// HALBORN

Circle - ERC20R Recoverable Wrapper Token

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **October 31st, 2023 - November 2nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	11/01/2023
0.2	Document Updates	11/01/2023
0.3	Final Draft	11/02/2023
0.4	Draft Review	11/08/2023
0.5	Draft Review	11/10/2023
1.0	Remediation Plan	11/23/2023
1.1	Remediation Plan Review	11/28/2023
1.2	Remediation Plan Review	11/28/2023

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

Circle engaged Halborn to conduct a security assessment on their smart contracts beginning on October 31st, 2023 and ending on November 2nd, 2023. The security assessment was scoped to the smart contracts provided in the circlefin/erc20r-wrapper GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The ERC20R Wrapper is a configurable mechanism to protect any ERC20 token from thefts, hacks, and accidental transactions

1.2 ASSESSMENT SUMMARY

Halborn was provided 1 week for the engagement and assigned a team of 2 full-time security engineers to review the security of the smart contracts in scope. The security team consists of a blockchain and smart contract security experts 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, which were partially addressed by Circle. The main ones were the following:

- Checks were introduced in the deleteAt function of the RecordUtil library to prevent doubly linked list corruption.
- The risk of allowing unsettled tokens transfers was accepted as the Circle. team, with a disclaimer that the protocols and dApps integrating ERC20R tokens should be aware of this risk.

- The risk of using unsettled record manipulation to bypass the freeze mechanism by front-running the freeze transaction was acknowledged, as the Circle. team stated they will be using flash bots to prevent front-running.

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. Project Name

- Repository: circlefin/erc20r-wrapper
- Commit ID: f13e897237a55c5714b0f4b101346b72752c2dee
- Smart contracts in scope:
    1. ERC20RWrapper.sol (contracts/ERC20RWrapper.sol)
    2. RecordUtil.sol (util/RecordUtil.sol)

