Chia CAT standard

Security Assessment

DATE

Jun 24, 2022

Prepared for:
Paul Hainsworth
Chia

Prepared by: **Gustavo Grieco and Troy Sargent**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Chia engaged Trail of Bits to review the security of its CAT standard. From May 31 to June 24, 2022, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	2
Low	0
Informational	0
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Patching	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-Chia-004**
  The sha256 operation concatenates and hashes its arguments, which can generate collisions if the values have variable length that affect the security or correctness of the ChiaLisp code.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Ann Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
May 31, 2022	Pre-project kickoff call
June 3, 2022	Status update meeting #1
June 10, 2022	Status update meeting #2
June 17, 2022	Status update meeting #3
June 24, 2022	Delivery of report draft
June 24, 2022	Report readout meeting
Month XX, 202X	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Chia's CAT standard. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are proper access controls used in the code that users interact with?
- Could an unauthorized user mint or melt tokens?
- Could one user withdraw another user's tokens?
- Could a user's funds be frozen or stuck?
- Are the CAT token proofs complete and sound?
- Can the inner puzzle interfere with the CAT expected behavior?
- Are there any centralization issues associated with the CAT standard or the TAIL program?
- Can the creator of the CAT block or force the owner of the coin to perform some transaction?
- Is the parsing of the output conditions safe and correct?

Project Targets

The engagement involved a review and testing of the following repositories.

chia_rs

Repository	https://github.com/Chia-Network/chia_rs/releases/tag/0.1.1
Version	`4faa802e37f00ae4ead622aef660e11c0d0ae978`
Type	Rust

chia-blockchain

Repository	https://github.com/Chia-Network/chia-blockchain/tree/1.3.5
Version	`0f5a6df4ffcd7b1d5b950b9f40c15b4e6045ee1b`
Type	Python

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- The CAT-1 standard: Chia developed a standard for fungible tokens using Chialisp smart contracts (called "puzzles" following their terminology). We manually reviewed the puzzle associated with this standard to uncover flaws that could allow an attacker to disrupt their expected behavior or break any internal invariants.
- The TAIL puzzle: this puzzle is an optional, but an important part of the CAT-1 standard, since it allows the creator of the token to define specific rules for minting and burning. While this puzzle is not specified, we reviewed development practices and interfaces provided by Chia in order to implement it, looking for aspects to improve and potential security pitfalls.
- The `chia_rs` crate. This rust code performs the parsing and some limited validation of the output conditions, as part of the consensus mechanism. We performed manual review and automated testing review of code to make sure it never panics and returns the expected values.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- The Chia full node implementation, including transaction parsing and consensus algorithms
- Differential testing between the Rust and Python implementation of the output condition parsing and processing.
- Farmer's frontrunning capabilities and the impact on CAT tokens

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
AFL.rs	An in-process, coverage-guided, evolutionary fuzzing engine based on AFL	Appendix C
Clippy	A collection of lints to catch common mistakes and improve Rust code.	Pedantic
Dylint	A tool for running custom Rust lints from dynamic libraries.	General and Restriction

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The CAT-1 standard performs a small number of arithmetic operations, which were manually verified.	Satisfactory
Auditing	If a CAT transaction fails, it can be difficult to determine the cause of it, if the creator of the code has not used an exception with an error message. This makes blockchain use and monitoring more difficult than expected.	Moderate
Authentication / Access Controls	The CAT-1 standard relies on coin announcements, but the hash collisions in concatenation of inputs using the sha256 opcode can break some important security properties (TOB-Chia-004)	Weak
Complexity Management	While the ChiaLisp code is splitted in different functions and contain reasonable amount of documentation, it can be difficult to follow (TOB-Chia-001, TOB-Chia-005)	Weak
Decentralization	The use of TAIL programs, with a limited amount of privileges, allow to increase the decentralization, at the cost of some additional complexity.	Satisfactory
Documentation	The ChiaLisp website contains an extensive amount of documentation, tutorial and security pitfalls to avoid when creating smart coins.	Strong
Front-Running	Farmers have incentives to front-run transactions to	Further

