Least Authority
PRIVACY MATTERS

Paloma Forwarder Contracts
Security Audit Report

Paloma Pay, Inc.

Final Report Version: 14 January 2021

Table of Contents

Overview

Background

Paloma Pay, Inc. has requested that Least Authority perform a security audit of the Paloma Forwarder Contracts. Paloma Pay is a non-custodial payment solution. The solution's primary goal is to have all transactions take place directly between customers and merchants, without the ability to prevent the movement of funds once a payment has been initiated. In order to achieve this goal, a Forwarder Contract (decentralized) is leveraged to act as a middle man, which forwards funds to the merchant's custodian wallet (exchange wallet).

Project Dates

- **January 4 - 6**: Code review *(Completed)*
- **January 8**: Delivery of Initial Audit Report *(Completed)*
- **January 10 - 13**: Verification Review *(Completed)*
- **January 14**: Final Audit Report delivered *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Alex Lewis, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Paloma Forwarder Contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Paloma Forwarder Contract: https://github.com/EastAgile/paloma_forwarder_contract

Specifically, we examined the Git revisions for our initial review:

```
cba4fcbb3308b0cd01480f7346cdbfea0b1f3c8b
```

For the verification, we examined the Git revision:

```
95ab2a3f331d1e8fa31ad4b82f9b0f7e1d0daf88
```

```
546ee80c1c8feae5d6895a8b14ac7a20cd06a428
```

```
5caa3c247971ee62921eebb445bbc50ebc98eab9
```

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Paloma Non-custodial Payment Proposal.docx: Provided to Least Authority via email on 20 December 2020
- EIP-1167: https://eips.ethereum.org/EIPS/eip-1167

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation and adherence to best practices;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the smart contracts intended use or disrupt the execution of the smart contracts;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team found the scope of the audit to be sufficient and identified no dependency concerns.

Code Quality

The code in the Paloma Forwarder Contracts repository is well-written, organized and follows Solidity security best practices, along with the OpenZeppelin standards.The system uses a modern compiler version 0.7.0, which helps to prevent the presence of outdated Solidity bugs in the code base. During our investigation and analysis, we did not identify any security critical issues. However, we have outlined suggestions that will make the system more robust by providing additional security and further adherence to best practices.

The test coverage is adequate and includes tests for fail cases. For example, attempting to initialize the Paloma Forwarder Contract more than once is highlighted as an error, as expected.

Documentation

The code base includes a few short code comments. Given that the complexity of the contracts code is very minimal, we do not consider this to be problematic. However, expanding code comments would facilitate an easier review and understanding of the contracts by future auditors and contributors to the project. For example, the cloning process would benefit from additional comments to help explain the intended functionality. As a result, we recommend that the comments follow the NatSpec guidelines (Suggestion 1) and to add clarifying comments that guide the reviewer to previous deployments as is the case with the proxy used in the cloning function.

In addition, there is no documentation external to the code comments. We recommend that the intended use case for the Paloma Forwarder Contracts be further documented in the `README.md`, which will facilitate an easier understanding for reviewers of the code (Suggestion 4).

System Design

Paloma Pay utilizes a proxy clone first developed by OpenZeppelin. This has been modified to work with the CREATE2 opcode to create a factory that can deploy off-chain generated forwarder contracts that have payments similar to the Coinbase USDC merchant system, and an additional salt is supplied to the cloning process to accomplish this. This simple proxy allows the Factory Contract to print many forwarder contracts at a low cost. The cloning process is a particularly complex aspect of the contract code, and special attention was given to reviewing whether it was comparable to the original implementation. We concluded that the implementation of CRATE2 in Paloma Pay correctly replicates the original functionality without any security issues. As noted previously, further context around the intended functionality and use case would be helpful to include in the code comments and documentation.

Overall, security considerations are apparent in the design. For example, the Paloma Pay team modified the Factory cloning process to only allow the owner to initiate the clones so that the salts provided would not be spammed by another entity. We commend the Paloma Pay team for restricting the access to an important function.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Suggestion 1: Use NatSpec Comments	Resolved
Suggestion 2: Consider Using A Two-Step Ownership Update Process	Resolved
Suggestion 3: Use Standard Initialization Function	Resolved
Suggestion 4: Supply Documentation	Resolved

Suggestions

Suggestion 1: Use NatSpec Comments

Synopsis

Functions present in the code base do not follow NatSpec guidelines for Solidity comments. Using NatSpec comments would help code reviewers and users easily understand the inputs and functionality of every method present, and therefore aid in identifying any potential issues.

Mitigation

Add comments to the beginning of each function in the NatSpec style, which is considered a Solidity best practice.

Status

The Paloma Pay team has implemented NatSpec comments for each contract and function.

Verification

Resolved.

Suggestion 2: Consider Using A Two-Step Ownership Update Process

Location

/contracts/ForwarderFactory.sol#L7

Synopsis

The standard OpenZeppelin ownership contracts have basic checks for input sanitation of the newly supplied owner address. However, they cannot check to ensure that the provided address is controlled by the intended target. If an address is supplied that passes the input checks, but is not actually controlled by the intended target (e.g. as a result of human error), the ownership of the Forwarder Factory Contract will be lost. The feasibility of this is reduced if the team does not plan to update the owner of the factory often. In addition, losing ownership of the factory does not result in loss of funds, which limits the impact of this scenario.

Mitigation

Update the `transferOwnership` function of the standard OpenZeppelin ownership contract. The recommended way to update ownership would require two contract calls, the first to a function that proposes the new ownership address, and the second call to another function from the new owner that accepts ownership. This will ensure that the intended target of the ownership update can successfully create transactions.

Status

The Paloma Pay team has responded that the Forwarder Factory Contract is a utility contract used by the system to generate contracts. If the reported scenario takes place, they are able to re-deploy and use a new Forwarder Factory Contract and it is acceptable to use standard ownership in this case as the contract is disposable.

We consider the Paloma Pay team's solution to be reasonable and this suggestion to be resolved without implementing the aforementioned mitigation strategy.

Verification

Resolved.

Suggestion 3: Use Standard Initialization Function

Location

/contracts/Forwarder.sol#L21

Synopsis

The Paloma Forwarder Contract is deployed via a proxy system where the constructor will not work, which is why it must be initialized with a custom `init` function created by the Paloma Pay team. While the functionality of this is minimal, and there are no apparent issues with the `init` function they have established, OpenZeppelin now provides a standard proxy initialization contract that will ensure that the function can only be called once.

Mitigation

Import the `initializable` contract from OpenZeppelin to replace the custom `init` function present in the `Forwarder` contract.

Status

The Paloma Pay team does not intend to implement the suggested mitigation and has responded that they were aware of OpenZeppelin's proxy initialization when developing the Forwarder Contract. They have decided against using it as it added more code and believe it does not reduce the security risk. They also state that OpenZeppelin's proxy initialization is designed for other types of contracts (e.g. upgradeable contracts) as opposed to the Minimal Proxy Contracts used by Paloma Pay, which are implemented in an effort to minimize the cost of deploying Forwarder Contracts.

With this goal in mind, the Paloma Pay team found that OpenZeppelin's proxy initialization added extra variables and code, thus increasing the gas cost. They have tested the OpenZeppelin proxy initialization against their own implementation and found the gas cost increased significantly: the OpenZeppelin proxy initialization cost 1,536,458 gas for generating 20 contracts while the Paloma Pay implementation cost 1,381,738. Due to the volume of contracts generated in production, this would result in an 11% increase in cost to scale the business.

Considering the higher than expected gas cost, we accept the Paloma Pay team's decision to not implement this suggestion.

Verification

Resolved.

Suggestion 4: Supply Documentation

Synopsis

The Paloma Forwarder Contracts repository that we examined does not contain any documentation. In particular, the intended use case for the Forwarder Contract is unclear, which increases complexity in reviewing the code (e.g. when trying to understand whether or not the Forwarder Contract should be permissioned or whether it is the intended use case to allow anyone to flush the Forwarder).

Mitigation

Provide a `README.md` file that contains basic use case information that provides context to reviewers of the intended uses of the Paloma Forwarder Contracts.

Status

The Paloma Pay team has expanded the `README.md` file as suggested. The `README.md` documentation now includes information on the interactions between contracts in addition to overview information on the Paloma Pay system and the specific role of the Forwarder Contract.

Verification

Resolved.

Recommendations

We commend the Paloma Pay team for their security considerations, as is demonstrated by a clean and well-structured code base that adheres to development best practices. We did not identify any security critical issues in the Paloma Forwarder contract, however, implementing the suggestions outlined above would further improve the security and transparency of the system.

The Paloma Pay team implemented additional code comments describing the functionality of each method, which aids in understanding and reviewing the system, thus facilitating the recognition of potential issues.

Finally, the Paloma Pay team provided valid reasoning for the decision against implementing the suggested mitigation strategies regarding using a two-step ownership update process and using a standard initialization function. As a result, we consider their current design choices to be adequate.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.