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	1	1	2

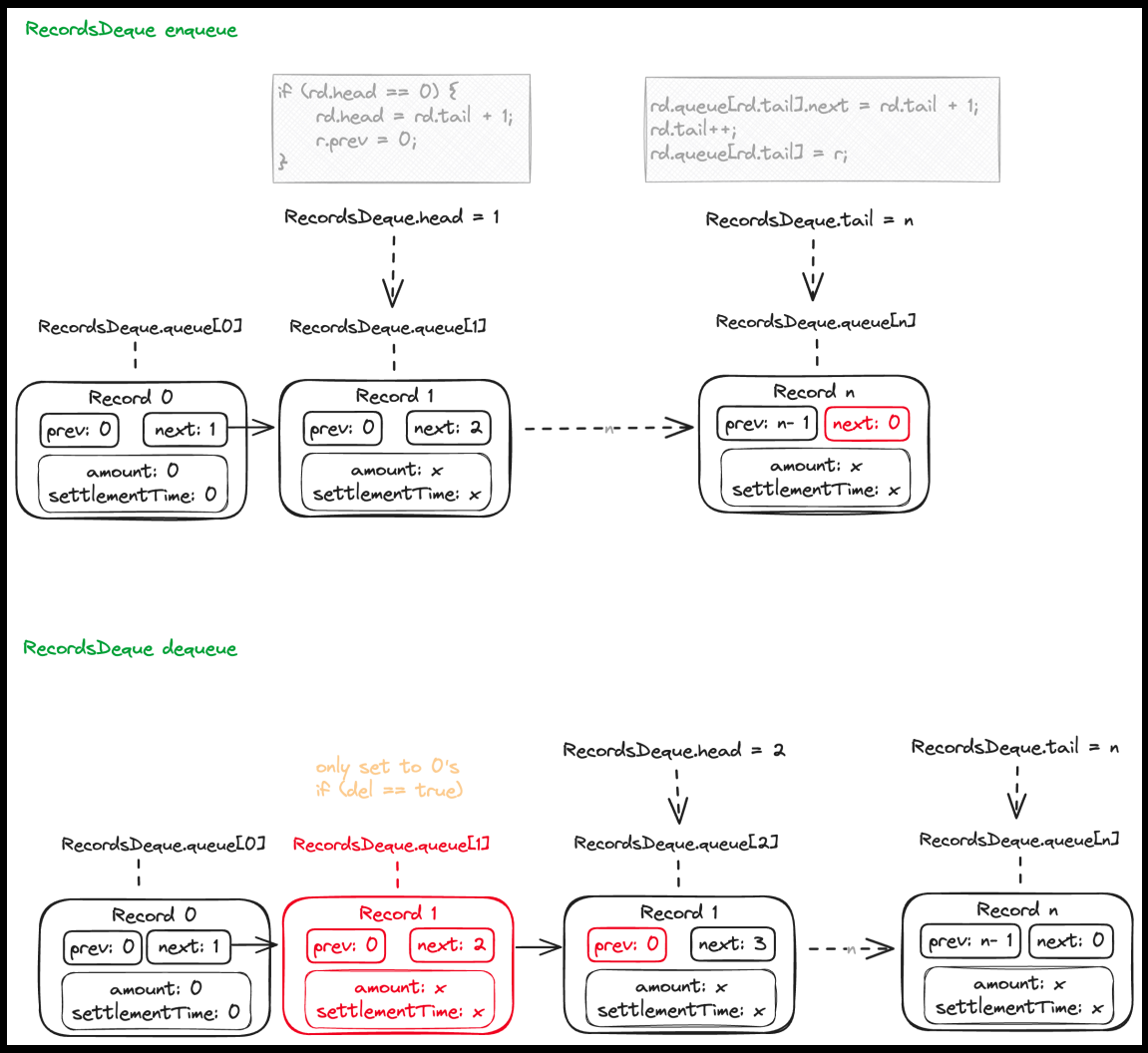
SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) DELETING INDEX 0 CORRUPTS DOUBLY LINKED LISTS	Critical (10)	SOLVED - 11/20/2023
(HAL-02) ALLOWANCE DOES NOT DISTINGUISH BETWEEN SETTLED AND UNSETTLED TOKENS	Medium (5.0)	RISK ACCEPTED
(HAL-03) UNHANDLED SELF-TRANSFER	Low (2.5)	SOLVED - 11/20/2023
(HAL-04) UNSETTLED RECORD MANIPULATION ALLOWS FREEZE BYPASS	Informational (0.0)	ACKNOWLEDGED
(HAL-05) TRANSFERRING UNSETTLED TOKENS CAN LEAD TO UNEXPECTED DEBT	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) DELETING INDEX 0 CORRUPTS DOUBLY LINKED LISTS - CRITICAL(10)

Description:

The `deleteAt` function in the `RecordsDequeLib` contract does not correctly handle the scenario when `rawIndex 0` is deleted. Index 0 is used as a sentinel node for managing the head of the linked list, and should never be deleted or altered during normal list operations. When deleting the last element of the list, the sentinel node at ‘record[0]‘‘, which should only be a reference and not an active node in the list, ends up pointing to the last deleted node. This can lead to an infinite loop in the linked list, as the sentinel’s prev field can be set to an active index. This corruption of the linked list can result in undefined behavior of the list operations and potentially lead to a denial of service due to an infinite loop or unintended reuse of “removed” indices.



Code Location:

Listing 1: contracts/util/RecordUtil.sol (Line 94)

```
94 function deleteAt(RecordsDeque storage rd, uint256 rawIndex)
↳ public {
95     uint256 next = rd.queue[rawIndex].next;
96     uint256 prev = rd.queue[rawIndex].prev;
97     if (rd.tail == rawIndex && rd.tail != rd.head) {
98         rd.tail = prev;
99     }
100    if (rd.head == rawIndex) {
101        rd.head = next;
102    }
103    rd.queue[prev].next = next;
```

```
104     rd.queue[next].prev = prev;
105     delete rd.queue[rawIndex];
106 }
```

Proof of Concept:

1. Four values are enqueued.
2. deleteAt is called for index 4, which will cause the index 0 to have its previous pointer set to the last active index, in this case 3.
3. deleteAt is called for index 1, which will cause the index 0 to have its next pointer set to 2.
4. deleteAt is called for index 0, which will cause both index 2 and 3 have both previous and next pointers pointing to the other index, which will effectively cause an infinite loop.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:C/D:N/Y:N/R:N/S:U (10)

Recommendation:

It is critically important to safeguard the sentinel node (index 0) from deletion. This can be achieved by adding a condition in the deleteAt function to check if rawIndex is 0 and, if so, revert the transaction or skip the deletion process for this specific index. Here is a potential safeguard to include in the deleteAt function:

Listing 2

```
1 function deleteAt(RecordsDeque storage rd, uint256 rawIndex)
↳ public {
2     require(rawIndex != 0, "Cannot delete sentinel node");
3
4     // ... existing logic ...
5 }
```

Moreover, to further solidify the integrity of the doubly linked list, it is advisable to adjust the deleteAt function to conditionally skip updating the next pointer of a node if the prev node is the sentinel (index 0). This prevents the sentinel node's next property from being incorrectly assigned during a delete operation, which is critical when the deletion occurs at the head of the list. Implementing this conditional logic will ensure that the sentinel node's role as a static reference point remains intact. The revised part of the deleteAt function could be implemented as follows:

Listing 3

```
function deleteAt(RecordsDeque storage rd, uint256 rawIndex)
↳ public {
    require(rawIndex != 0, "Cannot delete sentinel node");

    uint256 next = rd.queue[rawIndex].next;
    uint256 prev = rd.queue[rawIndex].prev;

    // Only update the previous node's 'next' if it's not the
↳ sentinel
    if (prev != 0) {
        rd.queue[prev].next = next;
    }

    // Only update the next node's 'prev' if it's not the sentinel
    if (next != 0) {
        rd.queue[next].prev = prev;
    }

    // ... existing logic ...
}
```

The conditional check for the sentinel index ensures that no operations on the next and prev pointers inadvertently create references to or from the sentinel node, besides the initial state meant for the start of the list. This exception in the logic will ensure that the sentinel node (index 0) remains an isolated reference point, which will effectively prevent the list from becoming corrupted or entering an infinite loop. This adjustment should be included along with the comprehensive testing

Remediation Plan:

SOLVED: The Circle team solved the issue in commit 78ea2d7 by preventing the deleteAt function from deleting the node at index 0, as well as updating only the previous node and the next node's prev values if they are not 0.

4.2 (HAL-02) ALLOWANCE DOES NOT DISTINGUISH BETWEEN SETTLED AND UNSETTLED TOKENS - MEDIUM (5.0)

Description:

The ERC20R contract distinguish between two types of balances:

- Settled tokens who have already passed the recovery time window and cannot be clawed back.
- Unsettled tokens who have not yet passed the time window and can be clawed back at any time during that window.

Therefore, settled tokens are considered to be more beneficial for the user because with the unsettled tokens there is the inherent risk for the tokens to be removed from their balance at any time.

However, the user, when setting allowance, cannot specify whether they want their balance to be removed from settled or unsettled, and it is left at the spender's discretion.

BVSS:

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

Code Location:

Listing 4: contracts/ERC20R/ERC20RWrapper.sol (Line 308)

```
301 function transferFrom(
302     address from,
303     address to,
304     uint256 value,
305     bool includeUnsettled
306 ) external virtual override returns (bool) {
307     address spender = msg.sender;
```

```
308         _spendAllowance(from, spender, value);
309         _transfer(from, to, value.toUint128(), includeUnsettled);
310         return true;
311     }
```

Recommendation:

Set two different mappings to distinguish between settled and unsettled allowance.

Remediation Plan:

RISK ACCEPTED: The Circle team accepted the risk of this finding because modifying the allowance logic could impact the user experience.

4.3 (HAL-03) UNHANDLED SELF-TRANSFER - LOW (2.5)

Description:

The _transfer function in the smart contract allows for transfers where the sender (from) and recipient (to) addresses are the same. This results in unnecessary gas consumption as the function performs all the operations of a transfer despite no actual transfer of funds occurring. Additionally, it creates an unsettled transfer record which then has to go through the recoverableWindow period before being considered settled, even though no asset movement occurs. This behavior increases the complexity of the contract and can lead to wasted resources and potential confusion.

