// Private Smart Contract Security Assessment

05.27.2024 - 06.26.2024

berps v0.1.5

Berachain

/ DRAFT /

HALBORN

berps v0.1.5 - Berachain

Prepared by: HALBORN

Last Updated 07/01/2024

Date of Engagement by: May 27th, 2024 - June 26th, 2024

Summary

0% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
11	0	0	3	3	5

TABLE OF CONTENTS

1. Introduction

Berachain engaged Halborn to conduct a security assessment on their smart contracts beginning on May 27th, 2024, and ending on June 26th, 2024. The security assessment was scoped to the smart contracts provided in the following GitHub repository:

- https://github.com/berachain/contracts-monorepo/tree/berps-v0.1.5

2. Assessment Summary

The team at Halborn was provided one and a half weeks for the engagement and assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that the smart contract functions operate as intended.

- Identify potential security issues within the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the `Berachain team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.

- Smart contract manual code review and walkthrough.

- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`)

- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.

- Manual testing by custom scripts.

- Static Analysis of security for scoped contract, and imported functions. (`Slither`)

- Testnet deployment. (`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: contracts-monorepo

(b) Assessed Commit ID: bfabd16

(c) Items in scope:

Out-of-Scope:

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	3	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-04 - INCONSISTENT REVERTS DUE TO INSUFFICIENT PYTH UPDATE FEE HANDLING	MEDIUM	-
HAL-05 - MISSING IMPLEMENTATION OF FEE TRANSFER TO POL SYSTEM	MEDIUM	-
HAL-10 - HARDCODED HONEY PRICE ENABLES DEVASTATING ARBITRAGE ATTACKS AND MEV EXPLOITATION	MEDIUM	-
HAL-01 - ISSUE IN NOTCONTRACT MODIFIER DUE TO FUTURE EVM CHANGES	LOW	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-02 - MISSING __UUPSUPGRADEABLE_INIT() IN THE CONTRACTS	LOW	-
HAL-07 - LACK OF STORAGE GAP IN UPGRADEABLE CONTRACTS	LOW	-
HAL-08 - INCOMPLETE NATSPEC DOCUMENTATION AND TEST COVERAGE	INFORMATIONAL	-
HAL-03 - UNLOCKED PRAGMA	INFORMATIONAL	-
HAL-06 - INCOMPATIBILITY RISK FOR EVM VERSIONS IN DIFFERENT CHAINS	INFORMATIONAL	-
HAL-09 - USE OWNABLE2STEP INSTEAD OF OWNABLE	INFORMATIONAL	-
HAL-11 - INCOMPLETE CANCELREASON CHECK IN EXECUTELIMITOPENORDERCALLBACK	INFORMATIONAL	-

7. FINDINGS & TECH DETAILS

7.1 (HAL-04) INCONSISTENT REVERTS DUE TO INSUFFICIENT PYTH UPDATE FEE HANDLING

// MEDIUM

Description

The `getPrice` function in the `Entrypoint` contract has a potential issue with the handling of the Pyth update fee. The function uses `useValue` to retrieve the fee amount required for updating the Pyth price feeds. However, if the `msg.value` sent with the transaction is greater than the required fee, the excess funds will be left in the contract.
This can lead to inconsistent reverts in certain scenarios:
1. If the contract does not have a mechanism to withdraw or manage the excess funds, they will accumulate over time.
2. If a subsequent call to `getPrice` is made with insufficient `msg.value`, even though the contract holds enough funds from previous calls, the transaction will still revert due to the insufficient fee.
The inconsistent reverts due to insufficient Pyth update fee handling can have the following impacts:
1. User experience: Users may face unexpected transaction reverts when calling the `getPrice` function, even if the contract holds sufficient funds from previous calls. This can lead to confusion and frustration for users interacting with the contract.
2. Stuck funds: If the contract accumulates excess funds from previous calls and does not provide a mechanism to withdraw or manage them, these funds may become stuck in the contract, potentially leading to financial losses.
3. Incorrect contract behavior: The inconsistent reverts can cause the contract to behave unexpectedly, deviating from its intended functionality and potentially leading to incorrect price updates or other undesired consequences.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.3)

Recommendation

To address this issue, it is recommended to implement a mechanism to handle the excess funds and ensure consistent behavior for the Pyth update fee. Here are a few possible approaches:
1. Refund excess funds: After updating the Pyth price feeds, any excess funds can be immediately refunded back to the sender. This ensures that the contract does not hold any excess funds and maintains a consistent state for the Pyth update fee.

```
function getPrice(...) internal returns (int64) {

// ...

uint256 fee = useValue(pyth.getUpdateFee(priceUpdateData));
pyth.updatePriceFeeds{ value: fee }(priceUpdateData);

// Refund excess funds

if (msg.value > fee) {

payable(msg.sender).transfer(msg.value - fee);

}

// ...

}
```

References

berachain/contracts-monorepo/src/berps/core/v0/Entrypoint.sol#L434

7.2 (HAL-05) MISSING IMPLEMENTATION OF FEE TRANSFER TO POL SYSTEM

// MEDIUM

Description

In the provided code for the Vault contract, there is a missing implementation for transferring fees to the PoL system. The code calculates the portion of fees that should be allocated to the PoL system based on the feesToPolP percentage, but the actual transfer of these fees is not implemented.
The specific line of code where this missing functionality is identified is marked with a "TODO" comment:

```
if (feesToPolP > 0) {

    feesToPol = (assets * feesToPolP) / PRECISION / 100;

    // TODO: transfer feesToPol to PoL

}
```

The fees intended for the PoL system are not being transferred to the appropriate destination, resulting in an incorrect distribution of fees. This can impact the overall functioning and incentive structure of the system.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.3)

Recommendation

Design and implement a secure and reliable method to transfer the feesToPol amount to the designated PoL system or contract.

References

berachain/contracts-monorepo/src/berps/core/v0/Vault.sol#L462

7.3 (HAL-10) HARDCODED HONEY PRICE ENABLES DEVASTATING ARBITRAGE ATTACKS AND MEV EXPLOITATION

// MEDIUM

Description

In The `Settlement` contract, The `honeyPrice()` function, responsible for providing the price of HONEY/USDC, is currently hardcoded with a fixed value of `1e10`. This static pricing mechanism opens the door to devastating arbitrage attacks and creates a breeding ground for MEV exploitation.
By hardcoding the price of HONEY/USDC, the contract operates under the flawed assumption that the price remains constant, completely disregarding the dynamic nature of market prices. This glaring discrepancy between the hardcoded price and the actual market price is a ticking time bomb waiting to be exploited by malicious actors.

```
    // TODO: use oracle for price of HONEY/USDC
    function honeyPrice() internal pure returns (int64) {
        return 1e10;
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:C (6.3)

Recommendation

Replace the hardcoded price with an oracle that provides real-time, accurate, and tamper-proof price data for HONEY/USDC.

References

berachain/contracts-monorepo/src/berps/core/v0/Settlement.sol#L623C1-L626C6

7.4 (HAL-01) ISSUE IN NOTCONTRACT MODIFIER DUE TO FUTURE EVM CHANGES

// LOW

Description

The `notContract` modifier in the `Entrypoint.sol` contract is used to ensure that calls are only made from externally owned accounts (EOAs) and not from other contracts. This is achieved by comparing `tx.origin` with `msg.sender`. However, the assumption that this modifier will always effectively distinguish between EOAs and contracts may not hold true in the future due to proposed changes in the Ethereum EVM.

The Ethereum Improvement Proposal (EIP) 3074 introduces two new EVM instructions: AUTH and AUTHCALL. These instructions allow an EOA to delegate control to a smart contract. With AUTH, a context variable `authorized` is set based on an ECDSA signature, and AUTHCALL sends a call as the authorized account. This means that a contract can initiate transactions on behalf of an EOA, effectively blurring the distinction between EOAs and contracts.

If EIP 3074 is implemented, the `notContract` modifier, which relies on the comparison of `tx.origin` and `msg.sender`, may no longer provide the intended protection against calls from contracts.

```
    // Modifiers
    modifier onlyGov() {
        isGov();
        _;
    }

    modifier notContract() {
        isNotContract();
        _;
    }

    modifier notDone() {
        isNotDone();
        _;
    }

    // Saving code size by calling these functions inside modifiers
    function isGov() private view {
        if (msg.sender != orders.gov())
BerpsErrors.Unauthorized.selector.revertWith();
    }

    function isNotContract() private view {
        if (tx.origin != msg.sender) BerpsErrors.Unauthorized.selector.revertWith();
    }
    function isNotDone() private view {
        if (isDone) BerpsErrors.Done.selector.revertWith();
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

To mitigate the potential risks associated with the `notContract` modifier and ensure the contract remains secure in the face of future EVM changes, consider the following recommendations:
1. Use OpenZeppelin's `isContract` function: Instead of relying on the comparison of `tx.origin` and `msg.sender`, use OpenZeppelin's `isContract` function to determine whether the caller is a contract or an EOA. This function provides a more reliable way to distinguish between the two.

```
function isNotContract() private view {

    if (Address.isContract(msg.sender))
BerpsErrors.Unauthorized.selector.revertWith();

}
```

References

berachain/contracts-monorepo/src/berps/core/v0/Entrypoint.sol#L103

7.5 (HAL-02) MISSING __UUPSUPGRADEABLE_INIT() IN THE CONTRACTS

// LOW

Description

Although the `__UUPSUpgradeable_init()` is empty in the current `"@openzeppelin/contracts-upgradeable" v5.0.2`, there may be additional logic in the future of the contract modified by OpenZeppelin themselves (see reference from OpenZeppelin team: https://forum.openzeppelin.com/t/uups-proxies-tutorial-solidity-javascript/7786/30). So the best practice now is to include `__UUPSUpgradeable_init()` in the initialize function.

```
    /// @inheritdoc IEntrypoint
    function initialize(
        address _pyth,
        address _orders,
        address _feesMarkets,
        address _feesAccrued,
        PythConfig calldata _pythCfg,
        uint256 _maxPosHoney
    )
        external
        initializer
    {
        if (
            _pyth == address(0) || _orders == address(0) || _feesMarkets ==
address(0) || _feesAccrued == address(0)
                || _maxPosHoney == 0
        ) BerpsErrors.WrongParams.selector.revertWith();

        orders = IOrders(_orders);
        feesMarkets = IFeesMarkets(_feesMarkets);
        feesAccrued = IFeesAccrued(_feesAccrued);

        pyth = IPyth(_pyth);
        _pythConfig = _pythCfg;
        maxPosHoney = _maxPosHoney;
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:C (3.1)

Recommendation

Consider adding `__UUPSUpgradeable_init()` into initialize function.

References

berachain/contracts-monorepo/src/berps/core/v0/Entrypoint.sol#L57-L58
berachain/contracts-monorepo/src/berps/core/v0/Markets.sol#L13

7.6 (HAL-07) LACK OF STORAGE GAP IN UPGRADEABLE CONTRACTS

// LOW

Description

The core contracs are designed to be used with the UUPS proxy pattern. However, it lacks storage gaps. Storage gaps are essential for ensuring that new state variables can be added in future upgrades without affecting the storage layout of inheriting child contracts. Without it, any addition of new state variables in future contract versions can lead to storage collisions.

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:C (2.5)

Recommendation

Consider adding a storage gap as the last storage variable to BorpaGateway contract.

7.7 (HAL-08) INCOMPLETE NATSPEC DOCUMENTATION AND TEST COVERAGE

// INFORMATIONAL

Description

At least all `public` and `external` functions that are not `view` or `pure` should have `NatSpec` comments. During the security review, it was observed that the multiple publicly-accessible functions did not have any `NatSpec` comments.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:F/S:U (0.8)

Recommendation

Consider adding `NatSpec` documentation to all functions, specially those that are publicly-accessible.

7.8 (HAL-03) UNLOCKED PRAGMA

// INFORMATIONAL

Description

The files in scope currently use floating pragma version `^0.8.21`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.21`, and less than `0.9.0`.
It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

Consider locking the pragma version to the same version used during development and testing.

7.9 (HAL-06) INCOMPATIBILITY RISK FOR EVM VERSIONS IN DIFFERENT CHAINS

// INFORMATIONAL

Description

From Solidity versions `0.8.20` through `0.8.24`, the default target EVM version is set to Shanghai, which results in the generation of bytecode that includes `PUSH0` opcodes. Starting with version `0.8.25`, the default EVM version shifts to Cancun, introducing new opcodes for transient storage, `TSTORE` and `TLOAD`.
In this aspect, it is crucial to select the appropriate EVM version when it's intended to deploy the contracts on networks other than the Ethereum mainnet, which may not support these opcodes. Failure to do so could lead to unsuccessful contract deployments or transaction execution issues.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Make sure to specify the target EVM version when using Solidity versions from `0.8.20` and above if deploying to chains that may not support newly introduced opcodes. Additionally, it is crucial to stay informed about the opcode support of different chains to ensure smooth deployment and compatibility.

7.10 (HAL-09) USE OWNABLE2STEP INSTEAD OF OWNABLE

// INFORMATIONAL

Description

The current ownership transfer process for all the contracts inheriting from Ownable or OwnableUpgradeable involves the current owner calling the [transferOwnership()](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.8/contracts/access/Ownable.sol#L69-L72) function:

```
function transferOwnership(address newOwner) public virtual onlyOwner {
require(newOwner != address(0), "Ownable: new owner is the zero address");
_setOwner(newOwner); }
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the onlyOwner modifier.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Consider using **Ownable2Step** instead of **Ownable** contract.

References

berachain/contracts-monorepo/src/berps/core/v0/Vault.sol#L9

7.11 (HAL-11) INCOMPLETE CANCELREASON CHECK IN EXECUTELIMITOPENORDERCALLBACK

// INFORMATIONAL

Description

The `executeLimitOpenOrderCallback` function in the `Settlement` contract does not consider the `CancelReason.SL_REACHED` case when determining the reason for canceling a limit open order. This can lead to incorrect behavior and potential loss of funds for traders.

Currently, the function checks for various cancel reasons, such as `CancelReason.PAUSED`, `CancelReason.IN_TIMEOUT`, and `CancelReason.NOT_HIT`. However, it fails to consider the scenario where the stop-loss (SL) price is reached before the limit order is executed.

The missing check for `CancelReason.SL_REACHED` can result in the following issue:

1. Incorrect order execution: If the market price reaches or crosses the stop-loss price before the limit price is reached, the limit order should be canceled. Without the `SL_REACHED` check, the order may still be executed, leading to unintended trades and potential losses for traders.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Ensure that the order is canceled if any of the cancel reasons, including `SL_REACHED`, are met.

References

berachain/contracts-monorepo/src/berps/interfaces/v0/ISettlement.sol#L21-L22
