Galaxy Fight Club

GCOIN

Smart Contract Audit Report

January 04, 2021

Introduction

1. About Galaxy Fight Club

Galaxy Fight Club (GFC) is the first-ever cross-IP, cross-platform (PC & mobile) PvP fighting game. They seek to pay homage to the battle-based video games of old while catering to modern, NFT-savvy players. The premise of GFC is simple: players log in using either their favorite third-party NFT (Bored Ape Yacht Club, CyberKongz, etc) or native character NFT, then wage war with other competitors to win rewards and earn payouts.

There is actually a precedent for this model: Nintendo's crossover fighting game Super Smash Bros. Of course, GFC's idea is a tad more ambitious since it seeks to bring characters from multiple rival platforms into the same ecosystem.
Players don't need to have a Genesis NFT or a third-party one to participate, of course: avatar-less competitors will be allocated a default fighter with base stats and a default, base-tier weapon when they sign up. If they perform well in the game, they can gradually increase their character's strength through buying/winning clothes and armor and obtaining a superior weapon, and leveraging the play to earn ecosystem.

Visit https://galaxyfightclub.com/ to know more about it.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The GFC team has provided the following doc for the purpose of audit:

1. https://whitepaper.galaxyfightclub.com/

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: Galaxy Fight Club
- Token Name: GCoin.sol,
- GitHub Address: https://github.com/dave-gfc/GCOIN-v2
- Commit Hash: ca6e1490fa540d5cc4eaa04f661fb943d703f805
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck, echinda

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

Admin/Owner Privileges can be misused either intentionally or unintentionally.

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	-
Closed	-	-	-

High Severity Issues

No issues were found.

Medium severity issues

No issues were found.

Low severity issues

No issues were found.

Recommendations/Informational

1. **Excessive use of digits can be avoided**
   Contract: GCOIN.sol
   Line no - 11
   Description:
   The above-mentioned lines have a large number of digits that makes it difficult to review and reduces the readability of the code.

   Recommendation:
   Ether Suffix can be used to symbolize the 10^18 zeros.

2. **External Visibility should be preferred**
   Contract: GCOIN.sol
   Line no: 25
   Explanation:
   Functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.

   Therefore, the following function must be marked as **external** within the GCOIN.sol contract:
   - mint()

```
24
25      function mint(address to, uint256 amount) public onlyMinter {
26          require(totalSupply() + amount <= TOKEN_MAX_CAP, "Exceeded TOKEN_MAX_CAP");
27          _mint(to, amount);
28      }
```

Recommendation:
If the **PUBLIC** visibility of the above-mentioned functions is not intended, then the **EXTERNAL** Visibility keyword should be preferred.

3. **Redundant comparison to boolean constants**
   Contract: TokenTimelock.sol
   Line no: 71, 88
   Description:
   Boolean constants can directly be used in conditional statements or require statements. Therefore, it's not considered a better practice to explicitly use TRUE or FALSE in the require statements.

```
83      function changeBeneficiary(uint256 id, address newBeneficiary) external {
84          // Check
85          VestingObj memory vestingObj = VestingInfo[id];
86          require(newBeneficiary != address(0), "New beneficiary cannot be address 0x0");
87          require(vestingObj.beneficiary == msg.sender, "You are not the beneficiary for this vestment");
88          require(vestingObj.redeemed == false, "This vestment have been redeemed");
89
```

   Recommendation:
   The equality to boolean constants could be removed from the above-mentioned line.

4. **Function initiates token transfer without checking the contract's balance.**
   Contract: TokenTimelock.sol
   Line no - 80
   Description:
   As per the current design of the redeem function, the vested token amount is transferred to the caller of the function at Line 80.
   This technically transfers the token from the contract to the user.

```
67      function redeem(uint256 id) external {
68          // Check
69          VestingObj memory vestingObj = VestingInfo[id];
70          require(vestingObj.beneficiary == msg.sender, "You are not the beneficiary for this vestment");
71          require(vestingObj.redeemed == false, "This vestment have been redeemed");
72          require(block.timestamp >= vestingObj.releaseTime, "Vestment is still locked");
73
74          // Effect
75          VestingInfo[id].redeemed = true;
76
77          emit VestmentRedeemed(VestingInfo[id]);
78
79          // Interact
80          IERC20(TOKEN).safeTransfer(msg.sender, vestingObj.amount);
81      }
```

   However, the function doesn't include any check to ensure whether or not the contract has the required amount of token balance to initiate this transfer.

