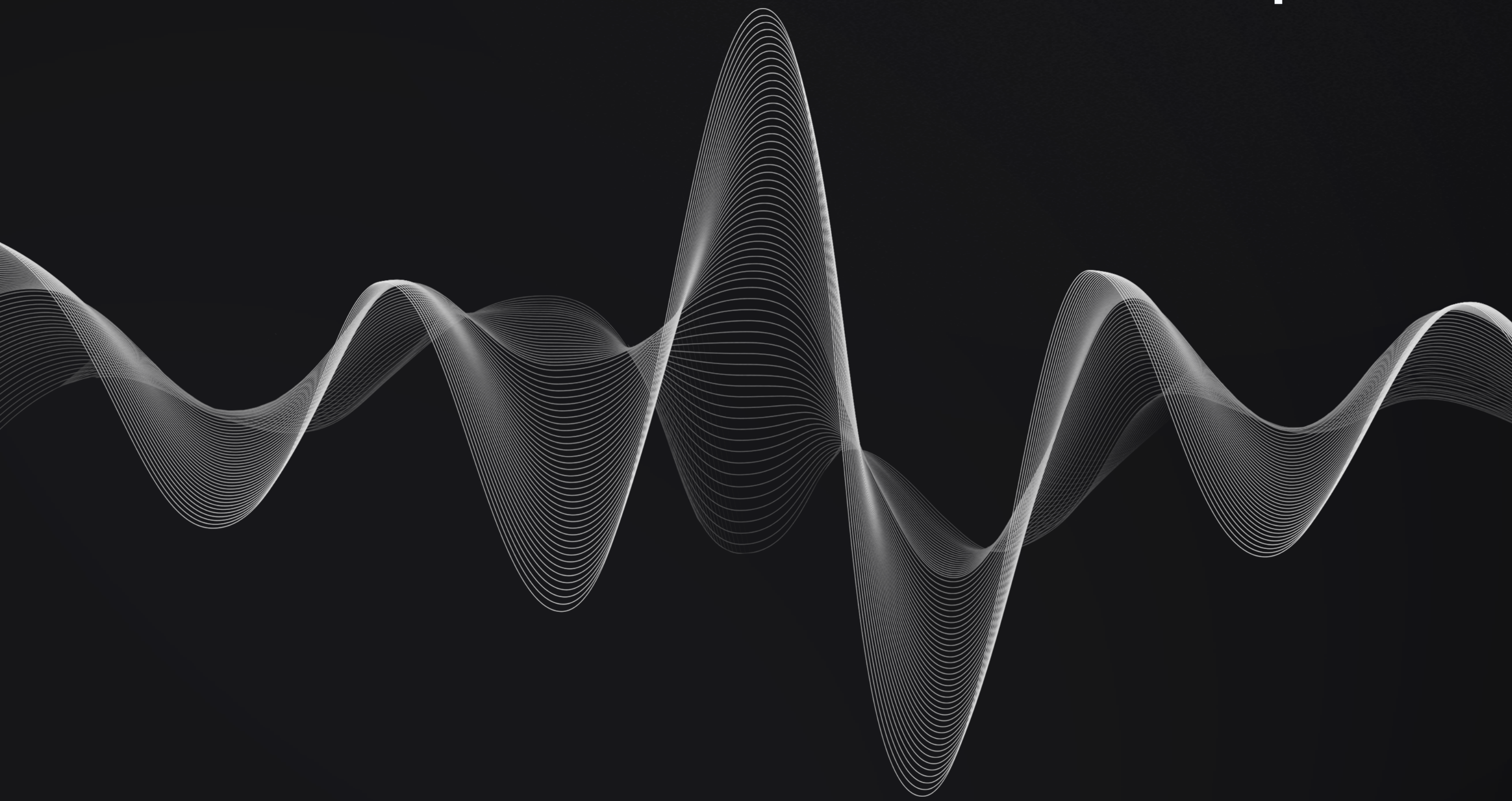
RESONANCE

UBET

UBET Sports

UBET Sports Betting Platform Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.0)

Audit_Report_UBET-SBP_FINAL_20

May 30, 2023	○	v0.1	Ilan Abitbol: Initial draft
May 6, 2023	○	v0.2	Ilan Abitbol: Added findings
May 8, 2023	○	v0.3	Michał Bazyli: Added findings
May 9, 2023	○	v0.4	João Simões: Final draft
May 9, 2023	●	v1.0	Charles Dray: Approved
May 9, 2023	○	v1.1	Michał Bazyli: Revisions
Jul 28, 2023	○	v1.2	João Simões: Reviewed findings
Sep 6, 2023	●	v2.0	Charles Dray: Published

Points of Contact	Daniel J. Im	Ubet Sports	daniel.im@sportsinference.com
	Alexander Kondratskiy	Ubet Sports	alex.kondratskiy@sportsinference.com
	Charles Dray	Resonance	charles@resonance.security
	Luis Lubeck	Resonance	luis@resonance.security
Testing Team	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security
	João Simões	Resonance	joao.simoes@resonance.security
	Michał Bazyli	Resonance	michal.bazyli@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Ubet Sports contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between May 30, 2023 and June 9, 2023. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 3 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted *10 days* to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Ubet Sports with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

Ubet Sports is a decentralized betting platform powered by smart contracts. The platform's fundamental principle is that correct bet outcomes result in payouts, while incorrect ones do not.

This platform operates on Ethereum-compatible chains **Polygon**, using **Solidity**, the programming language used for writing smart contracts.

The system is primarily composed of four key components: the end-user's interface for placing bets, the automated market maker (AMM) used to dynamically adjust odds and manage liquidity, the various smart contracts that provide the infrastructure for betting and payouts, and the on-chain registry which interfaces with the platform's token management system.

As a general workflow, a user's bet is forwarded to the appropriate MarketMaker contract through the user interface. This process involves the use of a transaction, which is propagated through the various components of the system. At the end of the chain, these transactions validate the outcome of sports events and trigger appropriate payouts to users.

For every odd, a market is created and the odds can be modified based:

- On liquidity provided inside our targeted market;
- Wager size;
- External market odds.

Concerning this last point, Ubet Sports platform utilizes a hybrid infrastructure combining blockchain technology with traditional cloud services provided by AWS.

One of the critical aspects of this hybrid infrastructure is the updating of sports betting odds. UBET Sports relies on external APIs such as LSportsRMQ and LSports Fixture REST API to access real-time sports data and odds. This data is then processed on the platform's internal AWS infrastructure to calculate the dynamic odds for each AMM.

Repository Coverage and Quality

This section of the report has been concealed by the request of the customer.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `SportsFI-UBet/ubet-contracts-v1/contracts`
- Hash: e26ddc8778662cbac3b9751c06362493f58ee298

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial-related attack vectors
- **UbetBucks, CashDistributor**. This ERC20 token is used exclusively for testnet and test mocking purposes and is upgradable. Any ERC20 token could be used for betting on the markets.

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- ▁ **"Quick Win"** Requires little work for a high impact on risk reduction.
- ▄ **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- █ **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	🟪	Frontrunning setParentPool() Results in Setting Malicious Parent Pool	█	Resolved
RES-02	🟥	Wages Of Bettors Can Be Replicated	█	Acknowledged
RES-03	🟥	Bypass Calculations Of calcReturnAmount() On removeCollateral()	▄	Resolved
RES-04	🟥	Integer Overflow On calcCostBasisReduction() Leads To Denial Of Service	▄	Resolved
RES-05	🟥	Casting Overflow	▄	Resolved
RES-06	🟧	Numerous Small Buy Orders Lead To Fee Bypass	▁	Resolved
RES-07	🟧	Bets Can Be Fronrunned To Keep Denying Every Users Bets On A Market	█	Acknowledged
RES-08	🟧	Minimum Buy Amount Check Bypass	▁	Acknowledged
RES-09	🟧	Missing Token Validation In ConditionalToken Can Lead To Unexpected Behavior	▁	Resolved
RES-10	🟧	Insufficient Usage Of Pausable Functionality	▄	Resolved
RES-11	🟩	Immutable interfaceId Leads to Impossibility of Upgrading Interfaces	▄	Acknowledged

RES-12	Different Validation Conditions When ERC1155 Tokens Are Received		Acknowledged
RES-13	isHalted() Allows Miners To Gain Unfair Advantage		Resolved
RES-14	Missing Validation Of poolValue On batchRemoveChildShares()		Resolved
RES-15	No Usage Of OpenZeppelin’s Math Library		Resolved
RES-16	Unnecessary Initialization Of Variables With Default Values		Acknowledged

Frontrunning setParentPool() Results in Setting Malicious Parent Pool

Critical **RES-UBET-SBP01** Transaction Ordering **Resolved**

Code Section

- `contracts/funding/ChildFundingPool.sol#L16-L30`

Description

Any `MarketMaker` market created in the Ubet Sports platform is a `ChildFundingPool`. This means that any market can be set with a parent pool through the function `setParentPool()`. This function allows the `_parent` storage variable to change only once, rendering next attempts to change the parent pool unfruitful. Once a parent has been set for a specific child pool, the only thing that can be changed is the amount of funding approved.

The creation of the market is an atomic operation in relation to the `setParentPool()` operation. This means that, any market may or may not have an associated parent pool. Because of this design decision, it is possible for a malicious actor to frontrun `setParentPool()` transactions and assign the parent pool of every market to their own controllable parent pool. This will severely impact various other future funding operations.

Recommendation

It is recommended to either perform access control based on a whitelist when setting a new parent pool, or set the parent pool immediately during the market's creation, mandating every child funding pool to have one parent pool.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Wages Of Bettors Can Be Replicated

High | **RES-UBET-SBP02** | Business Logic | **Acknowledged**

Code Section

- Not specified

Description

In the Ubet DeFi betting ecosystem, bets placed by any bettor are visible on the blockchain. This transparency, while typical for blockchain operations, opens the potential vulnerability of bets placed by famous or highly successful bettors being sniffed out and replicated. In other words, **less experienced or opportunistic bettors could observe and mirror the betting strategy of a well-known bettor**, hoping to achieve similar success. This could potentially impact the betting ecosystem in a few significant ways.

Threat Scenario Example:

Alice is a bettor on Ubet with a very high success rate. Her winning strategy is well-known within the Ubet community, and she's earned a significant reputation as a result. Bob, on the other hand, is a new bettor who's looking for a quick way to increase his profits. He discovers that he can see Alice's bets on the blockchain and decides to start replicating her strategy, placing the same bets as Alice.

Soon, word spreads within the community, and other bettors start replicating Alice's bets too. The market is now flooded with bets that mirror Alice's strategy. This begins to have multiple impacts on the Ubet ecosystem:

1. The odds skew significantly as they dynamically adjust to the increased number of similar bets.
2. The slippage increases, changing the odds unfavourably for other bettors.
3. The potential earnings for liquidity providers decrease as the pool of "bad" wagers shrinks.
4. The balance between bettors and liquidity providers is disrupted, impacting the health of the platform.
5. The diversity of betting strategies decreases, which could make the platform less appealing to those who enjoy creating unique betting strategies.

Recommendation

To mitigate this vulnerability, Ubet should consider implementing a Commit-Reveal scheme. In this mechanism, bettors first commit to their bet without disclosing specifics. Only when all bettors have committed their bets do they reveal the details. This would ensure that no one could sniff out and replicate the bets of a famous bettor before the bets are placed, preserving the diversity of betting strategies and maintaining the balance of the betting ecosystem.

Status

The issue was acknowledged by Ubet's team. The development team stated "Ubet aims to create more transparency and community, so being able to see other's bets is part of the experience. The long term vision is being able to buy and sell bets at any moment at the current price, which cannot work with a commit-reveal scheme.".

Bypass Calculations Of calcReturnAmount() On removeCollateral()

High **RES-UBET-SBP03** Business Logic **Resolved**

Code Section

- contracts/funding/ParentFundingPool.sol#L131-L156

Description

The function `removeCollateral()` is used to remove any collateral that was previously funded with `addFunding()`. This function makes use of the function `calcReturnAmount()` inside the `FundingMath` library to calculate how much collateral should be recovered by burning the necessary shares on the `ParentFundingPool`.

When multiple funders fund the same parent pool and any child pool requests funding from the parent pool using `requestFunding()`, any funder of the parent pool attempting to remove their collateral will only receive a percentage of the collateral proportionate to the total supply of shares on the parent pool. This effectively means that a funder may never be able to retrieve the full collateral amount in this situation.

However, a funder may bypass the calculations made on `calcReturnAmount()` and retrieve the full amount of collateral by first using the function `removeChildShares()` and then funding the parent pool again. On the next turn, the funder will be able to withdraw the full amount of collateral. This is possible because the variable funderTotalShares also bases its calculations on collateral that was already removed previously.

Recommendation

It is recommended to update the logic regarding the calculation of the variable `funderTotalShares` to not include collateral removals that were made using other means.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Integer Overflow On calcCostBasisReduction() Leads To Denial Of Service

High **RES-UBET-SBP04** Arithmetic Issues **Resolved**

Code Section

- `contracts/funding/FundingMath.sol#L78-L86`

Description

The function `calcCostBasisReduction()` is used during the burning of shares from the `FundingPool` to calculate how much to reduce the cost basis of the funder. This function multiplies `funderCostBasis` with `sharesToBurn` which both may contain very large values, since they directly depend on the decimals of the `collateralToken` used. As such, for values close to the maximum value of an `uint128`, this calculation may cause an integer overflow which may ultimately prevent legitimate users from removing any collateral.

Recommendation

It is recommended to enforce a maximum amount of decimals for collateral tokens being used in this protocol.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Casting Overflow

High **RES-UBET-SBP05** Arithmetic Issues **Resolved**

Code Section

- `contracts/funding/ParentFundingPool.sol#L148`
- `contracts/funding/ParentFundingPool.sol#L332-L333`

Description

Whenever a variable with a larger amount of representation bits, e.g. `uint256`, is cast down to a variable with a smaller amount of representation bits, e.g. `uint128`, casting overflows may occur. This type of overflow is not detected or reverted automatically by the Solidity language or the Ethereum Virtual Machine, and therefore, must be accounted for manually in the source code of the smart contract.

The functions `removeCollateral()` and `_removeChildShares()` are prone to casting overflows.

Recommendation

It is recommended to perform arithmetic calculations on variables with the same representation bits. Variable cast downs should be accounted for in the source code.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Numerous Small Buy Orders Lead To Fee Bypass

Medium **RES-UBET-SBP06** Data Validation **Resolved**

Code Section

- `contracts/markets/MarketMaker.sol#L267`

Description

In the current implementation of the `batchBuyAffiliate` function, it's possible for a user to create a multitude of small `BuyOrders`, each with a minimal `investmentAmount`, in an effort to bypass or minimize the incurred fees.

The `batchBuyAffiliate` called function executes each `BuyOrder` separately and calculates fees independently for each of them, which might result in rounding down to zero for each small transaction due to the lack of precision in Solidity. This, in turn, **would allow users to repeatedly execute small bets**, potentially bypassing fees that would otherwise be due for a single transaction of an equivalent total value.

This vulnerability would effectively allow a user to carry out small transactions without incurring the expected transaction fees, hence reducing the overall income for the platform and providing an unfair advantage to the bettor.

If we assume the `investmentAmount` is 1 UBCKS (smallest unit of UBCKS), which is 10^-6 UBCKS and we consider the feeDecimal to be 2%, or `0.02 * 10^18` (to account for 18 decimals precision) in the Solidity contract

1. Calculate the fee amount:

   feeAmount = (investmentAmount * feeDecimal) / ONE_DECIMAL

   = (1 * 0.02 * 10^18) / 10^18

   = 0.02

   Due to the precision limitations in Solidity, this `feeAmount` will be rounded down to `0`.

2. Calculate the resulting investment minus fees:

   investmentMinusFees = investmentAmount - feeAmount

   = 1 - 0

   = 1.

   This amount is the same as the initial investment. In terms of UBCKS, it will be equal to 10^-6 UBCKS.

Thus, for very small transactions, the fees are effectively bypassed due to the precision limitations of Solidity. A minimum bet limit or ensuring non-zero fees could help prevent this potential abuse.

Recommendation

Amend the conditional statement `if (investmentAmount == 0) revert InvalidInvestmentAmount();` in the `buyFor` function to also check for a minimum acceptable `investmentAmount`. Implementing a minimum transaction size for each individual `BuyOrder` within `batchBuyAffiliate` will discourage users from creating many small transactions to avoid fees, as the cost of gas would outweigh any potential savings. The modified conditional might look something like this: `if (investmentAmount < MIN_INVESTMENT) revert InvalidInvestmentAmount();`, where `MIN_INVESTMENT` is a constant representing the minimum acceptable transaction size.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Bets Can Be Fronrunned To Keep Denying Every Users Bets On A Market

Medium RES-UBET-SBP07 Denial of Service **Acknowledged**

Code Section

- `contracts/markets/MarketMaker.sol#L276`

Description

A well-resourced attacker can continuously front-run pending transactions and artificially inflate the slippage, causing the transactions to fail due to the `minOutcomeTokensToBuy` parameter check. This effectively could lead to a denial-of-service condition where legitimate users are unable to place bets.

Specifically, this issue is related to the following function:

- `buyFor()` inside smart contract `MarketMaker.sol`: This function is for buying conditional tokens as a bettor. It checks the `minOutcomeTokensToBuy` parameter which represents the minimum amount of outcome tokens that should be received from the transaction. If the calculated `outcomeTokensBought` is less than `minOutcomeTokensToBuy`, the transaction is reverted with `MinimumBuyAmountNotReached` error.

If the attacker front-runs the transaction and buy a bet just before the targeted user, it might change the odds and cause the slippage, hence making the targeted user's `minOutcomeTokensToBuy` check to fail.

Here's a potential threat scenario:

1. The attacker monitors the transactions in the Ethereum mempool and waits for the targeted User to place a large bet.
2. As soon as the targeted User's transaction is sent to the network but before it gets confirmed, the attacker, who has high resources, places a massive bet on the **same** outcome to the targeted User's bet.
3. Given Ubet's dynamic odds calculation, the attacker's bet changes the odds, and as a result, the expected return of the targeted User's bet.
4. When the Targeted User's transaction is mined, the `minOutcomeTokensToBuy` condition in the smart contract is checked. The targeted User's transaction might fail if the actual outcome tokens he's about to buy are less than the minimum he specified, due to the odds shift caused by the attacker's bet.
5. The attacker's strategy can therefore cause the targeted User's bet to be revert. The attacker can keep doing this to disrupt the market, causing a denial-of-service (DoS) condition, where valid users are persistently denied from placing bets.

Recommendation

With the Commit-Reveal mechanism in place, the confidentiality of bets is enhanced, and the information necessary for a successful front-running attack becomes inaccessible. In simple terms, even though an attacker may see the transactions on the blockchain, they won't know the specifics of the bets. They can see that bets are being placed but without any details about what the bets are.

When users place bets, what the blockchain publicly shows is something like this:

- Bet `0xwhatever` - 20 USDC
- Bet `0xdeadbeef` - 50 USDC
- Bet `0xblobblob` - 15 USDC

The hashes (like 0xwhatever, 0xdeadbeef, etc.) are commitments of the bet details, and do not reveal any information about the actual bet (the team chosen, the outcome predicted, etc.). Even if there's a famous bettor with a known address, the attacker only sees how much they bet, but not what they bet on.

So, with the commit-reveal mechanism, the attackers can potentially front-run the transactions, but they won't know what they are front-running. This effectively mitigates the vulnerability, making the platform more secure against front-running attacks.

However, it's crucial to note that this mechanism would introduce additional complexity and the users would have to submit two transactions for every bet: one to commit and one to reveal. This would mean increased gas costs for the users. Despite these complexities, the Commit-Reveal mechanism significantly improves the platform's security against front-running attacks.

Status

The issue was acknowledged by Ubet's team. The development team stated "The minOutcomeTokensBought parameter shields users from shifting odds because of frontrunners. Even though a bet can be denied in this case, it prevents the worse outcome of buying the bet at much worse odds and the loss of money. Regarding the commit-reveal mechanism - it only works in parimutuel betting where the odds are determined at the end of the betting period where the outcome and everyone's bets are revealed. It does not work in continuously priced markets such as ours where the odds are determined at the time of the bet.".

Minimum Buy Amount Check Bypass

Medium **RES-UBET-SBP08** Code Quality **Acknowledged**

Code Section

- `contracts/markets/MarketMaker.sol#L276`
- `contracts/conditions/ConditionalTokens.sol#L178`

Description

The buyFor function in the MarketMaker.sol contract enforces the check `if (outcomeTokensBought < minOutcomeTokensToBuy) revert MinimumBuyAmountNotReached();` to ensure that the minimum buy amount is reached before token purchases. However, when the market conditions that trigger the transaction reversal are met, users can bypass this check by directly calling the splitPosition function in ConditionalToken.sol to mint tokens, thus circumventing the mentioned validation.

Recommendation

To ensure the intended flow and restrict direct exchanges of collateral tokens for conditional tokens via the `ConditionalToken` contract, it is advisable to implement checks within the smart contract code. These checks should validate that the desired flow is followed, requiring users to go through the designated process, such as utilizing the MarketMaker contract for exchanging collateral tokens. By enforcing these checks, the protocol can maintain control over the token exchange mechanism, preventing potential misuse or bypassing of intended procedures.

Status

The issue was acknowledged by Ubet's team. The development team stated "This is technically possible, but does not actually give a meaningful advantage to exploiters. With ConditionalTokens it is always possible to create all outcomes of a condition from collateral by calling splitPosition*. This encodes the invariant that probabilities for all outcomes must sum to one. In the best case scenario the user can then either merge the tokens back to the same collateral, or wait until settlement, when they will also get the same amount of collateral back. There is no financial advantage to be gained from calling splitPosition in general. The case where buying from a MarketMaker has a worse price than splitPosition will be due to low liquidity. In either case, there isn't a financial benefit to be extracted.".*

Missing Token Validation In ConditionalToken Can Lead To Unexpected Behavior

Medium **RES-UBET-SBP09** Data Validation **Resolved**

Code Section

- `contracts/conditions/ConditionalTokens.sol#L178`
- `contracts/conditions/ConditionalTokens.sol#L208`
- `contracts/conditions/ConditionalTokens.sol#L265`
- `contracts/conditions/ConditionalTokens.sol#L291`

Description

The `ConditionalToken` contract enables protocol users to create conditional tokens by providing collateral. However, the functions within the contract lack validation for collateral tokens, which can result in unforeseen issues. For instance, it is possible to mint conditional tokens by exchanging them for a malicious ERC20 token. It is also possible to phish legitimate users to run arbitrary attacker-controlled functions.

Recommendation

To enhance the security of the protocol, it is advisable to validate all external contracts provided as parameters. This validation helps ensure that only trusted and verified contracts are used, mitigating potential risks and vulnerabilities.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Insufficient Usage Of Pausable Functionality

Medium	**RES-UBET-SBP10**	Access Control	**Resolved**

Code Section

- Not specified

Description

The majority of smart contracts used by the Ubet Betting Platform inherit from the contract `AdminExecutorAccess` which in turn inherits from OpenZeppelin's `Pausable`. This contract provides pausable functionalities to smart contracts in case of emergencies so that mostly critical operations may not be executed while the protocol is in a vulnerable state.

Throughout the several smart contracts, the pausable functionality is rarely used and is not included in most of the critical operations, such as, adding and removing funding, creating markets, reporting payouts, etc.

Recommendation

It is recommended to implement code logic that prevents normal critical actions to take place during protocol emergencies. For that, smart contract's `Pausable` modifiers `whenPaused` and `whenNotPaused` should be used to create different branches of logic that avoid changing the state of the platform to an even more unstable state.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Immutable interfaceId Leads to Impossibility of Upgrading Interfaces

Low

RES-UBET-SBP11 Code Quality **Acknowledged**

Code Section

- `contracts/funding/ChildFundingPool.sol#L14`
- `contracts/Markets/BatchBet.sol#L29`

Description

The smart contract `ChildFundingPool` and `BatchBet` contain hardcoded values on variables identifying the `interfaceId` of other smart contracts. This effectively means that the referenced interfaces can never be changed in the future. Since these variables are used in the `supportsInterface()` operation, it will break the purpose of using ERC165 when upgrading the respective smart contracts.

Recommendation

It is recommended not to use hardcoded values on immutable or constant variables as these will most likely break upgradeability and interoperability patterns across the blockchain.

Status

The issue was acknowledged by Ubet's team. The development team stated "We do not intend to change the existing interface - it is there for compatibility between components across upgrades of their implementations. The hardcoded ids ensure our tests catch inadvertent changes to the interface that break backward compatibility. If new functionality must be added, a new and separate interface will be created.".

Different Validation Conditions When ERC1155 Tokens Are Received

Low **RES-UBET-SBP12** Data Validation **Acknowledged**

Code Section

- `contracts/markets/MarketMaker.sol#L292-L319`

Description

The functions `onERC1155Received()` and `onERC1155BatchReceived()` perform similar functionalities, however, the latter performs them in batch in a single function. The functions are very similar in code, however, the function `onERC1155BatchReceived()` contains an additional validation of the from argument. This means that `onERC1155BatchReceived()` only returns successfully on mint operations while `onERC1155Received()` returns successfully both on mint and transfer operations. This difference creates ambiguities for users to make use of one function in detriment of the other.

Recommendation

It is recommended to perform the same set of actions on functions that are meant to be similar, being function calls, data validations, events emissions, and reverts.

Status

The issue was acknowledged by Ubet's team. The development team stated "The difference between the checks is because the batch is meant purely for liquidity provision/minting - when collateral is split into tokens for the liquidity pool. The transfer is allowed for single outcomes to support sells in the future, when a user transfers a single outcome conditional token in order to swap it for collateral.".

isHalted() Allows Miners To Gain Unfair Advantage

Low **RES-UBET-SBP13** Timestamp Dependence **Resolved**

Code Section

- `contracts/markets/MarketMaker.sol#L443-L445`

Description

The function `isHalted()` is used to verify if the deadline for the market trading has been reached. Whenever the market is halted functions such as `sell()`, `_updateFairPrices()`, `addFundingFor()`, `buyFor()` cannot be used. Regular legitimate users have no way of abusing this but miners on the other hand, control the block timestamp when the block is being mined. As such, malicious miners may operate on an interval buffer of about 30 seconds, to manipulate the `block.timestamp` value and execute the stated functions.

For example, a market that has been created with a `haltTime` coincident with the specific event's end time may allow miners to abuse the block.timestamp and enter or exit betting positions after or right before the event has ended. This way, they gain unfair advantage over regular platform users.

Recommendation

It is recommended to implement code logic that does not rely on `block.timestamp` or `block.number` to close the operations of a market. Some examples may be the implementation of a 2-step betting process, or the existence of an event oracle that collects data off-chain and relays the information about events into the smart contracts on a pull-based method.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Missing Validation Of poolValue On batchRemoveChildShares()

Info RES-UBET-SBP14 Data Validation **Resolved**

Code Section

- `contracts/funding/ParentFundingPool.sol#L161-L222`

Description

The functions `removeChildShares()` and `batchRemoveChildShares()` perform similar functionalities, however, the latter performs them in batch in a single function. The functions are very similar in code, however, the function `batchRemoveChildShares()` is missing the same check made on line 171 on `removeChildShares()`.

Recommendation

It is recommended to perform the same set of actions on functions that are meant to be similar, being function calls, data validations, events emissions, and reverts.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

No Usage Of OpenZeppelin’s Math Library

Info **RES-UBET-SBP15** Code Quality **Resolved**

Code Section

- `contracts/Math.sol`

Description

OpenZeppelin implements several standards used all across blockchain development, especially in Solidity and the Ethereum Virtual Machine. These standards primary goal is to unify and normalize development patterns so that the entire blockchain may be more understandable and readily usable.

The Ubet Betting Platform implements source code that mimics or is very similar to components already implemented by OpenZepellin, and could therefore, be switched with the more the standard well-known approach. Such source code includes:

- Function `ceildiv()` on `Math` library, can be substituted with OpenZeppellin’s implementation on `Math` library.
- Function `min()` on `Math` library, can be substituted with OpenZeppellin’s implementation on `Math` library.

It should be noted that the usage of bigger libraries does not constitute an overuse of gas, since only the functions that are used are included on the bytecode of the smart contract.

Recommendation

It is recommended to make use of implemented and audited standards that already solve the necessary functionalities.

Status

The issue has been fixed in 81895c7d36e1b620a85105839ee5168602a04e9f.

Unnecessary Initialization Of Variables With Default Values

Info **RES-UBET-SBP16** Gas Optimization **Acknowledged**

Code Section

- Not specified

Description

In the Solidity programming language, **all variables are automatically initialized to a default value corresponding to their type when they are declared.** For example, integer types are initialized to `0`, boolean types to `false`, and address types to `0x0000000000000000000000000000000000000000`. Explicitly initializing variables to these default values when they are declared is therefore redundant, and since each operation in a contract costs gas, it results in unnecessary gas costs. This could potentially impact the contract's efficiency and the cost of executing its functions. It's important to review the contract's code to identify any instances of this issue and optimize for gas efficiency.

Recommendation

Review your contract's code for variable declarations where the variable is explicitly initialized to the type's default value. Remove the explicit initialization and let Solidity automatically initialize the variable. Here's an example:

Before:

```
uint256 public counter = 0;
```

After:

```
uint256 public counter;
```

In both cases, `counter` will be initialized to 0. However, the latter declaration will cost less gas when the contract is deployed, improving the contract's efficiency.

Status

The issue was acknowledged by Ubet's team. The development team stated "We followed the recommendations of the slither static analysis tool. We also prefer to be explicit about all variable initialization even if it is zero, to ensure we are not forgetting to set something to a different value. We compile with optimizations turned on, so it should not make a difference to the bytecode or gas cost to explicitly initialize to 0.".

Proof of Concepts

RES-01 Frontrunning setParentPool() Results in Setting Malicious Parent Pool

ParentFundingPool.t.sol (added lines):

```
function testAddParentFrontrun() public {
    // Malicious
    ParentFundingPool parentPoolMalicious = new
↪   ParentFundingPool(makeAddr("random-user"), erc20);
    childPools[0].setParentPool(address(parentPoolMalicious), 0);

    // Legitimate
    childPools[0].setParentPool(address(parentPool), 100);
}
```

RES-03 Bypass Calculations Of calcReturnAmount() On removeCollateral()

MarketFundingPool.t.sol (added lines):

```
function testExploit1() public {
    // Create
    uint256 limit = 1000;
    MarketMakerFactory.PriceMarketParams[] memory params = new
↪   MarketMakerFactory.PriceMarketParams[](1);
    uint256 questionId = 0;
    params[0] = makeMarketParams(questionId, 3600);

    MarketMaker[] memory markets = createMarkets(access.fund.executor, limit,
↪   params);
    MarketMaker market = markets[0];

    vm.prank(access.fund.admin);
    marketFundingPool.setRequestLimit(limit);

    // Fund by Bob
    collateralToken.mintAndApprove(vm, bob, address(marketFundingPool), limit);
    vm.prank(bob);
    uint256 bobFundingShares = marketFundingPool.addFunding(limit);

    // Fund by Alice
    collateralToken.mintAndApprove(vm, alice, address(marketFundingPool), limit);
    vm.prank(alice);
    uint256 aliceFundingShares = marketFundingPool.addFunding(limit);

    // Add the child pool
    vm.prank(access.fund.executor);
    marketFundingPool.setApprovalForChild(address(market), limit);

    // increase limits
    vm.prank(access.fund.admin);
    marketFundingPool.setRequestLimit(limit);

    // Request funding into child pool
    vm.prank(address(market));
    marketFundingPool.requestFunding(limit);

    // Exploit Start //
    vm.prank(bob);
    marketFundingPool.removeChildShares(address(market), bobFundingShares);

    collateralToken.mintAndApprove(vm, bob, address(marketFundingPool), limit);
    vm.prank(bob);
    marketFundingPool.addFunding(limit);

    //vm.prank(address(market));
    //marketFundingPool.requestFunding(limit);
    // Exploit End //

    vm.prank(bob);
    marketFundingPool.removeCollateral(bobFundingShares);

    marketFundingPool.balanceOf(bob);
}
```

RES-04 Integer Overflow On calcCostBasisReduction() Leads To Denial Of Service

MarketFundingPool.t.sol (added lines):

```
function testExploit2() public {
    // Create
    uint256 limit = type(uint128).max;
    MarketMakerFactory.PriceMarketParams[] memory params = new
↪   MarketMakerFactory.PriceMarketParams[](1);
    uint256 questionId = 0;
    params[0] = makeMarketParams(questionId, 3600);

    MarketMaker[] memory markets = createMarkets(access.fund.executor, limit,
↪   params);
    MarketMaker market = markets[0];

    vm.prank(access.fund.admin);
    marketFundingPool.setRequestLimit(limit);

    // Fund by Bob
    collateralToken.mintAndApprove(vm, bob, address(marketFundingPool), limit);
    vm.prank(bob);
    uint256 bobFundingShares = marketFundingPool.addFunding(limit);
    collateralToken.mintAndApprove(vm, bob, address(marketFundingPool), limit);
    vm.prank(bob);
    marketFundingPool.addFunding(limit);
    // Fund by Alice
    collateralToken.mintAndApprove(vm, alice, address(marketFundingPool), limit);
    vm.prank(alice);
    uint256 aliceFundingShares = marketFundingPool.addFunding(limit);

    // Add the child pool
    vm.prank(access.fund.executor);
    marketFundingPool.setApprovalForChild(address(market), limit);

    // increase limits
    vm.prank(access.fund.admin);
    marketFundingPool.setRequestLimit(limit);

    // Request funding into child pool
    vm.prank(address(market));
    marketFundingPool.requestFunding(limit);

    // Exploit Start //
    vm.prank(bob);
    marketFundingPool.removeCollateral(bobFundingShares);
    // Exploit End //
}
```