the Owner permission. Renouncing ownership before transferring would result in the contract having no Owner, eliminating the ability to call privileged functions.

Code Location:

Listing 3: openzeppelin-contracts/contracts/access/Ownable.sol

```
61 function renounceOwnership() public virtual onlyOwner {
62     _transferOwnership(address(0));
63 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

It is recommended that the Owner cannot call renounceOwnership() without first transferring Ownership to another address. In addition, if a multi-signature wallet is used, the call to the renounceOwnership() function should be confirmed for two or more users.

Remediation Plan:

RISK ACCEPTED: The FortunaFi team made a business decision to accept the risk of this finding and not alter the contracts. The team stated that renouncing ownership will require approval from multiple administrators, which will be enforced through a change management policy and fine-grained role-based access control, and monitored through reports and alerts. All of which are supported natively on-chain via their MPC wallet.

4.6 (HAL-06) INCOMPATIBILITY WITH NON-STANDARD TOKENS - INFORMATIONAL (1.2)

Description:

It was identified that the OffchainFund contract assumes that the transferFrom() calls transfers the full amount of tokens. This may not be true if the tokens being transferred are fee-on-transfer tokens, causing the received amount to be lesser than the accounted amount. For example, DGX (Digix Gold Token) and CGT (CACHE Gold) tokens apply transfer fees, and the USDT (Tether) token also has a currently disabled fee feature.

It was also identified that the contract assumes that its token balance does not change over time without any token transfers, which may not be true if the tokens being transferred were deflationary/rebasing tokens. For example, the supply of AMPL (Ampleforth) tokens automatically increases or decreases every 24 hours to maintain the AMPL target price.

It was also identified that the contract assumes that the token transfers return true on success, which may not be true with some implementations not following the ERC20 standard. For example, USDT (Tether) does not return true on success. These tokens are incompatible with the protocol, as the token transfers revert in these cases.

The finding does not apply to the contract when USDC is used as the token, and therefore, the risk rating of this finding was lowered to informal and is only included as a cautionary note in case the contract is used with other tokens.

Code Location:

Listing 4: OffchainFund.sol

```
138 function refill(uint256 assets) external {
139     assert(usdc.transferFrom(_msgSender(), address(this), assets))
 ↳ ;
140
141     emit Refill(_msgSender(), epoch, assets);
142 }
```

Listing 5: OffchainFund.sol

```
146 function deposit(uint256 assets) external onlyRole(WHITELIST) {
147     require(assets > min, "deposit is less than the minimum");
148     require(
149         cap > pendingDeposits + assets,
150         "deposit would exceed epoch cap"
151     );
152
153     (bool valid, ) = _canProcessDeposit(_msgSender());
154     require(!valid, "user has unprocessed userDeposits");
155
156     assert(usdc.transferFrom(_msgSender(), address(this), assets))
 ↳ ;
```

Listing 6: OffchainFund.sol

```
180     pendingDeposits += assets;
181     userDeposits[_msgSender()].amount += assets;
182
183     emit Deposit(_msgSender(), epoch, assets);
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:N/R:P/S:U (1.2)

Recommendation:

It is recommended that all tokens are thoroughly checked and tested before they are used in the contract to avoid tokens that are incompatible with the contracts.

Consider using OpenZeppelin's SafeERC20 wrapper with the IERC20 interface to make the contract compatible with tokens that return no value.

Consider getting the exact received amount of the tokens being transferred by calculating the difference of the token balance before and after the transfer to make the contract compatible with transfer-on-fee tokens.

Remediation Plan:

ACKNOWLEDGED: The FortunaFi team acknowledged this finding. The team stated that the contract will be deployed with USDC as a token and cannot be modified. Hence, none of the risks listed will apply. Recommendations will be taken into account for any potential future contracts that support other tokens.

4.7 (HAL-07) MISSING EVENTS FOR CONTRACT OPERATIONS - INFORMATIONAL (0.8)

Description:

It was identified that the adjustCap(), adjustMin() and recover() functions from the OffchainFund contract do not emit any events. These functions can only be used by the Owner to manage the contract. As a result, it might be more difficult for blockchain monitoring systems to detect suspicious behavior related to these features.

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (0.8)

Recommendation:

Consider adding events for all important operations to help monitor the contract, as a monitoring system that tracks relevant events would allow the timely detection of compromised system components.

Remediation Plan:

ACKNOWLEDGED: The FortunaFi team acknowledged this finding. The team stated those functions are restricted to Owner only. Calls to these functions will require approval from multiple administrators, which will be enforced through a change management policy and fine-grained role-based access control, and will be monitored through reports and alerts. All of which are supported natively on-chain via their MPC wallet.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

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.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

src/OffchainFund.sol

Slither results for OffchainFund.sol	
Finding	**Impact**
OffchainFund._processRedeem(address) (contracts/OffchainFund.sol#305-376) performs a multiplication on the result of a division: - available = (shares * balance * 1e12) / currentRedemptions (contracts/OffchainFund.sol#326) - deduct = Math.min((available * 1e8) / currentPrice,userRedemptions[account].amount) (contracts/OffchainFund.sol#354-357)	Medium

