

Zendoo

Source Code Audit



Source Code Audit

Prepared for Horizen • June 2021

v210921

1. Executive Summary
2. Introduction
3. Scope
4. Assessment
 - 4.1 System design
 - 4.2 Threat scenarios
 - 4.3 Embedded crypto library interactions
5. Summary of Findings
6. Findings
 - ZOO-001 - Reachable assertion allows attackers to hijack the network
 - ZOO-002 - DoS attack by improper handling of compressed data
 - ZOO-003 - Malicious sidechains can block withdrawals to the mainchain
 - ZOO-004 - Sidechain certificates enable mainchain resource exhaustion attacks
 - ZOO-005 - Ceased sidechains enable mainchain resource exhaustion attacks
7. Appendix I
8. Disclaimer

1. Executive Summary

In May 2021, [Horizen](#) engaged [Coinspect](#) to audit the security of its Zendoo open sidechain platform. The objective of this audit was to evaluate the security of the framework and the Cross-Chain Transfer Protocol (CCTP) to identify vulnerabilities that might allow adversaries to take advantage of the mainchain and sidechains interactions.

The platform is composed of mainchain and sidechain modules. The focus of this audit were the mainchain modules, the security of the sidechains and their specific implementation were not evaluated during this audit.

The modifications introduced to the original Zcash codebase were found to be consistent with the existing security assumptions and defense mechanisms. The code was extensively documented with commentaries and several unit and integration tests are included in the repository.

Overall, Coinspect did not find any high risk security vulnerability that would directly result in stolen or lost user funds. However, some vulnerabilities are reported that can affect the mainchain availability (and all sidechains availability as a result) and could be abused by attackers to target specific nodes or mount network wide attacks that could weaken the integrity of the blockchain.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk	Zero Risk
5	0	0	0

Coinspect identified **four high risk issues** during the assessment. The root causes for these vulnerabilities can be categorized in two main groups:

1. Insufficient validation of untrusted data that can be exploited to crash the Zendoo node as a result of a panics in the Rust components.

2. Design issues related to the mainchain mempool transaction acceptance criteria in the context of malicious sidechains.

During August 2021, Coinspect verified these findings had been correctly addressed by the Horizen team and this report was updated to reflect those fixes.

During September 2021, Coinspect verified additional changes made by Horizen to the source code and this report was updated to reflect those changes in [Appendix I](#).

The present report details the tasks performed and the vulnerabilities found during this audit as well as several suggestions aimed at improving the overall code quality, and warnings regarding potential issues.

2. Introduction

Zendoo is a sidechain solution designed to enable blockchain scalability and extensibility by allowing users to create a parallel platform with a custom business logic bound to the Horizen public blockchain as the mainchain.

The audited version of the Zendoo client implements the Zendoo verifiable Cross-Chain Transfer Protocol (CCTP) that allows the creation of ad-hoc sidechains with customizable business rules. Users can transfer coins from the mainchain to the sidechains and back through cryptographically verifiable withdrawal certificates.

During this engagement, Coinspect consultants used a hands-on approach to evaluate the platform security, which included:

- Source code review of [Zendoo](#) client code
- Source code review of critical functionality used by the code in scope from the underlying cryptographic libraries
- Gray-box testing of selected functionality
- Rapid prototyping of potential attacks and proof of concept development

The primary objective of the assessment was to examine the changes made to the Zend client source code to identify and attempt to exploit security vulnerabilities that might allow adversaries to attack the mainchain or the sidechains and the funds secured by them.

3. Scope

The audit started on **May 19, 2021** with focus on the new code modifications introduced by Zendoo to incorporate the ability for the Horizen mainchain to interact with multiple sidechains. The security review was guided by the attack scenarios listed below in the assessment section, aiming at finding as many high risk security issues as possible in the assigned time, with code coverage not being the main priority.

The audit was conducted on the https://github.com/HorizenOfficial/zend_oo Github repository, as of commit `56b651f20780c1bbaa6b0b3ef5cbcdcc9b113bbf` on the `sidechains_integration_step2` branch:

```
commit 56b651f20780c1bbaa6b0b3ef5cbcdcc9b113bbf
(HEAD -> sidechains_integration_step2, origin/sidechains_integration_step2)
Merge: 529b26f54 b767c9a34
Author: albertog78 <34939252+albertog78@users.noreply.github.com>
Date:   Fri May 28 19:17:08 2021 +0200
```

Merge pull request #117 from HorizenOfficial/sidechains_integration_step3

Sidechains integration step temp PR

The `sidechains_integration_step2` branch included features merged from the following branches: `mbtr_introduction`, `ceased_sidechain_withdrawal`, `cert_custom_fields`, and `sc_commitment_tree_cumulative_hash`. These branches were selected by the client as the focus for this audit and introduced the following features to the Zend client:

1. Mainchain Backward Transfer Requests
2. Ceased Sidechain Withdrawals
3. Customizable Certificate Fields

4. Cumulative SCBlockTxCommTree

In a similar fashion, the following features implemented as Rust libraries embedded by the Zend client were targeted:

1. BitVector
2. ProofVerifier
3. SCTxCommTree

Additionally, critical functionality used by the code in scope was reviewed from the following libraries:

Repository	Branch	Commit / Date
zendoo-mc-cryptolib	sync_with_cctp_lib	ac1a8d59330953d9bfabf8c65b11b21bde6669f9 / May 28, 2021
zendoo-CCTP-lib	dev	f7aeeba5266a2a6d82e2186958d11ead165191ab / May 28, 2021
ginger-lib	development_tmp	b8b3a9feb8f1c4dde5ce3a3f2e951d597ec9d696 / May 28, 2021

During the audit, the [Zendoo's whitepaper](#) was utilized as a guide to the platform design and its expected behaviour:

[Zendoo-A_zk-SNARK-Verifiable-Cross-Chain-Transfer-Protocol.pdf](#)

All findings have been identified and reproduced with local builds of Zendoo client version **v2.1.0-beta4-56b651f20-dirty**.

The security audit was conducted on the most recent version of the platform, which is currently under development. New functionality is currently being implemented and some critical components are being modified, such as the SNARK proof verification system interface and the integration of asynchronous and batched

verification of proofs which are currently a work in progress. **It is recommended all these late additions to be reevaluated once completed.**

Neither the sidechain side libraries nor any specific sidechain implementation (such as Latus and its consensus protocol) were in scope for this audit.

4. Assessment

The source code reviewed was found to be clear and thoroughly documented with an abundant number of commentaries that make it easy to understand.

Zendoo introduced several modifications to the Zcash source code base, and these were found to be consistent with the overall design. Coinspect did not find any place where these changes affected the security assumptions nor the defense mechanisms in place.

The security patches from the upstream Zcash project have been backported to the Zendoo code repository. Specifically, the latest vulnerability identified with [CVE-2020-8806](#) (disclosed on Feb 2020) has been backported to Zendoo code at commit [ae5046b672523c3231504e23ab8c2dfed27cbdc4](#) (May 2020).

4.1 System design

This section briefly describes the overall system design, a more complete specification can be found in Zendoo's whitepaper.

The platform architecture is comprised of 2 main components:

1. The mainchain (MC from now on) Zendoo client
2. The sidechains (SC from now on)

These two components interact via the Zendoo's CCTP (Cross-Chain Transfer Protocol) which implements two main operations:

1. Forward transfers (FT): coins move from MC to SC
2. Backward transfers (BT): coins move back from SC to MC

The SCs must observe the MC, but the MC is agnostic about the SC implementation. Each SC is created with a sidechain creation transaction, which includes its own SNARK verification keys that are used to validate proofs included

in certificates submitted as special transactions to the mainchain. This keeps the mainchain totally agnostic from any sidechain rules and business logic.

Forward transfers are implemented as unspendable outputs, the SC observes these in the MC to mint coins on their side.

Backward transfer requests consist of sidechain generated proofs, which are batched in certificates and published to the mainchain. Certificate issuers are responsible for including the SC users backward transfers requests. These certificates are processed in withdrawal epochs whose duration is defined in the sidechain registration transaction.

Sidechains must send at least one certificate each epoch, or they are considered ceased. If more than one certificate from a SC is received for a certain epoch, the MC Zondoo client utilizes the `quality` parameter included in the certificate proof to determine which certificate will be applied for that epoch.

When a SC is ceased, users can withdraw their funds by submitting a CSW (Ceased Sidechain Withdrawal) transaction directly to the mainchain.

Another safeguard mechanism was incorporated to the design in order to protect users from a malicious majority in the sidechain from censoring their backward transfer requests in order to block them from recovering their funds. The MBTRs (Mainchain Backward Transfer Request) can be submitted by any user to the MC, which if the user ownership of the coins is verified can force the SC to include the corresponding BT in the next epoch certificate.

4.2 Threat scenarios

The threat scenarios and security mechanisms identified and evaluated by Coinspect experts included but were not limited to:

- Mainchain nodes attacks (e.g., denial of service)
- Malicious sidechains attacks to mainchain nodes (e.g., ill-behaved SNARK verification circuits)

- Broken consensus rules (e.g., chain split, malicious miner advantage)
- Double spend or create coins out of thin air in the mainchain or the sidechains
- Forward and backward transfers transaction validation rules
- Ceased Sidechain Withdrawals mechanism abuse (e.g., replaying withdrawal requests)
- Identification of possible malicious strategy to cease a sidechain (e.g. preventing certificates to be accepted by leveraging some bugs)
- Mainchain Backward Transfer Requests
- Identification of possible malicious strategy to delay block propagation by forging specific certificate elements (e.g. BitVector, proof data, etc) in order to increase block verification time
- Block processing performance impact of new features
- Certificate verification logic
- Sidechain registration processing
- Sidechain events handling
- Mainchain reorganization handling
- Coin maturity rules
- Sidechain balance accounting
- Modifications performed to the Zendo node mempool to accommodate sidechain related transactions and certificates
- Serialization and deserialization of data structures

During the audit Coinspect focused on several scenarios where a malicious sidechain was registered with the goal of attacking the MC Zendo client and the mainchain blockchain. Even when the MC node is designed to be agnostic of the SC operations, it is affected by several SC specific operations. In particular, many modifications had to be made to block processing and mempool acceptance criteria.

Coinspect auditors found design issues that enable adversaries to create malicious sidechains in order to spam the mainchain network for free by submitting as many SC related transactions as possible, which will be invalidated by a single transaction. This is because the SC related transactions mempool acceptance

criteria cannot guarantee the accepted transactions will be able to be executed in the near future.

In this respect, Coinspect found attackers can spam the network with sidechain transactions with no cost. This is detailed in [ZOO-004 Sidechain certificates enable mainchain resource exhaustion attacks](#).

It is worth noting that the MC can not trust the SC verification circuits to be correct or not malicious, so even when proofs are correct, special care must be taken when the processing of sidechain related transactions affects the mainchain Zendoo client state. An issue related to this was found that enables attackers to exhaust network resources by invalidating transactions in a ceased sidechain and is described in [ZOO-005 Ceased sidechains enable mainchain resource exhaustion attacks](#).

Zendoo guarantees there never can be more coins sent back from the SC to the MC than the amount of coins that were sent from MC to SC: the Zendoo client tracks a balance for each SC. Coinspect reviewed how this was implemented and did not find any vulnerabilities that could allow the creation of new coins.

The MBTR and CSW special transactions, designed to protect users from ill-behaved sidechains were dedicated time in order to establish if could be bypassed. Coinspect found certain sidechain related fees were validated in the MC before user transactions are accepted to the mempool. Because these fees can be set and updated by the SC certificate issuers, this could be abused by malicious sidechains to prevent users from recovering their funds by inflating these fees. This is further detailed in [ZOO-003 Malicious sidechains can prevent users from withdrawing funds to the mainchain](#).

Coinspect could not review a sidechain implementation with a fully working MBTR flow during this audit, and it is recommended the MBTR verification circuits and the whole mechanism is evaluated in the context of a working sidechain implementation as Latus when available.

Blockchain reorganization handling of SC related transactions, withdrawal certificates and events were reviewed to determine if the added complexity could lead to a vulnerability or broken consensus caused by edge cases but it was found to be correct.

4.3 Embedded crypto library interactions

Zendoo client embeds Rust libraries responsible for implementing the verification of SNARK proofs provided by sidechains certificate issuers.

The high level `zendoo-mc-crypto1ib` library is embedded by the Zendoo client and exposes the API which implements all the services needed. This library is implemented in `zendoo-mc-crypto1ib/src/lib.rs` and is responsible for converting memory buffers provided by the C++ code to Rust slices and passing them to the `zendoo-CCTP-1ib` which in turn relies on the lower level `ginger-1ib` code.

These libraries handle binary blobs, sometimes compressed, which are obtained from the network by the Zendoo client and are passed with almost no previous validation. For that reason, and because the failure in processing this untrusted data from the network results in a full node crash, Coinspect considered this an interesting target during the audit.

Coinspect auditors found it is possible to crash the Zendoo client node by providing a compressed `BitVector` that causes it to consume all system memory. This is detailed in [ZOO-002 DoS attack by improper handling of compressed data](#).

It is worth mentioning that as the `zendoo-mc-crypto1ib` Rust library externally exported API is designed to be embedded in the node, it has no alternative to trusting the memory pointers and buffer lengths (for variable length buffers) provided by the caller. These memory manipulations are performed in unsafe Rust blocks and could result in memory corruption and unexpected behaviour if the parameters passed by the node are not correct. Developers must be aware of this

fact, and never pass the `zendoo-mc-cryptolib` non-validated untrusted values obtained from the network.

Some not clearly identifiable constants were observed in Zendoo's client source code which lead to hard to maintain code. In particular, the magic constant 254 is shared between Zendoo client and the Rust libs:

1. `sidechaintypes.cpp` uses the constant 254
2. Rust libraries define `FIELD_CAPACITY` and `CAPACITY`, however the 254 constant is utilized in source code comments as well

Coinspect is calling attention to this issue and using this particular example as it is related to one of the high risk findings reported in this document: [ZOO-001 Reachable assertion allows attackers to hijack the network](#).

5. Summary of Findings

ID	Description	Risk	Fixed
ZOO-001	Reachable assertion allows attackers to hijack the network	High	✓
ZOO-002	DoS attack by improper handling of compressed data	High	✓
ZOO-003	Malicious sidechains can block withdrawals to the mainchain	High	✓
ZOO-004	Sidechain certificates enable mainchain resource exhaustion attacks	High	✓
ZOO-005	Ceased sidechains enable mainchain resource exhaustion attacks	High	✓

During August 2021 Coinspect verified all findings have been correctly fixed by Horizen. ZOO-003 is considered partially addressed as detailed below in the finding.

6. Findings

ZOO-001 Reachable assertion allows attackers to hijack the network		
Total Risk High	Impact High	Location cctp_primitives/src/bit_vector/merkle_tree.rs
Fixed ✓	Likelihood High	

Description

Attackers can selectively and reliably crash honest nodes by sending a specially crafted `BitVector` that results in a panic in the `zendoo-CTTP-lib` Rust library.

The ability to crash selected nodes could enable attackers to isolate parts of the network or cause the honest network hashing power to drop if miners are targeted. As a consequence, the security of the blockchain as a whole could be subverted.

If the provided `BitVector` size is not a power of 2, because of the `log2` function rounding in `ginger-lib`, the `assert` in the `merkle_root_from_bytes` function will fail resulting in the node crashing.

```
pub fn merkle_root_from_bytes(uncompressed_bit_vector: &[u8])
    -> Result<algebra::Fp256<algebra::fields::tweedle::FrParameters>, Error> {

    let bv = BitVec::from_bytes(&uncompressed_bit_vector);
    let bool_vector: Vec<bool> = bv.into_iter().map(|x| x).collect();

    // The bit vector may contain some padding bits at the end that have to be discarded
    let real_bit_vector_size: usize = bool_vector.len() - bool_vector.len()
        % FIELD_CAPACITY;

    let merkle_tree_height = log2(real_bit_vector_size / FIELD_CAPACITY) as usize; ←
    let num_leaves = 1 << merkle_tree_height;
    let mut mt = GingerMHT::init(
```

```

    merkle_tree_height,
    num_leaves,
);

let leaves = bool_vector[..real_bit_vector_size].to_field_elements()?;

assert_eq!(leaves.len(), num_leaves) ←

```

To reproduce this issue, modify the `sc_cert_custom_fields.py` test and change the `BIT_VECTOR_BUF` value to:

```

01425a68393141592653591dadce4d0000fe8180900000100008200030cc09aa6990
1b5403c5dc914e1424076b739340

```

The following output was obtained from the proof-of-concept attack script:

```

Create raw cert with good custom field elements for SC1...
thread '<unnamed>' panicked at 'assertion failed: `(left == right)`
  left: `4092`,
  right: `4096`,
/home/admin/.cargo/git/checkouts/zendoo-cctp-lib-852597c6711b5702/296ea2b/cctp_primitives/src/bit_vector/merkle_tree.rs:46:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
fatal runtime error: failed to initiate panic, error 5
Unexpected exception caught during testing: [Errno 104] Connection reset by peer ←
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/test_framework.py", line 122, in main
    self.run_test()
File "./jp_sc_cert_customfields.py", line 357, in run_test
    cert = self.nodes[0].sendrawcertificate(signed_cert['hex'])
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/authproxy.py", line 157, in __call__
    response = self._request('POST', self.__url.path, postdata)
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/authproxy.py", line 139, in _request
    return self._get_response()
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/authproxy.py", line 172, in _get_response
    http_response = self.__conn.getresponse()
File "/usr/lib/python2.7/httplib.py", line 1137, in getresponse
    response.begin()
File "/usr/lib/python2.7/httplib.py", line 448, in begin
    version, status, reason = self._read_status()
File "/usr/lib/python2.7/httplib.py", line 404, in _read_status
    line = self.fp.readline(_MAXLINE + 1)
File "/usr/lib/python2.7/socket.py", line 480, in readline
    data = self._sock.recv(self._rbufsize)
Stopping nodes
Traceback (most recent call last):