Resistance	increase their fees (TOB-Chia-002). We think this kind of issue requires more effort in order to understand the implications when using the CAT-1 standard for trades.	**investigation required**
Testing and Verification	While the test covers a variety of scenarios when using the CAT-1 standard, it is still unclear if they are enough. There is no notion of coverage in the Chialisp code, so it is difficult to measure systematically. On top of that, more advanced techniques like fuzzing are not straightforward to use, since executing the code requires specific values to run (which are verified using hashes) and has an unconstrained structure more similar to code than actual plain data.	**Further investigation required**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Chialisp compiler should warn when output conditions do not use input parameters	Data Validation	High
2	Fee for farmers should have a explicit field in transactions	Data Validation	High
3	Chialisp development can be error-prone	Data Validation	High
4	sha256 is prone to collisions	Data Validation	High
5	CAT1 inputs structures are not verified	Data Validation	Medium
6	sha256tree1 can be re-implemented incorrectly	Patching	Medium

Detailed Findings

1. Chialisp compiler should warn when output conditions do not use input parameters	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-Chia-001
Target: `chialisp compiler`	

Description

The chialisp compiler does not warn if a potential security issue is detected, despite it being widely documented.

Puzzle code should be resistant to manipulation by the farmers. Typically, they will change some parameter to redirect the coins into another address (puzzle hash). For instance, in the following code do not assert the puzzle hash in some of the output conditions:

```
(mod (
        TARGET_AMOUNT
        CASH_OUT_PUZZLE_HASH
        my_amount
        new_amount
        my_puzzlehash
     )

     ...

     (defun-inline cash_out (CASH_OUT_PUZZLE_HASH my_amount new_amount
my_puzzlehash)
      (list
        (list CREATE_COIN CASH_OUT_PUZZLE_HASH new_amount)
        (list CREATE_COIN my_puzzlehash 0)
        (list ASSERT_MY_AMOUNT my_amount)
        ; (list ASSERT_MY_PUZZLEHASH my_puzzlehash)
        (list CREATE_COIN_ANNOUNCEMENT new_amount)
      )
     )
     ...

)
```

Figure 1.1: An example of vulnerable code from the PiggyBank that does not assert the puzzle hash

Additionally, if the list of conditions outputs can be empty, the compiler should give a warning, since it could be accidentally introduced by the developer.

Exploit Scenario

Alice writes some puzzle code, but she forgets an assertion or an announcement check. The chialisp compiler will not warn about it, allowing Alice to deploy vulnerable code.

Recommendations

Short term, consider implementing a check to detect possible missing conditions. For instance:

1. Iterate over non-curried parameters and taint them.
2. Propagate the taint recursively.
3. Verify that each parameter taint reaches some certain outputs (e.g. asserts)

Long term, review common security issues to make sure the compiler warns the user about them.

2. Fee for farmers should have a explicit field in transactions

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-Chia-002
Target: `chia-blockchain`	

Description

The lack of explicit field to specify farmers fee provides an incentive to manipulate puzzle solutions.

In the Chia blockchain, farmers are participants that are responsible for including transactions in blocks. According to the documentation, if excess funds are sent in a transaction will be considered a fee for the farmer. This gives farmers a misaligned incentive to try to manipulate transactions with different solutions, in order to decrease the funds that are used during the transaction.

The following code shows an example of this issue, where a farmer could manipulate the amount of funds:

```
(mod (
        TARGET_AMOUNT
        CASH_OUT_PUZZLE_HASH
        my_amount
        new_amount
        my_puzzlehash
     )

     ...

     ; main
     if (>= new_amount TARGET_AMOUNT)
        ...
        (recreate_self my_amount new_amount my_puzzlehash)
     )
)
```

Figure 2.1: An `example of vulnerable code from the PiggyBank that do not assert the that the amount should always increase`

Exploit Scenario

Alice writes some puzzle code, but she forgets to enforce funds amounts. The chialisp compiler will not warn about it, allowing Alice to deploy vulnerable code.