While this doesn't break the intended behavior of the contract in any way, it leads to a scenario where inadequate error messages will be shown to the user if they try to withdraw an amount of token that is more than the contract actually holds.

Recommendation:
A require statement with an adequate error message could be included to ensure that the token transfer is only initiated if the contract has the required amount of balance.

Automated Audit Result

1. GCOIN.sol

```
Compiled with solc
Number of lines: 780 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 10 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 14
Number of informational issues: 13
Number of low issues: 0
Number of medium issues: 0
Number of high issues: 0

ERCs: ERC20

+-------+-------------+-------+--------------------+--------------+----------+
|  Name | # functions |  ERCS |     ERC20 info     | Complex code | Features |
+-------+-------------+-------+--------------------+--------------+----------+
| GCOIN |      49     | ERC20 |      Pausable      |      No      |          |
|       |             |       |     ∞ Minting      |              |          |
|       |             |       | Approve Race Cond. |              |          |
|       |             |       |                    |              |          |
+-------+-------------+-------+--------------------+--------------+----------+
contracts/GCOIN.sol analyzed (10 contracts)
```

2. MintOwnable.sol

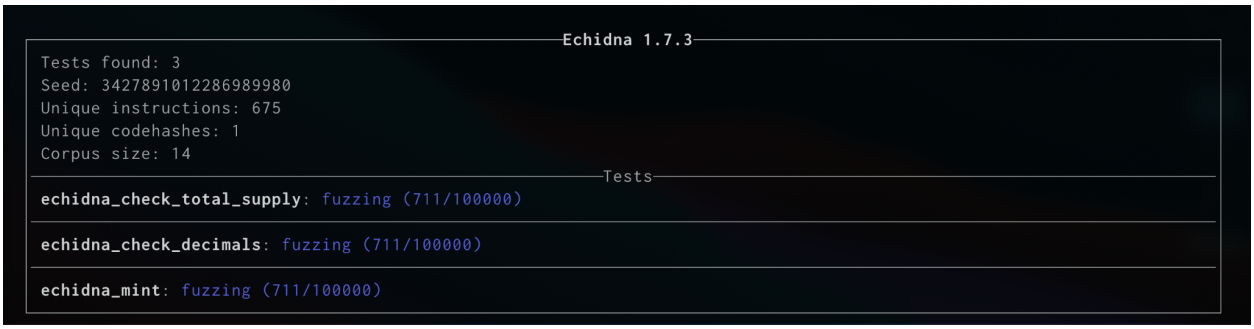
```
Compiled with solc
Number of lines: 29 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 1 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 0
Number of informational issues: 2
Number of low issues: 0
Number of medium issues: 0
Number of high issues: 0


+-------------+-------------+------+------------+--------------+----------+
|     Name    | # functions | ERCS | ERC20 info | Complex code | Features |
+-------------+-------------+------+------------+--------------+----------+
| MintOwnable |      3      |      |            |      No      |          |
+-------------+-------------+------+------------+--------------+----------+
contracts/MintOwnable.sol analyzed (1 contracts)
```

