

## SMART CONTRACT AUDIT REPORT

for

# JPYCoin

Prepared By: Yiqun Chen

PeckShield February 18, 2022

#### **Document Properties**

Client	JPYCoin	
Title	Smart Contract Audit Report	
Target	JPYC	
Version	1.0	
Author	Xuxian Jiang	
Auditors	Jing Wang, Xuxian Jiang	
Reviewed by	Yiqun Chen	
Approved by	Xuxian Jiang	
Classification	Public	

#### Version Info

	Description	Author	Date	Version
5	Final Release	Xuxian Jiang	February 18, 2022	1.0
didate	Release Candida	Xuxian Jiang	February 12, 2022	1.0-rc
dic	Release Candic	Xuxian Jiang	February 12, 2022	1.0-rc

#### Contact

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

Name	Yiqun Chen	
Phone	+86 183 5897 7782	
Email	contact@peckshield.com	

#### Contents

1	Intro	oduction	4
	1.1	About JPYCoin	4
	1.2	About PeckShield	5
	1.3	Methodology	5
	1.4	Disclaimer	7
2	Find	ings	8
	2.1	Summary	8
	2.2	Key Findings	9
3	ERC	20 Compliance Checks	10
4	Deta	ailed Results	13
	4.1	Fork-Compliant Domain Separator in JPYC	13
	4.2	Trust Issue Of Admin Roles	15
5	Con	clusion	18
Re	feren	ces	19

## 1 Introduction

Given the opportunity to review the design document and related source code of the JPYCoin protocol, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of some issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

#### 1.1 About JPYCoin

JPYCoin (JPYC) is a JPY-pegged stablecoin, legally dealt as a prepaid payment instrument in Japan. The audited token contract is an ERC20-compliant implementation deployable on Ethereum. It is proposed based on the observation that the JPY-pegged stablecoins have been far from practical use. JPYC is expected to expand the target of customers compared to ICB, which is for business use only, and streamline the cryptocurrency payment in buying and selling goods for the public use. The basic information of the audited contracts is as follows:

ltem	Description
lssuer	JPYCoin
Website	https://jpyc.jp/
Туре	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	February 18, 2022

Table 1.1: Basic Information of JPY	С
-------------------------------------	---

In the following, we show the Git repository and the commit hash value used in this audit:

• <a href="https://github.com/jcam1/JPYCv2.git">https://github.com/jcam1/JPYCv2.git</a> (fbe36c9)

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

• <a href="https://github.com/jcam1/JPYCv2.git">https://github.com/jcam1/JPYCv2.git</a> (2caec90)

#### 1.2 About PeckShield

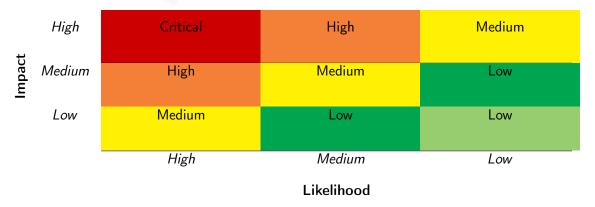
PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

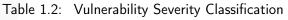
#### 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- <u>Likelihood</u> 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.





We perform the audit according to the following procedures:

- <u>Basic Coding Bugs</u>: 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.
- <u>ERC20 Compliance Checks</u>: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Category	Check Item	
	Constructor Mismatch	
	Ownership Takeover	
	Redundant Fallback Function	
-	Overflows & Underflows	
	Reentrancy	
	Money-Giving Bug	
	Blackhole	
	Unauthorized Self-Destruct	
Basic Coding Bugs	Revert DoS	
Dasic Counig Dugs	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	
	Approve / TransferFrom Race Condition	
ERC20 Compliance Checks	Compliance Checks (Section 3)	
	Avoiding Use of Variadic Byte Array	
	Using Fixed Compiler Version	
Additional Recommendations	Making Visibility Level Explicit	
	Making Type Inference Explicit	
	Adhering To Function Declaration Strictly	
	Following Other Best Practices	

Table 1.3: The Full List of Check Items

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 does not identify any issue, the contract is considered safe

regarding the check item. For any discovered issue, we might further 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.

#### 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.



## 2 Findings

#### 2.1 Summary

Here is a summary of our findings after analyzing the JPYC token contract. 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 logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings
Critical	0
High	0
Medium	1
Low	1
Informational	0
Total	2

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

#### 2.2 Key Findings

Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. Also, there is no critical or high severity issue, although the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 1 low-severity issue.

