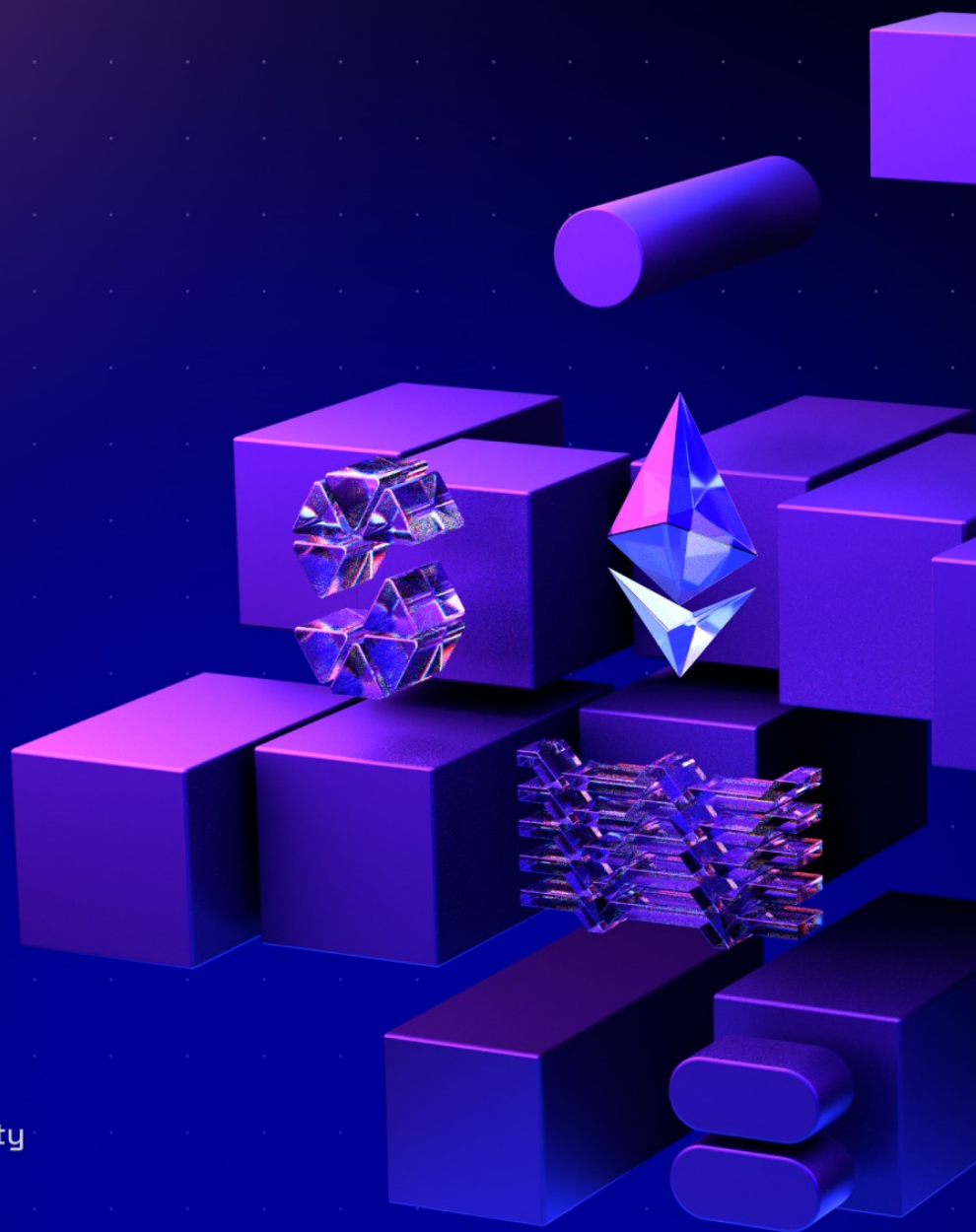
ackee
blockchain security

BetFin

Lottery

25.3.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0	Final Report	12.03.2025
1.1	Fix Review	25.03.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Michal Převrátil	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

BetFin Lottery is a blockchain-based lottery game that allows users to place bets with BET tokens on combinations of numbers and symbols. The winning selections are determined through Chainlink VRF's verifiable random function. The protocol incorporates a jackpot mechanism that accumulates a portion of all wagered tokens, creating additional incentives for players who match all winning selections.

Revision 1.0

BetFin engaged Ackee Blockchain Security to perform a security review of the BetFin protocol with a total time donation of 15 engineering days in a period between February 26 and March 12, 2025, with Michal Převrátil as the lead auditor. 5 engineering days were dedicated to manually-guided differential fuzzing using the Wake testing framework.

