# SMART CONTRACT AUDIT REPORT

for

# SUTERUSU

Prepared By: Shuxiao Wang

PeckShield

March 1, 2021

## Document Properties

| | |
|---|---|
| Client | Suterusu |
| Title | Smart Contract Audit Report |
| Target | Suterusu |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Huaguo Shi, Xuxian Jiang |
| Reviewed by | Shuxiao Wang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | March 1, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc | February 27, 2021 | Xuxian Jiang | Release Candidate #1 |
| 0.3 | February 20, 2021 | Xuxian Jiang | Additional Findings #2 |
| 0.2 | February 18, 2021 | Xuxian Jiang | Additional Findings #1 |
| 0.1 | February 15, 2021 | Xuxian Jiang | Initial Draft |

## Contact

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


| | |
|---|---|
| Name | Shuxiao Wang |
| Phone | +86 173 6454 5338 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Suterusu 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 Suterusu

Suterusu is a project built upon the state-of-the-art cryptographic technologies zk-Consnark. Suterusu is based on a new efficient range proof scheme with transparent setup, which serves as the foundation of the entire protocol. The Suterusu protocol can provide privacy protection for the users' transaction data without comprising efficiency. Suter Shield is a layer2 solution developed based on the Suterusu protocol and has already been integrated with Ethereum blockchain, Binance Smart Chain, and Huobi/Heco. Their integration is capable of providing privacy-protection service to millions of cryptocurrency users and return rewards to participating users.

The basic information of the Suterusu protocol is as follows:

Table 1.1: Basic Information of The Suterusu Protocol

| Item | Description |
|---:|---|
| Issuer | Suterusu |
| Website | https://suterusu.io/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | March 1, 2021 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the foundation of the entire protocol relies on the proposed efficient range proof scheme with transparent setup. And the proof and correctness of this scheme is not part of this audit.

- https://github.com/suterusu-team/suterusu-protocol.git (f883c1b)

## 1.2 About PeckShield

PeckShield Inc. [13] 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 |

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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.

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 |

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 classified into four categories accordingly, 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 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) [11], 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.

## 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 Suterusu implementation. 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 | 1 | ■ |
| Low | 4 | ■■■■ |
| Informational | 2 | ■■ |
| Total | 7 | |

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 that 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 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and and 2 informational recommendations.

Table 2.1: Key Suterusu Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Informational | Incompatibility with Deflationary/Rebasing Tokens | Business Logic | Confirmed |
| PVE-002 | Low | Possible Front-Running For Nonce Invalidation | Business Logic | Confirmed |
| PVE-003 | Low | Improved Sanity Checks For System Parameters | Coding Practices | Confirmed |
| PVE-004 | Low | Improved Ether Transfers | Business Logic | Confirmed |
| PVE-005 | Informational | Suggested Adherence Of The Checks-Effects-Interactions Pattern | Time and State | Confirmed |
| PVE-006 | Low | Support Of Chain-Specific User Registration | Business Logic | Confirmed |
| PVE-007 | Medium | Trust Issue of Admin Keys | Security Features | Confirmed |

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

# 3 | Detailed Results

## 3.1 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SuterERC20
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

### Description

In the Suterusu protocol, the SuterERC20 contract is designed to be the main entry for ERC20-related interaction with participating users. In particular, one entry routine, i.e., fund(), accepts user deposits of supported assets (e.g., DAI). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the SuterERC20 contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function fund(Utils.G1Point memory y, uint256 unitAmount, bytes memory encGuess)
        public {
        fundBase(y, unitAmount, encGuess);

        uint256 nativeAmount = toNativeAmount(unitAmount);

        // In order for the following to succeed, `msg.sender` have to first approve `
            this` to spend the nativeAmount.
        require(token.transferFrom(msg.sender, address(this), nativeAmount), "Native '
            transferFrom' failed.");
    }
```

Listing 3.1: SuterERC20::fund()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

**Recommendation** If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted `USDT`.

**Status** This issue has been confirmed. However, considering the fact that this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

## 3.2 Possible Front-Running For Nonce Invalidation

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SuterBase`
- Category: Business Logic [9]
- CWE subcategory: CWE-754 [5]

### Description

The `SuterBase` contract provides a `transfer()` function that allows users to authorize transfer by using the `Suterusu` protocol. The function has a nonce-based mechanism to detect and block replay attacks. To elaborate, we show below the implementation of the `transfer()` function.

Particularly, this function implements a rather straightforward logic in firstly validating the given arguments, next properly transferring the internal balances of involved accounts, then checking the freshness of the given nonce, and finally collecting the transfer fee (in addition to returned remaining assets back to the user).

```
    function transfer(Utils.G1Point[] memory C, Utils.G1Point memory D,
                      Utils.G1Point[] memory y, Utils.G1Point memory u,
                      bytes memory proof) public payable {

        uint256 startGas = gasleft();

        // TODO: check that sender and receiver should NOT be equal.
        uint256 size = y.length;
```

```
287         Utils.G1Point[] memory CLn = new Utils.G1Point[](size);
288         Utils.G1Point[] memory CRn = new Utils.G1Point[](size);
289         require(C.length == size, "Input array length mismatch!");


292         for (uint256 i = 0; i < size; i++) {
293             bytes32 yHash = keccak256(abi.encode(y[i]));
294             require(registered(yHash), "Account not yet registered.");
295             rollOver(yHash);
296             Utils.G1Point[2] memory scratch = pending[yHash];
297             pending[yHash][0] = scratch[0].pAdd(C[i]);
298             pending[yHash][1] = scratch[1].pAdd(D);
299             // pending[yHash] = scratch; // can't do this, so have to use 2 sstores
                    _anyway_ (as in above)

301             scratch = acc[yHash];
302             CLn[i] = scratch[0].pAdd(C[i]);
303             CRn[i] = scratch[1].pAdd(D);
304         }

306         bytes32 uHash = keccak256(abi.encode(u));
307         for (uint256 i = 0; i < nonceSet.length; i++) {
308             require(nonceSet[i] != uHash, "Nonce already seen!");
309         }
310         nonceSet.push(uHash);

312         require(transferverifier.verifyTransfer(CLn, CRn, C, D, y, lastGlobalUpdate, u,
                proof), "Transfer proof verification failed!");

314         uint256 usedGas = startGas - gasleft();

316         uint256 fee = (usedGas * TRANSFER_FEE_MULTIPLIER / TRANSFER_FEE_DIVIDEND) * tx.
                gasprice;
317         if (fee > 0) {
318             require(msg.value >= fee, "Not enough fee sent with the transfer transaction
                    .");
319             suterAgency.transfer(fee);
320             totalTransferFee = totalTransferFee + fee;
321         }
322         msg.sender.transfer(msg.value - fee);

324         emit TransferOccurred(y);
325     }
```

Listing 3.2: SuterBase::transfer()

During our analysis, we observe that the calculated nonce (line 36) is checked against the list of received nonce set, i.e., whether the new one is in the set of `nonceSet` (lines 307 – 309). If yes, it reverts the transaction by reporting back stale nonce.

Here comes the problem: when an user intends to invoke `transfer()` to perform the asset transfer by signing the transaction offline, but before the transaction is mined, it is possible for a malicious actor to observe it (by closely monitoring the transaction pool) and then possibly front-runs it by crafting a new transaction (with the same nonce) and offering a higher gas fee for block inclusion. The new transaction may perform a fresh `transfer()/burn(0)` call. If the front-running is successful, the crafted transaction essentially makes the given `nonce` included in the internal set, effectively invalidating the user transaction that is being front-run.

**Recommendation** It may be desirable to not expose the plain-text for the nonce calculation. Fortunately, this issue is mitigated as all nonces will be invalidated after the roll-over successfully advances the epoch.

**Status** This issue has been confirmed. Considering the difficulty and possible lean gains in exploiting the front-running, we agree with the team in keeping it as is.

## 3.3 Improved Sanity Checks For System Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SuterBase`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Suterusu` protocol is no exception. Specifically, if we examine the `SuterBase` contract, it has defined a number of protocol-wide risk parameters, e.g., `BURN_FEE_MULTIPLIER`, `TRANSFER_FEE_MULTIPLIER`, `setEpochBase`, and `setEpochLength`. The first two fee parameters affect the fees that have been charged on `burn` and `transfer` respectively and the last two defines how epochs are being computed. In the following, we show the corresponding routines that allow for their changes.

```
    function setBurnFeeStrategy(uint256 multiplier, uint256 dividend) public {
        require(msg.sender == suterAgency, "Permission denied: Only admin can change
            burn fee strategy.");
        BURN_FEE_MULTIPLIER = multiplier;
        BURN_FEE_DIVIDEND = dividend;
    }

    //function changeTransferFeeStrategy(uint256 multiplier, uint256 dividend, uint256
        nonce, uint256 c, uint256 s) public {
        //require(!usedFeeStrategyNonces[nonce], "Fee strategy nonce has been used!");
        //usedFeeStrategyNonces[nonce] = true;

        //Utils.G1Point memory K = Utils.g().pMul(s).pAdd(suterAgencyPublicKey.pMul(c.
            gNeg()));
```

```
        //// Use block number to avoid replay attack
        //uint256 challenge = uint256(keccak256(abi.encode(address(this), multiplier,
            dividend, "transfer", nonce, suterAgencyPublicKey, K))).gMod();
        //require(challenge == c, "Invalid signature for changing the transfer strategy
            .");
        //TRANSFER_FEE_MULTIPLIER = multiplier;
        //TRANSFER_FEE_DIVIDEND = dividend;
    //}

    function setTransferFeeStrategy(uint256 multiplier, uint256 dividend) public {
        require(msg.sender == suterAgency, "Permission denied: Only admin can change
            transfer fee strategy.");
        TRANSFER_FEE_MULTIPLIER = multiplier;
        TRANSFER_FEE_DIVIDEND = dividend;
    }

    function setEpochBase (uint256 _epochBase) public {
        require(msg.sender == suterAgency, "Permission denied: Only admin can change
            epoch base.");
        epochBase = _epochBase;
    }

    function setEpochLength (uint256 _epochLength) public {
        require(msg.sender == suterAgency, "Permission denied: Only admin can change
            epoch length.");
        epochLength = _epochLength;
    }
```

Listing 3.3: Various Setters In SuterBase

Our result shows the update logic on these fee parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large fee parameter (say more than 100%) will revert the `transfer()` operation.

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

**Status** The issue has been confirmed.

## 3.4 Improved Ether Transfers

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SuterBase
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

### Description

As described in Section 3.1, assets are transferred in or out with a number of helper routines such as transfer() and transferFrom(). While dealing with ERC20 tokens, we have examined related helper routines in their handling of non-standard ERC20 implementations. As for the case of transferring ETH, the Solidity function transfer() is used (lines 29/32 in the code snippet below). However, as described in [1], when the recipient happens to be a contract that implements a callback function containing EVM instructions such as SLOAD, the 2300 gas supplied with transfer() might not be sufficient, leading to an out-of-gas error.

```
22     function burn(Utils.G1Point memory y, uint256 unitAmount, Utils.G1Point memory u,
           bytes memory proof, bytes memory encGuess) public {
23         uint256 nativeAmount = toNativeAmount(unitAmount);
24         uint256 fee = nativeAmount * BURN_FEE_MULTIPLIER / BURN_FEE_DIVIDEND;

26         burnBase(y, unitAmount, u, proof, encGuess);

28         if (fee > 0) {
29             suterAgency.transfer(fee);
30             totalBurnFee = totalBurnFee + fee;
31         }
32         msg.sender.transfer(nativeAmount-fee);
33     }
```

Listing 3.4: SuterETH::burn()

As suggested in [1], we may consider avoiding the direct use of Solidity's transfer() as well. Note that we need to exercise extra caution during the use of call() as it may lead to side effects such as re-entrancy and gas token vulnerabilities. In other words, we need to specify the maximum allowed gas amount when making the (untrusted) external call().

**Recommendation** When transferring ETH, it is suggested to replace the Solidity function transfer() with call().

**Status** The issue has been confirmed.

## 3.5 Suggested Adherence Of The Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

### Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `SuterERC20` as an example, the `burn()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 38) starts before effecting the update on internal states (line 39), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `burn()` function.

```
31     function burn(Utils.G1Point memory y, uint256 unitAmount, Utils.G1Point memory u,
           bytes memory proof, bytes memory encGuess) public {
32         uint256 nativeAmount = toNativeAmount(unitAmount);
33         uint256 fee = nativeAmount * BURN_FEE_MULTIPLIER / BURN_FEE_DIVIDEND;
34
35         burnBase(y, unitAmount, u, proof, encGuess);
36
37         if (fee > 0) {
38             require(token.transfer(suterAgency, fee), "Fail to charge fee.");
39             totalBurnFee = totalBurnFee + fee;
40         }
41         require(token.transfer(msg.sender, nativeAmount - fee), "Fail to transfer tokens
               .");
42     }
```

Listing 3.5: SuterERC20::burn()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

**Recommendation** Apply necessary reentrancy prevention by following the known checks-effects-interactions pattern or making use of the common `nonReentrant` modifier.

**Status** The issue has been confirmed. The team plans to address it in the next upgrade.

## 3.6 Support Of Chain-Specific User Registration

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SuterBase`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

### Description

In the `Suterusu` protocol, there is a dedicated function `register()` to on-board a participating account. It comes to our attention that the challenge calculation takes the following form: `challenge = uint256 (keccak256(abi.encode(address(this), y, K))).gMod()` (line 149). It fails to take into account the important `chainID` information, which makes the differentiation from different `Ethereum`-alike chains impossible.

```
146     function register(Utils.G1Point memory y, uint256 c, uint256 s) public {
147         // allows y to participate. c, s should be a Schnorr signature on "this"
148         Utils.G1Point memory K = Utils.g().pMul(s).pAdd(y.pMul(c.gNeg()));
149         uint256 challenge = uint256(keccak256(abi.encode(address(this), y, K))).gMod();
150         require(challenge == c, "Invalid registration signature!");
151         bytes32 yHash = keccak256(abi.encode(y));
152         require(!registered(yHash), "Account already registered!");
153         // pending[yHash] = [y, Utils.g()]; // "not supported" yet, have to do the below
154 
155         /*
156             The following initial value of pending[yHash] is equivalent to an ElGamal
                encryption of m = 0, with nonce r = 1:
157             (mG + ry, rG) --> (y, G)
158             If we don't set pending in this way, then we can't differentiate two cases:
159             1. The account is not registered (both acc and pending are 0, because `
                mapping` has initial value for all keys)
160             2. The account has a total balance of 0 (both acc and pending are 0)
161 
162             With such a setting, we can guarantee that, once an account is registered,
                its `acc` and `pending` can never (crytographically negligible) BOTH
                equal to Point zero.
```

```
            NOTE: `pending` can be reset to Point zero after a roll over.
        */
        pending[yHash][0] = y;
        pending[yHash][1] = Utils.g();

        totalUsers = totalUsers + 1;
    }
```

Listing 3.6: SuterBase:: register ()

**Recommendation** Ensure the uniqueness of accounts in different blockchains, it is suggested to add the `chainID` information when calculating the challenge to validate a new account.

**Status** The issue has been confirmed.

## 3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

### Description

In the `Suterusu` protocol, there is an administrative-level account `suterAgency` that plays a critical role in governing and regulating the system-wide operations (e.g., fee collection, and parameter setting). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the implementation of `burn()` function. We notice a burn fee may be charged and transferred to the privileged `suterAgency` account.

```
    function burn(Utils.G1Point memory y, uint256 unitAmount, Utils.G1Point memory u,
        bytes memory proof, bytes memory encGuess) public {
        uint256 nativeAmount = toNativeAmount(unitAmount);
        uint256 fee = nativeAmount * BURN_FEE_MULTIPLIER / BURN_FEE_DIVIDEND;

        burnBase(y, unitAmount, u, proof, encGuess);

        if (fee > 0) {
            suterAgency.transfer(fee);
            totalBurnFee = totalBurnFee + fee;
        }
        msg.sender.transfer(nativeAmount-fee);
    }
```

Listing 3.7: SuterETH::burn()

As mentioned in Section 3.4, the ETH fee is collected via the Solidity function transfer() is used (line 29). And if the recipient happens to be a contract that implements a callback function containing EVM instructions such as SLOAD, the 2300 gas supplied with transfer() might not be sufficient, leading to an out-of-gas error. This error could further revert the burn() operation, hence potentially locking up user funds.

More specifically, a compromised suterAgency account would allow the attacker to add a malicious suterAgency to lock up the funds. Also, this privileged account has the authority to make various changes to a number of risk parameters (Section 3.3).

**Recommendation** Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations 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 confirmed. The team confirmed the plan to hold the admin key in a multi-sig account. All changed to privileged operations will be mitigated with necessary timelocks.

# 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Suterusu` protocol. The audited system presents a unique addition by providing a new efficient range proof scheme with transparent setup. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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] Steve Marx. Stop Using Solidity's transfer() Now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/.

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

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.