SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Telcoin
Prepared by: Sherlock
Lead Security Expert: hyh
Dates Audited: March 14 - March 17, 2023
Prepared on: March 25, 2023

Introduction

Telcoin leverages blockchain technology to provide access to low-cost, high-quality decentralized financial products for every mobile phone user in the world.

Scope

All contract within the `contracts` directory are within the scope of this audit, with the exception of all contracts within the `contracts/test` directory. These contracts are used for hardhat's unit tests only. There is a slight caveat to this however. Though many of the contracts in this directory are for testing purposes only and are either not contracts that Telcoin has deployed or is responsible for, some are slightly augmented versions of other contracts inside the scope. The reason for this is to facilitate testing. Namely, the `RootBridgeRelay.sol` is an existing contract behind a proxy. Due to the nature of how this contract is currently in use, it makes more sense to have hardcoded values when switching between implementations, rather than reinitializing the contracts. In the test version, a constructor is used instead to allow for passing in the addresses of these generated dependencies. When using manual review, we suggest auditors stick to all non-test based contracts. For automated tools and unit tests are used, the reverse may be beneficial.

telcoin-audit @ 4197d2547699d910238f1782572f4a95a1c40a2a

- telcoin-audit/contracts/bridge/RootBridgeRelay.sol
- telcoin-audit/contracts/interfaces/IBlacklist.sol
- telcoin-audit/contracts/interfaces/IFeeBuyback.sol
- telcoin-audit/contracts/interfaces/IPOSBridge.sol
- telcoin-audit/contracts/interfaces/IPlugin.sol
- telcoin-audit/contracts/interfaces/IRootBridgeRelay.sol
- telcoin-audit/contracts/interfaces/ISimplePlugin.sol
- telcoin-audit/contracts/stablecoin/Stablecoin.sol
- telcoin-audit/contracts/staking/FeeBuyback.sol
- telcoin-audit/contracts/staking/StakingModule.sol
- telcoin-audit/contracts/util/TieredOwnership.sol

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
7	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hyh
volodya
spyrosonic10
banditx0x
Tricko
dipp
J4de
gmx
0xAgro
Inspex
jonatascm
jasonxiale
0xGoodess
ddimitrov22

Issue H-1: Rogue plugin can become unremovable and halt all staking and claiming

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/67

Found by

hyh

Summary

StakingModule's plugin that turned rogue can deny any attempts of its removal and can effectively stop the contract, disturbing the whole range of StakingModule operations.

I.e. if any plugin turns malicious due to bug or upgrade altering its functionality vs one that was in place as of the time of its addition to StakingModule, such malicious plugin can halt StakingModule and freeze all the funds staked.

Vulnerability Detail

The reason is removePlugins() having `require(IPlugin(plugin).deactivated())` condition, which success is required.

Suppose that a plugin turned malicious (as a result of a bug or by owner's intent via upgrade), begin to permanently return `false` for the `deactivated()` call.

And, for instance, it can simultaneously return `2**256-1` in claim() to overflow the sum and revert the `IPlugin(plugin).requiresNotification()` calls.

Impact

As all StakingModule operations will be frozen and funds withdrawal be unavailable in this scenario it will be permanent freeze of funds for all the stakers.

Code Snippet

If a plugin turns rogue:

It can return `2**256-1` in claim() to overflow the sum:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L353-L366

```
    function _claim(address account, address to, bytes calldata auxData) private returns (uint256) {
        // balance of `to` before claiming
        uint256 balBefore = IERC20Upgradeable(tel).balanceOf(to);

        // call claim on all plugins and count the total amount claimed
        uint256 total;
        bytes[] memory parsedAuxData = parseAuxData(auxData);
        for (uint256 i = 0; i < nPlugins; i++) {
>>          try IPlugin(plugins[i]).claim(account, to, parsedAuxData[i]) returns
 ↪  (uint256 xClaimed) {
                total += xClaimed;
            } catch  {
                emit PluginClaimFailed(plugins[i]);
            }
        }
```

This will block slash(), claim(), fullClaimAndExit(), partialClaimAndExit() functions.

