VIRTUAL CAIM

UP-DOWN

Smart Contract Review

Deliverable: Smart Contract Audit Report

Security Assessment

October 2025

Disclaimer

The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the Company. The content, conclusions, and recommendations set out in this publication are elaborated in the specific for only project.

Virtual Caim does not guarantee the authenticity of the project or organization or team of members that are connected/owner behind the project nor the accuracy of the data included in this study. All representations, warranties, undertakings, and guarantees relating to the report are excluded, particularly concerning – but not limited to – the qualities of the assessed projects and products. Neither the Company nor any person acting on the Company's behalf may be held responsible for the use that may be made of the information contained herein.

Virtual Caim retains the right to display audit reports and other content elements as examples of their work in their portfolio and as content features in other projects with protecting all security purposes of customers. The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities are fixed - upon a decision of the Customer.

Report Summary

Title	**UPDOWN Smart Contract Audit**		
Project Owner	UpDown Team		
Classification	Public		
Reviewed by	Virtual Caim Private Limited	**Review date**	16/10/2025
Approved by	Virtual Caim Private Limited	**Approval date**	31/10/2025
		N° Pages	**25**

Overview

Background

UpDown team requested Virtual Caim to perform an Extensive Smart Contract Audit of their 'UPDOWN' Smart Contract.

Project Dates

The following is the project schedule for this review and report:

- **October 16**: Smart Contract Review Started *(Completed)*
- **October 17**: Initial Delivery of Audit Findings *(Completed)*
- **October 31**: Final Delivery of Audit Findings *(Completed)*

Coverage

Target Specification and Revision

For this audit, we performed the project's basic research, investigation by discussing the details with the project owner and developer and then reviewed the smart contracts of UPDOWN.

The following documentation & repositories were considered in -scope for the review:

UPDOWN Smart Contract (Deployed on Mainnet)	NA

Introduction

Given the opportunity to review UPDOWN's Contracts related smart contract source code, we in the report summary our methodical approach to evaluate all potential common security issues in the smart contract implementation, expose possible semantic irregularities between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is ready to use after resolving the mentioned issues and done functional testing by owner/developer themselves, as there might be issues related to business logic, security or performance which only can found/understand by them.

About Audit

Item	Description
Issuer	UPDOWN
Symbol	NA
Decimals	NA
Token Supply	NA
Website	NA
Type	ERC-20
Language	Solidity
Audit Test Method	Whitebox Testing
Latest Audit Report	October 31, 2025

Ownership Privileges:

PredictionGame.sol

OWNER_ROLE:

1. The owner can pause and unpause the contract.

2. The owner can modify the treasury fee to a maximum of 10%

3. The owner can change the operator's address at any time without restrictions.

4. The owner can set interval seconds for round duration, but only when the contract is paused.

5. The owner can claim accumulated treasury funds for any payment token at any time.

6. The owner can authorize contract upgrades through the proxy pattern.

OPERATOR_ROLE:

1. The operator can initialize genesis rounds for new token pairs with initial price data.

2. The operator can execute rounds, which lock the current round, end the previous round, calculate rewards, and start a new round.

