SMART CONTRACT AUDIT REPORT

for

Jump.Fun

Prepared By: Xiaomi Huang

PeckShield

November 17, 2024

Document Properties

Client	Jump.Fun
Title	Smart Contract Audit Report
Target	Jump.Fun
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 17, 2024	Xuxian Jiang	Final Release
1.0-rc	November 16, 2024	Xuxian Jiang	Release Candidate #1

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 Jump.Fun . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 6

2 Findings 9

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results 11

3.1 Improper Fee Collection During Token Selling . . . 11

3.2 Revisited Token Claim Logic in JUMPTreasury . . . 12

3.3 Improper Token Recovery Logic in JUMPTreasury . . . 13

3.4 Accommodation of Non-ERC20-Compliant Tokens . . . 14

3.5 Trust Issue of Admin Keys . . . 16

4 Conclusion 18

References 19

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Jump.Fun protocol, 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 version of smart contracts 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 Jump.Fun

Table 1.1: Basic Information of The Jump.Fun

Item	Description
Name	Jump.Fun
Website	https://jumpdotfun.gitbook.io/jump.fun
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 17, 2024

Jump.Fun is a next-level memecoin launchpad with one-click deployment, PvP liquidity war mode, and free 10 ETH liquidity loan on Uniswap. Through innovative features such as free liquidity loans and LiqWar mode, Jump.fun is significantly improving the odds for players to successfully catch pumps. Additionally, $JUMP token enables community ownership via fair launch guaranteeing same access for everyone. All platform revenue accrues back to $JUMP. Users can donate to get $JUMP as rewards and at the same time get a chance to get a share of the Fomo3D prize pool. Donated funds will be used to provide all creators with initial liquidity loans and a reward mechanism for community growth. All creators can launch tokens on Jump.fun's platform with a single click without permission, solving the problems of long fundraising time and high ETH fees for pre-market trading.

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

- https://github.com/JUMPFUNDev/JUMPFUN_Audit.git (e57e36d)

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

- https://github.com/JUMPFUNDev/JUMPFUN_Audit.git (412aceb3)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

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 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 1.3: 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

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 design and implementation of the `Jump.Fun` protocol. 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	2	■■
Informational	0	
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 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Jump.Fun Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Fee Collection During Token Selling	Business Logic	Resolved
PVE-002	Low	Revisited Token Claim Logic in JUMPTreasury	Business Logic	Resolved
PVE-003	Medium	Improper Token Recovery Logic in JUMPTreasury	Business Logic	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	Medium	Trust Issue on Admin Keys	Security Features	Mitigated

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 details.

3 | Detailed Results

3.1 Improper Fee Collection During Token Selling

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `JUMPERCFactory`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

`Jump.Fun` is a launchpad with its core `JUMPERCFactory` contract to deploy their own `memecoins`. These coins have the tax feature to prevent possible `MEV` issues. While reviewing the tax collection logic, we notice current implementation should be improved.

In the following, we show below the code snippet of the related `__handleTokenTransfer()` routine. As the name indicates, this routine handles the actual token transfer and collects tax if necessary. However, we notice the condition to determine whether a token is being sold should be revisited. Currently, the sell condition is `(feeRation >0)&& isPair(to)&& !isRouter(to)&& balanceOf(to)> 0` (line 624), which should be revised as `(feeRation >0)&& isPair(to)&& !isRouter(from)&& balanceOf(to)> 0`. In the meantime, we should point out that the created token can always be traded in other platforms with the possibility of bypassing the built-in taxes.

```
624        if((feeRation >0) && isPair(to) && !isRouter(to) && balanceOf(to) > 0){ //sell
625            if((amount > totalSupply() * whaleRate / basePoint) && (feeRation == 100)){
626                revert('Amount too Big');
627            }
628            uint256 feeAmount = amount * feeRation / basePoint;
629            _balances[address(this)] += feeAmount;
630            _balances[to] += (amount - feeAmount);
631            tradeCnt = tradeCnt + 1;
632            side = 2;
633
634            emit Transfer(from, to, amount - feeAmount);
635            emit Transfer(from, address(this), feeAmount);
```

```
            emit Trade(tx.origin, to, side, amount, feeAmount,block.timestamp);
        }
```

Listing 3.1: `JUMPERCFactory::__handleTokenTransfer()`

Recommendation Improve the above token-selling check logic.

Status This issue has been fixed by the following commit: `79c7b07`.

3.2 Revisited Token Claim Logic in JUMPTreasury

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `JUMPTreasury`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Jump.Fun` has its own protocol token, which can be minted via the `donate` feature. While reviewing the logic to calculate the amount to mint and claim, we notice current implementation can be improved.

In the following, we show below the implementation from the related `claim()` routine. As the name indicates, this routine is used to claim the protocol tokens. However, the new amount to claim should be computed as `canRelease = userInfo[user].mintTotal - userInfo[user].mintReleased`, not current `canRelease = userInfo[user].mintTotal` (line 662).

```
619     function claim() public nonReentrant {
620         address user = msg.sender;
621         require(userInfo[user].depositAmount  > 0,'Without Deposit');
622         require(block.timestamp > endTime,'Mint Not Finished');
623         require(!userInfo[user].isRefund,'Has Refunded');
624         require(userInfo[user].mintTotal > userInfo[user].mintReleased,'All Released');
625         //claim release token
626         uint canRelease = userInfo[user].mintTotal;
627         userInfo[user].mintReleased += canRelease;
628         IERC20(JUMPToken).transfer(user,canRelease);
629         userInfo[user].isStartClaim = true;
630         emit Claim(user, canRelease);
631     }
```

Listing 3.2: `JUMPTreasury::claim()`

Recommendation Revisit the above routine to compute the correct protocol token amount to claim.

Status This issue has been fixed by the following commit: `79c7b07`.

3.3 Improper Token Recovery Logic in JUMPTreasury

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

Description

To recover tokens that may accidently deposit into the protocol contracts, `Jump.Fun` supports the recovery of these tokens via a privileged function `recoverWrongTokens()`. Our analysis on this privileged function indicates it should be improved.

To elaborate, we show below the implementation of this `recoverWrongTokens()` routine. While the intention is to recover these so-called wrong tokens, there is a need to ensure legitimate tokens in the protocol contracts should not be withdrawn via this function.

```
    function recoverWrongTokens(address _tokenAddress, uint256 _tokenAmount) external
        onlyOwner {
        IERC20(_tokenAddress).transfer(address(msg.sender), _tokenAmount);
    }
```

Listing 3.3: `JUMPTreasury::recoverWrongTokens()`

Recommendation Revisit the above routine to ensure the exclusion legitimate tokens in the protocol contracts. In addition, we notice certain privileged functions that are mainly intended for test purpose and these functions should be removed as well before the production deployment.

Status This issue has been fixed by the following commit: `79c7b07`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- 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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX::transfer()/transferFrom()

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 the following, we show the `sellTokenForETHEx()` routine in the `MemeExchange` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(tokenAddr).approve(address(v2Router), amountIn)` (line 454) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
445     function sellTokenForETHEx(address tokenAddr,uint256 amountIn,uint256 slippage)
            public {
446         require(amountIn > 0,'Amount Zero');
447         require(slippage <= maxSlippage,'Slippage too large');
448         //if(slip == 0) slippage = 15;
449         IERC20(tokenAddr).transferFrom(msg.sender,address(this), amountIn);
450         address[] memory path = new address[](2);
451         path[0] = address(tokenAddr);
452         path[1] = address(WETH);
453
454         IERC20(tokenAddr).approve(address(v2Router), amountIn);
455         uint256[] memory amounts = v2Router.getAmountsOut(amountIn,path);
456         uint256 amountOutMin = (slippage == 0) ? 0 : (amounts[1] - amounts[1] * slippage
                 / basePoint);
457         v2Router.swapExactTokensForETH(
458             amountIn,
459             amountOutMin,
460             path,
461             address(msg.sender),
462             block.timestamp
463         );
464     }
```

Listing 3.5: `MemeExchange::sellTokenForETHEx()`

The same issue is also applicable to a number of other routines in related contracts, including `JUMPTreasury` and `JUMPRewards`.

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

Status This issue has been fixed by the following commit: `79c7b07`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The `Jump.Fun` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters, assign roles, as well as execute privileged operations). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setRmLqSlippage(uint256 _rmSlippage) public onlyOwner{
       rmSlippage = _rmSlippage;
       emit SetRmLqSlippage(msg.sender,_rmSlippage);
    }
    ...
    function setDebitLimit(uint256 _debitLimit) public onlyOwner{
        debitLimit = _debitLimit;
        emit SetDebitLimit(msg.sender,_debitLimit);
    }
    ...
    function lockLP(address tokenAddr,uint256 lockAmount) public onlyOwner{
        require(block.timestamp > endTime + protectRefundDuration,'Protect lockLP
            Duration');
        address pair = tokenPair[tokenAddr];
        uint256 lqBalance = IERC20(pair).balanceOf(address(this));
        if(lqBalance > lockAmount){
            IERC20(pair).safeTransfer(lockLPAddr, lockAmount);
        }
    }
    ...
    function lockETH(uint256 lockAmount) public onlyOwner{
        require(block.timestamp > endTime + protectRefundDuration,'Protect lockETH
            Duration');
        (bool success, ) = lockLPAddr.call{value: lockAmount}("");
        require(success, "lockLPAddr Unable to Withdraw ETH");
    }
    ...
    function setKeeper(address addr, bool active) public onlyOwner {
        keeperMap[addr] = active;
        emit SetKeeper(msg.sender,addr,active);
    }
```

Listing 3.6: Example Privileged Operations in `JUMPTreasury`

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 `owner` privilege 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. And revisit the weakened trust model to ensure the `multisig` support is not reduced.

Status This issue has been mitigated as the ownership will be controlled through a timelock contract. Also, emergency admin role will be a multi-sig only.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Jump.Fun` protocol, which is a next-level `memecoin` launchpad with one-click deployment, `PvP` liquidity war mode, and free `10 ETH` liquidity loan on `Uniswap`. Through innovative features such as free liquidity loans and `LiqWar` mode, `Jump.fun` is significantly improving the odds for players to successfully catch pumps. Additionally, `$JUMP` token enables community ownership via fair launch guaranteeing same access for everyone. All platform revenue accrues back to `$JUMP`. Users can donate to get `$JUMP` as rewards and at the same time get a chance to get a share of the `Fomo3D` prize pool. Donated funds will be used to provide all creators with initial liquidity loans and a reward mechanism for community growth. All creators can launch tokens on `Jump.fun`'s platform with a single click without permission, solving the problems of long fundraising time and high `ETH` fees for pre-market trading. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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.