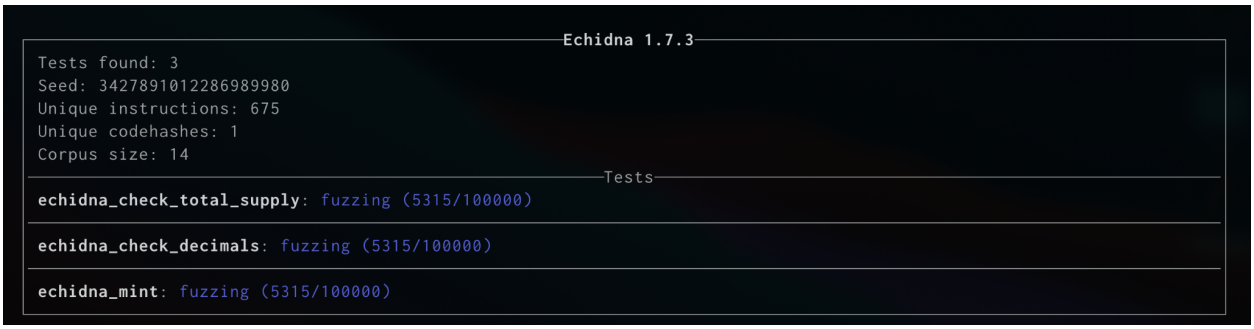
3. TokenTimelock.sol

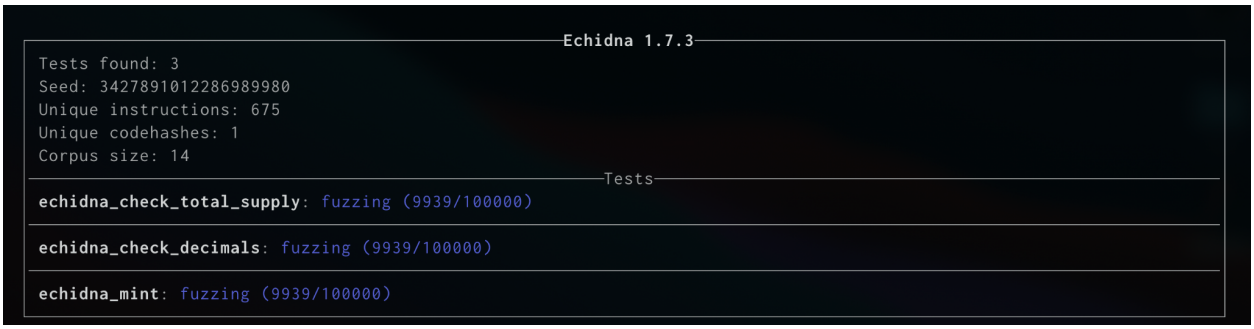
```
Compiled with solc
Number of lines: 586 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 6 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 2
Number of informational issues: 29
Number of low issues: 3
Number of medium issues: 0
Number of high issues: 0

ERCs: ERC20

+---------------+-------------+-------+--------------------+--------------+--------------------+
|      Name     | # functions |  ERCS |     ERC20 info     | Complex code |      Features      |
+---------------+-------------+-------+--------------------+--------------+--------------------+
|    Address    |      11     |       |                    |      No      |     Send ETH       |
|               |             |       |                    |              |   Delegatecall     |
|               |             |       |                    |              |     Assembly       |
|     IERC20    |      6      | ERC20 |     No Minting     |      No      |                    |
|               |             |       | Approve Race Cond. |              |                    |
|               |             |       |                    |              |                    |
|   SafeERC20   |      6      |       |                    |      No      |     Send ETH       |
|               |             |       |                    |              | Tokens interaction |
| TokenTimelock |      12     |       |                    |      No      | Tokens interaction |
+---------------+-------------+-------+--------------------+--------------+--------------------+
contracts/TokenTimelock.sol analyzed (6 contracts)
```