Also, it can revert the `IPlugin(plugin).requiresNotification()` call:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L485-L498

```
    /// @dev Calls `notifyStakeChange` on all plugins that require it. This is
 ↪  done in case any given plugin needs to do some stuff when a user exits.
    /// @param account Account that is exiting
    function _notifyStakeChangeAllPlugins(address account, uint256 amountBefore,
 ↪  uint256 amountAfter) private {
        // loop over all plugins
        for (uint256 i = 0; i < nPlugins; i++) {
            // only notify if the plugin requires
>>          if (IPlugin(plugins[i]).requiresNotification()) {
                try IPlugin(plugins[i]).notifyStakeChange(account, amountBefore,
 ↪  amountAfter) {}
                catch {
                    emit StakeChangeNotificationFailed(plugins[i]);
                }
            }
        }
    }
```

It will also block stake(), partialExit(), exit(), and migration claimAndExitFor(), stakeFor() functions.

As all involve _notifyStakeChangeAllPlugins(), for example:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L573-L575

```
    function claimAndExitFor(address account, address to, bytes calldata
 ↪  auxData) external onlyRole(MIGRATOR_ROLE) nonReentrant returns (uint256,
 ↪  uint256) {
>>      return (_claim(account, to, auxData), _exit(account, to));
    }
```

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L388-L406

```
    function _exit(address account, address to) private returns (uint256) {
        uint256 stakedAmt = _stakes[account].latest();

>>      _partialExit(account, to, stakedAmt);

        return stakedAmt;
    }

    function _partialExit(address account, address to, uint256 exitAmount)
 ↪  private checkpointProtection(account) {
        if (exitAmount == 0) {
            return;
        }

        uint256 stakedAmt = _stakes[account].latest();

        require(stakedAmt >= exitAmount, "StakingMoudle: Cannot exit more than
 ↪  is staked");

        // notify plugins
>>      _notifyStakeChangeAllPlugins(account, stakedAmt, stakedAmt - exitAmount);
```

Tool used

Manual Review

Recommendation

Consider adding `force` option to removePlugin(), for example:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L542-L555

```
    /// @notice Removes a plugin
-   function removePlugin(uint256 index) external onlyRole(PLUGIN_EDITOR_ROLE) {
+   function removePlugin(uint256 index, bool force) external
 ↪  onlyRole(PLUGIN_EDITOR_ROLE) {
        address plugin = plugins[index];

-       require(IPlugin(plugin).deactivated(), "StakingModule::removePlugin:
↪   Plugin is not deactivated");
+       require(force || IPlugin(plugin).deactivated(),
↪   "StakingModule::removePlugin: Plugin is not deactivated");

        pluginsMapping[plugin] = false;
        plugins[index] = plugins[nPlugins - 1];
        pluginIndicies[plugins[index]] = index;
        plugins.pop();
        nPlugins--;

        emit PluginRemoved(plugin, nPlugins);
    }
```

Discussion

dmitriia

Looks ok

Issue M-1: Account that is affiliated with a plugin can sometimes evade slashing

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/62

Found by

hyh

Summary

Rogue plugin can be a big staker itself or can collide with one and allow such staker to evade slashing in a number of scenarios, i.e. reduce the probability of slashing execution.

Vulnerability Detail

In order to achieve that the `plugin` can behave otherwise normally in all regards, but on observing staked amount reduction for a specific `account` it can revert `IPlugin(plugin).requiresNotification()`.

If a given `account` also partially mitigate the existence of `withdrawalDelay > 0` with the periodic renewal of withdrawal requests (without using any, just to have some window available), the overall probability of it to be able to withdraw while `plugin` is still in the system is noticeable.

This way the overall scenario is:

1. `plugin` and `account` collide and set up the monitoring
2. SLASHER's slash() for `account` is front-run with `plugin` tx switching its state so it is now reverting on `IPlugin(plugin).requiresNotification()`
3. slash() is reverted this way, `plugin` switches to a normal state (it basically sandwiches slashing with two txs, own state change forth and back)
4. SLASHER investigate with PLUGIN_EDITOR who the reverting plugin is
5. Meanwhile withdraw window `account` has requested beforehand is approaching and if it occurs before PLUGIN_EDITOR removes a plugin (the ability to do so is an another issue, here we suppose it's fixed and plugin is removable) the `account` will be able to withdraw fully
6. `account` exit() executes as `plugin` is in normal state and doesn't block anything

Impact

`account` have some chance to evade the slashing, withdrawing the whole stake before slashing can occur.

