hashlock.

Security Audit

No Gain No Pain (DeFi)

May 2025 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

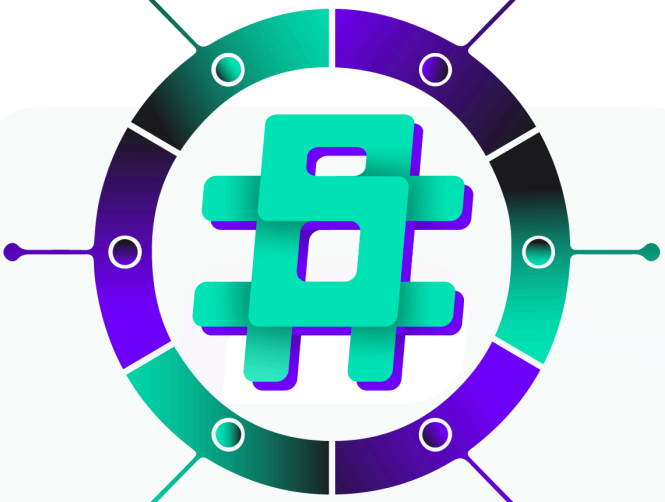
The No Gain No Pain team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

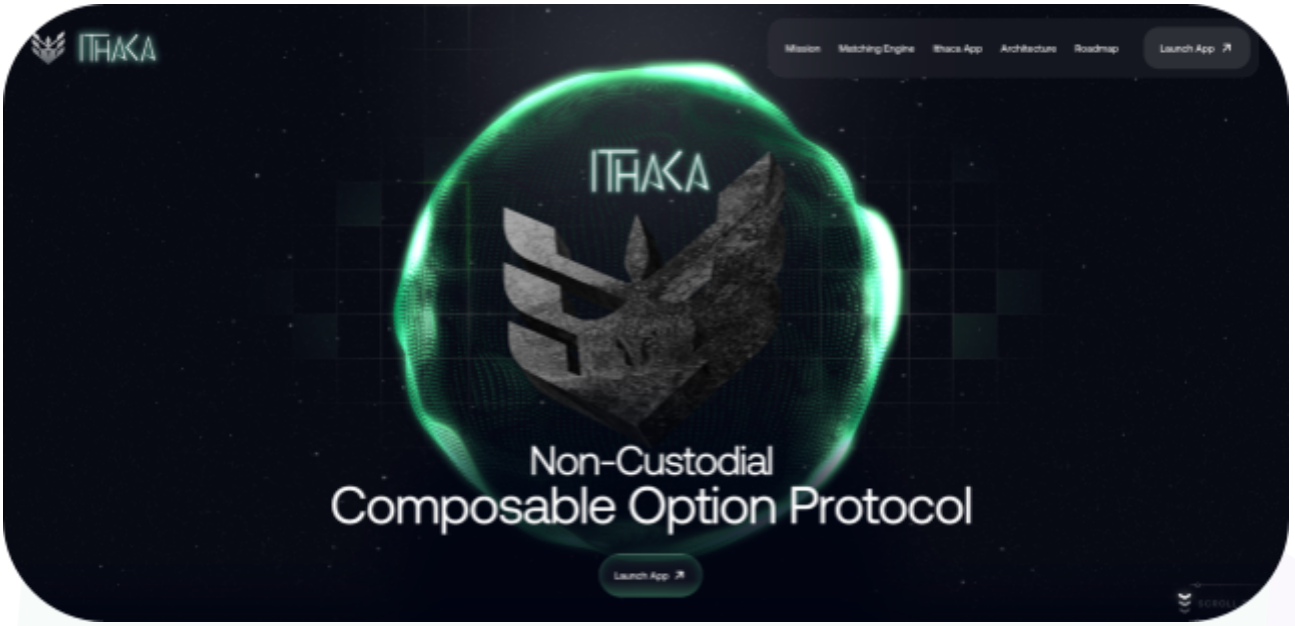
No Gain No Pain is a non-custodial DeFi dApp built on the Ithaca Protocol, a decentralized finance platform designed to enhance liquidity provision and decentralized governance. The protocol leverages advanced automated market-making (AMM) mechanisms to facilitate seamless trading and liquidity management, while empowering stakeholders through a robust governance model. No Gain No Pain enables traders to place short-term "Up" or "Down" bets on assets such as BTC, ETH, and SOL, with liquidity supplied by market makers via frequent batch auctions. Bet outcomes are determined by comparing the asset's closing price against a chosen spread.

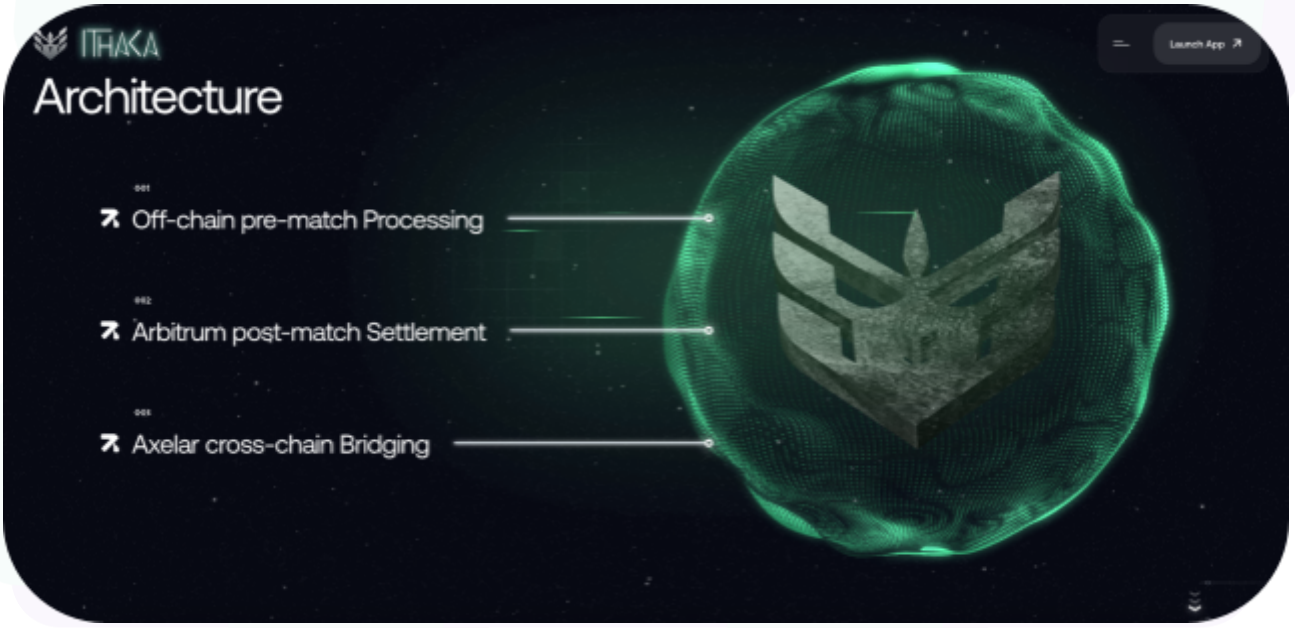
Project Name: No Gain No Pain
Project Type: DeFi, dApp
Compiler Version: 0.8.28
Website: https://www.ithacaprotocol.io/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the No Gain No Pain project, 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	**No Gain No Pain Smart Contracts**
Platform	**EVM / Solidity**
Audit Date	**May, 2025**
Contract 1	OrderV4.sol
Contract 1 MD5 Hash	cc78bc47c476d6d866b1fa71414de732
Contract 2	MakerVaultV3.sol
Contract 2 MD5 Hash	d2f6621cebfa43857fb9df979bebe79b
Contract 3	VaultV3.sol
Contract 3 MD5 Hash	3ed36d25c587d4ce04fb77c5a83f96d2
Audited GitHub Commit Hash	2ece62c448112c4a728ec415cee2508606bbab52
Fix Review GitHub Commit Hash	af4d6ab2037af54285ad10149f0ddc3694a1e507

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:

3 Medium severity vulnerabilities

4 Low severity vulnerabilities

3 QA

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
OrderV4.sol Allows users to: - Place price movement bets (UP/DOWN) - Receive payouts based on bet outcomes Allows coordinators to: - Create and settle bets - Determine outcomes using spot prices Allows admins to: - Manage coordinator address - Set fee percentages Features: - Multi-asset support - Spread-based outcomes - Partial payouts for "almost-win" - Fee management	**Contract achieves this functionality.**
MakerVaultV3.sol Allows makers to: - Register as market makers by staking Ithaca tokens - Deposit collateral for various assets - Withdraw available collateral - Stake additional Ithaca tokens Allows order contract to: - Transfer funds to taker vault - Adjust maker balances - Process fee payments to treasury Allows admins to: - Set minimum stake amounts	**Contract achieves this functionality.**

- Configure custom stake requirements per asset - Change the collateral asset Features: - Multi-asset collateral management - Stake-based maker registration - Locked balance tracking - Security against reentrancy	
VaultV3.sol Allows users to: - Deposit assets into the vault - Withdraw available assets - Check withdrawable balance Allows order contract to: - Adjust taker balances - Transfer assets to maker vault - Process fee payments to treasury Allows admins to: - Change the asset token - Initialize contract parameters Features: - Balance tracking for users - Locked balance management - Total asset availability tracking - Security against reentrancy	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the No Gain No Pain project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and 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 No Gain No Pain 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

Medium

[M-01] Contracts - Use safeTransfer and safeTransferFrom

Description

The contracts use `transferFrom` and `transfer` ERC20 to transfer tokens. Many tokens do not follow the EIP20 specification. Some tokens do not return a bool (e.g. USDT, BNB, OMG) on ERC20 methods. This will make the call break, making it impossible to use these tokens.

Vulnerability Details

Although the contracts handle the bool return value, and reverts an error if false, this is not sufficient when the asset does not follow the EIP20 specification

```
function transferFeeToTreasury(
    address _taker,
    address _treasury,
    uint256 _fee
  ) external onlyOrder {
    takerBalance[_taker] -= _fee;
    totalAssetAvailable -= _fee;

    bool success = asset.transfer(_treasury, _fee);
    if (!success) revert TransferFailed();
  }
```

Impact

This can lead to loss of funds, and users' tokens getting locked in the contract.

Recommendation

Use the safer implementation of token transfers to rule out the above issues. Look

at OpenZeppelin's implementation of SafeERC20.

Status

Resolved

[M-02] OrderV4#createNote - Replay a signature when creating a Note on another chain.

Description

The `createNote` function is designed to create a new bet for the taker if he has signed. During creation, the signature is checked for validity and if the signature is valid, locks the assets of the taker and maker and creates a bet. An attacker can reproduce the user's signature on another chain.

Vulnerability Details

Even though when creating a Note, the contract uses the openzeppelin `_useCheckedNonce` function to protect against signature replay and the address of this contract, an attacker can still replay the signature of the taker and create a Note with his signature.

If the Owner deploys contracts on different chains using `create2`, from the same address and with the same salt, the addresses of `OrderV4` contracts on different chains will be the same. Under these conditions, an attacker would just need to wait for the right nonce.

```
function createNote(
    Note calldata note,
    NoteAdditionalInfo calldata noteAdditionalInfo
  )
    external
    onlyCoordinator
    nonZeroAmount(note.amount)
    nonZeroAddress(note.taker)
    nonZeroAddress(note.maker)
  {
```

```
                uint256    takerBalance    =    traderVault.takerBalance(note.taker)    -
takerLockedBalance[note.taker];
    uint256 makerBalance = makerVault.getMakerCollateral(note.maker, note.asset) -
      makerLockedBalance[note.maker][note.asset];

    // check nonce uniqueness
    _useCheckedNonce(note.taker, note.nonce);
    // ...code
    bytes32 messageHash = getMessageHash(
      note.asset,
      note.direction,
      note.amount,
      note.expiryTime,
      note.taker,
      note.nonce,
      address(this)
    );

    bytes32 ethSignedMessageHash = messageHash.toEthSignedMessageHash();
    (address recovered, , ) = ethSignedMessageHash.tryRecover(note.takerSignature);
    if (recovered != note.taker) revert InvalidSignature();
```

Impact

Attackers can create a new Note using someone else's signatures, leading to token blocking from users and possible loss of tokens.

Recommendation

When recovering a signer, add a `block.chainid` argument to the `getMessageHash` function.

Status

Resolved

[M-03] OrderV4#settleNote - Incorrect fee calculation

Description

When the `settleNote` function is called, the winnings are distributed according to the status of the taker. If the taker wins, the difference between the winnings and his bet is transferred to him, and he will also pay the fee. But the fee is calculated incorrectly, which constantly leads to the fact that takers will overpay.

Vulnerability Details

If a user receives `Win` or `AlmostWin` status, the fee is not calculated correctly. In this case, the fee is calculated from the `winPayout` value, but the user receives only the difference between `winPayout` and `amount`, which is his profit. The fee should be taken from the taker's profit.

```
function settleNote(uint256 _noteId, uint256 _spotPrice) external onlyCoordinator {
    Note memory note = notes[_noteId];
    NoteAdditionalInfo memory noteAdditionalInfo = noteAdditionalInfos[_noteId];

    if (note.taker == address(0)) revert NoteNotAvailable();
    if (isNoteSettled[_noteId]) revert NoteAlreadySettled();

    if (note.expiryTime > block.timestamp) revert NoteNotExpired();
    NoteStatus status = getNoteStatus(note, noteAdditionalInfo, _spotPrice);

    uint256 winAmount = note.winPayout - note.amount;

    isNoteSettled[_noteId] = true;

    takerLockedBalance[note.taker] -= note.amount;
    makerLockedBalance[note.maker][note.asset] -= winAmount;
    uint256 payout;
    uint256 fee;

    if (status == NoteStatus.WIN) {
=>    payout = note.winPayout;
=>    fee = calculateFee(payout, Actor.TAKER);
      uint256 transferredAmount = payout - note.amount;
```

```
      // transfer transferred fund to the trader and reduce the transferred fund from the
maker
      traderVault.adjustTakerBalance(note.taker, transferredAmount, true);
      makerVault.adjustMakerBalance(note.maker, note.asset, transferredAmount, false);
      makerVault.transferToTakerVault(address(traderVault), transferredAmount);

      // fee is deducted from trader
      traderVault.transferFeeToTreasury(note.taker, treasury, fee);
```

Impact

Takers can lose more tokens than they win.

Recommendation

Change the fee calculation for `Win` and `AlmostWin` status so that the fee is taken from the profit.

Status

Resolved

Low

[L-01] MakerVaultV3,VaultV3 - Asset Inconsistency

Description

The system allows the use of potentially different ERC20 tokens in `MakerVaultV3` and `VaultV3` contracts. Both contracts have separately initialized asset variables which could point to different token contracts (e.g., USDC in one and USDT in another).

If different tokens with different decimals are set, this may result in incorrect payouts.

Recommendation

Consider adding an asset equality check when creating a note

Status

Resolved

[L-02] MakerVaultV3,VaultV3 - NonZeroAmount modifier bypass

Description

In these contracts there is a modifier `NonZeroAmount`, which checks that a null value is not passed to the function arguments. The following functions have this modifier and all these functions can be bypassed by simply passing `_amount = type(uint256).max` and still have a zero balance. `VaultV3::deposit()`, `VaultV3::withdraw()`, `MakerVaultV3::depositCollateral()`, `MakerVaultV3::withdrawCollateral`

```
function withdraw(uint256 _amount) external nonZeroAmount(_amount) nonReentrant {
    uint256 availableBalance = getWithdrawableBalance(msg.sender);

    if (_amount == type(uint256).max) {
      _amount = availableBalance;
    }

    if (_amount > availableBalance) revert InsufficientBalance();
```

```
    bool success = asset.transfer(msg.sender, _amount);
    if (!success) revert TransferFailed();

    takerBalance[msg.sender] -= _amount;
    totalAssetAvailable -= _amount;
    emit Withdrawn(msg.sender, _amount);
  }
```

This feature can lead to empty transactions. Users will just spend money on gas. It will also clog up the event log. Because this contract is logical, if the contract is updated in the future, the logic of these functions may change, creating a potential vulnerability. In the case of withdraw functions, this can happen when the user has all tokens locked and passes `type(uint256).max`

Recommendation

Consider adding a 0 value check after overriding the `_amount` argument

Status

Resolved

[L-03] OrderV4 - No check for 0 value

Description

The `createNote` and `settleNote` functions do not check if the difference between `winPayout` and `amount` is greater than 0. If they are equal, the taker will only pay the fee and will not get any win.

```
function settleNote(uint256 _noteId, uint256 _spotPrice) external onlyCoordinator {
    Note memory note = notes[_noteId];
    NoteAdditionalInfo memory noteAdditionalInfo = noteAdditionalInfos[_noteId];

    if (note.taker == address(0)) revert NoteNotAvailable();
    if (isNoteSettled[_noteId]) revert NoteAlreadySettled();

    if (note.expiryTime > block.timestamp) revert NoteNotExpired();
    NoteStatus status = getNoteStatus(note, noteAdditionalInfo, _spotPrice);
```

```
    uint256 winAmount = note.winPayout - note.amount;

    isNoteSettled[_noteId] = true;

    takerLockedBalance[note.taker] -= note.amount;
    makerLockedBalance[note.maker][note.asset] -= winAmount;
    uint256 payout;
    uint256 fee;

    if (status == NoteStatus.WIN) {
      payout = note.winPayout;
      fee = calculateFee(payout, Actor.TAKER);
      uint256 transferredAmount = payout - note.amount;
    //...code
```

This is possible because in the `createNote` function there is a strict condition that `winPayout` should not be less than `amount`, so it can be equal to it.

```
function createNote(
    Note calldata note,
    NoteAdditionalInfo calldata noteAdditionalInfo
  )
    external
    onlyCoordinator
    nonZeroAmount(note.amount)
    nonZeroAddress(note.taker)
    nonZeroAddress(note.maker)
  {
                uint256    takerBalance    =    traderVault.takerBalance(note.taker)    -
takerLockedBalance[note.taker];
    uint256 makerBalance = makerVault.getMakerCollateral(note.maker, note.asset) -
      makerLockedBalance[note.maker][note.asset];

    // check nonce uniqueness
    _useCheckedNonce(note.taker, note.nonce);

    // check if the payout is correct, must be more than wager
    if (note.winPayout < note.amount) revert InvalidPayout();
```

```
    // ...code
```

Recommendation

Add a check that the difference between values is greater than 0.

Status

Resolved

[L-04] MakerVaultV3,VaultV3 -Fee On Transfer Tokens May devastate contracts

Description

When you deposit assets, the MakerVaultV3 and VaultV3 contracts increase the mapping by exactly the transferred amount. This implementation does not take into account tokens that charge a fee for the transfer. Widely used tokens, such as USDC and USDT, have features that allow the owners of these tokens to include a fee at any time, which can lead to malfunctions in accounting for vault contracts.

Vulnerability Details

If tokens are used as assets that charge a fee for transfer, or the USDC/USDT token starts charging a fee, contracts such as `VaultV3` and `MakerVaultV3` will be subject to incorrect asset counting. If you deposit, the contracts will receive less than the actual amount. This will result in users not being able to withdraw their tokens as the vault will empty.

```
function deposit(uint256 _amount) external nonZeroAmount(_amount) nonReentrant {
    if (_amount == type(uint256).max) {
      _amount = asset.balanceOf(msg.sender);
    }
    asset.transferFrom(msg.sender, address(this), _amount);

    takerBalance[msg.sender] += _amount;
    totalAssetAvailable += _amount;

    emit Deposited(msg.sender, _amount);
```

```
}
```

Impact

Contracts could be devastated

Recommendation

Consider increasing the users' balance as the difference between the contract balance before and after the transfer

Status

Acknowledged

QA

[Q-01] VaultV3 - TYPO

Description

There is a typographical mistake in the comment to the mapping description

```
mapping(address => uint256) public takerBalance; // taker addresss => tradable asset => balance
```

Recommendation

Replace `"address"` with `"address"`.

Status

Resolved

[Q-02] VaultV3 - Inconsistency between commentary and implementation

Description

For mapping, provided description in the comment. But the problem is that the comment does not match the implementation.

```
mapping(address => uint256) public takerBalance; // taker addresss => tradable asset => balance
```

The comment suggests that the mapping contains nested mappings, but this is not the case. This mismatch leads to confusion.

Recommendation

Correct the discrepancy

Status

Resolved

[Q-03] Contracts- Checks-Effects-Interactions Pattern Not Followed

Description

Contracts do not strictly enforce the check-effect-interaction pattern in functions during token transfers. Typically, this pattern helps mitigate the risks of reentrancy attacks by ensuring that state updates occur before external calls. Although this issue is labeled as QA, because of the nonReentrant modifier, it is worth looking at as a best practice.

Recommendation

For optimal security and readability, refactor functions involving external token transfers to ensure internal state updates are completed before any external calls.

Status

Resolved

Centralisation

The No Gain No Pain project values security and utility, with decentralization being addressed through verifiable cryptographic guarantees.

The protocol is cryptographically verifiable: takers can confirm that both their entry and exit prices have been correctly referenced and used accurately to calculate rewards. While some owner-executable functions exist, centralization risk is mitigated by the quick resolution of stakes and the relatively small amounts at stake compared to protocols reliant on large TVLs and slower execution models.
Furthermore, the team is actively working with Chainlink to implement a more scalable solution to address this concern.

Conclusion

After Hashlock's analysis, the No Gain No Pain 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