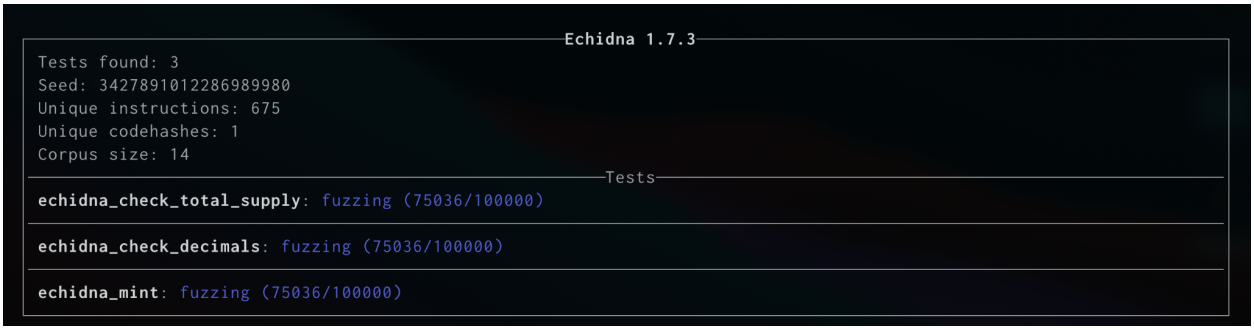
Fuzz Testing

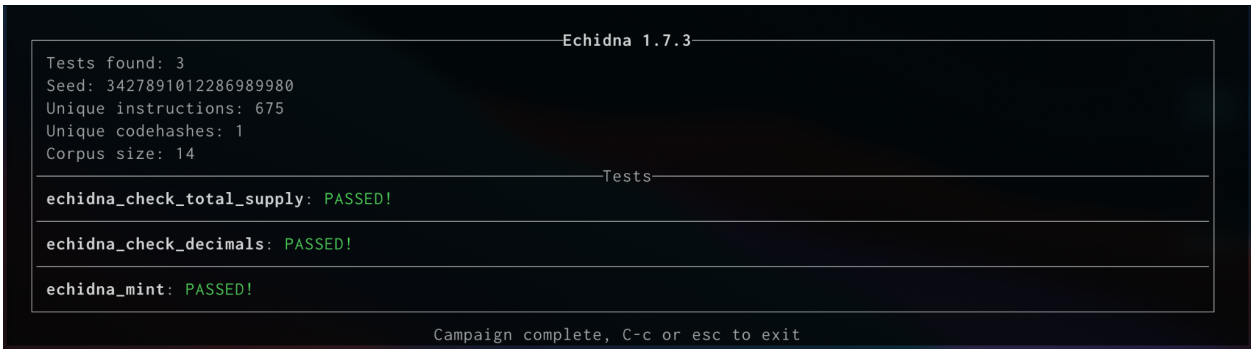
- totalSupply
  - ***PASS***
- decimals
  - ***PASS***
- Mint
  - ***PASS***
- setMinter
  - ***PASS***
- renouncedMinter
  - ***PASS***
- checkMinter
  - ***PASS***

