NU3 Specification and Blossom Implementation Audit

Zcash

February 6, 2020 – Version 1.1

Prepared for

Taylor Hornby

Benjamin Winston

Prepared by

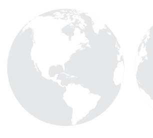
Gérald Doussot

Thomas Pornin

©2020 – NCC Group

Prepared by NCC Group Security Services, Inc. for Zcash. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.



Synopsis

In October 2019, the Electric Coin Company engaged NCC Group to conduct a review of two Zcash improvement proposals (ZIP 213 and ZIP 221) and of the implementation of ZIP 208 within the Zcash node implementation. ZIP 213 proposes a change to consensus rules to allow coinbase transactions to target shielded addresses. ZIP 221 describes a novel type of hash trees meant to support efficient validation of transactions by lightweight clients. The implementation of ZIP 208 applies a change of target pacing for block issuance, so that overall network latency is halved. Two consultants performed the engagement for a total of 15 person-days.

Scope

NCC Group's evaluation included:

- The ZIP 213 draft, as of commit `90ad18f`.[1]
- The ZIP 221 draft, as of commit `b6be377`.[2]
- The implementation of ZIP 208, as made in pull request 4025.[3] PR 4025 was merged into the main Zcash tree on August 9, 2019; the scope covered that set of commits, last of which was `b99003c` on August 7.

Limitations

ZIP 213 and 221, being draft specifications, do not have corresponding implementations. Therefore, analysis of potential security issues was based on how such implementations could plausibly be built. In particular, the assessment of the severity of the two issues found in the pseudocode of ZIP 221 was based on the estimated likelihood that they would result in flaws in the implementation that may fail to be detected by basic unit testing. Any specific future implementation may have different characteristics.

Key Findings

In ZIP 213, no issues were found. The description of the change appears to be fully correct and in line with its purported goals and rationale. Notably, the analysis found in the draft regarding coinbase transaction maturity clearly explains how anchors in shielded transactions act as an enforced dependency system on whole sequences of transactions, contrary to transparent transactions that are more free-standing and can be inserted in blocks in any order as long as no balance goes negative. This dependency mechanism justifies the removal of the coinbase maturity rule for shielded outputs.

In ZIP 221, two issues were found in the pseudocode for the operations on the hash tree; they are described in finding NCC-1908_Zcash-001 on page 5 and finding NCC-1908_Zcash-002 on page 6. These issues lead to wrong computations that might go unnoticed during development, since all implementations following that pseudocode would behave identically to each other, but would still exclude some block headers from the hash tree. A number of smaller issues and remarks were assembled in Appendix B on page 12; most of these are concerning consistency of the description or typography, and NCC Group deemed them unlikely to lead to any vulnerability in future implementations.

In PR 4025, no serious issues were uncovered. A very minor imperfection in the validation of a command-line parameter was detected, and is described in finding NCC-1908_Zcash-003 on page 8; it has no real consequence on security.

Strategic Recommendations

Since ZIP 221 pseudocode follows a Python syntax, NCC Group recommends converting that pseudocode into runnable Python code, so that extensive unit tests can be made, in particular with respect to edge cases (adding one or several nodes to an empty tree, removing all nodes from a tree, removing an exact full subtree, and so on).

[1] https://github.com/str4d/zips/blob/90ad18ff228e86830da965a5180cfc6e2acac520/zip-0213.rst
[2] https://github.com/therealyingtong/zips/blob/b6be3770c9cbd340b57d6dd6786d43ddf0610189/zip-0221.rst
[3] https://github.com/zcash/zcash/pull/4025

Dashboard

Target Metadata

Name	Zcash Protocol and Implementation

Engagement Data

Type	Specification and Implementation Review
Method	Code-assisted
Dates	2019-10-21 to 2019-10-31
Consultants	2
Level of Effort	15 person-days

Finding Breakdown

Critical issues	0
High issues	2
Medium issues	0
Low issues	0
Informational issues	1
Total issues	**3**

Category Breakdown

Cryptography	2
Data Validation	1

Component Breakdown

ZIP 208 implementation	1
ZIP 221	2

Key

Critical High Medium Low Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 10.