Code Location:

Listing 5: contracts/ERC20R/ERC20RWrapper.sol

```
456 function _transfer(
457         address from,
458         address to,
459         uint128 amount,
460         bool includeUnsettled
461     ) internal virtual {
462         // there is no need to check if `from` is the zero address,
 ↳  because it cannot
463         // call transfer, nor would anyone have approval to spend
 ↳ from it.
464         _checkNotZeroAddress(to);
465         _clean(from);
466         _clean(to);
467         uint256 unsettledUsed = 0;
468
469         uint256 spendable = spendableBalanceOf(from,
 ↳ includeUnsettled);
470         if (spendable < amount) { InsufficientSpendableFunds(
471                 from,
472                 spendable,
473                 amount,
474                 includeUnsettled
```

```
475             );
476         }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

To mitigate the unnecessary operations and gas consumption, the `_transfer` function should be modified to include a check at the beginning to determine if the `from` and `to` addresses are identical. If the addresses are the same, the function can return early, avoiding unnecessary updates to the unsettled records and state changes.

Remediation Plan:

SOLVED: The Circle solved the issue in commit ae4f195 by reverting on self-transfer.

4.4 (HAL-04) UNSETTLED RECORD MANIPULATION ALLOWS FREEZE BYPASS - INFORMATIONAL (0.0)

Description:

The vulnerability exists in the interaction between the transfer and freeze functions of the contract. The transfer function includes the ability to transfer unsettled tokens by appending new unsettled records to _unsettledRecords. The freeze function, meant to be a governance action to immobilize funds, checks for the existence of a settlement time in the records and then freezes the specified assets. However, it's possible to perform a transfer of unsettled tokens that 'front runs' the freeze function, effectively bypassing the intended freeze action.

This occurs because a transfer allowing the user to shuffle their unsettled funds and modify the unsettled records right before a freeze action takes place. Consequently, the freeze function attempts to freeze a record that has been altered, leading to a failure in freezing the correct assets as intended by governance.

The Proof of Concept (POC) provided demonstrates how the user can perform a self-transfer of unsettled funds (wrapper.transfer(USER2, 50, true)) immediately before the governance action (wrapper.freeze(suspensions)), which results in the freeze action failing due to the RecordNotFound error.

Code Location:

Listing 6: contracts/ERC20R/ERC20RWrapper.sol (Line 456)

```
456 function _transfer(
457     address from,
458     address to,
459     uint128 amount,
460     bool includeUnsettled
```

```
461 ) internal virtual {
462     // there is no need to check if `from` is the zero address,
 ↳ because it cannot
463     // call transfer, nor would anyone have approval to spend from
 ↳  it.
464     _checkNotZeroAddress(to);
465     _clean(from);
466     _clean(to);
467     uint256 unsettledUsed = 0;
468 
469     uint256 spendable = spendableBalanceOf(from, includeUnsettled)
 ↳ ;
470     if (spendable < amount) {
471         revert InsufficientSpendableFunds(
472             from,
473             spendable,
474             amount,
475             includeUnsettled
476         );
477     }
478 
479     if (includeUnsettled) {
480         unsettledUsed = _spendUnsettled(from, amount);
481     }
482 
483     uint256 rawIndex = _unsettledRecords[to].enqueue(
484         amount,
485         block.timestamp + recoverableWindow
486     );
487 
488     _accountState[from].balance -= amount;
489     _accountState[to].balance += amount;
490     _accountState[to].nonce++;
491     _accountState[from].nonce++;
492     emit Transfer(
493         from,
494         to,
495         unsettledUsed,
496         amount - unsettledUsed,
497         rawIndex
498     );
499 }
```

Proof of Concept:

1. Alice has an amount of tokens obtained through illegitimate methods.
2. Governance wants to freeze Alice tokens, so they can investigate them.
3. Alice front-runs the transaction, transferring the tokens to another of her wallets.

```
Running 1 test for test/ERC20RWrapper.t.sol:ERC20RWrapperTest
[FAIL. Reason: RecordNotFound(0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e, 1)] test_freeze_bunny(
) (gas: 743211)
Logs:
  Transfering 50 from alice to bob
  Frontrunning started
  Transfering 50 unsettled from bob to bob
  Frontrunning finished

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 2.50ms

Failing tests:
Encountered 1 failing test in test/ERC20RWrapper.t.sol:ERC20RWrapperTest
[FAIL. Reason: RecordNotFound(0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e, 1)] test_freeze_bunny(
) (gas: 743211)
```

BVSS:

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

Recommendation:

The core issue stems from the ability to transfer unsettled tokens, which allows users to alter the state of unsettled records in a manner that undermines the integrity of governance actions, specifically the freezing of funds. To address this vulnerability, it is recommended to consider a significant refactoring of the codebase that targets the mechanisms by which unsettled tokens are handled. Here are two potential approaches to refactoring:

1. **Remove the Ability to Transfer Unsettled Tokens:** Refactor the code to remove the functionality that allows for the transfer of unsettled tokens. This will simplify the state management and remove the ability for users to manipulate unsettled records to bypass governance actions. This could be the most straightforward and secure

approach, ensuring that once tokens are unsettled, they remain with the original owner until settled.

2. **Change the Freeze Functionality to Target Sender's Unsettled Records:** Instead of freezing the funds at the destination, change the logic to allow freezing from the sender's side. This means that governance would place a freeze on the sender's account, locking the unsettled tokens before they are transferred. This would require a reworking of the internal data structures and logic to track and freeze tokens from the point of their origin.

This refactoring will be non-trivial and should be approached with a comprehensive strategy involving code reviews, testing, audits, and possibly formal verification, especially since it pertains to the fundamental mechanisms of fund transfer and governance.

Remediation plan:

ACKNOWLEDGED: Although this issue was initially rated **critical**, the Circle team explained:

> When the governance goes to recover the stolen tokens, it will execute the transaction using the flashbots RPC so that front-running is not possible".

With this statement, the severity was decreased to **informational** as it is no longer possible to execute front-running, and the attacker does not have time to act before the action is settled on chain.

4.5 (HAL-05) TRANSFERRING UNSETTLED TOKENS CAN LEAD TO UNEXPECTED DEBT - INFORMATIONAL (0.0)

Description:

The current ERC20R contract allows transferring unsettled tokens, this means that the tokens have not passed the recovery time window and can be clawed back any time from any of the users.

This can lead to increased debt for DeFi protocols and other dApps whose tokens can be removed at any time. For example, if a malicious user sends their unsettled balance to another legitimate user, and this legitimate user uses these tokens in a liquidity pool. It is possible that when the original malicious user is disputed, the tokens are clawed back from the liquidity pool, the legitimate user then would have the LP tokens he exchanged in return while the pool balance would have decreased.

This would ultimately lead to a debt for the liquidity pool.

Proof Of Concept:

1. Alice has 10,000 tokens that were obtained through illegitimate methods.
2. Alice uses those 10,000 tokens in a liquidity pool that supports ERC20R tokens.
3. Alice receives in exchange 10,000 LP Tokens.
4. Tokens are disputed, and they are clawed back from the liquidity pool.
5. Alice still has the LP tokens and can withdraw other tokens from the pool.

BVSS:

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

Recommendation:

Disallow unsettled balance transfers.

Remediation plan:

ACKNOWLEDGED: Although this issue was initially rated **critical**, the score was decreased because the Circle team considered that protocols using ERC20R tokens with unsettled balance should be fully aware of the risks involved.

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:

Slither results for ERC20RWrapper.sol	
Finding	**Impact**
ERC20RWrapper.constructor(string,string,uint256,address,address,uint16).governanceAddress_ (contracts/ERC20R/ERC20RWrapper.sol#106) lacks a zero-check on : - governanceAddress = governanceAddress_ (contracts/ERC20R/ERC20RWrapper.sol#111)	Low

Finding	Impact
Reentrancy in ERC20RWrapper.wrap(uint256) (contracts/ERC20R/ERC20RWrapper.sol#218-228): External calls: - SafeERC20.safeTransferFrom(baseERC20,msg.sender,address(this),amount) (contracts/ERC20R/ERC20RWrapper.sol#220-225) State variables written after the call(s): - _mint(msg.sender,amount) (contracts/ERC20R/ERC20RWrapper.sol#226) - _accountState[account].balance += amount.toUint128() (contracts/ERC20R/ERC20RWrapper.sol#446) - _accountState[account].cachedUnsettled = unsettled (contracts/ERC20R/ERC20RWrapper.sol#383) - _accountState[account].cachedUnsettledFrozen = unsettledFrozen (contracts/ERC20R/ERC20RWrapper.sol#389) - _mint(msg.sender,amount) (contracts/ERC20R/ERC20RWrapper.sol#226) - _totalSupply += amount (contracts/ERC20R/ERC20RWrapper.sol#445) - _mint(msg.sender,amount) (contracts/ERC20R/ERC20RWrapper.sol#226) - _unsettledRecords[account].cacheIndex = 0 (contracts/ERC20R/ERC20RWrapper.sol#385-388) - _unsettledRecords[account].cacheIndex = cacheIndex (contracts/ERC20R/ERC20RWrapper.sol#385-388)	Low
Reentrancy in ERC20RWrapper.wrap(uint256) (contracts/ERC20R/ERC20RWrapper.sol#218-228): External calls: - SafeERC20.safeTransferFrom(baseERC20,msg.sender,address(this),amount) (contracts/ERC20R/ERC20RWrapper.sol#220-225) Event emitted after the call(s): - Wrap(msg.sender,amount) (contracts/ERC20R/ERC20RWrapper.sol#227)	Low
Reentrancy in ERC20RWrapper.unwrap(uint256) (contracts/ERC20R/ERC20RWrapper.sol#234-244): External calls: - SafeERC20.safeTransfer(baseERC20,msg.sender,amount) (contracts/ERC20R/ERC20RWrapper.sol#242) Event emitted after the call(s): - Unwrap(msg.sender,msg.sender,amount) (contracts/ERC20R/ERC20RWrapper.sol#243)	Low

Finding	Impact
Reentrancy in ERC20RWrapper.unwrapTo(address,uint256) (contracts/ERC20R/ERC20RWrapper.sol#250-259): External calls: - SafeERC20.safeTransfer(baseERC20,to,amount) (contracts/ERC20R/ERC20RWrapper.sol#256) Event emitted after the call(s): - Unwrap(msg.sender,to,amount) (contracts/ERC20R/ERC20RWrapper.sol#258)	Low
ERC20RWrapper._clean(address) (contracts/ERC20R/ERC20RWrapper.sol#333-390) uses timestamp for comparisons Dangerous comparisons: - r.settlementTime > block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#346)	Low
ERC20RWrapper._spendUnsettled(address,uint128) (contracts/ERC20R/ERC20RWrapper.sol#509-559) uses timestamp for comparisons Dangerous comparisons: - r.settlementTime <= block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#522)	Low
ERC20RWrapper.freeze(Suspension[]) (contracts/ERC20R/ERC20RWrapper.sol#594-620) uses timestamp for comparisons Dangerous comparisons: - r.settlementTime <= block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#606)	Low
ERC20RWrapper._unsettledBalanceOf(address) (contracts/ERC20R/ERC20RWrapper.sol#396-434) uses timestamp for comparisons Dangerous comparisons: - r.settlementTime <= block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#404) - _unsettledRecords[account].getAt(cacheIndex).settlementTime <= block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#412-413) - r_scope_0.settlementTime > block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#427)	Low
ERC20RWrapper.closeCase(bool,address,Suspension[]) (contracts/ERC20R/ERC20RWrapper.sol#629-688) uses timestamp for comparisons Dangerous comparisons: - pastSettlement = r.settlementTime <= block.timestamp (contracts/ERC20R/ERC20RWrapper.sol#676)	Low
End of table for ERC20RWrapper.sol	

Results summary:

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

THANK YOU FOR CHOOSING

// HALBORN