With the growth of the protocol and increasing of the number of plugins this probability will gradually raise as volatile behavior of a particular plugin can be more tricky to identify which can provide enough time for an `account`.

The cost of being removed can be bearable for `plugin` provided that the `account` stake saved is big enough.

Code Snippet

Plugin can revert the `IPlugin(plugin).requiresNotification()` call:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L485-L498

```
    /// @dev Calls `notifyStakeChange` on all plugins that require it. This is
 ↪  done in case any given plugin needs to do some stuff when a user exits.
    /// @param account Account that is exiting
    function _notifyStakeChangeAllPlugins(address account, uint256 amountBefore,
 ↪  uint256 amountAfter) private {
        // loop over all plugins
        for (uint256 i = 0; i < nPlugins; i++) {
            // only notify if the plugin requires
>>          if (IPlugin(plugins[i]).requiresNotification()) {
                try IPlugin(plugins[i]).notifyStakeChange(account, amountBefore,
 ↪  amountAfter) {}
                catch {
                    emit StakeChangeNotificationFailed(plugins[i]);
                }
            }
        }
    }
```

It will prohibit slashing as slash() calls _claimAndExit() that invokes _notifyStakeChangeAllPlugins():

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L510-L513

```
function slash(address account, uint amount, address to, bytes calldata auxData)
 ↪  external onlyRole(SLASHER_ROLE) nonReentrant {
    _claimAndExit(account, amount, to, auxData);
    emit Slashed(account, amount);
}
```

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L460-L471

```
    function _claimAndExit(address account, uint256 amount, address to, bytes
↪   calldata auxData) private checkpointProtection(account) {
        require(amount <= balanceOf(account, auxData), "Account has insufficient
↪   balance");

        // keep track of initial stake
        uint256 oldStake = _stakes[account].latest();
        // xClaimed = total amount claimed
        uint256 xClaimed = _claim(account, address(this), auxData);

        uint256 newStake = oldStake + xClaimed - amount;

        // notify all plugins that account's stake has changed (if the plugin
↪   requires)
>>      _notifyStakeChangeAllPlugins(account, oldStake, newStake);
```

If there is a `withdrawalDelay` the `account` can routinely renew withdrawal requests:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L231-L236

```
function requestWithdrawal() external {
    require(withdrawalDelay > 0, "StakingModule: Withdrawal delay is 0");
    require(block.timestamp > withdrawalRequestTimestamps[msg.sender] +
↪   withdrawalDelay + withdrawalWindow, "StakingModule: Withdrawal already
↪   pending");

    withdrawalRequestTimestamps[msg.sender] = block.timestamp;
}
```

This way there is a chance that `account` will be able to withdraw while SLASHER locates the reason of blocking and communicate with PLUGIN_EDITOR in order to remove the `plugin`.

Tool used

Manual Review

Recommendation

Consider adding `try-catch` to the requiresNotification() call, for example:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L485-L498

```
   /// @dev Calls `notifyStakeChange` on all plugins that require it. This is
↪  done in case any given plugin needs to do some stuff when a user exits.
   /// @param account Account that is exiting
   function _notifyStakeChangeAllPlugins(address account, uint256 amountBefore,
↪  uint256 amountAfter) private {
       // loop over all plugins
       for (uint256 i = 0; i < nPlugins; i++) {
           // only notify if the plugin requires
-          if (IPlugin(plugins[i]).requiresNotification()) {
+          bool notificationRequired;
+          try IPlugin(plugins[i]).requiresNotification() returns (bool req) {
↪  notificationRequired = req; }
+          catch  { emit StakeChangeNotificationFailed(plugins[i]); }
+          if (notificationRequired) {
               try IPlugin(plugins[i]).notifyStakeChange(account, amountBefore,
↪  amountAfter) {}
               catch {
                   emit StakeChangeNotificationFailed(plugins[i]);
               }
           }
       }
   }
```

Discussion

amshirif

https://github.com/telcoin/telcoin-audit/pull/5

dmitriia

Looks ok

Issue M-2: `slash` calls can be blocked, allowing malicious users to bypass the slashing mechanism.

