SMART CONTRACT AUDIT REPORT

for

BasePump

Prepared By: Xiaomi Huang

PeckShield

December 17, 2024

Document Properties

Client	BasePump
Title	Smart Contract Audit Report
Target	BasePump
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 17, 2024	Xuxian Jiang	Final Release
1.0-rc	December 6, 2024	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `BasePump` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About BasePump

`BasePump` is a decentralized platform designed to enable fair and transparent token creation and trading using a bonding curve mechanism. The platform will be deployed on the `Base` blockchain, a Layer 2 solution built on the `OP Stack`, providing scalability and low transaction costs. The goal is to offer a decentralized solution for token issuance, allowing projects and individuals to launch tokens without the need for presales, team allocations, or mint authority, ensuring a trustless environment for all participants. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of BasePump

Item	Description
Name	BasePump
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 17, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/basepumpOrg/basepump-contracts.git (325e8d0)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/basepumpOrg/basepump-contracts.git (5c29b47d)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `BasePump` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	1	■
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Ether Transfers in BondingCurveLogic	Coding Practices	Resolved
PVE-002	Medium	Possibly Earlier Pair Creation Before Completion	Business Logic	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Ether Transfers in BondingCurveLogic

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BondingCurveLogic
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

The BasePump protocol has a core BondingCurveLogic contract that allows for real-time discovery of token price. In the process of examining the token sell/buy logic, we notice a possible issue that may arise from the native coin transfer.

To elaborate, we show below the code snippet of the related sell() routine, which allows to sell tokens for the native coin (e.g., Ether). We notice that this routine directly calls the native transfer() routine (line 131) to transfer Ether. However, the transfer() is not recommended to use any more since the EIP-1884 may increase the gas cost and the 2300 gas limit may be exceeded. There is a helpful blog stop-using-soliditys-transfer-now that explains why the transfer() is not recommended any more.

```
109   function sell(
110     address recipient,
111     uint256 tokenAmount,
112     uint256 minimumEthAmount
113   ) public virtual whenNotPaused returns (uint256 ethAmount) {
114     if (tokenAmount == 0) {
115       revert BondingCurve__ZeroAmount();
116     }
117     uint256 rawAmount =
118       _calculateDelta(virtualTokenReserve, virtualEthReserve, tokenAmount);
119     if (rawAmount == 0) {
120       revert BondingCurve__ZeroAmount();
121     }
```

```
    uint256 feeAmount = rawAmount * FACTORY.fee() / 1_00_00;
    ethAmount = rawAmount - feeAmount;
    if (ethAmount < minimumEthAmount) {
      revert BondingCurve__Slippage(minimumEthAmount, ethAmount);
    }
    virtualEthReserve -= rawAmount;
    virtualTokenReserve += tokenAmount;
    TOKEN.transferFrom(msg.sender, address(this), tokenAmount);
    payable(FACTORY.feeReceiver()).transfer(feeAmount);
    payable(recipient).transfer(ethAmount);
    emit Trade(
      msg.sender,
      recipient,
      ethAmount,
      tokenAmount,
      feeAmount,
      virtualEthReserve,
      virtualTokenReserve,
      false
    );
  }
```

Listing 3.1: `BondingCurveLogic::sell()`

Recommendation Revisit the above-mentioned routine to transfer `ETH` using `call()`.

Status This issue has been resolved by following the suggestion.

3.2 Possibly Earlier Pair Creation Before Completion

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `BondingCurveLogic`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned earlier, the `BasePump` protocol has a core `BondingCurveLogic` contract that allows for real-time discovery of token price. Moreover, once the bonding process ends, liquidity will be automatically added to the respective pair in `UniswapV2`. While reviewing the final step of liquidity addition, we notice a possibility that the intended `token-weth` pair may be created with imbalanced pool before completion.

To elaborate, we show below the implementation of the related `_deployPool()` function. As the name indicates, this routine is invoked when the bonding curve is completed. However, it currently assumes the pair may not be created by others. With that, if a malicious user intentionally creates the pool earlier with an imbalanced state, the added liquidity may result in a loss as it is being added into an imbalanced pool. To fix, there is a need to better protect the liquidity to avoid being manipulated.

```
function _deployPool(
  uint256 poolTokenInput
) internal virtual returns (address pool) {
  IUniswapV2Router02 router = IUniswapV2Router02(FACTORY.UNISWAP_ROUTER());
  TOKEN.approve(address(router), poolTokenInput);
  router.addLiquidityETH{
    value: virtualEthReserve - INITIAL_VIRTUAL_ETH_RESERVE
  }(
    address(TOKEN),
    poolTokenInput,
    0,
    0,
    FACTORY.lpTokenReceiver(),
    block.timestamp
  );
  return
    IUniswapV2Factory(router.factory()).getPair(address(TOKEN), router.WETH());
}
```

Listing 3.2: `BondingCurveLogic::_deployPool()`

Recommendation Revise the above logic to properly complete the bonding curve and protect the liquidity.

Status The issue has been resolved as the team confirms it is intended balanced design.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `BasePump` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, pause/unpause gauges, and execute privileged operations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized.

In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
function deployBondingCurveLogic(
  bytes32 _salt
) external virtual onlyOwner {
  // Deploy BondingCurveLogic contract
  _deployNewBondingCurveLogic(_salt);
}
...
function setTargetUsdCap(
  uint256 newTargetUsdCap
) public virtual onlyOwner {
  if (newTargetUsdCap == 0) {
    revert BasePumpFactory__ZeroAmount();
  }
  targetUsdCap = newTargetUsdCap;
}
...
function setInitialVirtualTokenReserve(
  uint256 newInitialVirtualTokenReserve
) public virtual onlyOwner {
  if (newInitialVirtualTokenReserve == 0) {
    revert BasePumpFactory__ZeroAmount();
  }
  initialVirtualTokenReserve = newInitialVirtualTokenReserve;
}
...
function setTokenTotalSupply(
  uint256 newTokenTotalSupply
) public virtual onlyOwner {
  if (newTokenTotalSupply == 0) {
    revert BasePumpFactory__ZeroAmount();
  }
  tokenTotalSupply = newTokenTotalSupply;
}
...
function setFee(
  uint16 newFee
) public virtual onlyOwner {
  if (newFee >= 1_00_00) {
    revert BasePumpFactory__FeeTooHigh();
  }
  fee = newFee;
}
...
function setFeeReceiver(
  address newFeeReceiver
) public virtual onlyOwner {
  if (newFeeReceiver == address(0)) {
    revert BasePumpFactory__ZeroAddress();
  }
```

```
    feeReceiver = newFeeReceiver;
  }
  ...
  function setLpTokenReceiver(
    address newLpTokenReceiver
  ) public virtual onlyOwner {
    lpTokenReceiver = newLpTokenReceiver;
  }
```

Listing 3.3: Example Privileged Functions in BasePumpFactory

Note that if the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `BasePump` protocol, which is a decentralized platform designed to enable fair and transparent token creation and trading using a bonding curve mechanism. The platform will be deployed on the `Base` blockchain, a Layer 2 solution built on the `OP Stack`, providing scalability and low transaction costs. The goal is to offer a decentralized solution for token issuance, allowing projects and individuals to launch tokens without the need for presales, team allocations, or mint authority, ensuring a trustless environment for all participants. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.