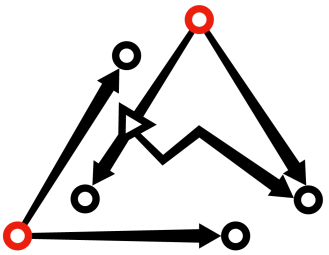

Least Authority

PRIVACY MATTERS

Offers Primitive

Security Audit Report

Chia Network

Final Audit Report: 20 September 2023

Table of Contents

Overview

Background

Chia Network has requested that Least Authority perform a security audit of their Offers Primitive.

Project Dates

- **March 20 - May 16, 2023:** Initial Code Review *(Completed)*
- **May 19, 2023:** Delivery of Initial Audit Report *(Completed)*
- **September 20, 2023:** Verification Review *(Completed)*
- **September 20, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nicole Ernst, Security Researcher and Engineer
- Nikos Iliakis, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- Ann-Christine Kycler, Security Researcher and Engineer

Coverage

Target Code and Revision

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

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

- Chia-Network / Chia-blockchain: https://github.com/Chia-Network/chia-blockchain
  - Trade Manager:
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/wallet/puzzles/settlement_payments.clsp
    - https://github.com/Chia-Network/chia-blockchain/tree/main/chia/wallet/trading
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/wallet/trade_manager.py
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/wallet/trade_record.py
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/wallet/outer_puzzles.py
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/wallet/nft_wallet/nft_wallet.py#L785-L1068
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/data_layer/data_layer_wallet.py#L1142-L1286
    - https://github.com/Chia-Network/chia-blockchain/blob/main/chia/wallet/puzzles/graftroot_dl_offers.clsp
  - Action Manager
    - https://github.com/Chia-Network/chia-blockchain/tree/quex.offer_refactor/chia/wallet/action_manager
    - https://github.com/Chia-Network/chia-blockchain/blob/quex.offer_refactor/chia/wallet/puzzles/add_wrapped_announcement.clsp
    - new `InnerDriver`/`OuterDriver` classes

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

- `Dac9ee506519fa53425986b0c6f1e4fd3ae97dea`

For the verification, we examined the Git revision:

- `b32128949f09ef3e08484fd36b911ba9df10a0dc`

For the review, this repository was cloned for use during the audit and for reference in this report:

- Chia-Network / Chia-blockchain:
  https://github.com/LeastAuthority/Chia-Network

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:

- Primitive:
  https://chialisp.com/offers
- RPC API:
  https://docs.chia.net/offer-rpc
- CLI API:
  https://docs.chia.net/offer-cli
- GUI Tutorial:
  https://docs.chia.net/guides/offers-gui-tutorial
- CLI Tutorial:
  https://docs.chia.net/guides/offers-cli-tutorial

In addition, this audit report references the following documents and links:

- S. Bratus, L. Hermerschmidt, S. M. Hallberg, M. E. Locasto, F. D. Momot, et al., "Curing the Vulnerable Parser: Design Patterns for Secure Input Handling." *USENIX*, 2017, [BHH+17]
- Hypothesis:
  https://hypothesis.readthedocs.io/en/latest

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Adversarial actions and other attacks;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) attacks and security exploits that would impact or disrupt execution;
- Vulnerabilities within individual components and whether the interactions between the components are secure;
- Exposure of any critical information during interaction with any external libraries;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other methods of exploitation;

- Data privacy, data leaking, and information integrity;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Chia Network's Offers Primitive enables users to exchange assets without requiring an intermediary and while ensuring both buyer and seller in any transaction receive what they expected in assets. Users wanting to sell or buy an asset such as a token or NFT create an `Offer`, which can be posted to the public. The `Offer` consists of a Bech32-encoded file including an encoded `Offer` object – an unfinished spend bundle. A counterparty wishing to accept the `Offer` creates a transaction by completing the spend bundle, at which point the transaction between seller and buyer is finalized on-chain.

Our team performed a comprehensive review of the Offers Primitive and investigated the areas of concern listed above. We considered a threat model with the attacker taking the role of each the maker, the taker, and the observer of an `Offer` within the Offers Primitive. We examined the mechanism that checks the validity of an `Offer` for security vulnerabilities. We investigated the use of nonces in the implementation and issues arising from the potential reuse of nonces in notarized payments. In addition, our team checked for security vulnerabilities resulting from the handling of multi-asset `Offers` and from `Offer` aggregation.

Our team checked the implementation of the Inner and Outer classes, files that interpret asset-specific wallet and puzzle code, as it relates to `Offers`. Furthermore, our team performed preliminary property-based testing on `Offers` to identify potential edge cases that could cause unexpected behavior.

Our team found that Offers Primitive is well-designed and implemented. Although we did not identify critical issues in the design and implementation, we found opportunities for improvement in the code quality, testing, and documentation, which all contribute to the overall security of the system.

System Design

The Offers Primitive is a decentralized protocol enabling users to exchange assets and sets of assets efficiently while protecting the counterparties in the transaction, and the network. `Offers` are exchanged in Bech32 format and can be created by anyone. Our team could not find a way to circumvent the checks put in place to prevent an attacker from causing harm to the system and/or its users. However, due to the decentralized and public nature of `Offer` creation and publication, we found that the design of the system can be improved by considering components within the system untrusted. This requires rigorous validation of inputs at well-considered points in the workflow. As a result, we recommend that the process of parsing an `Offer` file be improved to better prevent malformed `Offers` from being parsed in accordance with the "parsing before processing" principle (Issue B).

Code Quality

Our team performed a manual review of the files listed above and found that the code is well-organized, and there are comments explaining critical parts of the code. However, our team identified an implementation error that leads to crashes during `Offer` creation (Issue A). We also identified areas of improvement in error handling within the implementation (Suggestion 3), and recommend the use of a parent class for classes with similar functions (Suggestion 2). In addition, our team found unresolved TODOs in the codebase, which reduced the readability of the code and raised questions about its

completeness. We recommend that outstanding TODOs be resolved and removed from the codebase (Suggestion 7).

Tests

While the Offers Primitive contains sufficient coverage of integration tests, no unit tests were implemented. Inclusion of unit tests in the CI and continuous deployment workflows is considered best practice and aids in automating the process of verification. Higher test coverage increases trust in the system and enables early detection of bugs and errors. As a result, we recommend that the Chia team increase unit test coverage (Suggestion 4).

We also recommend refactoring the code to improve testability and readability (Suggestions 5), and implementing property-based tests to increase the likelihood of finding bugs in the handling of edge cases (Suggestion 6).

Documentation

The project documentation provided for this review is in need of improvement, as it accurately describes what the `Offers` protocol does but does not provide a sufficient description of how the functionality is implemented in the codebase. It is difficult to read the documentation and find the corresponding functionality in the code, and vice versa. We recommend that the language in the documentation match the language in the code and that key terminology be made more consistent to facilitate reasoning about the security of the implementation and checking for unintended behavior (Suggestion 1).

Code Comments

Our team found that some functions and components have sufficient code comments that describe their expected behavior. However, we recommend that code comments be improved in their consistency (Suggestion 8).

Scope

The scope of this review included all security-critical functionality of the Offers primitive. Our team assumed that user interfaces such as devices and wallets behave as expected. During the review, the Chia team stated that the functionalities that are in the `trade manager` are planned to be moved into the `action manager`. As a result, we recommend that a further review be performed by an independent security team once those changes are complete.

Dependencies

Our team did not identify any security issues in the use of dependencies.

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
Issue A: Improper Check of Balance Causes Crashes During Offer Creation	Resolved
Issue B: Offer Decoding May Lead To Unexpected Errors	Unresolved
Suggestion 1: Use Consistent Terminology Within Code	Unresolved

Suggestion 2: Create a Parent Class for Classes With Similar Functions and Names	Unresolved
Suggestion 3: Improve Error Handling	Unresolved
Suggestion 4: Increase Test Coverage	Unresolved
Suggestion 5: Refactor Code To Improve Input Validation, Ease of Testing	Unresolved
Suggestion 6: Perform Property-Based Testing on Decoding / Parsing Functionalities	Unresolved
Suggestion 7: Resolve TODOs in Codebase	Unresolved
Suggestion 8: Improve Code Comments	Unresolved

Issue A: Improper Check of Balance Causes Crashes During Offer Creation

Location

`chia/wallet/wallet.py#L654-L657`

Synopsis

The check for sufficient balance does not properly verify if the balance is spendable. This leads to crashes in the `Offer` creation code after the balance is checked.

Technical Details

`balance = await self.get_confirmed_balance()` should be `balance = await self.get_spendable_balance()`

Remediation

We recommend replacing the function `get_confirmed_balance` with `get_spendable_balance`.

Status

The Chia team has implemented the recommended remediation.

Verification

Resolved.

Issue B: Offer Decoding May Lead To Unexpected Errors

Location

`chia/wallet/trading/offer.py`

Synopsis

It is possible to create `Offers` with the constructor and transform them to Bech32, causing crashes in `from_bech32`. `Offers` that are Bech32-encoded should be treated as untrusted inputs and be properly parsed/validated.

Furthermore, it is possible to create an `Offer` that successfully translates to Bech32, gets successfully decoded, but then fails upon further usage. See [BHH+17] for design patterns for secure input handling.

Impact

Attackers may craft inputs that cause errors in the user's wallet on import. This could lead to unexpected system states or crashes.

Preconditions

An attacker would need to get a user to import a Bech32-encoded `Offer` file.

Feasibility

In order to produce a crash with an invalid `Offer`, the attacker would need to understand the data structures involved in the `Offers`.

Technical Details

Currently, there are several functions, such as `from_bech32`, `from_bytes`, and `parse`, that return objects of the class `Offer` within that class, which might operate on external input. Using property-based testing with `hypothesis`, our team generated strategies, which produced the simplest structures that would be accepted as inputs to the `Offer` constructor. This revealed that it is possible to generate `Offers` that will successfully be encoded with the `to_bech32` function, but then would either fail in the `from_bech32` function or – even more surprisingly – fail when functions are called on the `Offer` object that is returned by the decoding function. For an example of an `Offer` file that should be rejected as invalid before attempting to process, please refer to Appendix A.

Remediation

We suggest that there be one location that creates an `Offer` object from an external input, and that all validation be performed in that one location. This means that the `Offer` object returned should be a valid object, and no method call on an object should cause errors due to improper encoding, missing dictionary elements, improper coin expenditure, etc.

Status

At the time of the verification, the suggested remediation has not been resolved.

Verification

Unresolved.

Suggestions

Suggestion 1: Use Consistent Terminology Within Code

Location

`chia/wallet/trade_manager.py#L628`

Synopsis

Our team identified multiple instances of variables being named inconsistently within the implementation (e.g., `solver` and `sibling` in the keybase chat). In addition, there are instances in which function names do not provide sufficient insight to the expected behavior of the function.

Mitigation

We recommend consistent use of terminology in the codebase and documentation by adhering to a single naming convention and naming functions descriptively to indicate their intended purpose.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 2: Create a Parent Class for Classes With Similar Functions and Names

Location

`chia/wallet/action_manager/actions_aliases.py`

Synopsis

There are a number of classes that have functions and names in common (e.g., `` `from_solver` ``).

Mitigation

We recommend that a(n) parent/abstract class be created, and that these classes inherit/implement the interface to overload these functions. We further recommend checking that all the necessary functionality is implemented, and avoiding duplicates, across the entire codebase.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 3: Improve Error Handling

Location

`wallet/trading/offer.py#L150-L151`

`wallet/action_manager/protocols.py#L294-L298`

Synopsis

Error handling can be improved, as related to Issue A. It is not a recommended practice to pass on all exceptions because a large number of exceptions can be a sign of an attack on the system.

`Offer` decoding (`from_bech32`) is expected to throw a `ValueError: Invalid offer` exception. However, other errors during `Offer` decoding and decompression within that function are not caught and handled (see Issue B).

Mitigation

We recommend logging errors to identify possible attacks. For specific exceptions that commonly occur during normal operations, we recommend utilizing a typed `except` block.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 4: Increase Test Coverage

Synopsis

The Chia team utilizes integration testing to test `Offer` functionalities. While integration testing is a good approach to test that components work together correctly, testing individual components is essential to test for correct behavior early and often, as well as for performing regression testing. Individual tests (e.g. unit tests) require much less resources and can be run quickly.

Mitigation

We recommend testing the components of the `Offer` system individually with, for example, property-based tests.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 5: Refactor Code To Improve Input Validation, Ease of Testing

Location

Example (non-exhaustive):

`chia/wallet/trading/offer.py#L437-L535`

Synopsis

Separating different functionalities into different functions makes it easier to test certain functions individually and to understand the code for maintaining or auditing. In the Chia wallet codebase, there are many functions that perform different operations on input. For example, `to_valid_spend` in `offer.py` could consist of various parts. In addition, the serialization of the offered coins and the creation of the solution could be integrated into different functions and thereby tested separately.

Mitigation

We recommend the separation of concerns and loose coupling to simplify the testing process and improve code readability.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 6: Perform Property-Based Testing on Decoding / Parsing Functionalities

Synopsis

Property-based testing allows for testing a large number of inputs and edge cases that would be impractical to test manually. Furthermore, defining properties that the code should uphold can benefit the development process, as well as help uncover bugs. Property-based testing can also run as part of CI and help ensure that properties still hold even after code changes.

Mitigation

We recommend that the Chia team look into property-based testing, for example, for the `Offer` functionality, or correct the behavior of `trade` and `action manager`. This could be done by utilizing `hypothesis`, which our research team referred to in order to uncover inputs that could lead to unwanted behavior.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 7: Resolve TODOs in Codebase

Location

Examples (non-exhaustive):

`chia/wallet/nft_wallet/nft_wallet.py#L617`

`chia/wallet/nft_wallet/nft_wallet.py#L798`

`chia/wallet/trade_manager.py#L922`

Synopsis

There are many unresolved TODO items in the code comments of the in-scope files, which may lead to a lack of clarity and cause confusion about the completion of the implementation. Resolving TODOs prior to a comprehensive security audit of the code allows security researchers to better understand the full intended functionality of the code, indicates completion, and increases readability and comprehension.

However, note that resolving these TODOs could result in changes to the code, which might have an impact on the security of the project that is unpredictable at the time of the audit.

Mitigation

We recommend that TODOs be resolved or removed from the codebases.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 8: Improve Code Comments

Location

Examples (non-exhaustive):

chia/wallet/trading/offer.py#L464-L635

chia/wallet/trading/offer.py#L464-L635

chia/wallet/trade_manager.py#L935-L1017

Synopsis

Currently, the codebase lacks explanation in some areas. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. Comprehensive in-line documentation explaining, for example, expected function behavior and usage, input arguments, variables, and code branches can greatly benefit the readability, maintainability, and auditability of the codebase.

Mitigation

We recommend expanding and improving the code comments within the codebase to facilitate reasoning about the security properties of the system.

Status

At the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Appendix

Appendix A: Invalid Offer Example

The following input can be successfully created as an `Offer` object, en- and decoded to and from Bech32 and then still fail when the `get_coins_to_offer` function is called on it, for example. This is one of the examples generated using specialized `hypothesis` strategies that causes failures on a seemingly valid `Offer` object.

```
Offer(
      requested_payments={None: []},
      _bundle=SpendBundle(
            coin_spends=[CoinSpend(
                  coin=Coin(
                        parent_coin_info=<bytes32:
0000000000000000000000000000000000000000000000000000000000000000>,
                        puzzle_hash=<bytes32:
0000000000000000000000000000000000000000000000000000000000000000>,
                        amount=1,
                  ),
                  puzzle_reveal=SerializedProgram(00),
                  solution=SerializedProgram(80),
            )],
            aggregated_signature=<G2Element
c00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000>,
      ),
      driver_dict={<bytes32:
0000000000000000000000000000000000000000000000000000000000000000>: PuzzleInfo(
            info={'type': <bytes32:
0000000000000000000000000000000000000000000000000000000000000000>,
            'also': <bytes32:
0000000000000000000000000000000000000000000000000000000000000000>},
            ),
            None: PuzzleInfo(
```

```
            info={'type': <bytes32:
0000000000000000000000000000000000000000000000000000000000000000>,

            'also': <bytes32:
0000000000000000000000000000000000000000000000000000000000000000>},

            )},

      )
```

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 multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. 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. We are an international team that believes 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 include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website 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. As 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 and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and 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 before having 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 and comprehensive 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 Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, 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 a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.