ID	Severity	Title	Category	Status
PVE-001	Low	Fork-Compliant Domain Separator in	Business Logic	Confirmed
PVE-002	Medium	Trust Issue Of Admin Roles	Security Features	Confirmed

Table 2.1: Key JPYC Audit Findings

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

# 3 ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

ltem	Description	Status	
name()	Is declared as a public view function	1	
Returns a string, for example Tether USD		1	
symbol()	Is declared as a public view function	1	
symbol()	Returns the symbol by which the token contract should be known, for	1	
	example "USDT". It is usually 3 or 4 characters in length		
decimals()	Is declared as a public view function	1	
uecimais()	Returns decimals, which refers to how divisible a token can be, from $0$	1	
	(not at all divisible) to 18 (pretty much continuous) and even higher if		
	required		
totalSupply()	Is declared as a public view function	1	
totalSupply()	Returns the number of total supplied tokens, including the total minted	1	
	tokens (minus the total burned tokens) ever since the deployment		
balanceOf()	Is declared as a public view function	1	
balanceOI()	Anyone can query any address' balance, as all data on the blockchain is	1	
	public		
allowance()	Is declared as a public view function	1	
anowance()	Returns the amount which the spender is still allowed to withdraw from	1	
	the owner		

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited token contract. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted

ERC20 specification.

Table 3.2:	Key State-Changing	Functions Defin	ned in The ERC20	Specification
------------	--------------------	-----------------	------------------	---------------

ltem	Description	Status
	Is declared as a public function	1
	Returns a boolean value which accurately reflects the token transfer status	$\checkmark$
transfor()	Reverts if the caller does not have enough tokens to spend	1
transfer()	Allows zero amount transfers	1
	Emits Transfer() event when tokens are transferred successfully (include 0	1
	amount transfers)	
	Reverts while transferring to zero address	1
	Is declared as a public function	1
	Returns a boolean value which accurately reflects the token transfer status	1
	Reverts if the spender does not have enough token allowances to spend	1
	Updates the spender's token allowances when tokens are transferred suc-	1
transferFrom()	cessfully	
	Reverts if the from address does not have enough tokens to spend	1
	Allows zero amount transfers	$\checkmark$
	Emits Transfer() event when tokens are transferred successfully (include 0	$\checkmark$
	amount transfers)	
	Reverts while transferring from zero address	1
	Reverts while transferring to zero address	$\checkmark$
	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	1
approve()	Emits Approval() event when tokens are approved successfully	1
	Reverts while approving to zero address	1
Transfor() autor	Is emitted when tokens are transferred, including zero value transfers	1
Transfer() event	Is emitted with the from address set to $address(0x0)$ when new tokens	1
	are generated	
Approve() event	Is emitted on any successful call to approve()	1

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on trans-	
	fer()/transferFrom() calls	
Rebasing	The balanceOf() function returns a re-based balance instead of the actual	_
	stored amount of tokens owned by the specific address	
Pausible	The token contract allows the owner or privileged users to pause the token	1
	transfers and other operations	
Blacklistable	The token contract allows the owner or privileged users to blacklist a	1
	specific address such that token transfers and other operations related to	
	that address are prohibited	
Mintable	The token contract allows the owner or privileged users to mint tokens to	1
	a specific address	
Burnable	The token contract allows the owner or privileged users to burn tokens of	1
	a specific address	

## 4 Detailed Results

#### 4.1 Fork-Compliant Domain Separator in JPYC

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High

- Target: FiatTokenV1/V2
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

#### Description

The FiatTokenV1 token contract strictly follows the widely-accepted ERC20 specification (Section 3). In the meantime, we notice the support of EIP-2612 with the permit() function that allows for approvals to be made via secp256k1 signatures. Interestingly, we notice the state variable DOMAIN\_SEPARATOR is initialized once inside the initialize() function (line 105).

68	function initialize(
69	string memory tokenName,
70	string memory tokenSymbol,
71	string memory tokenCurrency,
72	uint8 tokenDecimals,
73	address newMasterMinter,
74	address newPauser,
75	address newBlacklister,
76	address newOwner
77	) public {
78	<pre>require(!initialized, "FiatToken: contract is already initialized");</pre>
79	require(
80	<pre>newMasterMinter != address(0),</pre>
81	"FiatToken: new masterMinter is the zero address"
82	);
83	require(
84	<pre>newPauser != address(0),</pre>
85	"FiatToken: new pauser is the zero address"
86	);
87	require(

```
88
                 newBlacklister != address(0),
                 "FiatToken: new blacklister is the zero address"
89
90
             );
91
             require(
92
                 newOwner != address(0),
93
                 "FiatToken: new owner is the zero address"
94
             );
95
96
             name = tokenName;
97
             symbol = tokenSymbol;
98
             currency = tokenCurrency;
99
             decimals = tokenDecimals;
100
             masterMinter = newMasterMinter;
101
             pauser = newPauser;
102
             blacklister = newBlacklister;
103
             _transferOwnership(newOwner);
104
             blacklisted[address(this)] = true;
105
             DOMAIN_SEPARATOR = EIP712.makeDomainSeparator(name, "1");
106
             initialized = true;
107
```