The audit was performed on the commit `b8662d0`[1] with the scope being all Solidity files inside the `src` directory.

A kick-off meeting with BetFin was held to provide a code walkthrough and discuss technical details, which made the review process more efficient. As there was no technical specification or documentation during the audit period, we reviewed the project based on our understanding of the protocol and information provided by BetFin.

We began our audit with a manual review of the codebase in parallel with manually-guided differential fuzzing using the Wake testing framework. The fuzzing yielded the I6 finding. We concluded our review with static analysis tools, including Wake, which yielded the I9 and I10 findings.

During the review, we paid special attention to:

- ensuring randomly selected numbers and symbols of winning tickets are uniformly distributed with no correlations;
- verifying access controls are properly applied;
- preventing reentrancy attacks;
- ensuring tokens cannot be locked inside the contracts;
- preventing denial-of-service attacks;
- optimizing code efficiency; and
- avoiding common issues such as data validation.

Our review resulted in 21 findings, ranging from Info to High severity. The most severe finding, H1, highlights concerns about the unsustainability of the current design, which either forces BetFin to pay for costly transactions vulnerable to griefing attacks or leaves significant amounts of tokens locked inside the contracts.

The code quality is overall good, but multiple changes can improve readability (I1, I9, I10). The codebase contains multiple checks validating the same property in different ways, ensuring system correctness. The code avoids using inline assembly.

The codebase could be improved in terms of gas optimization, with the most notable inefficiency being described in the I3 finding, saving up to 50% of gas costs for users placing and claiming bets.

Ackee Blockchain Security recommends BetFin:

- avoid building invariants around the fact that all tickets must be claimed before unlocking the remaining BET tokens;
- be cautious when designing permissionless batch operations to avoid griefing attacks;

- strictly follow standards such as ERC-721; and
- address reported findings.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

BetFin engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision. The review was performed on the commit `3333d36`[2].

19 out of 21 findings were fixed, I2 was partially fixed, and H1 was acknowledged with a fix planned for implementation outside the scope of this revision.

See Report Revision 1.1 for the description of the changes made in this revision.

[1] full commit hash: `b8662d0fca5376a197b18380c93722418d08227f`

[2] full commit hash: `3333d3668657c36839b9f6d7619d9e81851e1bbb`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	1	4	2	4	10	21

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Unsustainable claiming of all bets	High	1.0	Acknowledged
M1: Griefing on placing bets	Medium	1.0	Fixed
M2: Griefing on claiming multiple bets	Medium	1.0	Fixed
M3: Missing owner check in `tokenURI`	Medium	1.0	Fixed
M4: Unintended ERC-721 tokens can be permanently locked in MultiBet contract	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
L1: Griefing on refunding bets	Low	1.0	Fixed
L2: Tokens rounding imprecision	Low	1.0	Fixed
W1: Tickets can be edited after round is closed	Warning	1.0	Fixed
W2: Bets list lacks public accessibility for user verification	Warning	1.0	Fixed
W3: `setResult` does not include jackpot additional rewards	Warning	1.0	Fixed
W4: `TicketSold` event emits cumulative amount instead of individual ticket value	Warning	1.0	Fixed
I1: Explicit getters can be replaced with public state variables	Info	1.0	Fixed
I2: Unclear parameter naming in round creation	Info	1.0	Partially fixed
I3: Inefficient placing of bets	Info	1.0	Fixed
I4: Unnecessary inheritance of `ERC721URIStorage` extension	Info	1.0	Fixed

Finding title	Severity	Reported	Status
I5: Replace role-based access control with direct contract reference checks for critical functions	Info	1.0	Fixed
I6: Misleading event name	Info	1.0	Fixed
I7: Unused state variable	Info	1.0	Fixed
I8: Typographical error in error message description	Info	1.0	Fixed
I9: Variables can be immutable	Info	1.0	Fixed
I10: Unused using-for directives	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Michal Převrátil	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

BetFin Lottery is a simple lottery game that allows users to bet on numbers and a symbol. Users can place any number of bets, with each bet containing multiple tickets (1-9). Each ticket represents a single set of 5 numbers (1-25) and a symbol (1-5). The symbol unlocks additional rewards if the ticket is winning and the bet holding the ticket has at least 3 tickets.

The lottery uses Chainlink VRF to select the winning ticket, guaranteeing the randomness of the results. The protocol uses BET tokens for ticket payments and rewards. A portion of the received BET tokens is accumulated as an additional jackpot, which is only claimable if a player guesses all 5 numbers and the symbol correctly.

