SMART CONTRACT AUDIT REPORT

for

Juicebox Protocol (v2)

Prepared By: Patrick Lou

PeckShield

April 8, 2022

Document Properties

Client	Juicebox
Title	Smart Contract Audit Report
Target	Juicebox
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 8, 2022	Xuxian Jiang	Final Release
1.0-rc	February 19, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Juicebox` 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 Juicebox

The `Juicebox` protocol is a programmable treasury, which can be used by `DeFi` projects to to configure how its tokens should be minted when it receives money, and under what conditions funds can be distributed to preprogrammed addresses or claimed by its community. These rules can evolve over `funding cycles`, allowing people to bootstrap open-ended projects and add structure, constraints, and incentives over time as needed. The protocol is light enough for a group of friends, yet powerful enough for a global network of users sharing thousands of ETH. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Juicebox

Item	Description
Name	Juicebox
Website	https://www.juicebox.money/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	April 8, 2022

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

- https://github.com/jbx-protocol/juice-contracts-v2.git (16d1ba9)

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

- https://github.com/jbx-protocol/juice-contracts-v2.git (a9ad156)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [9], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 design and implementation of the `Juicebox` protocol smart contracts. 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	2	■ ■
Low	3	■ ■ ■
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 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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Authorization Consistency in Terminal Management	Business Logic	Resolved
PVE-002	Low	Improved Validation of JBSplits-Store::set()	Coding Practices	Resolved
PVE-003	Low	Improved Reentrancy Protection in JBETHPaymentTerminal	Time And State	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Medium	Improper Token Removal Logic in changeFor()	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 Authorization Consistency in Terminal Management

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: JBDirectory
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Juicebox protocol has a built-in JBDirectory contract to keep track of the terminals through which each project is currently accepting funds. While reviewing the logic to manage each project's terminals, we notice the current implementation may be improved.

To elaborate, we show below the related addTerminalsOf()/removeTerminalOf() functions. As the names indicate, they are proposed to add or remove a terminal from the given project. Note both functions require proper authorization. It comes to our attention that the first function requires requirePermissionAllowingOverride(projects.ownerOf(_projectId), _projectId, JBOperations.ADD_TERMINALS, msg.sender == address(controllerOf[_projectId])) (lines 233-238) while the second function requires requirePermission(projects.ownerOf(_projectId), _projectId, JBOperations.REMOVE_TERMINAL) (line 263). In other words, the addition of a new terminal will require the authorization of the project owner or controller. However, the terminal removal needs to be performed by the project owner. For consistency, the project controller should also be authorized to remove a terminal. Our analysis shows that the setPrimaryTerminalOf() function shares the same issue.

```
230  function addTerminalsOf(uint256 _projectId, IJBTerminal[] calldata _terminals)
231    external
232    override
233    requirePermissionAllowingOverride(
234      projects.ownerOf(_projectId),
235      _projectId,
236      JBOperations.ADD_TERMINALS,
237      msg.sender == address(controllerOf[_projectId])
```

```
  )
{
  for (uint256 _i = 0; _i < _terminals.length; _i++) {
    // Can't be the zero address.
    if (_terminals[_i] == IJBTerminal(address(0))) {
      revert ADD_TERMINAL_ZERO_ADDRESS();
    }

    _addTerminalIfNeeded(_projectId, _terminals[_i]);
  }
}
```

Listing 3.1: `JBDirectory::addTerminalsOf()`

```
function removeTerminalOf(uint256 _projectId, IJBTerminal _terminal)
  external
  override
  requirePermission(projects.ownerOf(_projectId), _projectId, JBOperations.
      REMOVE_TERMINAL)
{
  // Get a reference to the terminals of the project.
  IJBTerminal[] memory _terminals = _terminalsOf[_projectId];

  // Delete the stored terminals for the project.
  delete _terminalsOf[_projectId];

  // Repopulate the stored terminals for the project, omitting the one being deleted.
  for (uint256 _i; _i < _terminals.length; _i++)
    // Don't include the terminal being deleted.
    if (_terminals[_i] != _terminal) _terminalsOf[_projectId].push(_terminals[_i]);

  // If the terminal that is being removed is the primary terminal for the token,
      delete it from being primary terminal.
  if (_primaryTerminalOf[_projectId][_terminal.token()] == _terminal)
    delete _primaryTerminalOf[_projectId][_terminal.token()];

  emit RemoveTerminal(_projectId, _terminal, msg.sender);
}
```

Listing 3.2: `JBDirectory::removeTerminalOf()`

Recommendation Properly ensure the authorization consistency among the above functions.

Status This issue has been resolved as the above functions are refactored into other functions which do not exhibit the inconsistency.

3.2 Improved Validation of JBSplitsStore::set()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `JBSplitsStore`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

To efficiently manage the fund distributions, the `Juicebox` protocol has a built-in `JBSplitsStore` contract that uses a `JBSplit` data structure to split up payout distributions. While reviewing the current logic to set up a project's payout splits, the current implementation can be improved to better validate the given project.

In particular, the logic is implemented in the following `set()` function, which takes a number of split-related information and validates them for further processing. It comes to our attention that the given `_projectId` is verified using the following statement, i.e., `_splits[_i].projectId > type(uint56).max`, which can be further improved with the `_splits[_i].projectId > projects.count()`. The improvement allows for rigorous validation on the (untrusted) user input.

```
function set(
  uint256 _projectId,
  uint256 _domain,
  uint256 _group,
  JBSplit[] memory _splits
)
  external
  override
  requirePermissionAllowingOverride(
    projects.ownerOf(_projectId),
    _projectId,
    JBOperations.SET_SPLITS,
    address(directory.controllerOf(_projectId)) == msg.sender
  )
{
  // Get a reference to the project's current splits.
  JBSplit[] memory _currentSplits = _getStructsFor(_projectId, _domain, _group);

  // Check to see if all locked splits are included.
  for (uint256 _i = 0; _i < _currentSplits.length; _i++) {
    // If not locked, continue.
    if (block.timestamp >= _currentSplits[_i].lockedUntil) continue;

    // Keep a reference to whether or not the locked split being iterated on is
        included.
    bool _includesLocked = false;
```

```
    for (uint256 _j = 0; _j < _splits.length; _j++) {
      // Check for sameness.
      if (
        _splits[_j].percent == _currentSplits[_i].percent &&
        _splits[_j].beneficiary == _currentSplits[_i].beneficiary &&
        _splits[_j].allocator == _currentSplits[_i].allocator &&
        _splits[_j].projectId == _currentSplits[_i].projectId &&
        // Allow lock extension.
        _splits[_j].lockedUntil >= _currentSplits[_i].lockedUntil
      ) _includesLocked = true;
    }

    if (!_includesLocked) {
      revert PREVIOUS_LOCKED_SPLITS_NOT_INCLUDED();
    }
  }

  // Add up all the percents to make sure they cumulative are under 100%.
  uint256 _percentTotal = 0;

  for (uint256 _i = 0; _i < _splits.length; _i++) {
    // The percent should be greater than 0.
    if (_splits[_i].percent == 0) {
      revert INVALID_SPLIT_PERCENT();
    }
    // ProjectId should be within a uint56
    if (_splits[_i].projectId > type(uint56).max) {
      revert INVALID_PROJECT_ID();
    }

    // The allocator and the beneficiary shouldn't both be the zero address.
    if (
      _splits[_i].allocator == IJBSplitAllocator(address(0)) &&
      _splits[_i].beneficiary == address(0)
    ) {
      revert ALLOCATOR_AND_BENEFICIARY_ZERO_ADDRESS();
    }
    ...
  }

  // Set the new length of the splits.
  _splitCountOf[_projectId][_domain][_group] = _splits.length;
}
```

Listing 3.3: JBSplitsStore::set()

Recommendation Properly validate the given input to the above `set()` function.

Status This issue has been resolved as it is implemented for gas efficiency.

3.3 Improved Reentrancy Protection in JBETHPaymentTerminal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `JBETHPaymentTerminal`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the reentrancy protection is not consistently enforced. Using the `JBETHPaymentTerminal` contract as an example, the `pay()` function (see the code snippet below) and `addToBalanceOf()` are not protected with the `nonReentrant` modifier while other functions are all enforced with `nonReentrant`. For consistency and improved protection, any invocation of an external contract requires extra care in avoiding the above `re-entrancy`. With that, we also suggest to add reentrancy protection to the above two functions `pay()` and `addToBalanceOf()`.

```
function pay(
  uint256 _projectId,
  address _beneficiary,
  uint256 _minReturnedTokens,
  bool _preferClaimedTokens,
  string calldata _memo,
  bytes calldata _delegateMetadata
) external payable override {
  return
    _pay(
      msg.value,
      msg.sender,
      _projectId,
      _beneficiary,
      _minReturnedTokens,
      _preferClaimedTokens,
      _memo,
      _delegateMetadata
    );
```

```
}
```

Listing 3.4: `JBETHPaymentTerminal::pay()`

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the following commit: `944561f`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Juicebox` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, whitelist contracts, and migrate protocols). It also has the privilege to regulate or govern the flow of assets within the protocol.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show a representative privileged operation in the `JBController` protocol.

```
function migrate(uint256 _projectId, IJBController _to)
  external
  requirePermission(projects.ownerOf(_projectId), _projectId, JBOperations.
      MIGRATE_CONTROLLER)
  nonReentrant
{
  // This controller must be the project's current controller.
  if (directory.controllerOf(_projectId) != this) {
    revert CALLER_NOT_CURRENT_CONTROLLER();
  }

  // Get a reference to the project's current funding cycle.
  JBFundingCycle memory _fundingCycle = fundingCycleStore.currentOf(_projectId);

  // Migration must be allowed
  if (!_fundingCycle.controllerMigrationAllowed()) {
    revert MIGRATION_NOT_ALLOWED();
  }

  // All reserved tokens must be minted before migrating.
```

```
    if (uint256(_processedTokenTrackerOf[_projectId]) != tokenStore.totalSupplyOf(
        _projectId))
      _distributeReservedTokensOf(_projectId, '');

    // Make sure the new controller is prepped for the migration.
    _to.prepForMigrationOf(_projectId, this);

    // Set the new controller.
    directory.setControllerOf(_projectId, _to);

    emit Migrate(_projectId, _to, msg.sender);
  }
```

Listing 3.5: JBController::migrate()

We emphasize that the privilege assignment with various protocol contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. We point out that a compromised `owner` account would allow the attacker to invoke the above `migrate()` to move funds out of the current protocol, which directly undermines the assumption of the `Juicebox` protocol.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and the team clarifies that a contract's owner is not the same as a project's owner. This permission check makes sure only a project's owner can migrate it's treasury..

3.5 Improper Token Removal Logic in changeFor()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `JBTokenStore`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Juicebox` protocol has an essential `JBTokenStore` contract that manages token minting and burning for all projects. This contract also supports the swap of a given project's token for another, including the removal of the project's token. While reviewing the token removal logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related `changeFor()` function. This function is proposed to swap the current project's token for another, and transfer ownership of the current token to another address if needed. As mentioned earlier, it is also used to reset the project's token if deemed appropriate for removal. However, it comes to our attention that when the token is being removed, the new given argument of `_token` will be set as 0, which will unfortunately revert the transaction execution (line 229)! In other words, the current functionality of removing a project's token is broken.

```
209 function changeFor(
210   uint256 _projectId,
211   IJBToken _token,
212   address _newOwner
213 ) external override onlyController(_projectId) returns (IJBToken oldToken) {
214   // Can't remove the project's token if the project requires claiming tokens.
215   if (_token == IJBToken(address(0)) && requireClaimFor[_projectId])
216     revert CANT_REMOVE_TOKEN_IF_ITS_REQUIRED();

218   // Can't add a token that doesn't use 18 decimals.
219   if (_token.decimals() != 18) revert TOKENS_MUST_HAVE_18_DECIMALS();

221   // Get a reference to the current token for the project.
222   oldToken = tokenOf[_projectId];

224   // Store the new token.
225   tokenOf[_projectId] = _token;

227   // If there's a current token and a new owner was provided, transfer ownership of
        the old token to the new owner.
228   if (_newOwner != address(0) && oldToken != IJBToken(address(0)))
229     oldToken.transferOwnership(_newOwner);

231   emit Change(_projectId, _token, oldToken, _newOwner, msg.sender);
232 }
```

Listing 3.6: `JBTokenStore::changeFor()`

Recommendation Revisit the above logic to properly support the token removal design. An example revision is shown as follows:

```
209 function changeFor(
210   uint256 _projectId,
211   IJBToken _token,
212   address _newOwner
213 ) external override onlyController(_projectId) returns (IJBToken oldToken) {
214   // Can't remove the project's token if the project requires claiming tokens.
215   if (_token == IJBToken(address(0)) && requireClaimFor[_projectId])
216     revert CANT_REMOVE_TOKEN_IF_ITS_REQUIRED();

218   // Can't change to a token already in use.
219   if (projectOf[_token] != 0) revert TOKEN_ALREADY_IN_USE();

221   // Can't change to a token that doesn't use 18 decimals.
```

```
    if (_token != IJBToken(address(0)) && _token.decimals() != 18)
      revert TOKENS_MUST_HAVE_18_DECIMALS();

    // Get a reference to the current token for the project.
    oldToken = tokenOf[_projectId];

    // Store the new token.
    tokenOf[_projectId] = _token;

    // Store the project for the new token if the new token isn't the zero address.
    if (_token != IJBToken(address(0))) projectOf[_token] = _projectId;

    // Reset the project for the old token if it isn't the zero address.
    if (oldToken != IJBToken(address(0))) projectOf[oldToken] = 0;

    // If there's a current token and a new owner was provided, transfer ownership of
        the old token to the new owner.
    if (_newOwner != address(0) && oldToken != IJBToken(address(0)))
      oldToken.transferOwnership(_newOwner);

    emit Change(_projectId, _token, oldToken, _newOwner, msg.sender);
  }
```

Listing 3.7: `JBTokenStore::changeFor()`

Status This issue has been fixed in the following commit: `63afb1e`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Juicebox` protocol, which is a programmable treasury, which can be used by `DeFi` projects to to configure how its tokens should be minted when it receives money, and under what conditions funds can be distributed to preprogrammed addresses or claimed by its community. 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

PeckShield