



SMART CONTRACT AUDIT REPORT

for

NiftyConnect



Prepared By: Xiaomi Huang

PeckShield
May 3, 2022

Document Properties

Client	NiftyConnect
Title	Smart Contract Audit Report
Target	NiftyConnect
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 3, 2022	Xuxian Jiang	Final Release
1.0-rc	April 30, 2022	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About NiftyConnect	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Precise Memory Allocation in ExchangeCore::hashOrder()	11
3.2	Improved Logic in NiftyConnectExchange::buildCallDataInternal()	13
3.3	Accommodation of Non-ERC20-Compliant Tokens	14
3.4	Trust Issue of Admin Keys	15
3.5	Fork Resistant Domain Separator In ExchangeCore	17
4	Conclusion	19
	References	20

1 | Introduction

Given the opportunity to review the `NiftyConnect` design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of `NiftyConnect` can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About NiftyConnect

`NiftyConnect` is a decentralized exchange protocol forked from the `Wyvern Protocol`, with its exclusive support for `ERC721` and `ERC1155` assets. Through this protocol, users can buy and sell `ERC721` or `ERC1155` assets with lower exchange fee and without counterparty risk. All the market orders are on the chain which facilitates users to achieve the best market liquidity and benefits other market parties, including order relayers. Besides, this protocol natively supports collection-based orders and trait-based orders. The basic information of `NiftyConnect` is as follows:

Table 1.1: Basic Information of NiftyConnect

Item	Description
Name	NiftyConnect
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 3, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- <https://github.com/NiftyConnect/NiftyConnect-Contracts-Audit.git> (15f3565)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/NiftyConnect/NiftyConnect-Contracts-Audit.git> (047ed3e)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
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

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.




Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
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 of 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 exploitable 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `NiftyConnect` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	3	
Informational	1	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation,

Table 2.1: Key NiftyConnect Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Precise Memory Allocation in ExchangeCore::hashOrder()	Coding Practices	Resolved
PVE-002	Informational	Improved Logic in NiftyConnectExchange::buildCallDataInternal()	Business Logic	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Fork Resistant Domain Separator In ExchangeCore	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Precise Memory Allocation in ExchangeCore::hashOrder()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ExchangeCore
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

At the core of the NiftyConnect protocol is the ExchangeCore contract, which provides the essential functionality in validating and matching buy and sell orders. To facilitate the order management, the protocol computes the hash value of the given order. While analyzing the hash computation logic, we notice the current implementation can be improved.

In particular, we show below the related hashOrder() function. It implements a rather straightforward logic in allocating a memory buffer with 800 bytes, populating the order content in the allocated memory buffer, and then calculating its hash value with keccak256(). Our analysis shows that the order does not need to take 800 bytes. Instead, it only occupies 514 bytes with 8 Addresses, 4 Bytes32s, 7 UInts, and 2 UInt8Words or $(0x14 * 8) + (0x20 * 4) + (0x20 * 7) + 2 = 514!$

```

270     function hashOrder(Order memory order, uint nonce)
271     internal
272     pure
273     returns (bytes32 hash)
274     {
275         /* Unfortunately abi.encodePacked doesn't work here, stack size constraints. */
276         uint size = 800;
277         bytes memory array = new bytes(size);
278         uint index;
279         assembly {
280             index := add(array, 0x20)
281         }
282         index = ArrayUtils.unsafeWriteBytes32(index, _ORDER_TYPEHASH);

```

```

283     index = ArrayUtils.unsafeWriteAddressWord(index, order.exchange);
284     index = ArrayUtils.unsafeWriteAddressWord(index, order.maker);
285     index = ArrayUtils.unsafeWriteAddressWord(index, order.taker);
286     index = ArrayUtils.unsafeWriteAddressWord(index, order.makerRelayerFeeRecipient)
        ;
287     index = ArrayUtils.unsafeWriteAddressWord(index, order.takerRelayerFeeRecipient)
        ;
288     index = ArrayUtils.unsafeWriteUint8Word(index, uint8(order.side));
289     index = ArrayUtils.unsafeWriteUint8Word(index, uint8(order.saleKind));
290     index = ArrayUtils.unsafeWriteAddressWord(index, order.nftAddress);
291     index = ArrayUtils.unsafeWriteUint(index, order.tokenId);
292     index = ArrayUtils.unsafeWriteBytes32(index, keccak256(order.calldata));
293     index = ArrayUtils.unsafeWriteBytes32(index, keccak256(order.replacementPattern)
        );
294     index = ArrayUtils.unsafeWriteAddressWord(index, order.staticTarget);
295     index = ArrayUtils.unsafeWriteBytes32(index, keccak256(order.staticExtradata));
296     index = ArrayUtils.unsafeWriteAddressWord(index, order.paymentToken);
297     index = ArrayUtils.unsafeWriteUint(index, order.basePrice);
298     index = ArrayUtils.unsafeWriteUint(index, order.extra);
299     index = ArrayUtils.unsafeWriteUint(index, order.listingTime);
300     index = ArrayUtils.unsafeWriteUint(index, order.expirationTime);
301     index = ArrayUtils.unsafeWriteUint(index, order.salt);
302     index = ArrayUtils.unsafeWriteUint(index, nonce);
303     assembly {
304         hash := keccak256(add(array, 0x20), size)
305     }
306     return hash;
307 }