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/54

Found by

dipp, Tricko

Summary

A malicious user can block slashing by frontrunning `slash` with a call to `stake(1)` at the same block, allowing him to keep blocking calls to `slash` while waiting for his withdraw delay, effectively bypassing the slashing mechanism.

Vulnerability Detail

`StakingModule`'s `checkpointProtection` modifier reverts certain actions, like claims, if the accounts' stake was previously modified in the same block. A malicious user can exploit this to intentionally block calls to `slash`.

Consider the following scenario, where Alice has `SLASHER_ROLE` and Bob is the malicious user.

1. Alice calls `slash` on Bob's account.
2. Bob sees the transaction on the mempool and tries to frontrun it by staking 1 TEL. (See Proof of Concept section below for a simplified example of this scenario)

If Bob stake call is processed first (he can pay more gas to increase his odds of being placed before than Alice), his new stake is pushed to `_stakes[address(Bob)]`, and his latest checkpoint (`_stakes[address(Bob)]._checkpoints[numCheckpoints - 1]`) `blockNumber` field is updated to the current `block.number`. So when `slash` is being processed in the same block and calls internally `_claimAndExit` it will revert due to the `checkpointProtection` modifier check (See code snippet below).

```
modifier checkpointProtection(address account) {
    uint256 numCheckpoints = _stakes[account]._checkpoints.length;
    require(numCheckpoints == 0 || _stakes[account]._checkpoints[numCheckpoints
↪   - 1]._blockNumber != block.number, "StakingModule: Cannot exit in the same
↪   block as another stake or exit");
    _;
}
```

Bob can do this indefinitely, eventually becoming a gas war between Alice and Bob or until Alice tries to use Flashbots Protect or similar services to avoid the public mempool. More importantly, this can be leverage to block all `slash` attempts while waiting the time required to withdraw, so the malicious user could call `requestWithdrawal()`, then keep blocking all future `slash` calls while waiting for his `withdrawalDelay`, then proceed to withdraws his stake when `block.timestamp > withdrawalRequestTimestamps[msg.sender] + withdrawalDelay`. Therefore bypassing the slashing mechanism.

In this modified scenario

1. Alice calls `slash` on Bob's account.
2. Bob sees the transaction on the mempool and tries to frontrun it by staking 1 TEL.
3. Bob requests his withdraw (`requestWithdrawal()`)
4. Bob keeps monitoring the mempool for future calls to `slash` against his account, trying to frontrun each one of them.
5. When enough time has passed so that his withdraw is available, Bob calls `exit` or `fullClaimAndExit`

Impact

Slashing calls can be blocked by malicious user, allowing him to request his withdraw, wait until withdraw delay has passed (while blocking further calls to `slash`) and then withdraw his funds.

Classify this one as medium severity, because even though there are ways to avoid being frontrunned, like paying much more gas or using services like Flashbots Protect, none is certain to work because the malicious user can use the same methods to their advantage. And if the malicious user is successful, this would result in loss of funds to the protocol (i.e funds that should have been slashed, but user managed to withdraw them)

Proof of Concept

The POC below shows that staking prevents any future call to `slash` on the same block. To reproduce this POC just copy the code to a file on the test/ folder and run it.

```
const { expect } = require("chai")
const { ethers, upgrades } = require("hardhat")

const emptyBytes = []

describe("POC", () => {
  let deployer
  let alice
  let bob
  let telContract
  let stakingContract
  let SLASHER_ROLE

  beforeEach("setup", async () => {
    [deployer, alice, bob] = await ethers.getSigners()

    //Deployments
    const TELFactory = await ethers.getContractFactory("TestTelcoin", deployer)
    const StakingModuleFactory = await ethers.getContractFactory(
      "StakingModule",
      deployer
    )
    telContract = await TELFactory.deploy(deployer.address)
    await telContract.deployed()
    stakingContract = await upgrades.deployProxy(StakingModuleFactory, [
      telContract.address,
      3600,
      10
    ])

    //Grant SLASHER_ROLE to Alice
    SLASHER_ROLE = await stakingContract.SLASHER_ROLE()
    await stakingContract
      .connect(deployer)
      .grantRole(SLASHER_ROLE, alice.address)

    //Send some TEL tokens to Bob
    await telContract.connect(deployer).transfer(bob.address, 1)

    //Setup approvals
    await telContract
      .connect(bob)
      .approve(stakingContract.address, 1)
  })

  describe("POC", () => {
    it("should revert during slash", async () => {
      //Disable auto-mining and set interval to 0 necessary to guarantee both
    ↪ transactions
      //below are mined in the same block, reproducing the frontrunning scenario.
      await network.provider.send("evm_setAutomine", [false]);
      await network.provider.send("evm_setIntervalMining", [0]);

      //Bob stakes 1 TEL
      await stakingContract
        .connect(bob)
```

```
        .stake(1)

      //Turn on the auto-mining, so that after the next transaction is sent, the block is mined.
      await network.provider.send("evm_setAutomine", [true]);

      //Alice tries to slash Bob, but reverts.
      await expect(stakingContract
        .connect(alice)
        .slash(bob.address, 1, stakingContract.address, emptyBytes)).to.be.revertedWith(
          "StakingModule: Cannot exit in the same block as another stake or exit"
        )
    })
  })
})
```