Multiple betting rounds can coexist, but it is assumed the round deadlines will never be the same. The contracts are not expected to be deployed behind a proxy.

Trust Model

Players must trust BetFin to provide BET tokens for the rewards and the jackpot. Players must also trust that the BET token is minted and distributed correctly, as it is used for ticket payments and rewards distribution.

Chainlink VRF is trusted to provide true randomness for the winning ticket selection.

Fuzzing

A manually-guided differential stateful fuzz test was developed during the review to test the correctness and robustness of the system. The fuzz test employs fork testing technique to test the system with external contracts exactly as they are deployed in the deployment environment. This is crucial to detect any potential integration issues.

The differential fuzz test keeps its own Python state. Assertions are used to verify the Python state against the on-chain state in contracts.

The fuzzing focused on the following aspects:

- betting round state transitions (creation, active, finished, refunding);
- bet data consistency throughout lottery operations;
- randomness fulfillment and result processing;
- reward calculation and distribution mechanisms;
- refund process integrity;
- jackpot processing and additional reward distribution;
- token balances across all states and operations.

The list of all implemented execution flows and invariants is available in Appendix B.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

H1: Unsustainable claiming of all bets

High severity issue

Impact:	High	Likelihood:	Medium
Target:	Lottery.sol	Type:	N/A

Description

The `Lottery` contract reserves BET tokens from the associated staking contract to cover payouts of winning tickets. The remaining BET tokens are returned to the staking contract once all tickets are claimed.

Under the assumption that one ticket is always placed in each bet, on average approximately 92% of opened bets will not be winning. Non-winning bets provide no motivation for users to claim them.

However, due to the design of the `Lottery` contract, all tickets must be claimed to avoid locking a significant amount of BET tokens in the contract. As a consequence, the majority of opened bets must be claimed by BetFin.

Claiming approximately 550 bets consumes 30M gas units, which is the block gas limit of the Polygon mainnet.

Exploit scenario

Given the configuration of the `Lottery` contract, claiming of remaining bets may become unsustainable for BetFin. Additionally, malicious actors can execute griefing attacks to prevent BetFin from claiming the remaining bets, see finding M2.

Recommendation

Consider redesigning the protocol to payout winning bets directly from the staking contract, effectively removing the requirement to claim all bets.

Alternatively, introduce a new feature to motivate users to claim their bets.

Acknowledgment 1.1

The finding is expected to be resolved by providing motivation for users to claim their bets. However, the exact solution is out of scope of this audit.

Go back to Findings Summary

M1: Griefing on placing bets

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Lottery.sol	Type:	Griefing

Description

Each lottery round allows only one registered ticket with given numbers and symbol. Placing new bets with tickets costs BET tokens. However, editing existing tickets is possible at the cost of consumed gas only.

This creates a possibility for griefing attacks resulting in denial of service for other players.

Exploit scenario

Bob has a placed ticket in the current round.

Alice wants to place a new ticket with unique numbers `{1, 2, 3, 4, 5}` and symbol `3`.

Bob sees Alice's transaction in the mempool and submits a new transaction, changing his ticket to `{1, 2, 3, 4, 5}` and symbol `3`.

By paying a higher gas price, Bob frontruns Alice's transaction and prevents her from placing her bet.

Recommendation

Consider charging a fee in the form of BET tokens for editing tickets to discourage griefing attacks.

Alternatively, consider removing the editing functionality.

Fix 1.1

The issue was fixed by implementing a 10% BET token fee of the current round's ticket price for editing tickets.

Go back to Findings Summary

M2: Griefing on claiming multiple bets

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Lottery.sol	Type:	Griefing

Description

After the winning numbers are drawn, anyone can claim any unclaimed bets with tickets using the `claim` function in the `Lottery` contract. To claim multiple bets at once, users can use the `claimAll` function.

Since the bets can be claimed by anyone, malicious actors can exploit this for griefing attacks.

This issue is even more severe in the context of the H1 finding.

Exploit scenario

BetFin submits a transaction to claim all remaining unclaimed bets. Bob sees the transaction in the mempool and submits a new transaction to claim the last bet from the list to be claimed by BetFin. Bob pays a higher gas price to frontrun BetFin's transaction.

BetFin's transaction reverts, consuming a significant amount of gas since all bets are successfully claimed except the last one.