```

Listing 3.1: ExchangeCore::hashOrder()

Recommendation Improve the above hashOrder() routine for the improved memory buffer allocation to populate the order content.

Status The issue has been fixed by this commit: 047ed3e.

3.2 Improved Logic in NiftyConnectExchange::buildCallDataInternal()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: NiftyConnectExchange
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the NiftyConnect protocol, the NiftyConnectExchange contract is acting as the core exchange center. While examining one of its internal helper routines – buildCallDataInternal(), we notice its current implementation can be improved.

To elaborate, we show below its implementation. As the name indicates, this function facilitates to build the calldata, which will be used to instantiate an order with the required information. It comes to our attention that the requirement at the end, i.e., `require(merkleRoot!=bytes32(0x00))` (line 78), is redundant as there is an earlier requirement `require(uints[8]>=2&&merkleRoot!=bytes32(0x00))` (line 62). While the extra requirement does not impose any security implications, the redundant code can be safely improved.

```

50     function buildCallDataInternal(
51         address from,
52         address to,
53         address nftAddress,
54         uint[9] uints,
55         bytes32 merkleRoot)
56     internal view returns(bytes) {
57         bytes32[] memory merkleProof;
58         if (uints[8]==0) {
59             require(merkleRoot==bytes32(0x00), "invalid merkleRoot");
60             return buildCallData(uints[5],from,to,nftAddress,uints[6],uints[7],
                merkleRoot,merkleProof);
61         }
62         require(uints[8]>=2&&merkleRoot!=bytes32(0x00), "invalid merkle data");
63         uint256 merkleProofLength;
64         uint256 divResult = uints[8];
65         bool hasMod = false;
66         for(;divResult!=0;) {
67             uint256 tempDivResult = divResult/2;
68             if (SafeMath.mul(tempDivResult, 2)<divResult) {
69                 hasMod = true;
70             }
71             divResult=tempDivResult;
72             merkleProofLength++;
73         }

```

```

74     if (!hasMod) {
75         merkleProofLength--;
76     }
77     merkleProof = new bytes32[](merkleProofLength);
78     require(merkleRoot!=bytes32(0x00), "invalid merkleRoot");
79     return buildCallData(uints[5],from,to,nftAddress,uints[6],uints[7],merkleRoot,
        merkleProof);
80 }

```

Listing 3.2: NiftyConnectExchange::buildCallDataInternal()

Recommendation Remove the redundant requirement in the above `buildCallDataInternal()` function.

Status The issue has been fixed by this commit: [5c72bb7](#).

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: TokenTransferProxy
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of USDT's `transfer()`, the call will be unfortunately reverted.

```

126     function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
127         uint fee = (_value.mul(basisPointsRate)).div(10000);
128         if (fee > maximumFee) {
129             fee = maximumFee;
130         }
131         uint sendAmount = _value.sub(fee);
132         balances[msg.sender] = balances[msg.sender].sub(_value);
133         balances[_to] = balances[_to].add(sendAmount);
134         if (fee > 0) {
135             balances[owner] = balances[owner].add(fee);

```

```

136         Transfer(msg.sender, owner, fee);
137     }
138     Transfer(msg.sender, _to, sendAmount);
139 }

```

Listing 3.3: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In current implementation, if we examine the `TokenTransferProxy::transferFrom()` routine that is designed to transfer the funds after validating the caller. To accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()`, instead of `transferFrom()` (line 33).

```

28     function transferFrom(address token, address from, address to, uint amount)
29     public
30     returns (bool)
31     {
32         require(msg.sender==exchangeAddress, "not authorized");
33         return ERC20(token).transferFrom(from, to, amount);
34     }