3. The operator can pause the contract but cannot unpause it (only the owner can unpause).

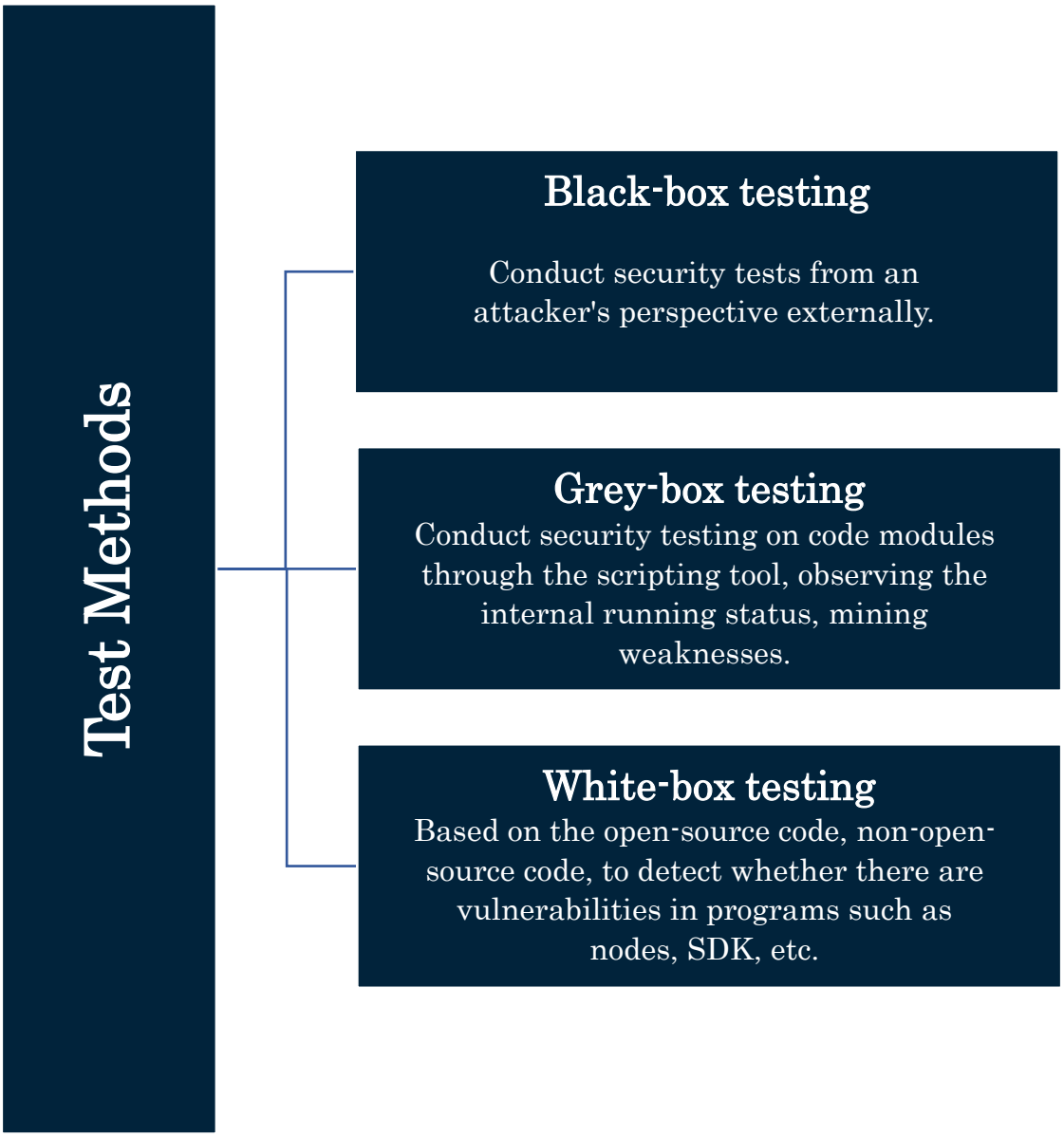
4. The operator has exclusive control over all price feeds and round progression, creating a single point of failure for price data integrity.

UPGRADER_ROLE (Proxy Admin):

1. The proxy admin can upgrade the contract implementation to completely new logic while preserving storage state.

2. The proxy admin can replace all contract behavior without user consent or time delays.

Test Method's Information



Vulnerability Severity Level Information

Level	Description
Critical	Critical severity vulnerabilities will have a significant effect on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.

List of Check Items

Basic Coding Bugs

- Constructor Mismatch
- Ownership Takeover
- Redundant Fallback Function
- Overflows & Underflows
- Reentrancy
- MONEY-Giving Bug
- Blackhole
- Unauthorized Self-Destruct
- Revert DoS
- Unchecked External Call
- Gasless Send
- Send Instead of Transfer
- Costly Loop
- (Unsafe) Use of Untrusted Libraries
- (Unsafe) Use of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

Semantic Consistency Checks

- Semantic Consistency Checks

Advanced DeFi Scrutiny

- Business Logics Review
- Functionality Checks
- Authentication Management
- Access Control & Authorization
- Oracle Security
- Digital Asset Escrow
- Kill-Switch Mechanism
- Operation Trails & Event Generation
- ERC20 Idiosyncrasies Handling
- Frontend-Contract Integration
- Deployment Consistency
- Holistic Risk Management

Additional Recommendations

- Avoiding Use of Variadic Byte Array
- Using Fixed Compiler Version
- Making Visibility Level Explicit
- Making Type Inference Explicit
- Adhering To Function Declaration Strictly
- Following Other Best Practices

Common Weakness Enumeration (CWE) Classifications Used in this Audit.

Configuration	• Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	• Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	• Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	• Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	• Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	• Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.