Listing 4.1: FiatTokenV1::initialize()

The DOMAIN\_SEPARATOR is used in the permit() function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this permit() routine, we realize the current implementation needs to be improved by recalculating the value of DOMAIN\_SEPARATOR inside the permit() function, for the very purpose of preventing crosschain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed DOMAIN\_SEPARATOR, a valid signature for one chain could be replayed on the other.

```
552
         function permit(
553
             address owner,
             address spender,
554
555
             uint256 value,
556
             uint256 deadline,
557
             uint8 v,
558
             bytes32 r,
559
             bytes32 s
560
         ) external whenNotPaused notBlacklisted(owner) notBlacklisted(spender) {
561
             _permit(owner, spender, value, deadline, v, r, s);
562
         }
563
564
         function _permit(
565
             address owner,
566
             address spender,
567
             uint256 value,
568
             uint256 deadline,
569
             uint8 v,
570
             bytes32 r,
571
             bytes32 s
572
         ) internal {
```

```
573
             require(deadline >= block.timestamp, "EIP2612: permit is expired");
574
575
             bytes memory data = abi.encode(
576
                 PERMIT_TYPEHASH,
577
                 owner,
578
                 spender,
579
                 value,
580
                 _permitNonces[owner]++,
581
                 deadline
             );
582
583
             require(
584
                 EIP712.recover(DOMAIN_SEPARATOR, v, r, s, data) == owner,
585
                 "EIP2612: invalid signature"
586
             );
587
588
             _approve(owner, spender, value);
589
```

Listing 4.2: FiatTokenV1::permit()

**Recommendation** Recalculate the value of DOMAIN\_SEPARATOR inside the permit() function. Note this issue is applicable to a number of functions (in both FiatTokenV1 and FiatTokenV2 contracts) that uses the DOMAIN\_SEPARATOR value, including transferWithAuthorization(), receiveWithAuthorization(), and cancelAuthorization().

Status The issue has been confirmed.

#### 4.2 Trust Issue Of Admin Roles

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium

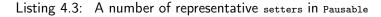
- Target: Multiple Contracts
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

#### Description

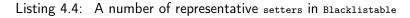
In JPYC, there are privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., privileged account management, account blacklisting/whitelisting, and token minting). In the following, we show the representative functions potentially affected by the privileged accounts.

67 /\*\*
68 \* @dev called by the owner to pause, triggers stopped state
69 \*/
70 function pause() external onlyPauser {

```
71
            paused = true;
72
            emit Pause();
73
        }
74
75
        /**
76
        * @dev called by the owner to unpause, returns to normal state
77
        */
78
        function unpause() external onlyPauser {
79
            paused = false;
80
            emit Unpause();
81
        }
82
83
        /**
84
        * @dev update the pauser role
85
         */
86
        function updatePauser(address _newPauser) external onlyOwner {
87
            require(
88
                _newPauser != address(0),
89
                "Pausable: new pauser is the zero address"
90
            );
91
            pauser = _newPauser;
92
            emit PauserChanged(pauser);
93
```



```
72
        /**
73
         * @dev Adds account to blacklist
74
         * Cparam _account The address to blacklist
75
        */
        function blacklist(address _account) external onlyBlacklister {
76
77
            blacklisted[_account] = true;
78
            emit Blacklisted(_account);
79
        }
80
81
        /**
82
        * @dev Removes account from blacklist
83
         * @param _account The address to remove from the blacklist
84
        */
85
        function unBlacklist(address _account) external onlyBlacklister {
86
            blacklisted[_account] = false;
87
            emit UnBlacklisted(_account);
88
        3
```



Moreover, it should be noted that current contracts are deployed behind a proxy and there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. At the same time the extra power to the owner may also be a counter-party risk to the token 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 token users.

**Recommendation** Make the list of extra privileges granted to privileged accounts explicit to JPYC users.

Status The issue has been confirmed.



## 5 Conclusion

In this security audit, we have examined the JPYCoin design and implementation. The smart contracts being audited are a JPY-pegged stablecoin, legally dealt as a prepaid payment instrument in Japan. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities and found no issue in these areas. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified three medium/low severity issues that were promptly confirmed and addressed by the team. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.



### References

- [1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.
- [2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/ definitions/841.html.
- [3] MITRE. CWE CATEGORY: 7PK Security Features. https://cwe.mitre.org/data/definitions/ 254.html.
- [4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840. html.
- [5] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP\_Risk\_Rating\_ Methodology.
- [6] PeckShield. PeckShield Inc. https://www.peckshield.com.