```

Listing 3.4: TokenTransferProxy::transferFrom()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: 5c72bb7.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the NiftyConnect protocol, there are special administrative accounts, i.e., `owner` and `governor`. These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., configure royalty rates and set various fees). They also have the privilege to control or govern the

flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine their related privileged accesses in current protocol.

```

30     function changeExchangeFeeRate(uint newExchangeFeeRate)
31     public
32     onlyGovernor
33     {
34         exchangeFeeRate = newExchangeFeeRate;
35     }

36
37     /**
38      * @dev Change the taker fee paid to the taker relay (owner only)
39      * @param newTakerRelayerFeeShare New fee to set in basis points
40      * @param newMakerRelayerFeeShare New fee to set in basis points
41      * @param newProtocolFeeShare New fee to set in basis points
42      */
43     function changeTakerRelayerFeeShare(uint newTakerRelayerFeeShare, uint
         newMakerRelayerFeeShare, uint newProtocolFeeShare)
44     public
45     onlyGovernor
46     {
47         require(SafeMath.add(SafeMath.add(newTakerRelayerFeeShare,
         newMakerRelayerFeeShare), newProtocolFeeShare) == INVERSE_BASIS_POINT, "
         invalid new fee share");
48         takerRelayerFeeShare = newTakerRelayerFeeShare;
49         makerRelayerFeeShare = newMakerRelayerFeeShare;
50         protocolFeeShare = newProtocolFeeShare;
51     }

52
53     /**
54      * @dev Change the protocol fee recipient (owner only)
55      * @param newProtocolFeeRecipient New protocol fee recipient address
56      */
57     function changeProtocolFeeRecipient(address newProtocolFeeRecipient)
58     public
59     onlyOwner
60     {
61         protocolFeeRecipient = newProtocolFeeRecipient;
62     }

```

Listing 3.5: Example Privileged Operations in `ExchangeCore`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the administrative privileges to the intended DAO-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies the planned development of a DAO contract as the owner/governor.

3.5 Fork Resistant Domain Separator In ExchangeCore

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: ExchangeCore
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the NiftyConnect protocol has the NiftyConnectExchange contract acting as the core exchange center to support the most DEX functionalities. While examining the implementation of the inherited ExchangeCore contract, it comes to our attention that its current EIP_712 support can be improved.

```

129     constructor () public {
130         require(keccak256("EIP712Domain(string name,string version,uint256 chainId,
            address verifyingContract)") == _EIP_712_DOMAIN_TYPEHASH);
131         require(keccak256(bytes(name)) == _NAME_HASH);
132         require(keccak256(bytes(version)) == _VERSION_HASH);
133         require(keccak256("Order(address exchange,address maker,address taker,address
            makerRelayerFeeRecipient,address takerRelayerFeeRecipient,uint8 side,uint8
            saleKind,address nftAddress,uint tokenId,bytes32 merkleRoot,bytes calldata,
            bytes replacementPattern,address staticTarget,bytes staticExtradata,address
            paymentToken,uint256 basePrice,uint256 extra,uint256 listingTime,uint256
            expirationTime,uint256 salt,uint256 nonce)") == _ORDER_TYPEHASH);
134         DOMAIN_SEPARATOR = _deriveDomainSeparator();
135     }

```

Listing 3.6: ExchangeCore::constructor()

In particular, there is an important state variable DOMAIN_SEPARATOR, which is only assigned in the constructor() function (lines 134). The DOMAIN_SEPARATOR is used in the hashToSign() function, which is designed to calculate hash of the content the user needs to sign. When analyzing this hashToSign() function, we notice the current implementation can be improved by recalculating the value of DOMAIN_SEPARATOR in the hashToSign() function. Otherwise, if there is a hard-fork, a valid signature for one chain could be replayed on the other (since DOMAIN_SEPARATOR is immutable).

```

315     function hashToSign(Order memory order, uint nonce)
316     internal
317     pure
318     returns (bytes32)
319     {

```

```
320     return keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR, hashOrder(order,  
321         nonce)));  
    }
```

Listing 3.7: ExchangeCore::hashToSign()

Recommendation Recalculate the value of DOMAIN_SEPARATOR inside the hashToSign() function.

Status This issue has been resolved as the chain-forking of the deployed mainnet is considered unlikely.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `NiftyConnect` protocol, which is a decentralized exchange protocol to allow users to buy and sell `ERC721` or `ERC1155` assets with lower exchange fee and without counterparty risk. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.