



SMART CONTRACT AUDIT REPORT

for

Super Blur Battlerz



Prepared By: Xiaomi Huang

PeckShield
February 29, 2024

Document Properties

Client	Super Blur Battlerz
Title	Smart Contract Audit Report
Target	Betting
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	February 29, 2024	Xuxian Jiang	Final Release
1.0-rc	February 29, 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 Super Blur Battlerz	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Possible Inconsistency in Ongoing Bets And Completing Bets	11
3.2	Revisited Reentrancy Protection in Betting	12
3.3	Trust Issue of Admin Keys	14
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related source code of the [Super](#) Blur Battlerz game 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 Super Blur Battlerz

[Super](#) Blur Battlerz is Horse Racing, but with Blur Farmers. The Betting game is created to have races between groups of Blur Farmers, and whoever scores the most Blur points during the race wins. Note this protocol was a Blast Big Bang contestant and won a Honorable Mention. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Super Blur Battlerz

Item	Description
Name	Super Blur Battlerz
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 29, 2024

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

- <https://github.com/SuperBlurBattlerz/smart-contract.git> (15d36ee)

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

- <https://github.com/SuperBlurBattlerz/smart-contract.git> (eda4564)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

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

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

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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. 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 [Super Blur Battlerz](#) game 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	1	
Low	1	
Informational	1	
Total	3	

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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational issue.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Inconsistency in Ongoing Bets And Completing Bets	Coding Practices	Resolved
PVE-002	Informational	Revisited Reentrancy Protection in Betting	Time and State	Resolved
PVE-003	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 Possible Inconsistency in Ongoing Bets And Completing Bets

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Betting
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

Description

The [Super](#) Blur Battlerz protocol allows for new race creation and betting in the key `Betting` contract. In the process of examining the betting logic, we notice there exists an inconsistency in proceeding the race or ending it.

In the following, we show the implementation of the related `betMore()` and `endRace()` routines. The first routine is used to bet on an ongoing race while the second routine ends the current race. It comes to our attention that the first routine is guarded with the starting and ending blocks, i.e., `bettingStartBlock` and `bettingEndBlock`. However, the second routine ends the race by ensuring `require(block.timestamp > race.raceEndTimestamp)` (line 285). For consistency, we suggest to make use of either block or timestamp for the enforcement. Or we need to enforce the following requirement in `betMore()`: `require(block.timestamp < race.raceEndTimestamp)`.

```
258     function betMore(address winner) external payable {
259         Race storage race = races[currentRaceIndex];

261         require(msg.value >= MIN_BET, "Bet too low.");
262         require(
263             block.number >= race.bettingStartBlock &&
264             block.number < race.bettingEndBlock,
265             "Outside of betting window."
266         );
267         require(!_isRacerInTheRace(race, winner), "This address isn't in the race.");
```

```
269         if (race.bets[winner][msg.sender] == 0) {
270             race.degens[winner].push(msg.sender);
271         }

273         race.bets[winner][msg.sender] += msg.value;
274         race.totalBets += msg.value;

276         allTimeDegens[msg.sender] = true;

278         emit NewBet(currentRaceIndex, msg.sender, winner, msg.value);
279     }

281     function endRace(address winner) external onlyRaceAdmin {
282         Race storage race = races[currentRaceIndex];

284         require(race.racers.length != 0, "Race has not been setup.");
285         require(block.timestamp > race.raceEndTimestamp, "Race end time hasn't been
           reached yet.");
286         require(race.winner == address(0), "Winner has already been set.");
287         require(!_isRacerInTheRace(race, winner), "This address isn't in the race.");

289         race.winner = winner;

291         emit RaceEnd(currentRaceIndex, winner);
292     }
```

Listing 3.1: Betting::betMore()/endRace()

Recommendation Improve the above routines to consistently enforce whether the race should be allowed to proceed or ended.

Status This issue has been fixed by the following commit: [eda4564](#).

3.2 Revisited Reentrancy Protection in Betting

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Betting
- Category: Time and State [6]
- 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 [11] exploit, and the Uniswap/Lendf.Me hack [10].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the Betting as an example, the `distributeFees()` function (see the code snippet below) is provided to call an external winner 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 winner contract (line 359) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

342     function distributeFees() public onlyRaceAdmin {
343         Race storage race = races[currentRaceIndex];
344
345         uint winnersCount = race.degens[race.winner].length;
346
347         require(race.winner != address(0), "Winner hasn't been set yet");
348         require(race.rewardsDistributedCount == winnersCount, "Rewards haven't been
           distributed yet.");
349         require(race.feesDistributed == false, "Fees have already been distributed.");
350
351         if (winnersCount == 0) {
352             _splitPoolWithWinner(race);
353             return;
354         }
355
356         race.feesDistributed = true;
357
358         if (race.winnerReward != 0) {
359             _sendEth(race.winner, race.winnerReward);
360         }
361
362         _closeRace();
363     }

```

Listing 3.2: Betting::distributeFees()

Fortunately, this `distributeFees()` routine updates the `race.feesDistributed` state in a reliable way. Moreover, the internal helper `_sendEth()` has the `nonReentrant` modifier for reentrancy protection. Meanwhile, we notice this `nonReentrant` modifier is embedded deep inside an internal helper routine and hence suggest to only add it to two public `distributeRewards()` routines.¹

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and utilizing the necessary `nonReentrant` modifier to block possible re-entrancy.

¹As mentioned earlier, the `distributeFees()` public routine does not need this `nonReentrant` modifier.

Status This issue has been fixed by the following commit: [eda4564](#).

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Betting
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In [Super](#) Blur Battlerz, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure various parameters, add race admins, and manage races). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
208     function setPointsOperator(address _pointsOperator) public onlyOwner {
209         BLAST_POINTS.configurePointsOperator(_pointsOperator);
210     }
211
212     function setOwner(address _owner) external onlyOwner {
213         owner = _owner;
214     }
215
216     function updateB1(address _b1) external onlyOwner {
217         beneficiary1 = _b1;
218     }
219
220     function updateB2(address _b2) external onlyOwner {
221         beneficiary2 = _b2;
222     }
223
224     function setRaceAdmin(address admin, bool flag) external onlyOwner {
225         raceAdmins[admin] = flag;
226     }
```

Listing 3.3: Example Privileged Functions in Betting

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 mitigated as the team makes use of a multisig to act as the privileged owner.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the [Super](#) Blur Battlerz game protocol, which is created to have races between groups of Blur Farmers, and whoever scores the most Blur points during the race wins. Note this protocol was a Blast Big Bang contestant and won a Honorable Mention. 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. <https://cwe.mitre.org/data/definitions/628.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 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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.
- [10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

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