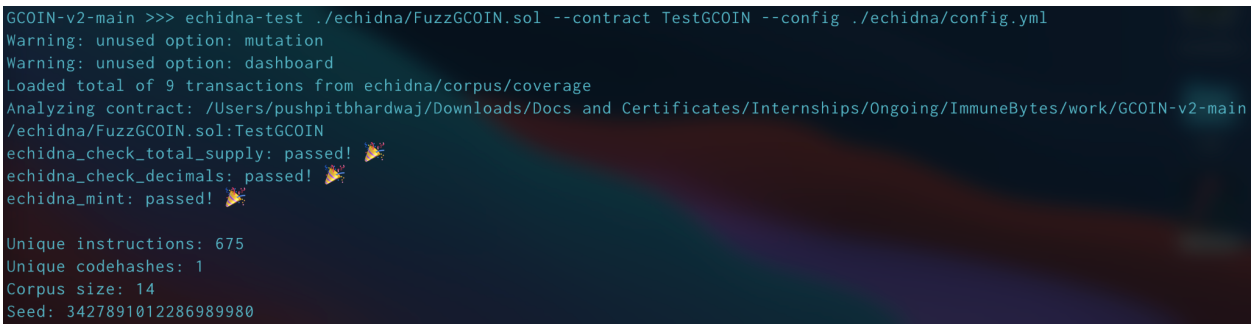


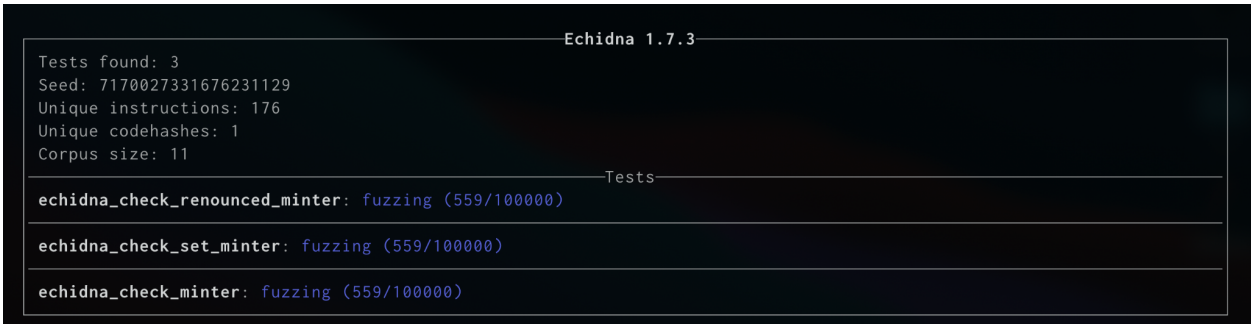


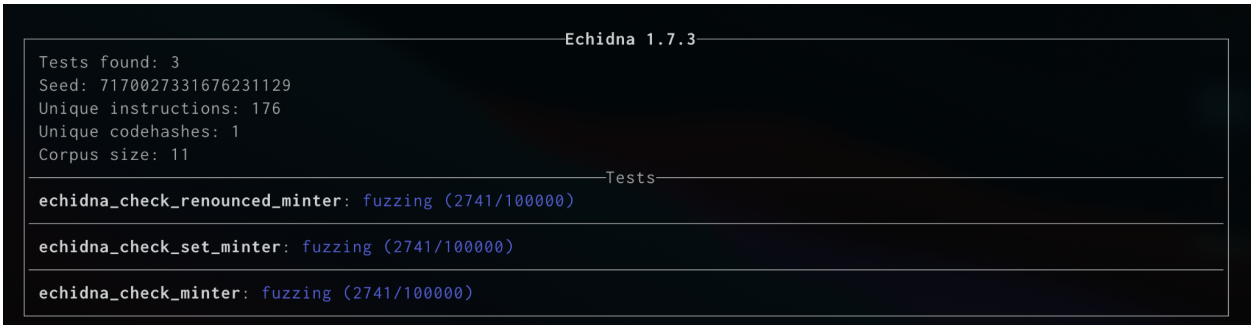


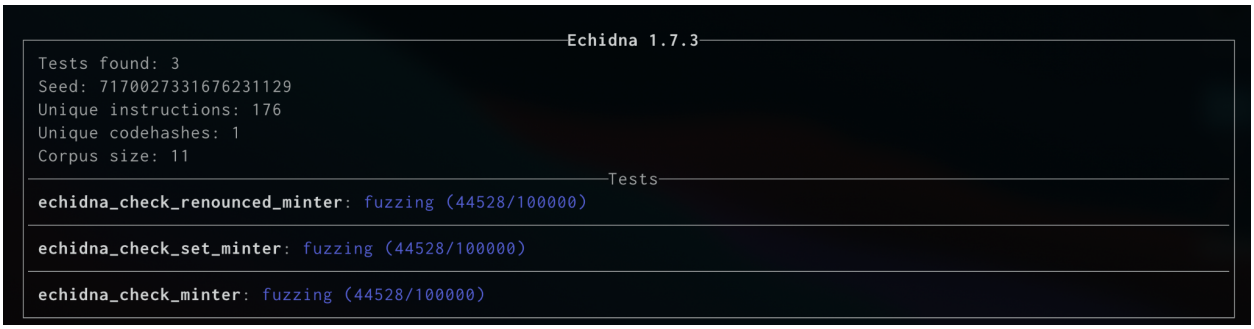


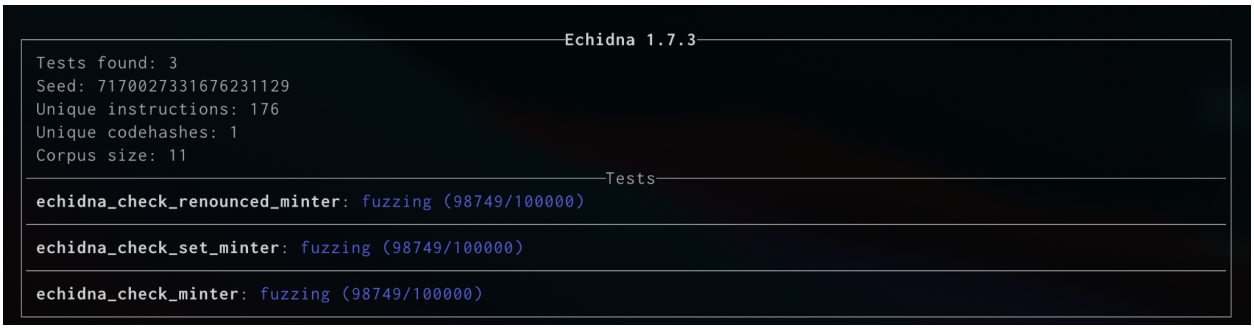


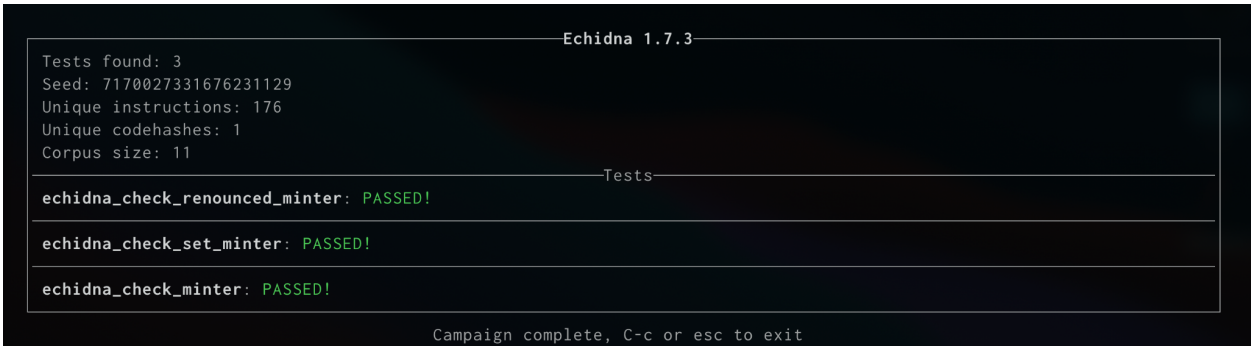













```
GCOIN-v2-main >>> echidna-test ./echidna/FuzzMintOwnable.sol --contract TestMintOwnable --config ./echidna/config.yml
Warning: unused option: mutation
Warning: unused option: dashboard
Loaded total of 9 transactions from echidna/corpus/coverage
Analyzing contract: /Users/pushpitbhardwaj/Downloads/Docs and Certificates/Internships/Ongoing/ImmuneBytes/work/GCOIN-v2-main
/echidna/FuzzMintOwnable.sol:TestMintOwnable
echidna_check_renounced_minter: passed! 🎉
echidna_check_set_minter: passed! 🎉
echidna_check_minter: passed! 🎉

Unique instructions: 176
Unique codehashes: 1
Corpus size: 11
Seed: 7170027331676231129
```

Concluding Remarks

While conducting the audits of the Galaxy Fight Club smart contracts, it was observed that the contracts contain no High, Medium and Low severity issues.

The recommendations given will improve the operations of the smart contract.

- ***The GFC contracts do not contain any issues.***

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the Galaxy Fight Club platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes