SMART CONTRACT AUDIT REPORT

for

FuelOnBlast

Prepared By: Xiaomi Huang

PeckShield

March 21, 2024

Document Properties

Client	FuelOnBlast
Title	Smart Contract Audit Report
Target	FuelOnBlast
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 21, 2024	Xuxian Jiang	Final Release
1.0-rc	March 20, 2024	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

Given the opportunity to review the design document and related smart contract source code of the `FuelOnBlast` 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 FuelOnBlast

`FuelOnBlast` is an innovative protocol designed to address the challenges faced by project creators when launching a token. The protocol empowers creators, letting them customize and create a token that meets their projects requirements. In addition, they can optionally choose to borrow their initial liquidity (aka `Fuel`). This eliminates the need for them to invest their own capital, allowing them to bring their ideas to life without financial constraints. The basic information of `FuelOnBlast` is as follows:

Table 1.1: Basic Information of FuelOnBlast

Item	Description
Target	FuelOnBlast
Website	https://fuelonblast.com/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 21, 2024

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

- https://github.com/fuelonblast/smart-contracts.git (64557be)

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

- https://github.com/fuelonblast/smart-contracts.git (0a33efb)

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 classified into four categories accordingly, 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.

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 `FuelOnBlast` 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	2	■ ■
Low	2	■ ■
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 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key FuelOnBlast Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possibly Inaccurate totalDistributedReward Accounting in TaxToken	Business Logic	Resolved
PVE-002	Medium	Possible Tax Evasion For Certain Wallets	Time and State	Resolved
PVE-003	Low	Improper totalPrivateRoundImported Accounting in PAWSVesting	Business Logic	Resolved
PVE-004	Informational	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Possibly Inaccurate totalDistributedReward Accounting in TaxToken

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

Description

The `FuelOnBlast` protocol has a core `TaxToken` contract that offers an `ERC20` token with taxation features. This token contract has a token-wide state, i.e., `totalDistributedRewards`, which is used to keep track of the total amount of distributed rewards. Our analysis shows this state may not accurately serve its purpose.

To elaborate, we show below the implementation of the affected routine, i.e., `_distributeRewards()`. As the names indicates, this routine is used to distribute the given amount of rewards. We notice that it may incorrectly take into account `undistributedRewards` twice. The proper approach is update `totalDistributedRewards` only once upon the entry of `_distributeRewards()`.

```
    function _distributeRewards(uint256 _amount) private {
        uint256 circulatingSupply = _getSupplyExcludingSpecialWallets();
        if (circulatingSupply != 0) {
            if (undistributedRewards != 0) {
                _amount += undistributedRewards;
                undistributedRewards = 0;
            }
            currentMagnifiedSharePerToken +=
                (TaxToolkit.MAGNIFICATION_FACTOR * _amount) /
                circulatingSupply;
        } else {
            /// @dev this edge case is possible if only whitelist addresses (and the
                pair) are holders
```

```
            undistributedRewards += _amount;
        }
        totalDistributedRewards += _amount;
    }
```

Listing 3.1: `TaxToken::_distributeRewards()`

Recommendation Revisit the above `_distributeRewards()` routine to properly update the `totalDistributedRewards` state.

Status The issue has been addressed in the following commit: `0a33efb`.

3.2 Possible Tax Evasion For Certain Wallets

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TaxToken`
- 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 `Uniswap/Lendf.Me` hack [12].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `TaxToken` as an example, the `_handleTax()` function (see the code snippet below) is provided to externally interact to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 457) start before effecting the update on internal state (in `_instantTransfer`), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` to evade possible tax collection.

```
448    function _handleTax() private instantTransfer {
449        TaxToolkit.CollectedTaxes memory taxes = _calculateTaxesToSwap();
450        if (!_isFueled) {
451            _distributeTaxes(taxes);
452            return;
```

```
        }
        uint256 outstanding = _loanManager.calculateOutstandingLoan(address(this));
        if (outstanding != 0) {
            /// @dev if the project owes fuel + interest.
            _pay(taxes.buy + taxes.sell + taxes.transfer, outstanding);
            return;
        }
        /// @dev "outstanding" could be 0 if it was paid off externally (either clawback
             or directly repaying the factory).
        _isFueled = false;
        _distributeTaxes(taxes);
    }
```

Listing 3.2: `TaxToken::_handleTax()`

Recommendation Revisit the above routine to ensure the tax collection will not be evaded.