Title	ID	Risk
Missing Last Processed Peak in `append()`	001	High
Wrong List of Peaks Returned by `get_peaks()`	002	High
Unchecked Possible Truncation	003	Informational

Finding Details

Finding **Missing Last Processed Peak in** `append()`

Risk **High** Impact: High, Exploitability: Undetermined

Identifier NCC-1908_Zcash-001

Category Cryptography

Component ZIP 221

Location Function `append()` in the ZIP 221 pseudocode.

Impact The specification may lead to an incorrect implementation of ZIP 221 where the computed MMR tree is incomplete, i.e. missing block header information. Simplified Payment Verification (SPV) clients use this information to validate the blockchain and the inclusion of a transaction in a block. If this information is incomplete, clients cannot determine accurately what amount of currency they own and can spend.

Description The ZIP 221 specification documents modifications to the Zcash block header format to include Merkle Mountain Range (MMR) commitments. The specification includes pseudocode for the algorithm of several routines to manipulate the MMR tree data structure. Specifically, the `append()` function permits to merge a leaf node to an existing MMR tree.

The `append()` function uses the `current` variable to operate iteratively in a loop on the peak being processed. Its content will be used in the next iteration of the loop. When the loop exits, the content of `current` is not added to the list of peaks to be bagged in `bag_peaks()`, therefore the resulting MMR tree will miss this final peak.

The issue is illustrated in the code snippet below:

```
def append(root: ZcashMMRNode, leaf: ZcashMMRNode) -> ZcashMMRNode:
    '''Append a leaf to an existing tree, return the new tree root'''
    # recursively find a list of peaks in the current tree
    peaks: List[ZcashMMRNode] = get_peaks(root)
    merged: List[ZcashMMRNode] = []

    # Merge peaks from right to left.
    # This will produce a list of peaks in reverse order
    current = leaf
    for peak in peaks[::-1]:
        current_leaves = current.latest_height - current.earliest_height + 1
        peak_leaves = peak.latest_height - peak.earliest_height + 1

        if current_leaves == peak_leaves:
            current = make_parent(peak, current)
        else:
            merged.append(current)
            current = peak

    # finally, bag the merged peaks
    return bag_peaks(merged[::-1])
```

Recommendation Ensure that all peaks are bagged. This can be achieved by adding the following statement before the last statement of the `append()` function:

```
merged.append(current)
```

Finding **Wrong List of Peaks Returned by `get_peaks()`**

Risk **High** Impact: High, Exploitability: Undetermined

Identifier NCC-1908_Zcash-002

Category Cryptography

Component ZIP 221

Location Function `get_peaks()` in ZIP 221 pseudocode.

Impact The `get_peaks()` function may return the wrong list of peaks, leading to a wrong MMR computation that could omit some block headers. If this information is incomplete, clients cannot determine accurately what amount of currency they own and can spend.

Description The `get_peaks()` function takes as input the root for the MMR, and extracts the peaks. It is the reverse operation of the `bag_peaks()` function. It is expressed recursively:

```
def get_peaks(node: ZcashMMRNode) -> List[ZcashMMRNode]:
    peaks: List[ZcashMMRNode] = []

    left_child = node.left_child
    right_child = node.right_child

    # find the number of leaves in the subtree
    left_leaves = left_child.latest_height - left_child.earliest_height + 1
    right_leaves = right_child.latest_height - right_child.earliest_height + 1

    if (left_leaves & (left_leaves - 1)) == 0:
        peaks.append(left_child)
    else:
        peaks.extend(get_peaks(left_child))

    if (right_leaves & (right_leaves - 1)) == 0:
        peaks.append(right_child)
    else:
        peaks.extend(get_peaks(right_child))

    return peaks
```

The principle is the following: a subtree with a number of leaves equal to 2^t for some integer t is complete, and thus should be a peak. The `get_peaks()` function starts from the root and explores down the tree to locate such subtrees.

However, there is one case that the function above does not cover properly: when the full tree is itself a single peak. If `get_peaks()` is invoked on the root of a tree with 2^t leaves, it will return *two* peaks, for the left and right children, respectively. However, it should return a single peak in that case. If `get_peaks()` returns two peaks instead of one, then `append()` will lead to a corrupted structure and compute the wrong root hash value, and possibly drop nodes from the tree.

