SMART CONTRACT AUDIT REPORT

for

EVERPAY

Prepared By: Shuxiao Wang

PeckShield

May 15, 2021

Document Properties

Client	everFinance
Title	Smart Contract Audit Report
Target	everPay
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	May 15, 2021	Yiqun Chen	Final Release
1.0-rc	May 7, 2021	Yiqun Chen	Release Candidate

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 **everPay** protocol, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About everPay

The `everPay` is an easy-to-use blockchain-based solution and an application protocol, which enables users to conveniently and reliably make payments and settlements as an Internet-scale application. The `everPay` team adopts a storage-based computation paradigm to build `everPay` as a trusted cross-chain payment and settlement protocol. The protocol is dedicated to improving user experience, lowering the threshold for development, and providing trusted decentralized financial applications for everyone.

The basic information of `everPay` is as follows:

Table 1.1: Basic Information of everPay

Item	Description
Issuer	everFinance
Website	https://everpay.io
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 15, 2021

In the following, we show the contract code deployed at the `Ethereum` with the following address:

- https://kovan.etherscan.io/address/0xc2835910467188351C479a3619585147CeAF48e0#code

And here is the revised contract code after all fixes have been checked in:

- https://kovan.etherscan.io/address/0xC38FbF9987f056D25E6eF40D973Ee65c25c95070#code

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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 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

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

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) [6], 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 Logic	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 everPay 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	1	■
Informational	1	■
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 1 medium-severity vulnerability, 1 low-severity vulnerabilities, and 1 informational recommendations.

Table 2.1: Key everPay Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Susceptible Replays Against EverPay::submit()	Coding Practices	Fixed
PVE-002	Informational	Suggested payable In EverPay::submit()/executes()	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Operator Admin Keys	Security Features	Confirmed

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 Susceptible Replays Against EverPay::submit()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `EverPay`
- Category: Coding Practices [4]
- CWE subcategory: CWE-841 [2]

Description

The `everPay` protocol allows owners to collect their signatures for a given proposal. Owners (saved in `owners`) and required threshold (saved in `required`) are set during the deployment of the `EverPay` contract. Once the collected (verified) signatures is larger than the required threshold, the intended transaction will be executed.

```
    function submit(
        uint256 proposalID, // ar tx id
        bytes32 everHash,
        address to,
        uint256 value,
        bytes memory data,
        bytes[] memory sigs
    ) public whenNotPaused returns (bytes32, bool) {
        bytes32 id = txHash(proposalID, everHash, to, value, data);
        require(!executed[id], "tx_executed");

        for (uint256 i = 0; i < sigs.length; i++) {
            address owner = ecAddress(id, sigs[i]);
            if (!isOwner[owner]) {
                emit SubmissionFailure(id, proposalID, everHash, owner, to, value, data)
                    ;
                continue;
            }

            confirmations[id][owner] = true;
            emit Submission(id, proposalID, everHash, owner, to, value, data);
```

```
        }

        if (!isConfirmed(id)) return (id, false);
        executed[id] = true;

        (bool ok, ) = to.call{value: value}(data);
        if (ok) {
            emit Execution(id, proposalID, everHash, to, value, data);
        } else {
            emit ExecutionFailure(id, proposalID, everHash, to, value, data);
        }

        return (id, true);
    }
```

Listing 3.1: EverPay::submit()

In particular, we show above the related `submit()` function. The function generates the `id` using required fields, i.e., `proposalID`, `everHash`, `to`, `value`, and `data`. However, this calculation assumes all collected signatures from different owners are in the same chain. The absence of a `EIP-712` domainSeparator with the `EIP-155` `chainID` in current calculation makes signature validation susceptible to possible replays across different chains.

Recommendation Add the `EIP-712` `domainSeparator` with the `chainID` into calculation.

Status This issue has been fixed.

3.2 Suggested payable In EverPay::submit()/executes()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `EverPay`
- Category: Coding Practices [5]
- CWE subcategory: CWE-841 [2]

Description

In the `everPay` contract, the `receive()` function receives ethers from the user deposit. This function is implicitly payable, and it will be called whenever the call data is empty (whether or not ether is received). The `submit()` function listed below allows users to set a proposal and collect signatures from `owners`, and finally transfer their deposits out according to the proposal if the number of valid signatures is large enough.

```
    function submit(
        uint256 proposalID, // ar tx id
        bytes32 everHash,
```

```
        address to,
        uint256 value,
        bytes memory data,
        bytes[] memory sigs
    ) public whenNotPaused returns (bytes32, bool) {
        bytes32 id = txHash(proposalID, everHash, to, value, data);
        require(!executed[id], "tx_executed");

        for (uint256 i = 0; i < sigs.length; i++) {
            address owner = ecAddress(id, sigs[i]);
            if (!isOwner[owner]) {
                emit SubmissionFailure(id, proposalID, everHash, owner, to, value, data);
                continue;
            }

            confirmations[id][owner] = true;
            emit Submission(id, proposalID, everHash, owner, to, value, data);
        }

        if (!isConfirmed(id)) return (id, false);
        executed[id] = true;

        (bool ok, ) = to.call{value: value}(data);
        if (ok) {
            emit Execution(id, proposalID, everHash, to, value, data);
        } else {
            emit ExecutionFailure(id, proposalID, everHash, to, value, data);
        }

        return (id, true);
    }
```

Listing 3.2: EverPay::submit()

However, users always have to firstly deposit then transfer to others. Considering the situation that some users may want to transfer ethers from their own address to others straightly without any deposits, we suggest adding a `payable` modifier to the `submit()` function. By adding it, the function can receive ethers and use them for the transaction. The whole process only needs one step, instead of current two, to complete. For the same reason, adding a `payable` modifier to the `executes()` is also suggested.

Recommendation Add `payable` modifiers to the `submit()` and `executes()` functions.

Status This issue has been resolved.

3.3 Trust Issue of Operator Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: EverPay
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the EverPay contract, there is an operator account that plays a critical role in governing and regulating the entire operation and maintenance. It has the privilege to pause the submit() function which enables owners to collect their signatures for a given proposal and send ethers as mentioned in Section 3.1.

In the following, we list the submit() function.

```
    function submit(
        uint256 proposalID, // ar tx id
        bytes32 everHash,
        address to,
        uint256 value,
        bytes memory data,
        bytes[] memory sigs
    ) public whenNotPaused returns (bytes32, bool) {
        bytes32 id = txHash(proposalID, everHash, to, value, data);
        require(!executed[id], "tx_executed");

        for (uint256 i = 0; i < sigs.length; i++) {
            address owner = ecAddress(id, sigs[i]);
            if (!isOwner[owner]) {
                emit SubmissionFailure(id, proposalID, everHash, owner, to, value, data)
                    ;
                continue;
            }

            confirmations[id][owner] = true;
            emit Submission(id, proposalID, everHash, owner, to, value, data);
        }

        if (!isConfirmed(id)) return (id, false);
        executed[id] = true;

        (bool ok, ) = to.call{value: value}(data);
        if (ok) {
            emit Execution(id, proposalID, everHash, to, value, data);
        } else {
            emit ExecutionFailure(id, proposalID, everHash, to, value, data);
        }
```

```
        return (id, true);
    }
```

Listing 3.3: EverPay::submit()

The only way for the user who wants to transfer his/her deposit out from the contract is calling the `submit()` function. However, the `operator` can enable/disable this important function whenever he/she wants. We understand the need of the privileged functions for contract operation and maintenance, but at the same time the extra power to the `operator` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making this privilege explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to `operator` explicit to everPay users.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `everPay` protocol. The `everPay` locks the assets of other public blockchains into a smart contract and maps them to corresponding assets, hence enabling users to make transfers and payments on its protocol. The current code base is well organized and 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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