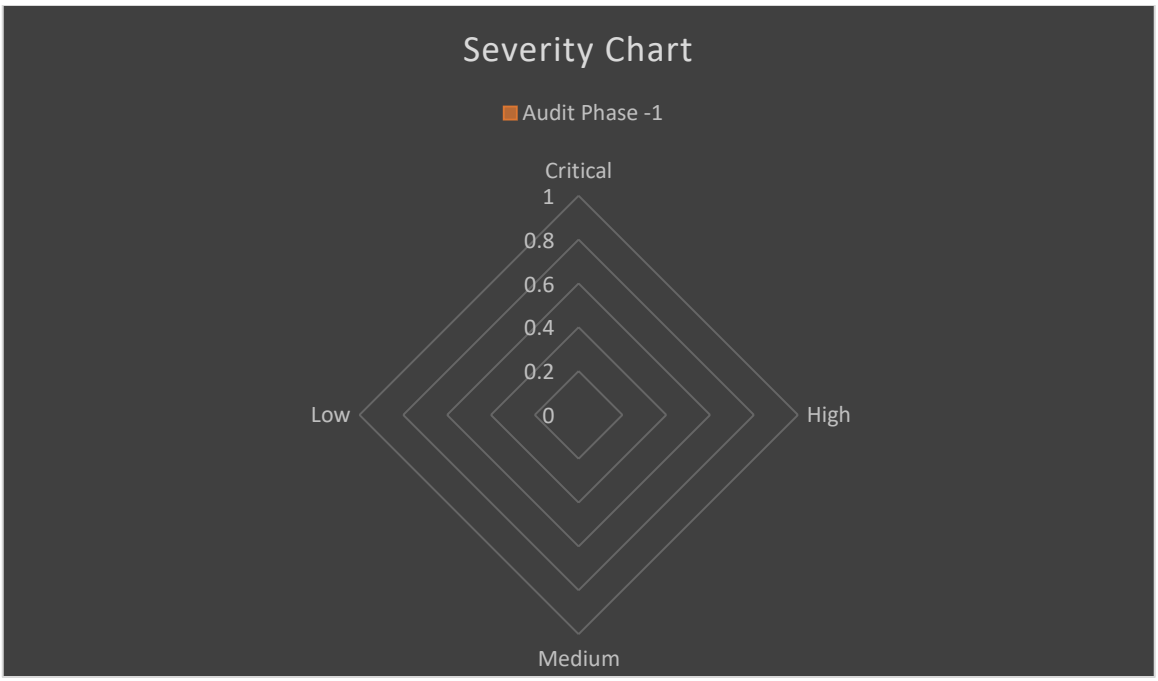
Resource Management	• Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	• Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	• Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	• Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	• Weaknesses in this category are related to improper use arguments or parameters within function calls.
Expression Issues	• Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	• Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an ex pilotable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

Findings

Summary

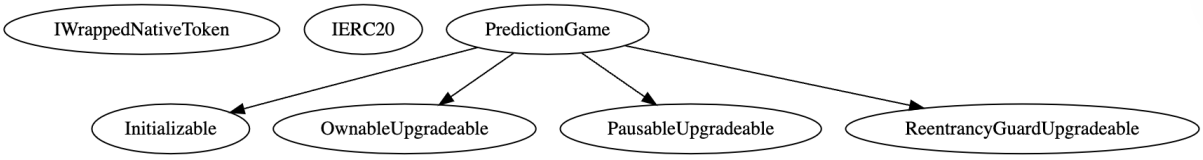
Here is a summary of our findings after scrutinizing the UPDOWN Smart Contract Review. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the Specific tools. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by tools. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	No. of Issues	Current Status
Critical	0	-
High	0	-
Medium	2	-
Low	4	-
Total	0 (Currently Open Issues)	



We have so far identified that there are potential issues with severity of **0 Critical, 0 High, 0 Medium, and 0 Low**. Overall, these smart contracts are well-designed and engineered.

Inheritance Tree



Detailed Results

Issues Checking Status

1. Missing Cooldown on Pause/Unpause Enables Rapid Toggle Attacks

- Severity: Medium
- Overview: Pause and unpause functions lack time-based restrictions, enabling unlimited state toggling without cooldown periods. This allows operators to repeatedly pause operations or selectively block transactions by pausing before execution. The absence of rate limiting creates potential for operational disruption and transaction manipulation without adequate controls.
- Status: Issue has been Resolved.
- POC:

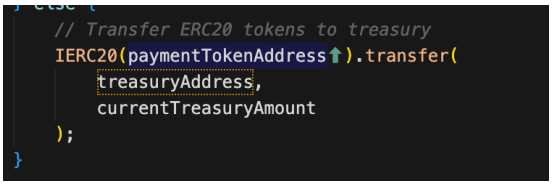
```
▷ Debug | ftrace | funcSig
function pause() external whenNotPaused onlyAdminOrOperator {
    _pause();
}

/**
 * @notice called by the admin to unpause, returns to normal state
 * Reset genesis state. Once paused, the rounds would need to be kickstarted by genesis
 */
▷ Debug | ftrace | funcSig
function unpause() external whenPaused onlyOwner {
    _unpause();
}
```

- Mitigation: Implement minimum time delay between pause state changes (e.g., 1 hour cooldown). Add daily toggle limit (e.g., maximum 3 pauses per day). Consider requiring timelock or multisig for pause operations to prevent instant manipulation.