An additional symptom of this issue is the fact that `get_peaks()` cannot work on a leaf node. If the MMR contains a single leaf, which is then the root, then `get_peaks()` starts by following null pointers to its inexistent left and right children, leading to adverse outcomes (segmentation fault, null pointer exception,... depending on the implementation language). The core

	conceptual issue is that `get_peaks()` assumes that it is called on an internal non-peak node, and expects its caller not to invoke it on a full tree or a single leaf node.
Recommendation	The following `get_peaks()` pseudocode would avoid the issue and work on all possible MMR roots, including single leaves:

```
def get_peaks(node: ZcashMMRNode) -> List[ZcashMMRNode]:
    peaks: List[ZcashMMRNode] = []

    # Get number of leaves.
    leaves = latest_height - earliest_height + 1

    # Check if the number of leaves is a power of two.
    if (leaves & (leaves - 1)) == 0:
        # Tree is full, hence a single peak. This also covers the
        # case of a single isolated leaf.
        peaks.append(node)
    else:
        # If the number of leaves is not a power of two, then this
        # node must be internal, and cannot be a peak.
        peaks.extend(get_peaks(left_child))
        peaks.extend(get_peaks(right_child))

    return peaks
```

Finding	**Unchecked Possible Truncation**
Risk	**Informational** Impact: Low, Exploitability: None
Identifier	NCC-1908_Zcash-003
Category	Data Validation
Component	ZIP 208 implementation
Location	`src/init.cpp`, lines 1068-1075
Impact	Despite explicit checks to the contrary, it is feasible to use the `-txexpirydelta` command-line parameter to set the lifetime of newly created transactions to a value lower than the expected minimum of four blocks.

Description

The `-txexpirydelta` command-line parameter is parsed into an integer and stored in a global variable through the following code:

```
    if (mapArgs.count("-txexpirydelta")) {
        int64_t expiryDelta = atoi64(mapArgs["-txexpirydelta"]);
        uint32_t minExpiryDelta = TX_EXPIRING_SOON_THRESHOLD + 1;
        if (expiryDelta < minExpiryDelta) {
            return InitError(strprintf(_("Invalid value for -txexpirydelta='%u' (
➜   must be least %u)"), expiryDelta, minExpiryDelta));
        }
        expiryDeltaArg = expiryDelta;
    }
```

The `atoi64()` function parses the argument string as a signed 64-bit integer, and therefore can yield any value in the -2^{63} to $2^{63}-1$ range. The test is against the value `minExpiryDelta`, which is 4 (`TX_EXPIRING_SOON_THRESHOLD` is defined in `main.h` with value 3), and aborts node startup if the value is too small.

However, the parsed value is then written into `expiryDeltaArg`, a global variable that has type `unsigned int` (wrapped into a Boost `optional` construction), as declared in `main.h`:

```
extern boost::optional<unsigned int> expiryDeltaArg;
```

On typical systems, `unsigned int` has size 32 bits. Therefore, the `int64_t` value is implicitly truncated (reduced modulo 2^{32}, as per the C and C++ rules). In particular, if the input argument string is "4294967298", then `expiryDelta` will successfully pass the test above (the value is $2^{32}+2$, which is way above 4), but the final value written in `expiryDeltaArg` will be 2, i.e. below the minimum.

Since this code is merely validation of a command-line launch parameter provided by the node owner, this finding does not have any practical security impact.

Recommendation

The test may be modified into the following:

```
    if (mapArgs.count("-txexpirydelta")) {
        int64_t expiryDelta = atoi64(mapArgs["-txexpirydelta"]);
        uint32_t minExpiryDelta = TX_EXPIRING_SOON_THRESHOLD + 1;
        uint32_t maxExpiryDelta = TX_EXPIRY_HEIGHT_THRESHOLD - 1;
```

```
        if (expiryDelta < minExpiryDelta || expiryDelta > maxExpiryDelta) {
            return InitError(strprintf(_("Invalid value for -txexpirydelta='%u' (
➜   must be least %u, at most %u)"), expiryDelta, minExpiryDelta, maxExpiryDelta)
➜   );
        }
        expiryDeltaArg = expiryDelta;
    }
```

The `TX_EXPIRY_HEIGHT_THRESHOLD` is defined in the consensus parameters (`src/consensus/consensus.h`) with the value $500{,}000{,}000$; since transactions with an expiry height beyond that value are unconditionally rejected by nodes, the `expiryDelta` parameter cannot logically have any value larger than that.

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: ZIP 221 Audit Notes

This section includes NCC Group's remarks about ZIP 221.[4] These remarks include typographical errors, and other comments on the text that did not meet NCC Group's standard of a security finding but are still worth mentioning. Each remark corresponds to a passage in the text; we list them below in appearance order.

In section "Terminology":

- "A Merkle mountain range (MMR) is binary hash tree" This should be: "... is a binary hash tree".

In section "Background":

- The background description explains tree construction in terms of order of insertion of nodes, assuming that inserted nodes can be leaves or internal nodes. However, this does not match usage of MMR for block headers, as described later on in ZIP 221: the block headers can only yield leaf nodes, and internal nodes are built afterwards. As such, the "insertion order" described in the background section tends to lead to confusion later on.
- The section that explains the computed position and height of a node based on its "insertion order" is valid only for MMR nodes before "bagging." The bagging process adds extra nodes to the tree that do not follow these rules.
- It is written that one can jump from a node to its right sibling by adding $2^{h+1} - 1$ to its position, and to its left sibling by subtracting 2^h. This seems dubious: the jump from left to right sibling, and the jump back from right to left sibling, should, by definition, be of the exact same amount.
- The use of "height" for the position of tree nodes is unfortunate, as the remainder of the document uses "height" in the sense of order of blocks in the blockchain. It would be clearer to speak of the "level" or "altitude" of nodes and peaks.
- "The MMR tree allow for ...". This should be: "The MMR trees allow for ..."
- "ZCash" should be spelled "Zcash". The miscapitalization occurs six times throughout ZIP 221; there is also a "zCash" in the list of references.

In section "Tree Node specification":

- In the list items 3, 4, 5 and 9, line breaks are missing, leading to a subclause appearing on the continuation of the previous line.
- List item 8 gives a formula that uses the `toTarget()` function, but that function is spelled `ToTarget()` in the Zcash specification[5] (section 7.6.4).
- In list item 8, no provision is made for integer overflows. It is extremely improbable that an overflow occurs: the computed work factors are, *on average*, equal to the computational efforts involved in the creation of the corresponding blocks, and an aggregate effort of 2^{256} or more is infeasible in practice. However, this deserves an explanatory note, e.g. to assert that computations modulo 2^{256} are fine here.

In section "Tree nodes and hashing (pseudocode)":

- The field names in the pseudocode do not match those given in the previous section. For instance, `hashSubtreeCommitment` becomes `subtree_commitment`. Though the specifications and pseudocode names are not hard to match to each other, it would be clearer and simpler if the exact same names were used.
- In the function `make_parent()`:
  - "`end_target=left_child.end_target`"; this should use `right_child.end_target`.

[4] https://github.com/therealyingtong/zips/blob/master/zip-0221.rst
[5] https://github.com/zcash/zips/raw/master/protocol/protocol.pdf

  - "`count_shielded_txs=left_child.count_shield + right_child.count_shield)`"; the left and right children do not have fields called `count_shield`, but `count_shielded_txs` (although, as explained previously, these should be named `nShieldedTxCount`, as in the previous section).

In section "Incremental push and pop (pseudocode)":

- The `append()` function calls `bag_peaks()`, but the latter is defined only in the next pseudocode blocks. The `bag_peaks()` function should be moved to this block.
- The text about block reorganizations explains that such an occurrence leads to removing some of the rightmost leaves from the MMR, an operation that can be done with cost $O(\log k)$, where k is the number of leaves in the right subtree of the MMR root. However, the pseudocode only shows how to remove a single leaf, with a function that fully un-bags and re-bags the tree, thus far from demonstrating this $O(\log k)$ performance.
- The `delete()` function uses an undefined variable called `tmp_root`; this appears to be a misnamed reference to the local variable `subtree_root`.

In section "Header modifications specification":

- In the second list item, `hashSaplingFinalRoot` should be `hashFinalSaplingRoot`.

In section "Rationale":

- The third paragraph ends abruptly with "we change the semantics of `hashSubtreeCommitment` in leaf nodes to commit." The end of the sentence appears to be missing.
- In "Non-FlyClient Commitments," item `hashLatestSaplingRoot`: "will descrie". This should be: "will describe".

In section "References":

- The fourth reference, to "ZCash reference light client protocol", is a broken link (or a link to a private document).