Recommendations

Short term, consider having a dedicated field in the transaction specify the fee for the farmer and return to the transaction sender any unused funds. Additionally, consider the recommendation from TOB-Chia-001 to automatically flag during compilation if the user forgets to enforce this type of constraints.

Long term, review the participant incentives to make sure they do not conflict with each other.

3. Chialisp development can be error-prone

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-Chia-003
Target: `chia-blockchain`	

Description

The lack of some high-level syntactic commands and the weak types in Chialisp make the language error prone to use.

Chialisp is the official programming language of the Chia blockchain. It is heavily based on Lisp, adding their own primitives. However, the language lacks certain features which could make developing and reviews more reliable.

For instance, the language lacks a let keyword to define local variables. The following code repeats the access to the elements of `inner_conditions`:

```
      (defun find_and_strip_tail_info (inner_conditions cat_mod_struct
tail_reveal_and_solution)
        (if inner_conditions
            (if (= (output_value_for_condition (f inner_conditions)) -113)  ; Checks this is
a CREATE_COIN of value -113
                (find_and_strip_tail_info
                  (r inner_conditions)
                  cat_mod_struct
                  (c (condition_tail_reveal (f inner_conditions)) (condition_tail_solution
(f inner_conditions)))
                )
```

Figure 3.1: An example of code from the CAT standard that can be more readable using a let statement

A proposal to reduce the mental load to develop and review this code will be to use a `let` statement:

```
      (defun find_and_strip_tail_info (inner_conditions cat_mod_struct
tail_reveal_and_solution)
        (if inner_conditions
            let (finner, rinner) = (f inner_conditions, r inner_conditions)
            (if (= (output_value_for_condition finner) -113)  ; Checks this is a CREATE_COIN
of value -113
                (find_and_strip_tail_info
                  rinner
                  cat_mod_struct
                  (c (condition_tail_reveal finner) (condition_tail_solution finner))
                )
```

Figure 3.1: An example of code from the CAT standard that can be more readable using a let statement

We found at least one proposal to improve the language with this feature.

Additionally, a stronger type system could make the language less error prone. For instance, one mistake in a currified variable could result in unexpected code if the ONE definition is not imported from `curry-and-treehash.clinc`:

```
(defun sha256tree1 (TREE)
           (if (l TREE)
               (sha256 2 (sha256tree1 (f TREE)) (sha256tree1 (r TREE)))
               (sha256 ONE TREE)))
```

Figure 3.2: An example of code where a currified value is misinterpreted

Furthermore, distinguishing which expressions are eagerly evaluated from lazily evaluated makes code more difficult to follow. It would benefit developers if the language made this behavior more explicit.

```
    (defun stager (
         cat_mod_struct
         inner_conditions
         lineage_proof
         inner_puzzle_hash
         my_id
         prev_coin_id
         this_coin_info
         next_coin_proof
         prev_subtotal
         extra_delta
     )
       /* snip */
         (find_and_strip_tail_info inner_conditions cat_mod_struct ())
       /* snip */
     )

     (stager
         ;; calculate cat_mod_struct, inner_puzzle_hash, coin_id
         (list MOD_HASH (sha256 ONE MOD_HASH) TAIL_PROGRAM_HASH)
         (a INNER_PUZZLE inner_puzzle_solution)
         lineage_proof
```

```
        (sha256tree1 INNER_PUZZLE)
        (sha256 (f this_coin_info) (f (r this_coin_info)) (f (r (r
this_coin_info))))
        prev_coin_id    ; ID
        this_coin_info  ; (parent_id puzzle_hash amount)
        next_coin_proof ; (parent_id innerpuzhash amount)
        prev_subtotal
        extra_delta
    )
)
```

Figure 3.3: An `example of evaluation precedence that is ambiguous`

Finally, the interpretation of hex values can be difficult to use:

```
( + 0xf 1)  ; evaluates to 16: 0xf is interpreted as 15
( + 0xff 1) ; evaluates to 0: 0xff is interpreted as -1
( + q 1)    ; evaluates to 2: q is interpreted as 1
```

Figure 3.4: An `example of statements that can be confusing`

Exploit Scenario

Alice writes some puzzle code, but she misspelled or confused some statements. The chialisp compiler will not warn about it, allowing Alice to deploy vulnerable code.

Recommendations

Short term, consider adding the syntactic sugar and enforcing types in a more strict approach.

Long term, review the features and design decisions of ChiaLisp to make sure they are not error-prone.

4. sha256 is prone to collisions

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-Chia-004
Target: `chia-blockchain, cat.clvm`	

Description

The `sha256` operation concatenates and hashes its arguments, which can generate collisions if the values have variable length.

Chia smart coins execute code using puzzles. Instead of including the puzzle source code in the smart coin, it is hashed using the sha256 algorithm. Chia gives the owner of the coin the responsibility to provide both the source that matches the hash and its input (also called "solution").

However, the approach to compute the sha256 hashes is prone to collisions, since the bytes are internally concatenated before hashing:

```
(sha256 "clvm")
   => 0xcf3eafb281c0e0e49e19c18b06939a6f7f128595289b08f60c68cef7c0e00b81
(sha256 "cl" "vm")
   => 0xcf3eafb281c0e0e49e19c18b06939a6f7f128595289b08f60c68cef7c0e00b81
```

Figure 4.1: An example of hash collision using sha256

The CAT standard is affected by sha256 concat collisions, for instance, when computing the result of the is_parent_cat, the user controls all the elements in lineage_proof:

```
      (defun-inline is_parent_cat (
        cat_mod_struct
        parent_id
        lineage_proof
      )
        (= parent_id
           (sha256 (f lineage_proof)
                   (cat_puzzle_hash cat_mod_struct (f (r lineage_proof)))
                   (f (r (r lineage_proof)))
           )
        )
      )
```

Figure 4.2: A hash collision using sha256 in the CAT standard

Exploit Scenario

Alice writes code for a coin that enforces the usage of a certain puzzle hash (e.g an inner puzzle). This hash is computed using `sha256` of several elements. Alice transfers the coin to Eve. Eve finds a way to split the elements used for the hash to actually use a part of the program instead of the complete one. As a result, Eve is able to bypass some of the Alice code.

Recommendations

Short term, consider implementing an opcode (or a library function) that computes the hash of a sequence of arguments without collisions produced by concatenations.

Long term, review how other blockchain programming languages implement hashing (e.g. Solidity recommends to use `encode` before hashing).

5. CAT1 inputs structures are not verified

Severity: **Medium**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-Chia-005
Target: `cat.clvm`	

Description

Some of the required inputs to start a CAT1 transaction are structured fields that are not property validated and can lead to unexpected interactions.

The `lineage_proof` is used to check if the parent coin is a CAT. The documentation states that it consists of three elements:

```
     ;;;;;;;;;;;;;;;;;;;;;;;;;;; lineage checking

     ;; return true iff parent of `this_coin_info` is provably a cat
     ;; A 'lineage proof' consists of (parent_parent_id parent_INNER_puzzle_hash
parent_amount)
     ;; We use this information to construct a coin who's puzzle has been wrapped in this
MOD and verify that,
     ;; once wrapped, it matches our parent coin's ID.
     (defun-inline is_parent_cat (
       cat_mod_struct
       parent_id
       lineage_proof
     )
       (= parent_id
          (sha256 (f lineage_proof)
                  (cat_puzzle_hash cat_mod_struct (f (r lineage_proof)))
                  (f (r (r lineage_proof)))
          )
       )
     )
```

Figure 5.2: Usage of the `lineage_proof` in CAT1

It is also directly provided to the tail program:

```
               (a  (f tail_reveal_and_solution)
                   (list
                     Truths
                     parent_is_cat
                     lineage_proof ; Lineage proof is only guaranteed to be true if
parent_is_cat
                     extra_delta
                     inner_conditions
                     (r tail_reveal_and_solution)
                   )
```

Figure 5.3: `Call to the tail program in CAT1`

However, it is never checked for size. Thus, the lineage checking could succeed despite containing extra elements, though it is unclear what could happen during execution of the tail puzzle.

Finally, the lack of checks in the structure of the input parameters implies the existence of an infinite number of puzzle solutions (or hashes), which may be an undesirable property, depending on the use case of the token.

Other CAT1 inputs which are not structurally checked are `this_coin_info` and `next_coin_proof.`

Exploit Scenario

Alice writes code for a CAT that contains a Tail, which enforces that lineage proofs have exactly three arguments. Bob obtains Alice coins and tries to transfer providing a lineage proof, which is larger than expected, blocking the transaction to be performed.

Recommendations

Short term, consider validating the structure and size of every CAT input and document the expected properties.

Long term, consider implementing a stricter type system and validating arguments at runtime.

6. sha256tree1 can be re-implemented incorrectly

Severity: **Medium**	Difficulty: **High**
Type: Patching	Finding ID: TOB-Chia-006
Target: `TAIL programs`	

Description

An essential function to hash a chialisp program is not part of any library and can be incorrectly implemented by users, potentially introducing a security issue.

The CAT1 standard relies on the `sha256tree1` function in order to correctly hash all the elements in a chialisp program:

```
    (defun sha256tree1 (TREE)
          (if (l TREE)
              (sha256 2 (sha256tree1 (f TREE)) (sha256tree1 (r TREE)))
              (sha256 ONE TREE)))
```

Figure 6.1: `Correct implementation of sha256tree1`

While this function is implemented correctly in the documentation, it is not part of any library, relying on users to copy it or reimplement it. If users are not careful, they can introduce a non-trivial collision issue:

```
    (defun sha256tree1 (TREE)
          (if (l TREE)
              (sha256 (sha256tree1 (f TREE)) (sha256tree1 (r TREE)))
              (sha256 TREE)))
```

Figure 6.2: `Incorrect implementation of sha256tree1`

The user omitted the constants one and two. If the pre-image of the hash of internal leaves is known, then a malicious user can provide a list of bytes that will be hashed to obtain the same value. This can allow bypassing checks inside the TAIL programs, if it is used.

Exploit Scenario

Alice writes code for a CAT that contains a TAIL, which requires the usage of `sha256tree1`. Since this function is not provided, she writes her own implementation, introducing a collision bug. Eve uses this issue to force Alice code to behave in an unexpected way.

Recommendations
Short term, consider providing `sha256tree1` as part of the standard library or even as an opcode.

Long term, review the standard library in the ChiaLisp to make sure users have the code they need to keep their coin safe.

Summary of Recommendations

ChiaLisp and the CAT1 standard are a work in progress with a future iteration planned. Trail of Bits recommends that Chia address the findings detailed in this report and take the following additional steps prior to deployment:

- Make sure the CAT standard properly validates inputs using hash without concat collisions. This will avoid malicious user breaking important code invariants (TOB-Chia-004)
- Review the feature and design decision in ChiaLisp, considering:
    - Detection of common security issues by the compiler. This will allow users to catch security or correctness issues during the development process. (TOB-Chia-001)
    - Stronger type system. This will allow users to catch security or correctness issues during the development process. (TOB-Chia-003, TOB-Chia-005)
    - Expanded standard library. This will make sure users have the code they need to keep their coin safe. (TOB-Chia-006)

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels

Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels

Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Fuzzer-based test cases

Trail of Bits has included a fuzz test that uses `afl.rs`, an in-process, coverage-guided, evolutionary fuzzing engine based on AFL. The test covers the parsing and validation of conditions. Figure B.1 shows part of the `afl.rs` test:

```
fn main() {
    afl::fuzz!(|data: &[u8]| {
        if data.len() < 4 {
            return;
        }
        let (f, d) = data.split_at(4);
        let mut allocator = Allocator::new();
        let _program = match node_from_bytes(&mut allocator, d) {
            Err(_) => { return; },
            Ok(r) => test_cb(as_u32_be(f.try_into().expect("slice with incorrect length")),
r, allocator),
        };
    });
}
…
fn test_cb(
    f: u32,
    n: NodePtr,
    a: Allocator,
) -> () {
    match parse_spends(&a, n, 11000000000, f) {
        Ok(list) => {
            for s in &list.spends {
                println!("output: {:?}", s);
            }
        }
        Err(e) => { println!("error: {:?}", e); },
    }
}
```

Figure C.1: Part of the test that parses arbitrary spend outputs.

The following `cargo` invocation is needed for building the fuzzing test:

```
$ cargo afl build
```

After the build finishes, the fuzzer can be executed using the `target/debug/fuzz_conditions` binary. For instance, to run the afl.rs using an initial corpus in the `corpus` directory and save the results in `out`:

```
$ cargo afl fuzz -i corpus -o out target/debug/fuzz_conditions
```

Check the `afl.rs` documentation on how to adjust its parameters.

Investigating non-deterministic executions

From time to time, AFL.rs re-executes every input, without any mutation. If the coverage changes from each execution, then it flags the input as "unstable". Once the fuzzing campaign started, we noticed the instability of a number of inputs. In general, instability in a program execution means that there is some non-deterministic code. Since the code under review was part of the consensus algorithm in Chia, it must be investigated to make sure it will not impact the security of the chain.

We created a patch to replace the non-deterministic use of `HashSet` by `BTreeSet` in chia_rs, in order to determinate if this was the source of the detected instability

```
diff --git a/chia_rs/src/gen/conditions.rs b/chia_rs/src/gen/conditions.rs
index ee793ca..4a74810 100644
--- a/chia_rs/src/gen/conditions.rs
+++ b/chia_rs/src/gen/conditions.rs
@@ -21,7 +21,7 @@ use clvmr::cost::Cost;
 use clvmr::op_utils::u64_from_bytes;
 use clvmr::sha2::Sha256;
 use std::cmp::max;
-use std::collections::HashSet;
+use std::collections::BTreeSet;
 use std::hash::{Hash, Hasher};
 use std::sync::Arc;

@@ -309,6 +309,8 @@ fn parse_args(
 }

 #[derive(Debug)]
+#[derive(Ord)]
```

```
+#[derive(PartialOrd)]
 pub struct NewCoin {
     pub puzzle_hash: Vec<u8>,
     pub amount: u64,
@@ -347,7 +349,7 @@ pub struct Spend {
     pub height_relative: Option<u32>,
     pub seconds_relative: u64,
     // all coins created by this spend. Duplicates are consensus failures
-    pub create_coin: HashSet<NewCoin>,
+    pub create_coin: BTreeSet<NewCoin>,
     // Agg Sig Me conditions
     pub agg_sig_me: Vec<(NodePtr, NodePtr)>,
 }
@@ -379,18 +381,18 @@ pub struct SpendBundleConditions {
 struct ParseState {
     // hashing of the announcements is deferred until parsing is complete. This
     // means less work up-front, in case parsing/validation fails
-    announce_coin: HashSet<(Arc<[u8; 32]>, NodePtr)>,
-    announce_puzzle: HashSet<(NodePtr, NodePtr)>,
+    announce_coin: BTreeSet<(Arc<[u8; 32]>, NodePtr)>,
+    announce_puzzle: BTreeSet<(NodePtr, NodePtr)>,

     // the assert announcements are checked once everything has been parsed and
     // validated.
-    assert_coin: HashSet<NodePtr>,
-    assert_puzzle: HashSet<NodePtr>,
+    assert_coin: BTreeSet<NodePtr>,
+    assert_puzzle: BTreeSet<NodePtr>,

     // all coin IDs that have been spent so far. When we parse a spend we also
     // compute the coin ID, and stick it in this set. It's reference counted
     // since it may also be referenced by announcements
-    spent_coins: HashSet<Arc<[u8; 32]>>,
+    spent_coins: BTreeSet<Arc<[u8; 32]>>,

     // this object tracks which ranges of the heap we've scanned for zeros, to
     // avoid scanning the same ranges multiple times.
@@ -400,11 +402,11 @@ struct ParseState {
 impl ParseState {
     fn new() -> ParseState {
         ParseState {
-            announce_coin: HashSet::new(),
-             announce_puzzle: HashSet::new(),
-             assert_coin: HashSet::new(),
-             assert_puzzle: HashSet::new(),
-             spent_coins: HashSet::new(),
+             announce_coin: BTreeSet::new(),
+             announce_puzzle: BTreeSet::new(),
+             assert_coin: BTreeSet::new(),
+             assert_puzzle: BTreeSet::new(),
+             spent_coins: BTreeSet::new(),
              range_cache: RangeSet::new(),
          }
      }
@@ -446,7 +448,7 @@ fn parse_spend_conditions(
          puzzle_hash,
          height_relative: None,
          seconds_relative: 0,
-         create_coin: HashSet::new(),
+         create_coin: BTreeSet::new(),
          agg_sig_me: Vec::new(),
      };

@@ -600,7 +602,7 @@ pub fn parse_spends(
      // check all the assert announcements
      // if there are no asserts, there is no need to hash all the announcements
      if !state.assert_coin.is_empty() {
-         let mut announcements = HashSet::<[u8; 32]>::new();
+         let mut announcements = BTreeSet::<[u8; 32]>::new();

          for (coin_id, announce) in state.announce_coin {
              let mut hasher = Sha256::new();
@@ -620,7 +622,7 @@ pub fn parse_spends(
      }

      if !state.assert_puzzle.is_empty() {
-         let mut announcements = HashSet::<[u8; 32]>::new();
+         let mut announcements = BTreeSet::<[u8; 32]>::new();

          for (puzzle_hash, announce) in state.announce_puzzle {
              let mut hasher = Sha256::new();
```

Figure C.2: Code modifications to minimize code instability

Fortunately, this reduced instability almost to 0%. The remaining unstable code was caused by AFL.rs when hooking the panic events, so it cannot be reduced further.

Measuring coverage

Regardless of how inputs are generated, it's important to measure a fuzzing campaign's coverage after its run. To perform this measure, we used the support provided by [Rust source-based code coverage feature](). This instrumentation is compatible with the AFL's one, so the same source code can be used. In order to compute the coverage report for an `input` file, it is necessary to run the following commands:

```
$ LLVM_PROFILE_FILE="input.profraw" ./target/debug/fuzz_conditions < input
$ llvm-profdata merge -sparse input.profraw -o cov.profdata
$ llvm-cov show ./target/debug/fuzz_conditions -instr-profile=cov.profdata >
report.txt
```

It is worth mentioning that the llvm-profdata and the llvm-cov used should be ones provided by the rust toolkit (usually located inside the toolchains files). Other versions of the llvm tools will fail to run.

```
  207|       |        ASSERT_PUZZLE_ANNOUNCEMENT => {
  208|    428|            let id = sanitize_hash(
  209|    433|                a,
  210|    433|                first(a, c)?,
  211|       |                32,
  212|    432|                ErrorCode::AssertPuzzleAnnouncementFailed,
  213|      4|            )?;
  214|    428|            if (flags & STRICT_ARGS_COUNT) != 0 {
  215|     12|                check_nil(a, rest(a, c)?)?;
  216|    416|            }
  217|    427|            Ok(Condition::AssertPuzzleAnnouncement(id))
  218|       |        }
```

Figure C.3: Part of the coverage generated after a fuzzing campaign.

Integrating fuzzing and coverage measurement into the development cycle

Once the fuzzing procedure has been tuned to be fast and efficient, it should be properly integrated in the development cycle to catch bugs. We recommend the following procedure to integrate fuzzing using a CI system:

1. After the initial fuzzing campaign, save the corpus generated by every test. We provide these initial corpora.
2. For every internal milestone, new feature, or public release, re-run the fuzzing campaign starting with each test's current corpora for at least 24 hours.[1]
3. Update the corpora with the new inputs generated.

Note that, over time, the corpora will come to represent thousands of CPU hours of refinement, and will be very valuable for guiding efficient code coverage during fuzz testing. An attacker could also use them to quickly identify vulnerable code, so avoid this additional risk by keeping fuzzing corpora in an access-controlled storage location rather than a public repository. Some CI systems allow maintainers to keep a cache to accelerate building and testing. The corpora could be included in such a cache, if they are not very large.

Instead of performing the fuzzing in-house, you can use the `oss-fuzz` project, which lets you perform automatic fuzzing campaigns with Google's extensive testing infrastructure. `oss-fuzz` is free for widely used open-source software. They may accept Chia as another project.

On the plus side, Google provides all their infrastructure for free and will notify project maintainers any time a change in the source code introduces a new issue. The received reports include essential important information such as minimized test cases and backtraces.

However, there are some downsides: If `oss-fuzz` discovers critical issues, Google employees will be the first to know, even before the Chia project's own developers. Google policy also requires the bug report to be made public after 90 days, which may or may not be in the best interests of Chia. Weigh these benefits and risks when deciding whether to request Google's free fuzzing resources.

[1] For more on fuzz-driven development, see this CppCon 2017 talk by Kostya Serebryany of Google.

D. Incident response plan recommendations

In this appendix, we provide recommendations around the formulation of an incident response plan.

- Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).
    - Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.
- Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.
    - Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.
- Clearly describe the intended process of code deployment.
- Consider whether and under what circumstances your company will make affected users whole after certain issues occur.
    - Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.
- Document how you plan to keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.
    - For each language and component, describe the noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord/Slack channel with a bot that will post the latest vulnerability news; this will help the team keep track of updates all in one place. Also consider assigning specific team members to keep track of the vulnerability news of a specific component of the system.
- Consider scenarios involving issues that would indirectly affect the system.
- Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.).
    - Some issues may require collaboration with external parties to efficiently remediate them.
- Define contract behavior that is considered abnormal for off-chain monitoring.
    - Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.
- Combine issues and determine whether new detection and mitigation scenarios are needed.
- Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.
    - Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry

runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

E. Static Analysis for ChiaLisp

The ChiaLisp language has a weak type system and several footguns. In order to improve the security of ChiaLisp puzzles, we recommend statically analyzing code at compile-time and emitting warnings for suspect code and erroring out for unsafe code. The following are a sample of analyses and checks that would be valuable:

- **Warn if input arguments can be manipulated by farmers.** When insufficient validation is performed, node operators can replace function inputs and transfer coins to themselves. If function inputs are not signed and do not appear in assert conditions, the compiler should warn developers.
- **Require explicit type conversions.** If an identifier is used but not declared, the string will be silently coerced into bytes. This introduces undefined behavior into ChiaLisp puzzles when, for example, a constant is not imported from a library but is used. Instead, throw an error when strings are used as they are likely undeclared identifiers. If the value should be treated as bytes, require explicit type casting.
- **Warn if a puzzle's condition output is empty.** While it is not certainly a vulnerability, it is uncommon for a puzzle to not output conditions e.g. validating user inputs. Giving developer's feedback that output conditions are likely important to include may help prevent bugs.
- **Validate structure and size of input.** It is common to retrieve elements at specific indices of lists and perform some operation on each element. However, function inputs are not guaranteed to conform to the anticipated length, and this could lead to unexpected behavior. Consider using annotations to validate function inputs by implementing a stricter type system that would reduce the likelihood of bugs. Additionally, consider checking structure automatically on output conditions to make sure they will be correctly parsed by the nodes.
- **Warn if hash collisions are possible.** If dynamic length arguments controlled by the user are hashed, recommend prepending magic values or enforcing strict length.

F. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- In the same manner that `ONE` is a constant for 1, create a constant `TWO` and replace the usage of 2 below:

```
   (defun sha256tree1 (TREE)
       (if (l TREE)
          (sha256 2 (sha256tree1 (f TREE)) (sha256tree1 (r TREE)))
          (sha256 ONE TREE)))
[...]
```

Figure F.1: `chia-blockchain/chia/wallet/puzzles/cat.clvm#L133-L136`

- If the unused parameter this_coin_info is not required, remove it from the function signature:

```
  (defun check_lineage_or_run_tail_program
    (
      this_coin_info
      tail_reveal_and_solution
      parent_is_cat  ; flag which says whether or not the parent CAT check ran and passed
      lineage_proof
      Truths
      extra_delta
      inner_conditions
    )
[...]
```

Figure F.2: `chia-blockchain/chia/wallet/puzzles/cat.clvm#L307-L316`