2. **Unchecked ERC20 Transfer Return Value in Treasury Claim Function**

- Severity: Medium
- Overview: The Treasury balance is cleared before ERC20 transfer, and the transfer return value is not checked. Non-standard tokens that return false instead of reverting cause silent transfer failure while the treasury balance remains zero, resulting in permanent loss of protocol revenue with no recovery mechanism available.
- Status: Issue has been Resolved.
- POC:



- Mitigation: Use OpenZeppelin SafeERC20.safeTransfer() or add require() to check the return value. Move treasuryAmount clearing after successful transfer to follow the CEI pattern. This prevents silent failures with non-standard ERC20 tokens.

3. Duplicate User Rounds Entry Due to Zero-Value Bet Logic Flaw

- Severity: Low
- Overview: When minBetAmount is zero or uninitialized, users can place a zero-value bet followed by a positive bet in the same round. The condition betInfo.amount == betAmount evaluates true for both transactions, causing the epoch to be added twice to the userRounds array, creating duplicate entries and breaking user history tracking.
- Status: Issue has been Resolved.
- POC:

```
// Only add to userRounds if this is the first bet for this round
if (betInfo.amount == betAmount) {
    userRounds[paymentTokenAddress⬆][gameTokenAddress⬆][msg.sender].push(
        epoch⬆
    );
}
```

- Mitigation: Add explicit zero-amount check: if (betInfo.amount == betAmount && betAmount > 0). Alternatively, enforce minBetAmount > 0 in setMinBetAmount() initialization and validation to prevent zero-value bets entirely.

4. Missing Return Value Check on ERC20 Transfer in Claim Function

- Severity: Low
- Overview: The ERC20 transfer() call does not check the return value. While most standard tokens revert on failure, some non-standard ERC20 implementations (like certain USDT versions) return false instead of reverting. If transfer fails silently, users are marked as claimed but receive no tokens, resulting in permanent fund loss.
- Status: Issue has been Resolved.
- POC:

```
        // Transfer ERC20 tokens to user
        IERC20(paymentTokenAddress).transfer(msg.sender, reward);
    }
}
```

- Mitigation: Use OpenZeppelin's SafeERC20 library: using SafeERC20 for IERC20; IERC20(paymentTokenAddress).safeTransfer(msg.sender, reward);. Alternatively, add explicit check: require(IERC20(...).transfer(...), "Transfer failed"); to ensure transaction reverts on failure.

5. Unbounded Array Length Allows Gas Griefing in Claim Function

- Severity: Low
- Overview: The claim function accepts an arbitrary-length epochs array without validation. Users or attackers can submit extremely large arrays, causing transactions to exceed block gas limits, resulting in failed claims and wasted gas fees. While funds remain safe, legitimate users may be unable to claim rewards efficiently.
- Status: Issue has been Resolved.
- POC:

```
for (uint256 i = 0; i < epochs⬆.length; i++) {
    require(
        rounds[paymentTokenAddress⬆][gameTokenAddress⬆][epochs⬆[i]]
            .startTimestamp != 0,
        "Round has not started"
    );
    require(
        block.timestamp >
            rounds[paymentTokenAddress⬆][gameTokenAddress⬆][epochs⬆[i]]
                .endTimestamp,
        "Round has not ended"
    );
```

- Mitigation: Add array length validation before the loop: require(epochs.length > 0 && epochs. length <= 50, "Invalid epochs length");. Choose an appropriate maximum based on gas profiling. Consider implementing pagination or batch claiming with reasonable limits.