Code Snippet

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L109-L113

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L510-L513

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L460-L483

Tool used

Manual Review

Recommendation

Consider implementing a specific version of `_claimAndExit` without the `checkpointProtection` modifier, to be used inside the `slash` function.

Discussion

amshirif

Two different issues are in the same PR because they both stem from the same modifier.

amshirif

https://github.com/telcoin/telcoin-audit/pull/6

dmitriia

Looks ok

Issue M-3: Front Run of addBlackList() function

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/43

Found by

0xAgro, J4de, gmx, Inspex

Summary

Front Run of addBlackList() function

Vulnerability Detail

Front running can be done either by sending a tx with a higher gas price (usually tx are ordered in a block by the gas price / total fee), or by paying an additional fee to the validator if they manage to run their tx without reverting (i.e. by sending additional ETH to block.coinbase, hoping validator will notice it).

Impact

Malicious user could listen the mempool in order to check if he sees a tx of blacklisting for his address , if it happens he could front run this tx by sending a tx with higher gas fee to transfer his funds to prevent them to be removed by removeBlackFunds() function

Code Snippet

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/stablecoin/Stablecoin.sol#L159

Tool used

Manual Review

Recommendation

Use the same mechanism as in StakingModule.sol to prevent user from withdrawing their funds if blacklisted so that front running won't be useful

Issue M-4: StakingModule's stakedByAt() can report erroneous values

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/36

Found by

hyh

Summary

stakedByAt() is vulnerable to producing manipulated readings when staking was performed more than once in the same block.

For example, if there were two stake() calls in one block, then stakedByAt() will report the state resulting from only the first one due to Checkpoints returning the value of the first checkpoint of the block.

Vulnerability Detail

Currently only exiting can't be carried out more than once in the same block, staking can happen more than once.

This will yield lower than actual stakedByAt() and balanceOfAt() readings whenever several staking calls happened in one block.

Impact

Erroneous readings can and most probably will impact downstream systems and can lead to their user's losses.

Reading other system's balance is a common component of decision making in a typical Vault contract. StakingModule can be a strategy therein and readings of the current holdings of the Vault will impact the course of its actions. Vault can have been depositing more than once in a block say as a result of actions of their users, for example it could been two deposits from different users in the same block, and parts of each of them was staked with StakingModule.

By having stakedByAt() reported value associated with the first deposit only, StakingModule biases the actions of the Vault, which can lead to losses for its users and then to removal of Telcoin integration, which is loss of market share that can later translates to Telcoin value.

Code Snippet

Historical requests used in stakedByAt() are vulnerable to stale Checkpoints readings:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L218-L223

```
/// @return Amount staked by an account at a specific block number excluding
↪  claimable yield.
/// @param account Account to query staked amount
/// @param blockNumber Block at which to query staked amount
function stakedByAt(address account, uint256 blockNumber) public view returns
↪  (uint256) {
    return _stakes[account].getAtBlock(blockNumber);
}
```

stakedByAt() is used for historical balance readings:

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L174-L176

```
function balanceOfAt(address account, uint256 blockNumber, bytes calldata
↪  auxData) external view returns (uint256) {
    return stakedByAt(account, blockNumber) + claimableAt(account, blockNumber,
↪  auxData);
}
```

checkpointProtection() is added to some functions (_partialExit() and _claimAndExit()), but _stake() is left unprotected:

https://github.com/sherlock-audit/2022-11-telcoin-judging/issues/83

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L109-L113

```
modifier checkpointProtection(address account) {
    uint256 numCheckpoints = _stakes[account]._checkpoints.length;
    require(numCheckpoints == 0 || _stakes[account]._checkpoints[numCheckpoints
↪  - 1]._blockNumber != block.number, "StakingModule: Cannot exit in the same
↪  block as another stake or exit");
    _;
}
```

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L429-L439

```
function _stake(address account, address from, uint256 amount) private {
    require(amount > 0, "Cannot stake 0");

    uint256 stakedBefore = _stakes[account].latest();
    uint256 stakedAfter = stakedBefore + amount;

    // notify plugins
    _notifyStakeChangeAllPlugins(account, stakedBefore, stakedAfter);

    // update _stakes
    _stakes[account].push(stakedAfter);
```

I.e. what currently implemented is a fix for the flash loan vector, but stakedByAt() readings can still be wrong.

Tool used

Manual Review

Recommendation

Consider adding the checkpointProtection() check to _stake():

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L429-L439

```
-    function _stake(address account, address from, uint256 amount) private {
+    function _stake(address account, address from, uint256 amount) private
↪    checkpointProtection(account) {
         require(amount > 0, "Cannot stake 0");

         uint256 stakedBefore = _stakes[account].latest();
         uint256 stakedAfter = stakedBefore + amount;

         // notify plugins
         _notifyStakeChangeAllPlugins(account, stakedBefore, stakedAfter);

         // update _stakes
         _stakes[account].push(stakedAfter);
```

This way both staking and unstaking will be restricted to once per block due to the usage of Checkpoints, which needs to be documented as a known limitation.

Discussion

amshirif

Two different issues are in the same PR because they both stem from the same modifier.

amshirif

https://github.com/telcoin/telcoin-audit/pull/6

dmitriia

Looks ok, but since `checkpointProtection` is removed it needs to be documented that stakedByAt() and balanceOfAt() return first known state instead of the final state of any block due to Checkpoints logic.

Issue M-5: Withdraw delay can be bypassed

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/23

Found by

banditx0x, spyrosonic10

Summary

StakingModule has core feature around staking, claim and withdraw. All these features has core and essential mechanism which is `delayed withdrawal`. In ideal scenario, user will stake X amount of token and will call `requestWithdrawal` when user want to withdraw his/her stake. `requestWithdrawal` will record user's request to withdraw and allow this user to withdraw only after `withdrawalDelay` is passed and during `withdrawalWindow` only. User can call `requestWithdrawal` in well advance before staking and this will allow user to bypass `withdrawalDelay`.

Vulnerability Detail

Withdraw locking/delaying is core feature of this contract and it can be exploited very easily.

User can call `requestWithdrawal` before staking tokens and this will set user's `withdrawalRequestTimestamps`. Once `withdrawalDelay` is passed user can easily stake and unstake without locking time.

One would suggest that easy fix is to check `staked > 0` during call to `requestWithdrawal` and that should solve this issue. No, it will not. Assume `staked>0` check is added in `requestWithdrawal` then user will stake 1 wei and call `requestWithdrawal` and this will result in almost same scenario.

Why would this happen? Because there is no relationship between stake and `withdrawalRequestTimestamps`.

Impact

`withdrawalDelay` can be bypassed

Code Snippet

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L231-L236

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/StakingModule.sol#L240-L246

POC

```
it.only("should bypass withdrawal delay", async () => {
  const delay = 60
  const window = 30
  // Set time delay
  await stakingContract.connect(deployer).grantRole(SLASHER_ROLE,
↪   slasher.address)
  await stakingContract.connect(slasher).setWithdrawDelayAndWindow(delay, window)
  await helpers.mine(1)
  // Request withdrawal
  await stakingContract.requestWithdrawal()
  // Increase time to pass timed delay
  await helpers.time.increase(delay)
  // Stake some tokens
  bobStakeTx3 = await stakingContract.connect(bob).stake(bobAmtStake)
  // Check there is non-zero staked balance
  expect(await stakingContract.balanceOf(bob.address, emptyBytes)).gt(0)
  // Claim and exit without wait.
  await stakingContract.fullClaimAndExit(emptyBytes)
})
```