Finding	Impact
Reentrancy in OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184): External calls: - assert(bool)(usdc.transferFrom(_msgSender(),address(this),assets)) (contracts/OffchainFund.sol#156) State variables written after the call(s): - pendingDeposits += assets (contracts/OffchainFund.sol#180)OffchainFund.pendingDeposits (contracts/OffchainFund.sol#107) can be used in cross function reentrancies: - OffchainFund._processRedeem(address) (contracts/OffchainFund.sol#305-376) - OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184) - OffchainFund.drain() (contracts/OffchainFund.sol#405-425) - OffchainFund.pendingDeposits (contracts/OffchainFund.sol#107) - userDeposits[_msgSender()].epoch = epoch + 1 (contracts/OffchainFund.sol#169) OffchainFund.userDeposits (contracts/OffchainFund.sol#119) can be used in cross function reentrancies: - OffchainFund._canProcessDeposit(address) (contracts/OffchainFund.sol#246-256) - OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184) - OffchainFund.processDeposit(address) (contracts/OffchainFund.sol#196-230) - OffchainFund.userDeposits (contracts/OffchainFund.sol#119) - userDeposits[_msgSender()].epoch = epoch (contracts/OffchainFund.sol#174) OffchainFund.userDeposits (contracts/OffchainFund.sol#119) can be used in cross function reentrancies: - OffchainFund._canProcessDeposit(address) (contracts/OffchainFund.sol#246-256) - OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184) - OffchainFund.processDeposit(address) (contracts/OffchainFund.sol#196-230) - OffchainFund.userDeposits (contracts/OffchainFund.sol#119) - userDeposits[_msgSender()].amount += assets (contracts/OffchainFund.sol#181) OffchainFund.userDeposits (contracts/OffchainFund.sol#119) can be used in cross function reentrancies: - OffchainFund._canProcessDeposit(address) (contracts/OffchainFund.sol#246-256) - OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184)	Medium

Finding	Impact
IOffchainFund.nav().nav (contracts/OffchainFund.sol#34) shadows: - IOffchainFund.nav() (contracts/OffchainFund.sol#34) (function)	Low
IOffchainFund.cap().cap (contracts/OffchainFund.sol#38) shadows: - IOffchainFund.cap() (contracts/OffchainFund.sol#38) (function)	Low
OffchainFund.adjustCap(uint256) (contracts/OffchainFund.sol#466-468) should emit an event for: - cap = cap_ (contracts/OffchainFund.sol#467)	Low
OffchainFund.adjustMin(uint256) (contracts/OffchainFund.sol#472-474) should emit an event for: - min = min_ (contracts/OffchainFund.sol#473)	Low
OffchainFund.processRedeem(address) (contracts/OffchainFund.sol#284-300) has external calls inside a loop: assert(bool)(usdc.transfer(account,assets)) (contracts/OffchainFund.sol#289)	Low
OffchainFund._processRedeem(address) (contracts/OffchainFund.sol#305-376) has external calls inside a loop: contractsUsdcBalance = usdc.balanceOf(address(this)) (contracts/OffchainFund.sol#319)	Low
Reentrancy in OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184): External calls: - assert(bool)(usdc.transferFrom(_msgSender(),address(this),assets)) (contracts/OffchainFund.sol#156) State variables written after the call(s): - postDrainDepositCount = postDrainDepositCount + 1 (contracts/OffchainFund.sol#170-172) - postDrainDepositCount = postDrainDepositCount (contracts/OffchainFund.sol#170-172) - preDrainDepositCount = preDrainDepositCount + 1 (contracts/OffchainFund.sol#175-177) - preDrainDepositCount = preDrainDepositCount (contracts/OffchainFund.sol#175-177)	Low
Reentrancy in OffchainFund.deposit(uint256) (contracts/OffchainFund.sol#146-184): External calls: - assert(bool)(usdc.transferFrom(_msgSender(),address(this),assets)) (contracts/OffchainFund.sol#156) Event emitted after the call(s): - Deposit(_msgSender(),epoch,assets) (contracts/OffchainFund.sol#183)	Low

Finding	Impact
Reentrancy in OffchainFund.refill(uint256) (contracts/OffchainFund.sol#138-142): External calls: - assert(bool)(usdc.transferFrom(_msgSender(),address(this),assets)) (contracts/OffchainFund.sol#139) Event emitted after the call(s): - Refill(_msgSender(),epoch,assets) (contracts/OffchainFund.sol#141)	Low
Reentrancy in OffchainFund.processRedeem(address) (contracts/OffchainFund.sol#284-300): External calls: - assert(bool)(usdc.transfer(account,assets)) (contracts/OffchainFund.sol#289) Event emitted after the call(s): - ProcessRedeem(_msgSender(),account,epoch,shares,assets,currentPrice,processedInFull) (contracts/OffchainFund.sol#291-299)	Low
Reentrancy in OffchainFund.drain() (contracts/OffchainFund.sol#405-425): External calls: - assert(bool)(usdc.transfer(_msgSender(),assets)) (contracts/OffchainFund.sol#422) Event emitted after the call(s): - Drain(_msgSender(),epoch,assets,shares) (contracts/OffchainFund.sol#424)	Low
End of table for OffchainFund.sol	

The findings obtained as a result of the Slither scan were reviewed. The vulnerabilities were not included in the report because they were determined false positives.

THANK YOU FOR CHOOSING

// HALBORN