pSeudo Ethereum

Smart Contract Security Audit

Prepared by ShellBoxes

May 15th, 2023 - May 18th, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	pSeudo Ethereum
Version	1.0
Classification	Public

Scope

Files	MD5 Hash
FactoryInspector.sol	edf59130bc11e81904582ef83dd2dc1d
pSeudoEthereum.sol	cb871ab2f5f4e83581e42e8e60cd4dfd

Re-Audit

Files	MD5 Hash
FactoryInspector.sol	5ad638756518cb4503f95fe4f78a69c8
PseudoEth.sol	25bc39df1535cfd4112fbccc21b4e7cf

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

pSeudo Ethereum engaged ShellBoxes to conduct a security assessment on the pSeudo Ethereum beginning on May 15th, 2023 and ending May 18th, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About pSeudo Ethereum

pSeudo Ethereum (pEth) is an ERC-20 token that primarily functions as a 'wrapper' for the Ethereum native currency on the Ethereum network.

Issuer	pSeudo Ethereum
Type	Solidity Smart Contract
Documentation	PSeudo Ethereum Litepaper V1
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This frame-

work is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Disclaimer

In the process of the conducted audit, it is important to note that certain interfaces were utilized within the contract code but were not included within the scope of the audit. These interfaces include IUniswapV2Pair.sol, IUniswapV2Factory.sol, and IUniswapV3Factory.sol.

As such, while these interfaces play a role in the functioning of the contract, they were not individually assessed for security and correctness. The audit makes the assumption that these interfaces are correctly implemented and the called contracts are the valid Uniswap contracts, and the findings of this audit are based on this premise.

2.2 Summary

The following is a synopsis of our conclusions from our analysis of the pSeudo Ethereum implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.3 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 3 high-severity, 1 medium-severity, 8 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Potential Bypass of maxLoadedPseudo Limitation	HIGH	Acknowledged
SHB.2. Bypassing Preload Expiration by Minimal Preloading	HIGH	Fixed
SHB.3. Potential Fee Bypass for Small Amounts Due to Rounding Error	HIGH	Acknowledged

SHB.4. Precision Loss Due to Division Before Multiplication	MEDIUM	Fixed
SHB.5. Race Condition in The Tax And Interest Rates	LOW	Acknowledged
SHB.6. Missing Check for Duplicate Before Pushing A Factory	LOW	Fixed
SHB.7. Potential Denial of Service (DoS) Due to The Use Of The .transfer function For Sending Ether	LOW	Fixed
SHB.8. Potential Race Condition in ERC20 approve Function	LOW	Acknowledged
SHB.9. Renounce Ownership Risk	LOW	Fixed
SHB.10. The owner Can Control The Fee Structure And The Contract's Parameters	LOW	Acknowledged
SHB.11. Missing Transfer Event	LOW	Acknowledged
SHB.12. Floating pragma	LOW	Fixed

3 Finding Details

SHB.1 Potential Bypass of maxLoadedPseudo Limitation

- Severity : HIGH
- Status : Acknowledged
- Likelihood : 3
- Impact : 2

Description:

The maxLoadedPseudo limitation is a security measure in the pSeudoEthereum contract that sets a maximum limit to the number of pseudo tokens that can be preloaded by a user. The idea is to prevent potential abuses where a user might preload a significant amount of tokens, potentially disrupting the system's balance.

However, it seems like there might be a loophole allowing a user to bypass this limit. This is done by preloading an amount up to the maxLoadedPseudo limit, injecting it into a pair, and then repeating the process. This way, they can preload more than the intended maximum limit, bit by bit. A malicious user could potentially use this bypass to preload an excessive amount of pseudo tokens, causing potential imbalance or other unforeseen issues in the system.

Files Affected:

SHB.1.1: pSeudoEthereum.sol

```
287 /// @notice Pre-load pSuedo pEth into a wallet to enable liquidity
        ↪ provisioning
288 function preLoadPseudo(uint256 amount) public {
289     address pseudoRecipient = _msgSender();
290     uint256 currentBalance = injectorBalance(pseudoRecipient);
291     require(currentBalance + amount <= maxLoadedPseudo, 'You have
            ↪ reached maximum pre-loaded Pseudo');
292     //Reset to 0 in case of expired pseudo.
```

```
293     if (currentBalance == 0 && injectors[pseudoRecipient].loadedAmount
            ↪ != 0)
294         injectors[pseudoRecipient].loadedAmount = 0;
295     injectors[pseudoRecipient].loadedAmount += amount;
296     injectors[pseudoRecipient].loadedBlock = block.number;
297     emit PseudoPreLoaded(pseudoRecipient, amount);
298 }
```

Recommendation:

To address this, you might want to include the number of pseudo tokens injected across all pairs in the maxLoadedPseudo check. This could be tracked in a mapping that stores the total injected amount for each user, which is updated every time they inject tokens into a pair.

Updates

The Pseudo Ethereum team acknowledged the issue, stating that the behavior is accepted by the business logic, as the maxLoadedPseudo is implemented to limit the preload amount per pair.

SHB.2 Bypassing Preload Expiration by Minimal Preloading

- Severity : HIGH
- Status : Fixed
- Likelihood : 3
- Impact : 2

Description:

This finding pertains to a potential loophole in the preLoadPseudo function of the smart contract, which may allow a user to bypass the expiration of their preloaded pseudo tokens. The smart contract seems to have a built-in mechanism to expire preloaded pseudo tokens if a

certain number of blocks (loadedBlockLimit) have passed since the preload. This mechanism is implemented in the injectorBalance function.

However, the preLoadPseudo function updates the loadedBlock every time a user preloads pseudo tokens, regardless of the amount. This means that a user could potentially preload an insignificant amount of pseudo tokens (even a single wei of pEth) to refresh the loadedBlock, thereby extending the lifespan of their preloaded balance. This could potentially result in indefinite "keeping alive" of preloaded balances, bypassing the intended expiration mechanism.

Files Affected:

SHB.2.1: pSeudoEthereum.sol

```
287 /// @notice Pre-load pSuedo pEth into a wallet to enable liquidity
        ↪ provisioning
288 function preLoadPseudo(uint256 amount) public {
289     address pseudoRecipient = _msgSender();
290     uint256 currentBalance = injectorBalance(pseudoRecipient);
291     require(currentBalance + amount <= maxLoadedPseudo, 'You have
            ↪ reached maximum pre-loaded Pseudo');
292     //Reset to 0 in case of expired pseudo.
293     if (currentBalance == 0 && injectors[pseudoRecipient].loadedAmount
            ↪ != 0)
294         injectors[pseudoRecipient].loadedAmount = 0;
295     injectors[pseudoRecipient].loadedAmount += amount;
296     injectors[pseudoRecipient].loadedBlock = block.number;
297     emit PseudoPreLoaded(pseudoRecipient, amount);
298 }
```

SHB.2.2: pSeudoEthereum.sol

```
332 function injectorBalance(address injectorAddress) public view returns (
        ↪ uint256) {
333     if (injectors[injectorAddress].loadedAmount != 0)
334         return block.number - injectors[injectorAddress].loadedBlock <=
              ↪ loadedBlockLimit ? injectors[injectorAddress].loadedAmount
```

```
            ↪ : 0;
335     return 0;
336 }
```

Recommendation:

To address this issue, you might want to consider only updating the loadedBlock when the preloaded balance was previously zero. This way, refreshing the loadedBlock would only be possible when the user preloads new tokens after their previous balance has expired, thereby preserving the intended expiration mechanism.

Updates

The Pseudo Ethereum team has resolved the issue by only updating the loadedBlock attribute if the loadedAmount is equal to zero.

SHB.3 Potential Fee Bypass for Small Amounts Due to Rounding Error

- Severity : HIGH
- Likelihood : 2
- Status : Acknowledged
- Impact : 3

Description:

In Solidity, integer division truncates downwards. As a result, if the fee rate is relatively small compared to the transaction amount, the calculated fee may round down to zero. This is a potential vulnerability in the feeCalc function where the calculation of _feeAmount takes place.

Exploit Scenario:

Let's say the amount is 199, taxRate is 5, and feeDenominator is 1000. The fee calculation would be as follows:

SHB.3.1: pSeudoEthereum.sol

```
_feeAmount = (amount * taxRate) / feeDenominator
_feeAmount = (199 * 5) / 1000
_feeAmount = 950 / 1000
_feeAmount = 0
```

Files Affected:

SHB.3.2: pSeudoEthereum.sol

```
277 } else {
278     uint256 _feeAmount = (amount*taxRate)/feeDenominator;
279     _balances[_feeRecipient] += _feeAmount;
280     _balances[from] -= _feeAmount;
281     amount -= _feeAmount;
282     emit Transfer(from, _feeRecipient, _feeAmount);
283 }
284 return amount;
```

Recommendation:

Consider requiring amount * taxRate to be higher than the feeDenominator to avoid rounding errors that leads to bypassing the fees.

Updates

The Pseudo Ethereum team acknowledged the issue, stating that they accept receiving no fees for amounts that are less than 199 Wei.

SHB.4 Precision Loss Due to Division Before Multiplication

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The function calculatePairInterestRate in the provided smart contract code has a potential precision loss issue due to the division operation performed before multiplication. This sequence can lead to truncation errors and the loss of fractional values because Solidity performs integer division, where decimal points are not preserved.

The division operation currentBalance/100 is performed before the multiplication operation with interestRate. This can cause precision loss because the division may truncate some decimal values, which are then not included in the subsequent multiplication.

Files Affected:

SHB.4.1: pSeudoEthereum.sol

```
318 function calculatePairInterestRate (address pairAddress) private {
319     uint256 currentBalance = injectedPairs[pairAddress].injectedAmount
        ↪ >= 100 ? injectedPairs[pairAddress].injectedAmount : 100;
320     uint256 numerator = (currentBalance/100)*interestRate;
321     injectedPairs[pairAddress].secondsInterestRate = numerator >=
        ↪ annualSeconds ? numerator/annualSeconds : 1;
322 }
```

Recommendation:

A better practice in such scenarios would be to switch the sequence of operations: perform multiplication before division. This change will minimize the truncation errors.

Updates

The Pseudo Ethereum team resolved the issue by performing multiplication operations before division to increase precision.

SHB.4.2: pSeudoEthereum.sol

```
316 function calculatePairInterestRate (address pairAddress) private {
317     uint256 currentBalance = injectedPairs[pairAddress].injectedAmount
        ↪ >= 100 ? injectedPairs[pairAddress].injectedAmount : 100;
318     uint256 numerator = (currentBalance*interestRate)/100;
319     injectedPairs[pairAddress].secondsInterestRate = numerator >=
        ↪ annualSeconds ? numerator/annualSeconds : 1;
320 }
```

SHB.5 Race Condition in The Tax And Interest Rates

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 2

Description:

A race condition occurs when the behavior of a system depends on the sequence or timing of other uncontrollable events. It becomes a major issue when it leads to unauthorized behavior, especially in a decentralized system like a blockchain where anyone can call public functions in any order.

In this project, you have a function feeCalc that is used to calculate the interest for v3 pairs and taxes for v2 pairs. The tax and interest calculations depend on taxRate, interestRate, and feeDenominator variables that can be changed by the contract owner.

The race condition can occur if the contract owner changes these variables while feeCalc is already executed and the miner swaps the order of the transactions. As a result, users may end up paying a different tax or interest rate than what they expected at the time of initiating the transaction.

Files Affected:

SHB.5.1: pSeudoEthereum.sol

```
260 function feeCalc(uint256 amount, address pairAddress, address from)
        ↪ private returns (uint256) {
261     if (injectedPairs[pairAddress].isInterestBearing) {
262         uint256 timeSince;
263         unchecked {
264             timeSince = block.timestamp - injectedPairs[pairAddress].
                ↪ interestCalcTimestamp;
265         }
266         if (timeSince > interestFrequency) {
267             uint256 interest = timeSince * injectedPairs[pairAddress].
                ↪ secondsInterestRate;
268             interest = _balances[pairAddress] >= interest ? interest :
                ↪ _balances[pairAddress];
269             // Unchecked due to ternary above
270             unchecked {
271                 _balances[pairAddress] -= interest;
272                 _balances[_feeRecipient] += interest;
273             }
274             injectedPairs[pairAddress].injectedAmount += interest;
275             injectedPairs[pairAddress].interestCalcTimestamp = block.
                ↪ timestamp;
276         }
277     } else {
278         uint256 _feeAmount = (amount*taxRate)/feeDenominator;
279         _balances[_feeRecipient] += _feeAmount;
280         _balances[from] -= _feeAmount;
281         amount -= _feeAmount;
282         emit Transfer(from, _feeRecipient, _feeAmount);
283     }
284     return amount;
285 }
```

SHB.5.2: pSeudoEthereum.sol

```
416 function changeTaxRate(uint8 newTaxRate) public onlyOwner {
417     require(newTaxRate <= 100, 'Tax can not be higher than 1%');
418     uint8 oldTaxRate = taxRate;
419     taxRate = newTaxRate;
420     emit TaxRateChange(oldTaxRate, newTaxRate);
421 }
```

SHB.5.3: pSeudoEthereum.sol

```
430 function changeInterestRate(uint8 newInterestRate) public onlyOwner {
431     require(newInterestRate <= 100, 'Interest rate can not be higher
        ↪ than 100% PA');
432     uint8 oldInterestRate = interestRate;
433     interestRate = newInterestRate;
434     emit InterestRateChange(oldInterestRate, newInterestRate);
435 }
```

Recommendation:

Consider adding the rates as arguments to the external function and then verifying that they are the same as the one stored in the contract. Also, it is recommended to notify the community by any change in the fee structure.

Updates

The Pseudo Ethereum team acknowledged the issue, stating that they accept this risk.

SHB.6 Missing Check for Duplicate Before Pushing A Factory

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The function addFactory is used to add an address to either the _v3Factories or _v2Factories arrays depending on the isV3 boolean. However, the function currently does not check if the provided address is already present in the respective array before pushing it. This could lead to unintentional duplication of addresses in the array.

In the case of this contract, the potential issue with this missing check is that duplicate factory addresses will occupy unnecessary space on the blockchain, leading to an increased gas cost when iterating over these arrays. It may also cause logical errors elsewhere in the contract if the presence of an address in these arrays is used as a check for certain conditions.

Files Affected:

SHB.6.1: FactoryInspector.sol

```
137 function addFactory(address factory, bool isV3) public onlyOwner {
138     if (isV3) {
139         _v3Factories.push(factory);
140     } else {
141         _v2Factories.push(factory);
142     }
143 }
```

Recommendation:

A solution would be to introduce a check to ensure that the factory address isn't already present in the array before pushing it. However, to prevent looping over these arrays (which could be expensive in terms of gas cost if the arrays are large), you could use mappings to store the factory addresses, as mappings in Solidity can check the existence of a key in constant time. The mappings could be used in conjunction with the arrays if you still need to maintain an iterable list of all factories.

Updates

The Pseudo Ethereum team resolved the issue, by implementing a mapping called factoryAddresses to store the factory addresses and avoid duplication in the _v2Factories and _v3Factories arrays.

SHB.6.2: FactoryInspector.sol

```
139 function addFactory(address factory, bool isV3) public onlyOwner {
140     require(!factoryAddresses[factory], 'Factory already added');
141     if (isV3) {
142         _v3Factories.push(factory);
143         factoryAddresses[factory] = true;
144     } else {
145         _v2Factories.push(factory);
146         factoryAddresses[factory] = true;
147     }
148 }
```

SHB.7 Potential Denial of Service (DoS) Due to The Use Of The .transfer function For Sending Ether

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The Ethereum network uses gas as a measure of computational effort. Every operation that takes place in the network requires a certain amount of gas to execute. When transferring Ether using the .transfer() function in Solidity, a hardcoded gas limit of 2300 is automatically set.

While this amount of gas is generally enough for a simple Ether transfer, it might not be sufficient under certain circumstances, particularly if the recipient is a contract that has a

fallback function. If the fallback function has operations that require more than 2300 gas, it will fail, causing the entire transaction to revert.

Moreover, Ethereum's gas costs for different opcodes may change due to hard forks or network upgrades. If such a change increases the gas cost for an Ether transfer above 2300, all contracts using the .transfer() function would break, potentially causing a network-wide DoS attack.

This hardcoded gas limit can also cause problems when deploying to some Layer 2 protocols. Layer 2 solutions often have different gas economics than the main Ethereum chain, so a hardcoded gas limit of 2300 might not be appropriate.

Files Affected:

SHB.7.1: pSeudoEthereum.sol

```
122 function withdraw(uint wad) public payable {
123     require(_balances[msg.sender] >= wad);
124     _balances[msg.sender] -= wad;
125     payable(msg.sender).transfer(wad);
126     emit Withdrawal(msg.sender, wad);
127 }
```

Recommendation:

A more flexible and safer approach is to use the .callvalue: amount("") method for sending Ether, which forwards all available gas (or a specified amount), and checks the success of the call afterwards. However the use of call should be implemented carefully as it can expose the contract to reentrancy attacks. This can be resolved by changing the state before sending the ether or the use of the nonReentrant modifier from the ReentrancyGuard contract.

Updates

The Pseudo Ethereum team resolved the issue by implementing the use of .callvalue: amount("") method instead of the .transfer.

SHB.7.2: pSeudoEthereum.sol

```
119 function withdraw(uint wad) public payable {
120     _balances[msg.sender] -= wad;
121     (bool sent, ) = msg.sender.call{value: wad}("");
122     require(sent, "Failed to send Eth");
123     emit Withdrawal(msg.sender, wad);
124 }
```

SHB.8 Potential Race Condition in ERC20 approve Function

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 2

Description:

The ERC20 standard includes an approve function, which allows a token owner to approve another address to spend up to a certain number of tokens on their behalf. However, there is a known potential race condition in this function, which can occur in the following sequence of events:

The issue arises from the fact that the approve function can only set the allowance to a certain value, it doesn't have the functionality to increase or decrease it. If the token owner wants to change the allowance, they must first set it to zero and then set it to the new value. This creates a window of opportunity where the spender could execute a transaction, leading to a potential race condition.

Exploit Scenario:

- Alice approves Bob to spend 10 of her tokens.
- Alice decides to decrease Bob's allowance to 5 tokens
- Bob decides to spend 10 of Alice's tokens before Alice's last transaction gets mined.
- Bob can then spend an additional 5 allowing him to spend 15 tokens in total.

Files Affected:

SHB.8.1: pSeudoEthereum.sol

```
159 function approve(address spender, uint256 amount) public virtual
        ↪ override returns (bool) {
160     address owner = _msgSender();
161     _approve(owner, spender, amount);
162     return true;
163 }
```

Recommendation:

A common mitigation to this problem is to make use of the already defined decreaseAllowance and increaseAllowance functions in the token contract. This allows the token owner to change the allowance without having to set it to zero first, effectively eliminating the race condition.

Updates

The Pseudo Ethereum team acknowledged the issue, stating that they accept the risk due to its existence in all the ERC20 tokens.

SHB.9 Renounce Ownership Risk

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The contract inherits from the Ownable pattern, which includes a renounceOwnership function. This function, if called, can result in the contract having no owner, causing a Denial of Service (DoS) for the functions with the onlyOwner modifier.

In the current implementation, the contract is Ownable, and the renounceOwnership function allows the contract owner to permanently relinquish ownership. If the ownership is renounced, the contract will not have an owner, and any function with the onlyOwner modifier will become unreachable. This scenario could lead to a Denial of Service (DoS) on these functions, as no one would be able to execute them, effectively rendering them useless.

Files Affected:

SHB.9.1: pSeudoEthereum.sol

```
50 contract PseudoEthereum is IERC20, Ownable {
```

SHB.9.2: FactoryInspector.sol

```
45 contract FactoryInspector is Ownable {
```

Recommendation:

To mitigate this risk, consider either removing the renounceOwnership function or replacing it with a safer alternative, such as allowing ownership transfer to a predefined address, like a multisig wallet or a timelock contract. This approach will maintain control over the contract and prevent a potential DoS on the functions with the onlyOwner modifier.

Updates

The Pseudo Ethereum team resolved the issue, by removing the renounceOwnership function.

SHB.10 The owner Can Control The Fee Structure And The Contract's Parameters

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 1

Description:

The contract provides the owner with the ability to manipulate various aspects of the contract, such as the fee structure, the preloaded threshold, the recipient of the fee, the block limit for preloads, the maximum amount of pseudo tokens that can be preloaded, the tax rate, the frequency and rate of interest, and the factory inspector.

While the implementation of onlyOwner restrictions and input requirements can prevent unauthorized manipulation and extreme parameter changes, the overall control that the owner has on these key contract parameters introduce risks associated with centralized power.

One risk in this situation is that if the owner's address is compromised, the attacker could change these parameters within the limitations set by the contract. This could be detrimental for the users of the contract. Additionally, the owner could potentially act maliciously or negligently themselves, which could also negatively impact the users.

Files Affected:

All functions with onlyOwner modifier.

Recommendation:

It is recommended to consider implementing a Decentralized Autonomous Organization (DAO) to manage these parameters. A DAO operates under the rules encoded as a computer program and is typically controlled by the token holders who vote on proposals. This would effectively decentralize the decision-making process and reduce the risks associated with having a single account with overarching control.

Updates

The Pseudo Ethereum team acknowledged the issue, stating that they are planning to implement a DAO which will control the contract's parameters.

SHB.11 Missing Transfer Event

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 1

Description:

In the feeCalc function, there's a condition where tokens are transferred from the sender to the fee recipient, but no Transfer event is emitted. This could lead to discrepancies when someone tries to track the token's movements. The ERC-20 standard specifies that a Transfer event should be emitted any time tokens move, so this could be seen as a violation of the standard.

Files Affected:

SHB.11.1: pSeudoEthereum.sol

```
260 function feeCalc(uint256 amount, address pairAddress, address from)
        ↪ private returns (uint256) {
261     if (injectedPairs[pairAddress].isInterestBearing) {
262         uint256 timeSince;
263         unchecked {
264             timeSince = block.timestamp - injectedPairs[pairAddress].
                ↪ interestCalcTimestamp;
265         }
266         if (timeSince > interestFrequency) {
267             uint256 interest = timeSince * injectedPairs[pairAddress].
                ↪ secondsInterestRate;
268             interest = _balances[pairAddress] >= interest ? interest :
                ↪ _balances[pairAddress];
269             // Unchecked due to ternary above
270             unchecked {
271                 _balances[pairAddress] -= interest;
```

```
272                 _balances[_feeRecipient] += interest;
273             }
274             injectedPairs[pairAddress].injectedAmount += interest;
275             injectedPairs[pairAddress].interestCalcTimestamp = block.
                    ↪ timestamp;
276         }
277     } else {
```

Recommendation:

Consider emitting the Transfer event in the first if condition of the feeCalc function to fully adhere to the ERC20 standard.

Updates

The Pseudo Ethereum team acknowledged the issue, stating that it was decided not to include a transfer event because from an external PoV, the balanceOf of the from address (the pair contract) does not change, due to a decrease in Actual pEth and a subsequent increase in pSeudo pEth. It was felt that to have a transfer event emitted, but the balance of the from account does not actually change, which could cause confusion.

SHB.12 Floating pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The contract makes use of the floating-point pragma 0.8.17. Contracts should be deployed using the same compiler version. Locking the pragma helps ensure that contracts will not unintentionally be deployed using another pragma, which in some cases may be an obsolete version, that may introduce issues to the contract system.

Files Affected:

SHB.12.1: pSeudoEthereum.sol

```
3 pragma solidity ^0.8.17;
```

SHB.12.2: FactoryInspector.sol

```
3 pragma solidity ^0.8.17;
```

Recommendation:

Consider locking the pragma version. It is advised that the floating pragma should not be used in production.

Updates

The Pseudo Ethereum team resolved the issue by locking the pragma version to 0.8.17.

SHB.12.3: pSeudoEthereum.sol

```
3 pragma solidity 0.8.17;
```

SHB.12.4: FactoryInspector.sol

```
3 pragma solidity 0.8.17;
```

4 Best Practices

BP.1 Make feeDenominator a Constant or Immutable Value

Description:

The feeDenominator variable in the contract is used as a fixed denominator to calculate the fees. Currently, this variable is not declared as a constant or immutable. The constant or immutable keyword in Solidity provides an advantage in terms of gas cost because values are directly inserted into the bytecode, instead of being read from storage, which can be more expensive in terms of gas cost.

Files Affected:

BP.1.1: pSeudoEthereum.sol

```
80 uint16 public feeDenominator = 1000;
```

Status - Fixed

The Pseudo Ethereum team implemented the best practice by declaring the feeDenominator as a constant.

BP.1.2: pSeudoEthereum.sol

```
77 uint16 public constant feeDenominator = 1000;
```

BP.2 Unnecessary Require Verifications

Description:

The contract contains several require statements, some of them are no longer necessary, starting from Solidity 0.8.0, as the compiler now includes automatic overflow and underflow checks.

Files Affected:

BP.2.1: pSeudoEthereum.sol

```
122 function withdraw(uint wad) public payable {
123     require(_balances[msg.sender] >= wad);
124     _balances[msg.sender] -= wad;
```

BP.2.2: pSeudoEthereum.sol

```
341 require(injectedPairBalance >= amount, "Requested amount exceeds
      ↪ injected balance");
342 injectedPairs[injectedAccount].injectedAmount = injectedPairBalance -
      ↪ amount;
```

Status - Partially Fixed

The Pseudo Ethereum team partially implemented the best practice, by removing the require statement from the withdraw and keeping it in the removeInjectedLiquidity function to provide an error message for the user.

BP.2.3: pSeudoEthereum.sol

```
119 function withdraw(uint wad) public payable {
120     _balances[msg.sender] -= wad;
121     (bool sent, ) = msg.sender.call{value: wad}("");
122     require(sent, "Failed to send Eth");
123     emit Withdrawal(msg.sender, wad);
124 }
```

BP.3 Invalid Error Message in Pair Verification

Description:

The contract includes an invalid error message in the require statement that checks the validity of a pair. The error message currently provided is 'ERC20: transfer amount exceeds balance', which is misleading as it doesn't pertain to the actual condition being checked. Instead, the code is verifying whether a pair is valid or not.

To maintain code clarity and ensure accurate feedback, it is recommended to revise the error message to more accurately reflect the condition being verified. A more suitable message could be 'Invalid pair', as it indicates the actual condition being checked.

Files Affected:

BP.3.1: pSeudoEthereum.sol

```
236 (bool isValidPair, bool isInterestBearing) = factoryInspector.
      ↪ isValidPair(to);
237 require(isValidPair, 'ERC20: transfer amount exceeds balance');
```

Status - Acknowledged

The Pseudo Ethereum team acknowledged the best practice, stating that the error message is included in the case where the user's balance is lower than the amount.

BP.4 Sub-optimal Array Element Removal For Gas Cost Optimization

Description:

The issue observed in this section of the contract pertains to the inefficient removal of elements from an array. In the provided code, when the removeFactory function is invoked by the contract owner, it shifts all subsequent elements of the array (_v3Factories or _v2Factories depending on the isV3 boolean value) one position to the left, starting from the specified factoryIndex. The function then uses the pop() method to remove the last element of the array.

The problem with this approach is that it is unnecessarily expensive in terms of gas cost, particularly for large arrays. Shifting elements one by one involves a substantial number of write operations, each of which incurs a high gas cost due to the use of the SSTORE opcode. This approach could lead to significant, unnecessary expenses for the contract owner who invokes this function.

A more efficient and gas-optimized solution for deleting an element from an array in Solidity is to move the last element of the array to the position of the element to be deleted, and

then pop the last element. This approach reduces the number of write operations to just two, regardless of the size of the array.

Files Affected:

BP.4.1: pSeudoEthereum.sol

```
145 function removeFactory(uint factoryIndex, bool isV3) public onlyOwner {
146     if (isV3) {
147         if (factoryIndex >= _v3Factories.length) return;
148         for (uint i = factoryIndex; i < _v3Factories.length-1; i++){
149             _v3Factories[i] = _v3Factories[i+1];
150         }
151         _v3Factories.pop();
152     } else {
153         if (factoryIndex >= _v2Factories.length) return;
154         for (uint i = factoryIndex; i < _v2Factories.length-1; i++){
155             _v2Factories[i] = _v2Factories[i+1];
156         }
157         _v2Factories.pop();
158     }
159 }
```

Status - Fixed

The Pseudo Ethereum team implemented the best practice by removing the loop and moving the last element of the array to the position of the element to be deleted, and then popping the last element.

BP.4.2: pSeudoEthereum.sol

```
150 function removeFactory(uint factoryIndex, bool isV3) public onlyOwner {
151     address factoryAddress;
152     if (isV3) {
153         if (factoryIndex >= _v3Factories.length) return;
154         factoryAddress = _v3Factories[factoryIndex];
```

```
155         if (_v3Factories.length > 1) {
156             _v3Factories[factoryIndex] = _v3Factories[_v3Factories.length
                ↪ -1];
157         }
158         _v3Factories.pop();
159         factoryAddresses[factoryAddress] = false;
160     } else {
161         if (factoryIndex >= _v2Factories.length) return;
162         factoryAddress = _v2Factories[factoryIndex];
163         if (_v2Factories.length > 1) {
164             _v2Factories[factoryIndex] = _v2Factories[_v2Factories.length
                ↪ -1];
165         }
166         _v2Factories.pop();
167         factoryAddresses[factoryAddress] = false;
168     }
169 }
```

BP.5 Use External Visibility Over Public

Description:

In this context, it's recommended to use external over public for functions that are not called inside the contract. This is because when you call a public function, the EVM loads all the arguments into memory, which costs a certain amount of gas. However, external functions read directly from calldata, which is a less costly operation in terms of gas.

By changing the function visibility from public to external, you can save gas when the function is called externally. This is especially beneficial in scenarios where these functions are frequently or primarily invoked externally, or where the functions have a large number of parameters.

Remember that external functions cannot be called internally, unless they are invoked using this.functionName(). This is because external functions can only be called from other contracts or transactions, not from other functions within the same contract.

Files Affected:

- name()
- symbol()
- decimals()
- totalSupply()
- balanceOf()
- transfer()
- approve()
- transferFrom()
- increaseAllowance()
- decreaseAllowance()
- actualBalance()
- replaceInjectedLiquidty()
- changeInjector()
- flipInterestBearing()
- changePreLoadThreshold()
- changeFeeRecipient()
- changeLoadedBlockLimit()
- changeMaxLoadedPseudo()
- changeTaxRate()
- changeInterestFrequency()
- changeInterestRate()
- changeFactoryInspector()

Status - Fixed

The Pseudo Ethereum team implemented the best practice, by changing the visibility of the mentionned functions to external.

BP.6 Standardized Import of Uniswap Interfaces

Description:

To ensure consistency, security, and compatibility with the Uniswap protocol, we recommend adopting a standardized approach for importing Uniswap interfaces. By replacing the existing import statements for Uniswap interfaces with the standardized import statements provided by Uniswap.

- Uniswap V2 Factory Interface:
  import "@uniswap/v2-core/contracts/interfaces/IUniswapV2Factory.sol";
- Uniswap V2 Pair Interface:
  import "@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol";
- Uniswap V3 Factory Interface:
  import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol";

Files Affected:

BP.6.1: pSeudoEthereum.sol

```
6 import './IUniswapV2Pair.sol';
7 import './IUniswapV2Factory.sol';
8 import './IUniswapV3Factory.sol';
```

BP.6.2: FactoryInspector.sol

```
5 import './IUniswapV2Pair.sol';
6 import './IUniswapV2Factory.sol';
7 import './IUniswapV3Factory.sol';
```

Status - Fixed

The Pseudo Ethereum team implemented the best practice, by importing the uniswap core contracts from the uniswap library.

BP.6.3: pSeudoEthereum.sol

```
6 import '@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol';
7 import '@uniswap/v2-core/contracts/interfaces/IUniswapV2Factory.sol';
8 import '@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol';
```

BP.6.4: FactoryInspector.sol

```
5 import '@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol';
6 import '@uniswap/v2-core/contracts/interfaces/IUniswapV2Factory.sol';
7 import '@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol';
```

5 Tests

Results:

→ Pseudo Eth Test

✓ Confirm correct owner (45ms)

✓ Load factory address

✓ Load Fee Recipient

✓ Deposit Eth, Get Peth (60ms)

✓ Send pEth back, get Eth (67ms)

✓ Load v2 factory address into inspector

✓ Load v3 factory address into inspector (47ms)

✓ Check factories loaded correctly (54ms)

✓ Pre Load psuedo into injector address (102ms)

✓ Test loaded pseudo block limit (122ms)

✓ Load pseudo again, make sure its not doubled up (64ms)

✓ Transfer to another address (87ms)

× Revert on transfering loaded psuedo to non pair address

✓ Revert on adding liquidity when requesting more actual than owned (1182ms)

✓ Create pair / add liquidity (1997ms)

✓ Buy guineapig token (120ms)

- ✓ Check fee earnt from previous buy is correct
- ✓ Set pair address to interest bearing (39ms)
- ✓ Advance time and ensure interest has been charged (194ms)
- ✓ Owner sells more tokens than in liqudity, gets proportional pEth back (183ms)
- ✓ First buyer tries to sell but fails because no actual peth left (97ms)
- ✓ Replace injected liquidity (85ms)
- ✓ Remove injected liquidity (118ms)
- ✓ Check all Psuedo only owners revert on incorrect address (157ms)
- ✓ Check all factory inspector only owners revert on incorrect address (53ms)
- ✓ Change factory inspector (41ms)
- ✓ Remove v2 factory (38ms)
- ✓ Revert factory inspector on 0x0 address
- ✓ Revert interest rate change on too high interest rate
- ✓ Revert tax rate change on too high tax rate
- ✓ Revert fee recipient on 0 address
- ✓ Revert fee recipient on self address
- ✓ Revert pre-loading on too much amount
- ✓ Revert on load block limit too short
- ✓ Revert on load threshold being 0

✓ Revert on transfer to 0 address

35 passing (10s),1 failing

Coverage:

The code coverage results were obtained by running npx hardhat coverage in the pEth project. We found the following results :

- Statements Coverage : 65.6%
- Branches Coverage : 55.56%
- Functions Coverage : 66.67%
- Lines Coverage : 64.71%

6 Conclusion

In this audit, we examined the design and implementation of pSeudo Ethereum contract and discovered several issues of varying severity. pSeudo Ethereum team addressed 6 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised pSeudo Ethereum Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

For a Contract Audit, contact us at contact@shellboxes.com