Tool used

Manual Review

Recommendation

Consider resetting `withdrawalRequestTimestamps` when user stake any amount of token.

```
function stake(uint256 amount) external nonReentrant {
    _stake({
        account: msg.sender,
        from: msg.sender,
        amount: amount
    });
    withdrawalRequestTimestamps = 0;
}
```

Discussion

dmitriia

Looks ok

hrishibhat

Considering this issue as a valid medium. As it just bypasses the delay mechanism. No direct funds are lost or any other significant impact for the issue to classify as high.

Issue M-6: FeeBuyback.submit() method may fail if all allowance is not used by referral contract

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/22

Found by

0xGoodess, spyrosonic10, ddimitrov22, jonatascm, jasonxiale

Summary

Inside `submit()` method of `FeeBuyback.sol`, if token is `_telcoin` then it safeApprove to `_referral` contract. If `_referral` contract do not use all allowance then `submit()` method will fail in next call.

Vulnerability Detail

`SafeApprove()` method of library `SafeERC20Upgradeable` revert in following scenario.

```
require((value == 0) || (token.allowance(address(this), spender) == 0),
"SafeERC20: approve from non-zero to non-zero allowance");
```

Submit method is doing `safeApproval` of Telcoin to referral contract. If referral contract do not use full allowance then subsequent call to submit() method will fails because of `SafeERC20: approve from non-zero to non-zero allowance`. `FeeBuyback` contract should not trust or assume that referral contract will use all allowance. If it does not use all allowance in `increaseClaimableBy()` method then submit() method will revert in next call. This vulnerability exists at two places in `submit()` method. Link given in code snippet section.

Impact

Submit() call will fail until referral contract do not use all allowance.

Code Snippet

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/FeeBuyback.sol#L63-L64

https://github.com/sherlock-audit/2023-02-telcoin/blob/main/telcoin-audit/contracts/staking/FeeBuyback.sol#L63-L64

Tool used

Manual Review

Recommendation

Reset allowance to 0 before non-zero approval.

```
_telcoin.safeApprove(address(_referral), 0);
_telcoin.safeApprove(address(_referral), _telcoin.balanceOf(address(this)));
```

Discussion

amshirif

https://github.com/telcoin/telcoin-audit/pull/3

dmitriia

Looks ok

Issue M-7: `transferERCToBridge` will not work for some tokens that don't support approve `2**256 - 1` amount.

Source: https://github.com/sherlock-audit/2023-02-telcoin-judging/issues/1

Found by

volodya

Summary

`transferERCToBridge` will not work for some tokens that don't support approve `2**256 - 1` amount.

Vulnerability Detail

Impact

There are tokens that don't support approve spender `2**256 - 1` amount. So the transferERCToBridge will not work for some tokens like UNI who will revert when approve `2**256 - 1` amount. `Uni` is on the list that project promise to support

```
    function approve(address spender, uint rawAmount) external returns (bool) {
        uint96 amount;
        if (rawAmount == uint(-1)) {
            amount = uint96(-1);
        } else {
345:        amount = safe96(rawAmount, "Uni::approve: amount exceeds 96 bits");
        }

        allowances[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }
```

code#L345

Code Snippet

```
31:  uint256 constant public MAX_INT = 2**256 - 1;

70:  if (balance > IERC20Upgradeable(token).allowance(recipient,
 ↪  PREDICATE_ADDRESS)) {IERC20Upgradeable(token).safeApprove(PREDICATE_ADDRESS,
 ↪  MAX_INT);}
```

RootBridgeRelay.sol#L70

Tool used

Manual Review

Recommendation

```
if (balance > IERC20Upgradeable(token).allowance(recipient, PREDICATE_ADDRESS)) {
IERC20Upgradeable(token).safeApprove(PREDICATE_ADDRESS,
      balance - IERC20Upgradeable(token).allowance(recipient, PREDICATE_ADDRESS)
    );}

}
```

Discussion

dmitriia

Looks ok (PR#11)