Status The issue has been addressed in the following commit: `fb12be3`.

3.3 Improper totalPrivateRoundImported Accounting in PAWSVesting

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PAWSVesting`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `FuelOnBlast` protocol has a key `PAWSVesting` contract that acts as the `PAWS` vesting contract. Within this vesting contract, there is an important state, i.e., `totalPrivateRoundImported`, which is used to keep track of the total amount of imported private rounds. The private rounds may be imported in multiple times and adjusted with two routines. While examining these two routines, we notice this important is not properly udpated.

To elaborate, we show below the related code snippet from these two routines, i.e., `importPrivateRound()` and `removeImport()`. As their names indicate, the first routine is used to import a private round and the second routine is used to remove an existing import. However, while the first routine properly update the `totalPrivateRoundImported`, the second routine does not.

```
    function importPrivateRound(Data[] calldata _dataList) external onlyOwner
        onlyBeforeRefining {
        uint88 total = _importPrivateRound(_dataList);
```

```
        if (PRIVATE_ROUND_TOTAL < totalPrivateRoundImported + total) revert
            RoundLimitReached();
        totalPrivateRoundImported += total;
        emit ImportedPrivateRound(_dataList);
    }

    /**
     * @notice Remove an imported vesting schedule (only Owner and before refining).
     * @param _address Wallet address.
     */
    function removeImport(address _address) external onlyOwner onlyBeforeRefining {
        if (_privateRoundVesting[_address].amount == 0) revert Errors.Unavailable();
        delete _privateRoundVesting[_address];
        emit RemovedImport(_address);
    }
```

Listing 3.3: `PAWSVesting::importPrivateRound()/removeImport()`

Recommendation Revisit the above `removeImport()` routine to properly update the global `totalPrivateRoundImported` state.

Status The issue has been addressed in the following commit: `fb12be3`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PAWSAirdrop`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
```

```
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.4: 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.

In current implementation, if we examine the `PAWSAirdrop::withdrawUndistributedPAWS()` routine that is designed to withdraw undistributed `PAWS`. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 88).

```
81    /**
82     * @notice Withdraw undistributed $PAWS (only Owner).
83     * @param _amount Amount to withdraw.
84     */
85    function withdrawUndistributedPAWS(uint256 _amount) external onlyOwner {
86        uint256 undistributed = calculateUndistributedPAWS();
87        if (_amount == 0  undistributed < _amount) revert Errors.InvalidAmount();
88        paws.transfer(_msgSender(), _amount);
89        emit UndistributedPAWSWithdrawal(_amount, undistributed - _amount);
90    }
```

Listing 3.5: PAWSAirdrop::withdrawUndistributedPAWS()

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

Status This issue has been resolved with the use of only `ERC20`-compliant tokens: `fb12be3`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `FuelOnBlast` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters and setting up minters). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function addMinter(address _minter) external onlyOwner {
        if (!isInitialized) revert Errors.NotInitialized();
        if (_minter == address(0)) revert Errors.InvalidAddress();
        if (_isMinter[_minter]) revert AlreadyAdded();
        _isMinter[_minter] = true;
        emit MinterAdded(_minter);
    }

    /**
     * @notice Remove a minter (only Owner).
     * @param _minter Minter address.
     */
    function removeMinter(address _minter) external onlyOwner {
        if (!_isMinter[_minter]) revert AlreadyRemoved();
        delete _isMinter[_minter];
        emit MinterRemoved(_minter);
    }

    /**
     * @notice Remove stuck native tokens (only Owner).
     * @param _receiver Address to send to native tokens to.
     */
    function removeStuckNativeToken(address _receiver) external onlyOwner {
        if (_receiver == address(0)) revert Errors.InvalidAddress();
        uint256 balance = address(this).balance;
        if (balance == 0) revert Errors.NothingToWithdraw();
        _safeNativeTransferOrRevert(_receiver, balance);
        emit RemovedStuckNativeToken(_receiver, balance);
    }
```

Listing 3.6: Example Privileged Operations in `Factory`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the `owner`.

4 Conclusion

In this audit, we have analyzed the design and implementation of `FuelOnBlast`, which is an innovative protocol designed to address the challenges faced by project creators when launching a token. The protocol empowers creators, letting them customize and create a token that meets their projects requirements. In addition, they can optionally choose to borrow their initial liquidity (aka `Fuel`). This eliminates the need for them to invest their own capital, allowing them to bring their ideas to life without financial constraints. 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-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.