6. Missing Treasury Address Validation in Initialize Function

- Severity: Low
- Overview: The initialize function accepts the treasuryAddress parameter without validating it is not address(0). If the contract is initialized with zero address as the treasury, all protocol fees will be permanently sent to the burn address. No setTreasuryAddress function exists to fix this post-deployment, making it irreversible.
- Status: Issue has been Resolved.
- POC:

```
ftrace | funcSig
function initialize(
    address _operatorAddress,
    address _treasuryAddress,
    address _wrappedNativeToken,
    uint256 _intervalSeconds,
    uint256 _treasuryFee
) public initializer {
    require(_treasuryFee <= MAX_TREASURY_FEE, "Treasury fee too high");
    require(
        _wrappedNativeToken != address(0),
        "Wrapped token address cannot be zero"
    );
```

- Mitigation: Add validation before assignment: require(_treasuryAddress != address(0), "Invalid treasury address"); at line 216. This ensures the treasury address is valid during initialization, preventing permanent loss of all protocol revenue.

7. Floating Pragma Version

- Severity: Low
- Overview: All contracts use a floating pragma ^0.8.26 instead of locking to a specific compiler version. The contracts use pragma solidity ^0.8.26, which allows compilation with any version from 0.8.30 up to (but not including) 0.9.0. This creates deployment inconsistencies where different compiler versions may produce different bytecode, introducing subtle bugs or behavioral changes across deployments.
- Status: Issue has been Resolved.
- POC:

- Mitigation: Lock the pragma to a specific compiler version using pragma solidity 0.8.26; to ensure consistent compilation and deployment behavior across all environments.

Basic Coding Bugs

No.	Name	Description	Severity	Result
1.	Constructor Mismatch	Whether the contract name and its constructor are not identical to each other.	Critical	PASSED
2.	Ownership Takeover	Whether the set owner function is not protected.	Critical	PASSED
3.	Redundant Fallback Function	Whether the contract has a redundant fallback function.	Critical	PASSED
4.	Overflows & Underflows	Whether the contract has general overflow or underflow vulnerabilities	Critical	PASSED
5.	Reentrancy	Reentrancy is an issue when code can call back into your contract and change state, such as withdrawing ETHs	High	PASSED
6.	MONEY-Giving Bug	Whether the contract returns funds to an arbitrary address	High	PASSED
7.	Blackhole	Whether the contract locks ETH indefinitely: merely in without out	High	PASSED
8.	Unauthorized Self-Destruct	Whether the contract can be killed by any arbitrary address	Medium	PASSED
9.	Revert DoS	Whether the contract is vulnerable to DoS attack because	Medium	PASSED

		of unexpected revert		
10.	Unchecked External Call	Whether the contract has any external call without checking the return value	Medium	PASSED
11.	Gasless Send	Whether the contract is vulnerable to gasless send	Medium	PASSED
12.	Send Instead of Transfer	Whether the contract uses send instead of transfer	Medium	PASSED
13.	Costly Loop	Whether the contract has any costly loop which may lead to Out-Of-Gas exception	Medium	PASSED
14.	(Unsafe) Use of Untrusted Libraries	Whether the contract use any suspicious libraries	Medium	PASSED
15.	(Unsafe) Use of Predictable Variables	Whether the contract contains any randomness variable, but its value can be predicated	Medium	PASSED
16.	Transaction Ordering Dependence	Whether the final state of the contract depends on the order of the transactions	Medium	PASSED
17.	Deprecated Uses	Whether the contract use the deprecated tx.origin to perform the authorization	Medium	PASSED
18.	Semantic Consistency Checks	Whether the semantic of the white paper is different from the implementation of the contract	Critical	PASSED

Conclusion

In this audit, we thoroughly analyzed Dwin Intertrade Company Limited's 'UPDOWN' Smart Contract. The current code base is well organized and identified issues were resolved in this phase of testing of Smart Contract.

Meanwhile, we need to call attention to the fact that smart contracts are still in an early but exciting stage of development. To improve this report, we greatly appreciate any constructive feedback or suggestions on our methodology, audit findings, or potential gaps in scope/coverage.

About Virtual Caim

Just like our other parallel journey at eNebula Solution, we believe that people have a fundamental need for security and that the use of secure solutions enables every person to use the Internet and every other connected technology more freely. We aim to provide security consulting services to help others make their solutions more resistant to unauthorized access to data & inadvertent manipulation of the system. We support teams from the design phase through production to launch and surely after.

The Virtual Caim is specifically incorporated to handle all kind of Security related operations, our Highly Qualified and Certified security team has skills for reviewing coding languages like Solidity, Rust, Go, Python, Haskell, C, C++, and JavaScript for common security vulnerabilities & specific attack vectors. The team has been reviewing the implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code & networks and build custom tools as necessary.

Although we are a small team, we surely believe that we can have a momentous impact on the world by being translucent & open about the work we do.

For more information about our other security services and consulting,
please visit -- https://virtualcaim.com/
& Mail us at – audit@virtualcaim.com