Recommendation

Skip claiming a given bet if it is already claimed in the `claimAll` function.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

M3: Missing owner check in `tokenURI`

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Lottery.sol	Type:	Standards violation

Description

Listing 1. Excerpt from Lottery

```
356   function tokenURI(uint256 tokenId) public view virtual override(ERC721,
    ERC721URIStorage) returns (string memory) {
357 return string.concat(uri, "/", Strings.toString(tokenId), ".json");
358   }
```

The `tokenURI` function lacks the `_requireOwned()` check that verifies token existence. While the direct impact is limited, this implementation violates the ERC-721 specification, which explicitly requires that `tokenURI` must revert if the token does not exist.

Exploit scenario

Alice creates a marketplace dApp that integrates with this Lottery contract. When Bob queries the `tokenURI` function for a non-existent token ID, the function returns a URI instead of reverting as required by the ERC-721 specification. Alice's marketplace incorrectly displays this token as valid. Charlie, relying on this information, attempts to interact with the non-existent token, resulting in failed transactions and a poor user experience. Additionally, third-party indexers may incorrectly catalog non-existent tokens, causing inconsistencies across the ecosystem.

Recommendation

Add the `_requireOwned()` check at the beginning of the `tokenURI` function to enforce token existence verification and comply with the ERC-721 standard.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

M4: Unintended ERC-721 tokens can be permanently locked in MultiBet contract

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	MultiBet.sol	Type:	Data validation

Description

Listing 2. Excerpt from MultiBet

```
42 function onERC721Received(address, address, uint256, bytes calldata) external
   pure override returns (bytes4) {
43     return IERC721Receiver.onERC721Received.selector;
44 }
```

The `MultiBet` contract implements functionality to receive any ERC-721 token.

While this function is intended for receiving BetFin Pass tokens used for permission control in the Core contract, the implementation lacks validation of the token contract address.

As a result, any ERC-721 token sent to this contract will be permanently locked without a recovery mechanism.

Exploit scenario

Alice sends an unintended ERC-721 token to the `MultiBet` contract. The token becomes permanently locked in the contract with no recovery mechanism.

Recommendation

Either:

- implement token address validation to accept only BetFin Pass tokens; or

- add a privileged recovery function for ERC-721 tokens.

Fix 1.1

The issue was fixed by only accepting BetFin Pass tokens in the `onERC721Received` function.

Go back to Findings Summary

L1: Griefing on refunding bets

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	LotteryRound.sol	Type:	Griefing

Description

Each lottery round may be refunded if randomness from Chainlink VRF is not requested within a given time window or if the response is not received within a given time window.

Anyone can permissionlessly refund any number of bets, sending BET tokens paid for the bets back to their owners.

Bets can be refunded in batches, given the offset in the array and the number of bets to refund. Refunding already refunded bets reverts the transaction.

This design allows a griefing attack on refunding bets, as described in the following scenario.

Exploit scenario

BetFin sends a transaction refunding 100 bets. Bob sees the transaction in the mempool and submits a new transaction refunding only the last bet from the list. Bob pays a higher gas price to frontrun BetFin's transaction.

BetFin's transaction reverts, consuming a significant amount of gas since all bets are successfully processed until the last one, which has already been refunded.

Recommendation

Skip refunding a given bet if it is already refunded in the `LotteryRound.refund` function.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

L2: Tokens rounding imprecision

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	LotteryRound.sol	Type:	Arithmetics

Description

Each `LotteryRound` contract collects BET tokens from users for placing bets. 4% of the collected tokens is used as an additional jackpot while the rest is sent to the associated staking contract.

Listing 3. Excerpt from LotteryRound.processJackpot

```
197 // calculate 4% of bets
198 uint256 jackpot = ticketsCount * ticketPrice * 4 / 100;
199 // send jackpot to lottery
200 require(IERC20(lottery.getToken()).transfer(address(lottery), jackpot),
    "LR11");
```

Remaining tokens are sent when claiming a given bet.

Listing 4. Excerpt from LotteryRound.claim

```
344 // calculate amount to send
345 uint256 amountToSend = ticketPrice * bet.getTicketsCount() * 96 / 100;
346 // send tokens - 4% to staking
347 IERC20(lottery.getToken()).transfer(lottery.getStaking(), amountToSend);
```

The current approach requires claiming all bets and having the ticket price as a multiple of 25 to avoid locked tokens in the `LotteryRound` contract.

Exploit scenario

BetFin sets the ticket price to 33 BET tokens. 100 tickets are sold in the current round, and 3300 BET tokens are collected.

The `processJackpot` function is called and 132 BET tokens are added to the jackpot.

When claiming a bet with a single ticket, `33 * 96 // 100 = 31` BET tokens are sent to the staking contract. Assuming all bets contain a single ticket, a total amount of 3100 BET tokens are sent to the staking contract.

100 BET tokens are now permanently locked in the `LotteryRound` contract.

Recommendation

Send the remaining 96% of the collected tokens in the `processJackpot` function and calculate the amount based on the difference between the total collected tokens and the jackpot.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

W1: Tickets can be edited after round is closed

Impact:	Warning	Likelihood:	N/A
Target:	LotteryRound.sol	Type:	Logic error

Description

After placing a lottery bet, the owner of the bet can edit tickets in the bet. Editing the tickets should only be possible before the round is closed.

Listing 5. Excerpt from LotteryRound.editTickets

```
126 // check if round is closed
127 require(status == 1, "LR02");
```

The round closed check is made using the `status` variable. However, the `status` variable is not updated when the round is closed. Instead, the `getStatus` function returns the correct status based on the current timestamp using the `isOpen` function.

Listing 6. Excerpt from LotteryRound.getStatus

```
333 if (!isOpen() && status == 1) {
334     return 5;
335 }
```

Editing tickets is possible after the round is closed but before the randomness from Chainlink VRF is requested. This makes it a non-security issue.

Recommendation

Use the `getStatus` function to check if the round is closed.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

W2: Bets list lacks public accessibility for user verification

Impact:	Warning	Likelihood:	N/A
Target:	LotteryRound.sol	Type:	Function visibility

Description

Listing 7. Excerpt from LotteryRound

```
54 address[] private bets;
```

The `bets` list in the `LotteryRound` contract is marked as private, preventing direct access to this data. The contract lacks view functions that expose this information.

Users interacting with the `refund` function require an index in the bets list as a parameter. Without visibility into the bets list, users cannot verify their position or confirm refund eligibility.

Recommendation

Implement one of these solutions:

- add a view function to expose information from the `bets` list; or
- change the `bets` list visibility to public.

Fix 1.1

The `bets` list visibility was changed to public.

Go back to Findings Summary

W3: `setResult` does not include jackpot additional rewards

Impact:	Warning	Likelihood:	N/A
Target:	Lottery.sol	Type:	Logic error

Description

Listing 8. Excerpt from Lottery._claim

```
323       if (winAmount > 0) {
324           if (jackpot) {
325               // transfer jackpot to player
326               token.transfer(bet.getPlayer(), winAmount +
    additionalJackpot);
327               // emit Jackpot event
328               emit JackpotWon(roundAddress, additionalJackpot);
329               // reset additional jackpot
330               additionalJackpot = 0;
331           } else {
332               // transfer win amount to player
333               token.transfer(bet.getPlayer(), winAmount);
334           }
335       }
336       // set bet result and status
337       bet.setResult(winAmount);
338       // increase claimed amount
339       claimedByRound[roundAddress] += winAmount;
340       // increment claimed tickets
341       bool allClaimed = round.claim(betAddress);
342       // check if all tickets are claimed
343       if (allClaimed) {
344           // transfer back to staking =  initial amount - claimed amount
345           uint256 toSend = amount * MAX_SHARES -
    claimedByRound[roundAddress];
346           // transfer to staking
347           token.transfer(address(staking), toSend);
348       }
349 emit TicketClaimed(betAddress, winAmount);
```

When a jackpot occurs with additional rewards, there is a discrepancy

between the displayed and received amounts. The ERC-721 token representing the bet displays the initial result amount, while users receive additional tokens from the jackpot rewards.

The `TicketClaimed` event omits information about these additional jackpot rewards, preventing off-chain systems from accurately tracking the total amount received by users.

Recommendation

Implement the following:

- include the additional jackpot rewards in the `TicketClaimed` event; and
- update the token metadata to reflect the total reward amount, including jackpot bonuses.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

W4: `TicketSold` event emits cumulative amount instead of individual ticket value

Impact:	Warning	Likelihood:	N/A
Target:	LotteryRound.sol	Type:	Arithmetics

Description

The `Lottery.placeBet` function returns the address of the bet contract.

Listing 9. Excerpt from LotteryRound.registerBet

```
113       // update ticket counter
114       ticketsCount += count;
115       // update bet counter
116       betsCount++;
117       // push bet to bets
118       bets.push(_bet);
119       // check balance of round - should not happen, but anyway
120       require(IERC20(lottery.getToken()).balanceOf(address(this)) >=
    ticketsCount * ticketPrice, "LR04");
121 // emit event
122 emit TicketSold(_bet, ticketsCount * ticketPrice);
```

The `LotteryRound.TicketSold(address indexed bet, uint256 amount)` event is the only event that includes information about the bet. However, the `amount` parameter represents the product of the round ticket price and the round total sold ticket amount, rather than the individual ticket value. This design makes tracking specific ticket sales difficult.

This issue becomes particularly problematic when using the `MultiBet` contract, as the `placeBet` return value is not utilized. In such cases, identifying the corresponding bet contract address requires either tracking the event emission order or calculating the total sold amount, which increases the complexity of integration with external systems.

Recommendation

Implement one of these solutions:

- emit a new event that includes the individual bet value and bet contract address;
- modify the `TicketSold` event to include both the individual ticket value and cumulative amount; or
- document this behavior in the protocol's technical documentation.

Fix 1.1

The issue was fixed by emitting the current bet value instead of the cumulative amount in the `TicketSold` event.

Go back to Findings Summary

I1: Explicit getters can be replaced with public state variables

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol, LotteryBet.sol, LotteryRound.sol	Type:	Code quality

Description

The codebase contains explicit getter functions that return state variable values. These functions can be replaced with public state variables, which automatically generate equivalent getter functions.

In the `LotteryBet` contract:

Listing 10. Excerpt from LotteryBet

```
54 function getPlayer() external view override returns (address) {
55     return player;
56 }
```

Listing 11. Excerpt from LotteryBet

```
61 function getAmount() external view override returns (uint256) {
62     return amount;
63 }
```

In the `LotteryRound` contract:

Listing 12. Excerpt from LotteryRound

```
253 function getTicketsCount() external view returns (uint256) {
254     return ticketsCount;
255 }
```

Listing 13. Excerpt from LotteryRound

```
257 function getBetsCount() external view returns (uint256) {
258     return betsCount;
259 }
```

Listing 14. Excerpt from LotteryRound

```
261 function getFinish() external view returns (uint256) {
262     return finish;
263 }
```

Replacing these explicit getters with public state variables would improve code readability.

Recommendation

Replace explicit getter functions with public state variables where appropriate. For example:

```
// Instead of:
address private player;
function getPlayer() external view returns (address) {
    return player;
}

// Use:
address public player;
```

Fix 1.1

The explicit getters were replaced with public state variables where possible. Due to interface compatibility requirements, some getters were retained.

Go back to Findings Summary

I2: Unclear parameter naming in round creation

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol	Type:	Code quality

Description

The `Lottery` contract implements a `createRound` function that initiates a new lottery round. The function accepts a single parameter named `_timestamp`, which represents the end time of the round.

Listing 15. Excerpt from Lottery

```
142 function createRound(uint256 _timestamp) external onlyRole(SERVICE) returns
    (address) {
```

The same variable name is also used in the `RoundCreated` event.

Listing 16. Excerpt from Lottery

```
66 event RoundCreated(address indexed round, uint256 indexed timestamp);
```

The parameter name `_timestamp` lacks specificity, as it does not indicate that it represents the round's end time.

Recommendation

Rename the `_timestamp` parameter to `_endTimestamp` or `_roundEndTime` to clearly indicate its purpose.

Partial solution 1.1

The `_timestamp` parameter in the `createRound` function was renamed to `_finish`.

The `timestamp` parameter in the `RoundCreated` event was kept as is.

Go back to Findings Summary

I3: Inefficient placing of bets

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol	Type:	Gas optimization

Description

The `Lottery` contract creates a new instance of the `LotteryBet` contract for each bet placement.

Listing 17. Excerpt from Lottery.placeBet

```
126 // create bet contract
127 LotteryBet bet = new LotteryBet(_player, amount, address(this), tokenId,
    _round);
```

This approach is inefficient as it requires deploying a new contract for each bet, resulting in higher gas costs. Implementation of OpenZeppelin's `Clones` library would reduce the gas costs by up to 50%.

Recommendation

Implement a cloning mechanism for the `LotteryBet` contract using a singleton pattern:

- modify immutable variables in the `LotteryBet` contract to be mutable;
- create an `initialize` function in the `LotteryBet` contract with the current constructor logic;
- **add a new check to the `initialize` function ensuring it can only be called once;**
- petrify the `LotteryBet` singleton in its constructor (optional);
- modify the `Lottery` constructor to accept a `LotteryBet` singleton address;
- implement OpenZeppelin's `Clones` library in the `placeBet` function to clone

the singleton and call `initialize` on the clone; and

- add a `setLotteryBetImplementation` function in the `Lottery` contract, restricted to `SERVICE` calls (optional).

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

I4: Unnecessary inheritance of `ERC721URIStorage` extension

Impact:	Info	Likelihood:	N/A
Target:	ERC721URIStorage	Type:	Configuration

Description

The contract inherits the `ERC721URIStorage` extension from OpenZeppelin but does not utilize any of its functionality. This inheritance unnecessarily increases the contract's bytecode size and deployment cost.

The contract implements its own token URI logic, making the `ERC721URIStorage` extension redundant.

Recommendation

Remove the `ERC721URIStorage` extension inheritance and update the `Lottery.supportsInterface` function accordingly.

Fix 1.1

The issue was fixed by removing the `ERC721URIStorage` base contract inheritance.

Go back to Findings Summary

I5: Replace role-based access control with direct contract reference checks for critical functions

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol	Type:	Trust model

Description

The current design of the `Lottery` contract creates a significant security vulnerability in case of `DEFAULT_ADMIN_ROLE` compromise. An attacker with `DEFAULT_ADMIN_ROLE` privileges can drain all tokens from the lottery contract.

The attack sequence is as follows:

1. The attacker grants both `CORE` and `SERVICE` roles to controlled accounts;
2. using the compromised `CORE` role, places bets without transferring tokens to the contract;
3. uses the compromised `SERVICE` role to interrupt the randomness fulfillment process;
4. claims refunds for fraudulent bets, draining legitimate user funds; and
5. extracts all remaining tokens from the `Lottery` contract.

This implementation violates the principle of least privilege and introduces unnecessary centralization risk.

Recommendation

Replace role-based access control with direct sender verification for critical functions since the `Core` contract address is set at construction time and remains immutable.

Implement direct address comparison (`require(msg.sender == coreContract, "Only Core can call")`) for functions involving token transfers. This ensures

that core payment verification remains secure even if role-based access control is compromised.

Fix 1.1

The `CORE` role was removed, and the `Core` contract address was used for `placeBet` function access control.

Go back to Findings Summary

I6: Misleading event name

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol	Type:	Logic error

Description

Listing 18. Excerpt from Lottery

```
162 function removeConsumer(address _round) external onlyRole(SERVICE) {
163     require(rounds[_round], "LT02");
164     uint256 status = LotteryRound(_round).getStatus();
165     require(status == 4 || status == 6, "LT13");
166     coordinator.removeConsumer(subscriptionId, address(_round));
167     emit RoundFinished(_round);
168 }
```

The `removeConsumer` function in the `Lottery` contract emits a `RoundFinished` event.

The function succeeds only when the round is finished or refunding has started. However, the main functionality is only to remove a consumer from the round.

This discrepancy between the event name and the function's actual behavior may mislead developers, off-chain monitoring systems, and users who rely on these events.

Recommendation

Implement one of these solutions:

- rename the event to accurately reflect the function's purpose, such as `ConsumerRemoved` or `ConsumerCleanup`;
- create a new event that accurately describes the function's behavior; or

- document this event emission pattern in the codebase.

Fix 1.1

The issue was fixed by renaming the `RoundFinished` event to `RoundRemoved`.

Go back to Findings Summary

I7: Unused state variable

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol	Type:	Unused code

Description

Listing 19. Excerpt from Lottery

```
54 CoreInterface private core;
```

The `Lottery.core` state variable is never used after being set in the constructor.

Recommendation

Either:

- remove the state variable; or
- use the state variable (as suggested in I5).

Fix 1.1

The `core` state variable is now used according to the I5 recommendation.

Go back to Findings Summary

I8: Typographical error in error message description

Impact:	Info	Likelihood:	N/A
Target:	LotteryRound.sol	Type:	Code quality

Description

The `LotteryRound` contract contains a typographical error in its custom error message.

Listing 20. Excerpt from LotteryRound.sol

```
27 * LR12: invalidat round status to request
```

The error message uses the non-existent word ”invalidat” instead of ”invalid”.

Recommendation

Replace ”invalidat” with ”invalid” in the error message.

Fix 1.1

The typo was fixed.

Go back to Findings Summary

I9: Variables can be immutable

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol, MultiBet.sol	Type:	Code quality

Description

The codebase contains multiple variables that can be made immutable. These variables are assigned only once during contract deployment and never modified afterward. Additionally, none of the contracts is expected to be deployed behind a proxy.

See Appendix B for the complete list of affected variables.

Recommendation

Declare the variables as `immutable`.

Fix 1.1

All of the variables were declared as `immutable`.

Go back to Findings Summary

I10: Unused using-for directives

Impact:	Info	Likelihood:	N/A
Target:	Lottery.sol, LotteryRound.sol	Type:	Code quality

Description

The codebase contains two `using-for` directives of the SafeERC20 library without any usage. Since the code only interacts with the `BET` ERC-20 token which is fully compliant, it is safe not to use the library and remove the `using-for` directives.

See Appendix B for the complete list of affected `using-for` directives.

Recommendation

Remove the unused `using-for` directives.

Fix 1.1

The unused `using-for` directives were removed.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Member's Name	Position
Michal Převrátil	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

This revision implements the fixes for previously identified findings and removes the round deadline postponement feature from the system.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, BetFin: Lottery, 25.3.2025.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Fuzzing

The following table lists all implemented execution flows in the Wake fuzzing framework.

ID	Flow	Added
F1	Creating a new lottery round	1.0
F2	Editing tickets	1.0
F3	Transferring lottery tickets between addresses	1.0
F4	Updating the finish timestamp	1.0
F5	Setting and updating ticket prices	1.0
F6	Placing multiple bets via the `MultiBet` contract	1.0
F7	Placing bets through partner integrations	1.0
F8	Requesting randomness	1.0
F9	Fulfilling randomness via Chainlink VRF	1.0
F10	Processing the jackpot	1.0
F11	Claiming rewards	1.0
F12	Recovering from failed or stuck rounds	1.0
F13	Initiating the refund process for a round	1.0
F14	Executing refunds	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Lottery view functions return expected values across all states	1.0	Success
IV2	Round view functions return expected values across all states	1.0	Success
IV3	Round status transitions follow the expected state machine	1.0	Success
IV4	Bet view functions return expected values across all states	1.0	Success
IV5	Bet data remains consistent throughout lottery operations	1.0	**Fail** (W3)
IV6	Claimable rewards are correctly calculated based on winning tickets	1.0	Success
IV7	Accounts' and contracts' ERC20 token balances reconcile correctly after all lottery operations	1.0	Success
IV8	Accounts' and contracts' native token balances reconcile correctly after all lottery operations	1.0	Success
IV9	Event emission correctness and consistency	1.0	**Fail** (I6)

Table 5. Wake fuzzing invariants

B.2. Detectors

```
wake detect unused-using-for

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
 37  * LT13: Invalid round status to remove
 38  */
 39 contract Lottery is GameInterface, AccessControl, ERC721, ERC721Enumera
❱ 40     using SafeERC20 for IERC20;
 41
 42     bytes32 public constant SERVICE = keccak256("SERVICE");
 43     bytes32 public constant ROUND = keccak256("ROUND");
src/Lottery.sol

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
 29  * LR14: could not recover funds - conditions are not met
 30  */
 31 contract LotteryRound is VRFConsumerBaseV2Plus {
❱ 32     using SafeERC20 for IERC20;
 33
 34     uint256 public constant REQUEST_PERIOD = 1 hours;
 35     uint256 public constant RECOVER_PERIOD = 36 hours;
src/LotteryRound.sol
```

Figure 1. Unused using-for directives

wake detect variable-can-be-immutable

```
[INFO][HIGH] Variable can be immutable [variable-can-be-immutable]
  50      uint256 private immutable created;
  51      IVRFCoordinatorV2Plus private immutable coordinator;
  52      bytes32 private immutable keyHash;
❱ 53      StakingInterface private staking;
  54      CoreInterface private core;
  55      ERC20 private token;
src/Lottery.sol

[INFO][HIGH] Variable can be immutable [variable-can-be-immutable]
  51      IVRFCoordinatorV2Plus private immutable coordinator;
  52      bytes32 private immutable keyHash;
  53      StakingInterface private staking;
❱ 54      CoreInterface private core;
  55      ERC20 private token;
  56
  57      uint256 public additionalJackpot;
src/Lottery.sol

[INFO][HIGH] Variable can be immutable [variable-can-be-immutable]
  52      bytes32 private immutable keyHash;
  53      StakingInterface private staking;
  54      CoreInterface private core;
❱ 55      ERC20 private token;
  56
  57      uint256 public additionalJackpot;
  58      uint256 public subscriptionId;
src/Lottery.sol

[INFO][HIGH] Variable can be immutable [variable-can-be-immutable]
  10   * MB01 - invalid length of input data
  11   */
  12  contract MultiBet is IERC721Receiver {
❱ 13      Token public token;
  14      address public core;
  15
  16      constructor(address _token, address _core) {
src/MultiBet.sol

[INFO][HIGH] Variable can be immutable [variable-can-be-immutable]
  11   */
  12  contract MultiBet is IERC721Receiver {
  13      Token public token;
❱ 14      address public core;
  15
  16      constructor(address _token, address _core) {
  17          token = Token(_token);
src/MultiBet.sol
```

Figure 2. Variables that can be made immutable

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz