SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Fore Protocol
Date: 24 October, 2023

Document

Name	Smart Contract Code Review and Security Analysis Report for Fore Protocol
Approved By	Paul Fomichov | Lead Solidity SC Auditor at Hacken OU
Tags	Non-fungible Token; Marketplace; Factory; Gamify
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.foreprotocol.io/
Changelog	14.08.2023 - Initial Review 29.09.2023 - Second Review 24.10.2023 - Third Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Fore Protocol (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

FORE Protocol is a peer-to-peer predictions protocol that enables users to create, participate in, and validate prediction markets in a dynamic and play-to-earn format. FORE Protocol provides the architecture to allow users to create a prediction market for almost any outcome, and rewards them for doing so through the redistribution of protocol fees.

The files in the scope:

- *ERC721NFTMarketV1.sol* – base contract for *ForeNftMarketplace.sol*
- *ICollectionWhitelistChecker.sol* - checks if a token can be listed on the *ForeNftMarketplace.sol*
- *ForeNftMarketplace.sol* - the FORE Protocol NFT Marketplace
- *ForeProtocol.sol* - main protocol contract responsible for the storing and validation of the markets data, minting Verifier NFT and increasing or decreasing the power of the Verifier NFT
- *IForeProtocol.sol* - interface for the ForeProtocol.sol
- *IMarketConfig.sol* - interface for the MarketConfig.sol
- *IProtocolConfig.sol* - interface for the ProtocolConfig.sol
- *MarketConfig.sol* - records the values applied for each marketplace separately
- *ProtocolConfig.sol* - stores configuration of the ForeProtocol.sol, applies changes for all markers
- *BasicFactory.sol* - factory contract responsible for creation new BasicMarket.sol
- *BasicMarket.sol* - base protocol prediction market contract
- *library/MarketLib.sol* - library for the *BasicMarket.sol*
- *ForeVerifiers.sol* - Analyst NFT holding, which will allow to verify market results
- *IForeVerifiers.sol* - interface for the ForeVerifiers.sol

Privileged roles

- ForeNftMarketplace.sol:
    - *Owner* - can recover ERC20/NFTs tokens sent to the contract by mistake, can set/update admin and treasury address
    - *Admin* - can add a new , close existing amd modify collections, can update minimum and maximum prices for a token

- ProtocolConfig.sol:
    - *Owner* - can set factory statuses, edit tiers, update market configurations, can change foundation and high guard accounts, marketplace contract address, verifier mint price and market creation price
- BasicMarket.sol:
    - *Factory* - can initialize new market
    - *Verifiers* - can perform verification
    - *High guard* - can resolve a dispute
    - *Market Creator* - can withdraw market creator reward
- ForeProtocol.sol:
    - *Owner* - can change base URI
    - *Whitelisted factory* - can create new markets
- ForeVerifiers.sol:
    - *Owner - can change base URI, protocol contract, enables or disables transferability feature*
    - *Protocol - can mint token with defined power*
    - *Fore operator - can increase validation for chosen Id, increase token power, can transfer verifier NFT's from any address*
    - *Fore market - can decrease token power*
    - *Verifier token Id owner - can decrease power*

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are detailed:
  - Project overview is detailed.
  - All roles in the system are described.
  - Use cases are described and detailed.
  - For each contract all futures are described.
  - All interactions are described.
- Technical description is robust:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **9** out of **10**.

- The development environment is configured.
- Solidity Style Guide violations.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Interactions with several users are tested thoroughly.

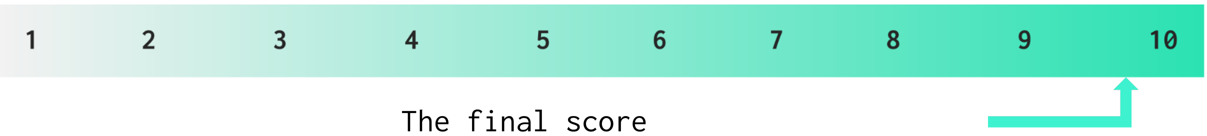
Security score

As a result of the audit, the code contains **1** low severity issue. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.8**.The system users should acknowledge all the risks summed up in the risks section of the report.



The final score

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
14 August 2023	5	6	5	5
29 September 2023	1	0	1	2
24 October 2023	1	0	0	0

Risks

- **Dispute outcomes rely solely on the decisions of the HighGuard**. This centralization can lead to potential biases or inaccuracies in dispute resolutions, putting the fairness of the system at risk.
- Market creators are tasked with setting accurate timelines for predictions. **If a prediction remains active even after real-world results are known**, malicious actors could exploit this oversight, voting with certainty and gaining undue benefits at the expense of regular users. It is crucial to ensure timeline accuracy to maintain the fairness of the prediction market.
- In the event of a dispute, users' tokens and verifiers' NFTs are held in the contract. Access and retrieval of these assets are paused until the dispute is resolved by the HighGuard. This may result in unforeseen delays in accessing assets.
- The platform's prediction market may become "one-sided" when all participants vote for the same outcome. In such cases, the market automatically closes as 'INVALID'. This approach has potential drawbacks:
    - Highly predictable events may lead all users to choose the same result. This can unexpectedly invalidate the market, potentially causing dissatisfaction among participants.
    - When a market becomes one-sided, there's no opposing side to distribute rewards from. As a result, participants may not receive the rewards they anticipated.
- Incorrect market results might be accepted if no dispute is raised within the given 12-hour window. This can lead to users losing rewards to the wrong side and genuine validators being wrongly penalized with potential NFT burns. **Users are advised to actively review validation outcomes and initiate disputes when discrepancies are observed.**
- In case a market is marked as INVALID, the market creator is not only denied any rewards but also forfeits the fee initially paid to create the market. This means market creators have financial disincentives in scenarios where the market outcome is deemed INVALID, potentially leading to financial losses for them.

- The prediction market has **no caps on the maximum amount** a single verifier can contribute towards market verification. As a result, individual verifiers, if sufficiently funded, can influence a large portion or even the entirety of the market's verification.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Failed	L02
Authorization through tx.origin	tx.origin should not be used for authorization.	Not Relevant	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Not Relevant	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Failed	I01
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

C01. Mishandled Edge Case; Data Consistency

Impact	High
Likelihood	High

In the prediction market's verification system, the market is considered "verified" if the larger side's verified amount equals the total amount of the smaller side. This method can be easily manipulated, especially when there is a significant difference between sides. Additionally, a single verifier with a large enough verification power or in markets where one side has a small size can unduly influence the verification outcome.

It can lead to loss of trust in the prediction market's fairness, economic misalignment and potential for manipulation. Verifiers might be incentivized to select the side with larger amounts, skewing outcomes. Markets with large imbalances might end up being verified based on just a few or even a single verifier's input.

Example:

m.sideA = 100 and m.sideB = 1000000.

For the market to be considered "verified" in this scenario:

- If m.verifiedB is at least 100 (total of m.sideA), it is considered verified. This is easily achievable and will likely be the common scenario due to the huge discrepancy.
- Alternatively, if m.verifiedA is at least 1000000, it will also be verified, but this is improbable given the initial imbalance.

In this scenario, due to the huge discrepancy in the amounts, it is most likely that the verification will be achieved through the first condition (m.sideA <= m.verifiedB).

Path: ./contracts/protocol/markets/basic/library/MarketLib.sol : _isVerified()

Recommendation: introduce a dynamic verification threshold based on a percentage of the total amount in both market sides, instead of a fixed threshold. Implement a cap on the maximum amount that can be verified by a single verifier in the market. For instance, no verifier should be able to verify more than a certain percentage (e.g., 10%) of the total market. This prevents undue influence by a single verifier and ensures collective decision-making in the verification process.

Found in: a643ce0

Status: Mitigated (The prediction market's functionality has been revised. Now, a market is deemed "verified" when the verified amount on either side is greater than the total market size. However, it is important to note that a cap on the maximum amount verifiable by an individual verifier has not been introduced.) (Revised commit: 910f87a)

C02. Denial of Service Vulnerability

Impact	High
Likelihood	High

The withdrawVerificationReward function attempts to transfer tokens from its own address using the transferFrom method. Regular ERC20 implementations typically do not allow for this kind of transfer without prior approval calls, leading to potential denial-of-service (DoS) attacks as the function can be made to fail consistently.

Users might be unable to withdraw their rewards, leading to financial losses.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol: withdrawVerificationReward()

Recommendation: replace the transferFrom method with a direct transfer method. Direct transfers from the contract's own balance do not need any allowances and are more straightforward.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

C03. Unauthorized Access To Critical Functions

Impact	High
Likelihood	High

The withdrawVerificationReward function lacks proper access controls, enabling any external party to dictate the mode of withdrawal for the verifier's rewards. This design flaw can be exploited by a malicious actor to control the power distribution among verifier NFTs, potentially gaining undue advantages in future prediction markets.

A malicious actor can prevent certain verifiers from increasing the power of their NFTs. By selectively increasing power for chosen verifiers, an attacker can rig subsequent prediction markets in their favor. Verifiers might face unintended financial consequences or lose opportunities to augment the power of their NFTs.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol: withdrawVerificationReward()

Recommendation: implement proper access controls to restrict the calling of withdrawVerificationReward. Ensure that only the NFT owner or an authorized protocol entity can determine the withdrawal mode. Divide the function into specialized functions. One function should be dedicated to allowing verifiers to either withdraw their rewards or increase their NFT power. Another function can cater to administrators or protocol entities for handling incorrect votes and potential NFT burns. This separation ensures clarity and reduces the attack surface.

Found in: a643ce0

Status: Mitigated (The updated function now enforces a condition that restricts its calling. Specifically, only the associated verifier (linked with the given v.tokenId) or the highGuard entity can invoke the function, ensuring enhanced security.

Recommended structural division of the function into multiple specialized functions has not been implemented. Instead, the decision was made to retain the function's holistic approach, combining reward withdrawal, NFT power increment, and NFT burn mechanisms.

By resolving the access control vulnerability, the risk associated with unauthorized entities manipulating the function and potentially rigging prediction markets has been substantially mitigated.)

C04. Denial of Service Vulnerability

Impact	High
Likelihood	High

The withdrawVerificationReward function tries to execute transfers and burns using the foreVerifiers contract's tokens, but the foreVerifiers contract lacks the ability to grant allowances, resulting in denial-of-service (DoS) vulnerabilities.

Key platform functionalities can become paralyzed due to this oversight, leading to operational disruptions.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol: withdrawVerificationReward()

Recommendation: implement external methods in the foreVerifiers contract that allow trusted markets to request the transfer or burn of assets. These methods should be safeguarded to be callable only by trusted entities (e.g., onlyMarket modifier).

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

C05. Data Consistency

Impact	High

Likelihood	**High**

The function upgradeTier enables a user to upgrade their NFT tier if they meet the verificationsDone requirement. However, the function does not check if the tier they are upgrading to actually exists in the _tiers mapping. As a result, a user might upgrade their NFT to a non-existent tier. If the owner later defines new tiers, this could result in data inconsistency where some users have upgraded to tiers they should not have been able to.

Users might possess NFTs of tiers that they should not have been able to attain based on their verificationsDone count. If the owner of the ProtocolConfig.sol later tries to define new tiers, they would have to handle the already upgraded NFTs, which might now belong to tiers they should not. This could lead to a loss of trust from the user base when they realize that tier upgrades are not consistent or fair.

Path: ./contracts/protocol/ForeProtocol.sol: upgradeTier()

Recommendation: shift from a mapping to an array for storing tiers. This will allow for better control over indices and easier checks for valid tiers. In the upgradeTier function, before allowing the upgrade, check that the tier to which the user is upgrading actually exists. When editing tiers, always ensure the changes maintain data consistency and no user can be in a tier they should not.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

C06. Denial of Service Vulnerability; Invalid Calculations

Impact	High
Likelihood	High

In instances when a verifier votes for an incorrect side and is thus eligible for a penalty, the function designed to calculate the amounts for toDisputeCreator and toHighGuard incorrectly uses the multipliedPowerOf function instead of the powerOf function for calculating amounts to transfer. This discrepancy becomes pronounced in scenarios where the NFT id multiplier exceeds 100%. Such a difference can induce a Denial of Service (DoS) at the line :

```
foreVerifiers.marketBurn(power - toDisputeCreator - toHighGuard);
```

due to the actual power being less than the multipliedPower, which does not genuinely reflect the number of tokens held by that particular NFT id.

When a verifier, who has voted inappropriately, faces a penalty, the protocol mandates that its NFT should be burned. Consequently, the tokens held by this NFT are then redistributed among the Dispute Creator and the High Guard, with any residual tokens being incinerated. The crux of the vulnerability stems from the miscalculation of the amounts designated for toDisputeCreator and toHighGuard, which are presently determined using the multipliedPowerOf function. For an accurate representation of the tokens associated with the penalized NFT, the powerOf function should be employed.

By misapplying the multipliedPowerOf function, there is a risk of over-allocating assets to both toDisputeCreator and toHighGuard, which surpasses the true tokens retained by the NFT. This inconsistency can initiate a Denial of Service (DoS) during the token burn process, thereby possibly disrupting standard operations on the platform, especially during penalty enforcement procedures.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol: withdrawVerificationReward(), calculateVerificationReward

Recommendation: for accurate token calculations during penalty enforcement, the multipliedPowerOf function should be supplanted with the powerOf function in the withdrawVerificationReward and calculateVerificationReward methods.

Incorporate unit tests tailored to ensure the precise distribution of tokens and calculation in scenarios where an NFT verifier is subjected to penalties and multipliedPowerOf of the NFT exceed 100%.

Found in: 4672889

Status: Fixed (Recommended changes are applied. *powerOf* function is used to calculate the amounts to transfer.) (Revised commit: 910f87a)

■■■ High

H01. Mishandled Edge Case; Data Consistency

Impact	High
Likelihood	Medium

The fore operator holds overriding control on the verification NFTs (vNFTs), which creates a centralized point of vulnerability. If the owner's private keys are compromised, an attacker can take over any vNFT, potentially devaluing them and extracting tokens.

A vulnerability arises from the isApprovedForAll and _transfer() functions, which always approve an action if it originates from a foreOperator. With the setFactoryStatus function, there is a risk of introducing an erroneous address as a whitelisted factory or, if the owner's keys are compromised, a malicious address. This address can then be used to operate on any vNFT.

It can lead to malicious or unauthorized control over any verifier NFT, regardless of the actual owner, ability to reset the power of any chosen NFT to its initial state and as a result receiving Fore tokens that can be sold on the DEX.

Path: ./contracts/verifiers/ForeVerifiers.sol : isApprovedForAll(), _transfer()

Recommendation: re-evaluate the need for a universal override in the isApprovedForAll function. Restricting permissions based on roles and use cases can be more secure.

Found in: a643ce0

Status: Mitigated (We keep it by design. Fore Operator will be a MultiSign wallet.)

H02. Requirements Violation; Data Consistency

Impact	High
Likelihood	Medium

The ProtocolConfig contract allows setting of market validation and dispute time periods without ensuring they align with the documentation constraints. Consequently, a market can be created with periods not matching the documented standards.

With periods not restricted, malicious actors might exploit this flexibility, especially if shorter periods don't provide stakeholders adequate time to react. Users might make decisions based on the documentation's defined standards. Deviations can lead to unanticipated actions and losses.

Discrepancies between implementation and documentation erode platform trustworthiness.

Path: ./contracts/protocol/config/ProtocolConfig.sol: constructor(), _setConfig()

Recommendation: implement checks within _setConfig to ensure disputePeriodP is up to 12 hours and verificationPeriodP is up to 24 hours. Revise the constructor's default values for disputePeriodP and verificationPeriodP to ensure they are within the recommended bounds.

Found in: a643ce0

Status: Fixed (Minimum validation and dispute period is now set to 12 hours, maximum for both is set to 96 hours. Default value for both is 24 hours.) (Revised commit: 4672889)

H03. Coarse-Grained Access Control

Impact	High
Likelihood	Medium

The project's design gives sole privilege to the highGuard address to resolve disputes. While the protocolConfig contract's owner can change the highGuard, the exclusive reliance on a single address poses significant security risks. If control over the highGuard private key is compromised, an attacker could manipulate prediction market results for their advantage.

An attacker with control over the highGuard address could unilaterally resolve disputes in their favor, potentially manipulating market outcomes and compromising the integrity of the entire system. Given the attacker's ability to sway market results, they could profit dishonestly, leading to loss for honest participants.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol: resolveDispute()

Recommendation: instead of a single address, require multiple trusted entities to confirm a dispute resolution. This would significantly reduce the risk of manipulation, even if one of the trusted entities gets compromised.

Found in: a643ce0

Status: Mitigated (The HighGuard will be a MultiSign wallet.Therefore we keep this by design.)

H04. Requirements Violation

Impact	Medium
Likelihood	High

Although it is stated that the highGuard address is a bunch of NFT holders, there is no such implementation in the contract. The flexibility of the high guard address allows the system owner to specify any entity, without the requirement for it to be a multi-sig wallet configuration or limited solely to NFT holders.

Path: ./contracts/protocol/config/ProtocolConfig.sol

Recommendation: implement the required functionality or edit the documentation.

Found in: a643ce0

Status: Fixed (Documentation was updated) (Revised commit: 4672889)

H05. Requirement Violation; Data Consistency

Impact	High
Likelihood	Medium

The editTier function lacks comprehensive checks to maintain the ordered and hierarchical structure of minVerifications and multipliers across tiers, potentially allowing for inconsistent tier configurations.

There is a possibility of setting minVerifications for a tier to a value greater than its subsequent tier or lesser than its previous tier, leading to inconsistency in tier structuring.

Similarly, the function does not restrict setting the multiplier value such that it might not maintain a proper hierarchical progression when compared to adjacent tiers. This can cause financial inconsistencies in multiplier calculations across tiers.

Path: ./contracts/protocol/config/ProtocolConfig.sol: editTier()

Recommendation: separate the validation checks for tierIndex == 0 and tierIndex > 0 to avoid oversight. For tierIndex > 0, in addition to checking that the new minVerifications is greater than the previous tier's minVerifications, also ensure that it is less than the next tier's minVerifications (if it exists). For multiplier, enforce a check that ensures a proper order relative to the previous and next tier's multipliers. For the last tier, validate that if there is no subsequent tier, there should be no limitation on its minVerifications or multiplier relative to higher tiers, but they should still be greater than the previous tier's values.

Found in: a643ce0

Status: Fixed (Revised commit: 910f87a)

■■ Medium

M01. Redundant Memory Allocation

Impact	Low
Likelihood	High

The line MarketLib.Market memory m = market; creates an in-memory copy of the storage variable market. Given the size of the struct (multiple variables), this can lead to a significant Gas overhead.

Only one attribute of m (endPredictionTimestamp) is accessed afterward. It is wasteful to create an entire in-memory copy of the market for this purpose.

It will lead to increased Gas cost for every invocation of the function, making predictions more expensive for users.

Path: ./contracts/protocol/markets/basic/library/MarketLib.sol: _predict()

Recommendation: directly access the endPredictionTimestamp from the storage variable market instead of creating a memory copy.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

M02. Best Practices Violation

Impact	Low
Likelihood	High

All events are declared in the MarketLib.sol library. This is unconventional. Usually, main contracts declare events, as users and tools primarily expect to find them there.

Users or tools monitoring events might overlook or not expect events declared in libraries. This can lead to missed logs or additional tracking efforts.

Path: ./contracts/protocol/markets/basic/library/MarketLib.sol

./contracts/protocol/markets/basic/BasicMarket.sol

Recommendation: declare all events in the main contract, even if the library emits them. This way, anyone inspecting the main contract can identify all possible emitted events in one location.

Found in: a643ce0

Status: Fixed (The issue was fixed by introducing the event declarations in the main contract.) (Revised commit: 910f87a)

M03. Absence of ReentrancyGuard for ERC721 Functions

Impact	High
Likelihood	Low

The project's contracts do not utilize the ReentrancyGuard for functions that interact with ERC721 tokens. Although the project adheres to the Checks-Effects-Interactions (CEI) pattern, which can help prevent reentrancy attacks, it remains best practice to implement ReentrancyGuard as an additional security layer.

Without the explicit use of ReentrancyGuard, functions are potentially more exposed to reentrancy attacks even if the CEI pattern is followed. Not using the ReentrancyGuard is a deviation from accepted smart contract development best practices.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol

Recommendation: incorporate the ReentrancyGuard modifier in all functions that interact with ERC721 tokens. This ensures an added layer of security against potential reentrancy attacks.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

M04. Requirements Violation; Data Consistency

Impact	High
Likelihood	Low

The project's NFT marketplace documentation specifies that it should exclusively use the native project ERC20 token for transactions. However, an inherited method, buyTokenUsingBNB, allows for purchases using chain native currency. Although the native ForeToken does not support the deposit method, and a call to buyTokenUsingBNB would fail, bridged tokens on other chains might contain a fallback function, opening a door to unintended behavior.

Successful purchases using native chain currency, instead of the expected ERC20 token, can cause financial discrepancies. Participants may take advantage of this inconsistency, leading to potential losses for others.

Path: ./contracts/external/pancake-nft-markets/ERC721NFTMarketV1.sol: buyTokenUsingBNB()

Recommendation: override the buyTokenUsingBNB method in the ForeNftMarketplace.sol contract to explicitly disallow its execution. It should revert immediately if someone attempts to call it.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

M05. Mishandled Edge Case

Impact	High
Likelihood	Low

The functions withdrawPredictionReward, withdrawVerificationReward, and marketCreatorFeeWithdraw in the contract compute reward values based on various market, verification, and user details. Due to potential rounding discrepancies in mathematical operations, the computed amounts (toWithdraw, toVerifier, etc.) could occasionally surpass the contract's available balance.

If the calculated reward values are higher than the balance, the associated transfer functions will fail. This can hinder users from withdrawing their rightfully earned rewards, potentially eroding trust in the system.

Path: ./contracts/protocol/markets/basic/BasicMarket.sol: withdrawPredictionReward(), withdrawVerificationReward(), marketCreatorFeeWithdraw()

Recommendation: incorporate validation checks within withdrawPredictionReward, withdrawVerificationReward, and marketCreatorFeeWithdraw functions to ascertain that the computed reward values do not exceed the contract's balance. Should rounding cause the reward values to be larger than the available balance, adjust them to equal the contract's balance. Implementing this safeguard ensures the transfer functions operate seamlessly, providing a consistent user experience and preventing undue failures.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

M06. Accumulation of Dust Values

Impact	Low
Likelihood	High

Due to integer division when splitting the toVerifiers fee, there is a possibility of truncating small residual balances, leading to minor discrepancies in token distribution.

Path: ./contracts/protocol/markets/basic/library/MarketLib.sol: closeMarket()

Recommendation: implement a mechanism to handle the division more accurately. Consider rounding up or down consistently, or distribute the residual amounts in subsequent transactions. Alternatively, allocate potential dust to a predetermined category (e.g., toBurn).

Found in: a643ce0

Status: Mitigated (We agree that this could happen, but the amount of FORE that could be left as dust is very minimal.)

■ Low

L01. Missing Events on Critical State Updates

Impact	Low
Likelihood	Low

Critical state changes should emit events for tracking things off-chain.

This can lead to inability for users to subscribe events and check what is going on with the project.

Path: ./contracts/protocol/ForeProtocol.sol: editBaseUri()

ForeVerifiers.sol : editBaseUri(), mintWithPower(), increaseValidation()

Recommendation: emit events on critical state changes.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

L02. Race Condition

Impact	Medium
Likelihood	Low

The functions for market creation and minting verifier NFTs retrieve fees from the ProtocolConfig.sol contract. The owner can change these fees, leading to unpredictability for users. Users might end up paying more than anticipated if the fee is updated during their transaction.

Users might not have sufficient balance for the new fee, leading to failed transactions and wasted Gas fees.

Path: ./contracts/protocol/markets/basic/BasicFactory.sol: createMarket()

ForeProtocol : mintVerifier()

Recommendation: add an additional parameter to the createMarket and mintVerifier functions for the expected fee. Within the function, compare the provided fee against the current fee in the protocol. If they do not match, revert the transaction. This ensures users always pay what they expect.

Found in: a643ce0

Status: Acknowledged (Noted, no changes have been made) (Revised commit: 910f87a)

L03. Unsafe Minting of ERC721 Tokens

Impact	Medium
Likelihood	Low

The createMarket function uses the _mint method to create and assign ERC721 tokens. This method might not be safe when the receiver is a contract without ERC721 support, potentially leading to lost tokens.

If the receiver address is a contract that does not support ERC721 tokens, the minted tokens can become permanently inaccessible.

Path: ./contracts/protocol/ForeProtocol.so: createMarket()

Recommendation: use _safeMint: Switch to the _safeMint method, which contains an internal check to ensure the receiving address can handle ERC721 tokens. If the receiving address is a contract, _safeMint will ensure it implements the required ERC721 functions.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

L04. Unchecked Transfer

Impact	Medium
Likelihood	Low

The project's contracts do not utilize the SafeERC20 library for managing ERC20 token transfers. While all transfers within the protocol employ its native standard ERC20 token, not adhering to best practices can introduce risks, especially when bridged tokens from other chains are considered.

Not using the SafeERC20 library is a deviation from the accepted and recommended best practices for smart contract development.

Path: ./contracts/protocol/ForeProtocol.sol: mintVerifier(), buyPower()

./contracts/protocol/markets/basic/BasicFactory.sol: createMarket()

./contracts/protocol/markets/basic/BasicMarket.sol: predict(), openDispute(), resolveDispute(), _closeMarket(), withdrawPredictionReward(), withdrawVerificationReward(), marketCreatorFeeWithdraw()

./contracts/verifiers/ForeVerifiers.sol: decreasePower()

Recommendation: integrate the SafeERC20 library into the project's contracts. This library wraps around the standard ERC20 functions and reverts transactions if any operation fails, providing a safer mechanism for token operations.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

L05. Redundant Code

Impact	Low
Likelihood	Low

The current implementation of the market closure function executes several unnecessary operations when the market result is ResultType.INVALID. These additional operations waste gas and computational resources.

This lead to increased gas costs for users when handling an INVALID result.

Path: ./contracts/protocol/markets/basic/library/MarketLib.sol: closeMarket()

./contracts/protocol/markets/basic/BasicMarket.sol: closeMarket()

Recommendation: in the closeMarket function of the MarketLib.sol library, the check for ResultType.INVALID should be moved up right after emitting the CloseMarket event. This avoids unnecessary calculations and returns immediately. In the _closeMarket function of the BasicMarket.sol, immediately after obtaining the results from MarketLib.closeMarket, insert an additional check for the INVALID result type. If detected, avoid performing any redundant operations.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

Informational

I01. Style Guide Violation

The provided projects should follow the official guidelines.

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Within a grouping, place the view and pure functions last.

It is best practice to cover all functions with NatSpec annotation and to follow the Solidity naming convention. This will increase overall code quality and readability.

Path: ./contracts/

Recommendation: follow the official Solidity guidelines.

Found in: a643ce0

Status: Acknowledged (Noted and will be adjusted. However for this second iteration of the audit we didn't

make this changes) (Revised commit: 910f87a)

I02. Typo in require Statement

The error message in the require statement:

```
require(multiplier > 0, "ProtocolConfig: 1st tier multiplier musst bu greater than zero");
```

contains a typographical error. Specifically, the word "musst" should be "must" and "bu" should be "be".

The following NatSpec contains a typo.

```
    ///@param pA Prediction contribution for side A

    ///@param pA Prediction contribution for side B
```

The 'pA' that is on the second line needed to be 'pB'.

Path: ./contracts/protocol/config/ProtocolConfig.sol: editTier()

./contracts/protocol/markets/basic/library/MarketLib.sol: calculatePredictionReward()

Recommendation: correct the typos in the code.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

I03. Missing Zero Address Validation

Address parameters are being used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Path: ./contracts/protocol/config/ProtocolConfig.sol: constructor(), setFactoryStatus(), setFoundationWallet(), setHighGuard(), setMarketplace()

Recommendation: implement zero address checks.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

I04. State Variables Can Be Declared Immutable

In the BasicFactory.sol contract, variable *foreProtocol* value is set in the constructor.

In the BasicMarket.sol contract, variable *factory* value is set in the constructor.

Those variables can be declared immutable.

This will lower the Gas taxes.

Path:./contracts/protocol/markets/basic/BasicFactory.sol

./contracts/protocol/markets/basic/BasicMarket.sol

Recommendation: declare mentioned variables as immutable.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

I05. Floating Pragma

The project uses floating pragmas ^0.8.7, ^0.8.4 and ^0.8.0.

Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Path: ./contracts/

Recommendation: consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

I06. Redundant Import

The import of Strings.sol in the ForeVerifiers.sol contract is unnecessary for the contract.

The import of IForeProtocol.sol in the ForeProtocol.sol contract is unnecessary for the contract.

Unused imports should be removed from the contracts. Unused imports are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can decrease readability.

Path: ./contracts/verifiers/ForeVerifiers.sol

Recommendation: remove the redundant import.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

I07. Redundant Block

The section checking protocol.isForeMarket(msg.sender) permits the market to fully reduce power. However, the market does not implement functionality to call decreasePower(). This creates a discrepancy.

Path: ./contracts/verifiers/ForeVerifiers.sol: decreasePower()

Recommendation: remove the redundant code to ensure clarity. If the market isn't intended to decrease power, eliminate the corresponding conditions.

Found in: a643ce0

Status: Mitigated (no changes have been made as we will use this function in a future iteration of the market contracts)

I09. Outdated Encoder Use

The statement pragma abicoder v2 is outdated.

Path: ./contracts/external/pancake-nft-markets/ERC721NFTMarketV1.sol

Recommendation: Use “pragma abicoder v2” instead or use a contemporary compiler version. The Solidity versions >= 0.8.0 uses Abicode v2 by default.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

I10. Empty Code Block

An empty code block is detected on lines 26-28.

The presence of an empty code block might lead to confusion, reduced code readability, potential misinterpretation and implication of unfinalized code, without immediate functional consequences.

Path: ./contracts/marketplace/ForeNftMarketplace.sol

Recommendation: remove the empty code block.

Found in: a643ce0

Status: Fixed (Revised commit: 4672889)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	Medium
Medium Likelihood	**High**	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and in most cases cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/FOREProtocol/contracts
Commit	a643ce084d338aa5e8cca6613d8bbb9f55e696ba
Whitepaper	Link
Requirements	https://docs.foreprotocol.io/home/documentation
Technical Requirements	https://docs.foreprotocol.io/home/documentation
Contracts	File: contracts/external/pancake-nft-markets/ERC721NFTMarketV1.sol SHA3: db22708b84144cf05c5563daf3e938381347d01d9637e3c923fc603664fe9b74 File: contracts/marketplace/ForeNftMarketplace.sol SHA3: 3ea6e0d83cfa7cc454c51221d22f6c24e2590bbeddcd5a2bd4282f7cb69e6af1 File: contracts/protocol/ForeProtocol.sol SHA3: f9d64999f0fde2892e1aef205353f7655f2cde1eb58d8d6d3484a0c79f2e7c5c File: contracts/protocol/config/IMarketConfig.sol SHA3: 30fd1ce166d6e4a9fe26444b6e166d4711eaa09e8e6b6e0aa120d721f15d696a File: contracts/protocol/config/MarketConfig.sol SHA3: 6fe5ebc60b144190e29934c20ffeba40cd713cfb096fa8dcef4b80bcf4aa12e2 File: contracts/protocol/config/ProtocolConfig.sol SHA3: 4dbb907795e66ddf277308e50ad302c25d6839d5e1c9b402d0c1a3d4c39a5bec File: contracts/protocol/markets/basic/BasicFactory.sol SHA3: 79f1f39e454fb98e7ab02e512a9d4a7a911e29a7202b3b07588a6c161e3ba92a File: contracts/protocol/markets/basic/BasicMarket.sol SHA3: 29146f5633ad6965a455974ef9a9306758bac058ef1363a3945aad6412eb420f File: contracts/protocol/markets/basic/library/MarketLib.sol SHA3: 3c887b9c19df13b2b328305648928ab178338bba1a005c90c8d9b974b546f38b File: contracts/verifiers/ForeVerifiers.sol SHA3: a7c2f6079b1d24765594e02a21b24045f7783e84025626661fe540605a9decbf File: contracts/verifiers/IForeVerifiers.sol SHA3: e1952f4d10c5ffd9516648f626c3a1cf8fd9969e568a4fb6698ab088cac3d130 File: contracts/protocol/config/IProtocolConfig.sol SHA3: 56ddc1276b1b5dc7f003cfa0685e095effae7ba9f3b7606706f1bee80d0f47e3 File: contracts/protocol/IForeProtocol.sol SHA3: 79a0fbc38dc11cd5dd2fcd8d217bc29aa2d4cc7e5f64b7a79f3854158f8824c2

Second review scope

Repository	https://github.com/FOREProtocol/contracts
Commit	4672889a61a9cd4455aac1d9680fe2cb3eaa3fea
Whitepaper	Link
Requirements	https://docs.foreprotocol.io/home/documentation
Technical Requirements	https://docs.foreprotocol.io/home/documentation
Contracts	File: contracts/external/pancake-nft-markets/ERC721NFTMarketV1.sol SHA3: 63d4abdfdc0f4ae9dd1ff97c02b5a98d3bf171a15cdbf779136dc338f61ec87b File: contracts/marketplace/ForeNftMarketplace.sol SHA3: c61ac96dfe4db443ca9c8053282f19ac868067b64b1613de1dc2305dd5dc309b File: contracts/protocol/ForeProtocol.sol SHA3: 7e3df5c6166a5734ce3524d425a52ce2b6ba0cb2ed0f4cf2da685854614470a6 File: contracts/protocol/IForeProtocol.sol SHA3: 94b04e53fcb5a3be02d8052cc85d199cec9bc1f64ab54389349632fde815adc3 File: contracts/protocol/config/IMarketConfig.sol SHA3: deb2e1c69381303431e397e9bdfaaf202251ec820d34c06bf187dbbbece0f084 File: contracts/protocol/config/IProtocolConfig.sol SHA3: 0d24f195f97f187a4d5985b922813b27d1e418c2851ddf34380ec651b8333518 File: contracts/protocol/config/MarketConfig.sol SHA3: 125825f9a19c9056fb8dd7782860f29ef77af500595190741e49cb3021364f53 File: contracts/protocol/config/ProtocolConfig.sol SHA3: 2888d66cf4296055637d8692c4033d6149c908a2f2850003909e2950b92af025 File: contracts/protocol/markets/basic/BasicFactory.sol SHA3: b5bf30a13e60bd361435adf2a03f7d980c9a37d56490b208750d770dade34dbb File: contracts/protocol/markets/basic/BasicMarket.sol SHA3: e8a39ff00c736bc306efa9db958dc11fbc82e42c1da24e930c5aa2344e1c1ec7 File: contracts/protocol/markets/basic/library/MarketLib.sol SHA3: 3652451ed079a7c0da48058f79a1f201aec4ea9b60ab12199d3cdfd9e0a1f67c File: contracts/verifiers/ForeVerifiers.sol SHA3: 854907873a5d588d09bef675dae5ba2689667746fd3731abaf52c1254421905d File: contracts/verifiers/IForeVerifiers.sol SHA3: 6349deb059fe449ee8955d61a8003885ebe62cd44ab7c6272dfcd1d3dbed9fe1

Third review scope

Repository	https://github.com/FOREProtocol/contracts
Commit	910f87a02874128e12b94637b6b5514c790c7bf2

Whitepaper	Link
Requirements	https://docs.foreprotocol.io/home/documentation
Technical Requirements	https://docs.foreprotocol.io/home/documentation
Contracts	File: contracts/external/pancake-nft-markets/ERC721NFTMarketV1.sol SHA3: 63d4abdfdc0f4ae9dd1ff97c02b5a98d3bf171a15cdbf779136dc338f61ec87b File: contracts/marketplace/ForeNftMarketplace.sol SHA3: c61ac96dfe4db443ca9c8053282f19ac868067b64b1613de1dc2305dd5dc309b File: contracts/protocol/ForeProtocol.sol SHA3: 7e3df5c6166a5734ce3524d425a52ce2b6ba0cb2ed0f4cf2da685854614470a6 File: contracts/protocol/IForeProtocol.sol SHA3: 94b04e53fcb5a3be02d8052cc85d199cec9bc1f64ab54389349632fde815adc3 File: contracts/protocol/config/IMarketConfig.sol SHA3: deb2e1c69381303431e397e9bdfaaf202251ec820d34c06bf187dbbbece0f084 File: contracts/protocol/config/IProtocolConfig.sol SHA3: 0d24f195f97f187a4d5985b922813b27d1e418c2851ddf34380ec651b8333518 File: contracts/protocol/config/MarketConfig.sol SHA3: 125825f9a19c9056fb8dd7782860f29ef77af500595190741e49cb3021364f53 File: contracts/protocol/config/ProtocolConfig.sol SHA3: 2888d66cf4296055637d8692c4033d6149c908a2f2850003909e2950b92af025 File: contracts/protocol/markets/basic/BasicFactory.sol SHA3: b5bf30a13e60bd361435adf2a03f7d980c9a37d56490b208750d770dade34dbb File: contracts/protocol/markets/basic/BasicMarket.sol SHA3: e8a39ff00c736bc306efa9db958dc11fbc82e42c1da24e930c5aa2344e1c1ec7 File: contracts/protocol/markets/basic/library/MarketLib.sol SHA3: 3652451ed079a7c0da48058f79a1f201aec4ea9b60ab12199d3cdfd9e0a1f67c File: contracts/verifiers/ForeVerifiers.sol SHA3: 854907873a5d588d09bef675dae5ba2689667746fd3731abaf52c1254421905d File: contracts/verifiers/IForeVerifiers.sol SHA3: 6349deb059fe449ee8955d61a8003885ebe62cd44ab7c6272dfcd1d3dbed9fe1