```

```

File "./jp_sc_cert_customfields.py", line 402, in <module>
    sc_cert_customfields().main()
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/test_framework.py", line 141, in main
    stop_nodes(self.nodes)
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/util.py", line 296, in stop_nodes
    node.stop()
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/authproxy.py", line 157, in __call__
    response = self._request('POST', self.__url.path, postdata)
File "/home/admin/zend_oo/qa/rpc-tests/test_framework/authproxy.py", line 126, in _request
    self.__conn.request(method, path, postdata, headers)
File "/usr/lib/python2.7/httplib.py", line 1058, in request
    self._send_request(method, url, body, headers)
File "/usr/lib/python2.7/httplib.py", line 1092, in _send_request
    self.putrequest(method, url, **skips)
File "/usr/lib/python2.7/httplib.py", line 934, in putrequest
    raise CannotSendRequest()
httplib.CannotSendRequest

```

This is the output captured from the node debug log:

```

Create raw cert with good custom field elements for SC1...] #####
2021-06-16 00:10:44.512858 [139766713227008] IsCertApplicableToState():1150 - called:
cert[147fe04681108ef5071e3c6c20b983ed6f7580bfe29868f72722721cc7ff4157],
scId[3e123a24e8e9ce28efc519252dd72
7e16365d41480874c7c1c6bae39ea076b32]
2021-06-16 00:10:44.512917 [139766713227008] GetSidechain():695 - FetchedSidechain:
scId[3e123a24e8e9ce28efc519252dd727e16365d41480874c7c1c6bae39ea076b32]
2021-06-16 00:10:44.512946 [139766713227008] GetSidechain():695 - FetchedSidechain:
scId[3e123a24e8e9ce28efc519252dd727e16365d41480874c7c1c6bae39ea076b32]
2021-06-16 00:10:44.512964 [139766713227008] GetSidechain():695 - FetchedSidechain:
scId[3e123a24e8e9ce28efc519252dd727e16365d41480874c7c1c6bae39ea076b32]
2021-06-16 00:10:44.512981 [139766713227008] GetSidechain():695 - FetchedSidechain:
scId[3e123a24e8e9ce28efc519252dd727e16365d41480874c7c1c6bae39ea076b32]
2021-06-16 00:10:44.513001 [139766713227008] vRawData.size() 48 > 1111
cfg.getMaxCompressedSizeBytes() ?
2021-06-16 00:10:44.513017 [139766713227008] decompressing, expected nBitVectorSizeBytes 129921
- The node crashes without logging anything else - 

```

Recommendation

Check `real_bit_vector_size` is divisible by 2 before using `log2` or remove the `assert` if not really needed.

Status

This finding was correctly addressed in [PR #131](#), which includes new related test cases.

ZOO-002 DoS attack by improper handling of compressed data

Total Risk

High

Fixed



Impact

High

Likelihood

High

Location

cctp_primitives/src/bit_vector/compression.rs

Description

Attackers can selectively crash arbitrary nodes at will by forcing them to decompress a specially crafted compressed `BitVector` that would consume system memory resulting in an out of memory (OOM) error.

This can be exploited when the compression algorithm used is `Bzip2`.

Even though the code does check the decompressed size is below the maximum expected size, this check is performed after the buffer has been decompressed. The user-provided buffer is passed to the Rust dependency in charge of decompressing `Bzip2` streams:

```
pub fn decompress_bit_vector(compressed_bit_vector: &[u8], expected_size: usize) -> Result<Vec<u8>, Error> {

    println!("Decompressing bit vector...");
    println!("Algorithm: {}, size: {}, expected decompressed size: {}, address: {:p}",
compressed_bit_vector[0], compressed_bit_vector.len(), expected_size, compressed_bit_vector);

    println!("Bit vector content:");
    println!("{:x?}", compressed_bit_vector);

    let mut raw_bit_vector_result = match compressed_bit_vector[0].try_into() {
        Ok(CompressionAlgorithm::Uncompressed) => Ok(compressed_bit_vector[1..].to_vec()),
        Ok(CompressionAlgorithm::Bzip2) => bzip2_decompress(&compressed_bit_vector[1..]),
        Ok(CompressionAlgorithm::Gzip) => gzip_decompress(&compressed_bit_vector[1..]),
        Err(_) => Err("Compression algorithm not supported")?
    };

    if raw_bit_vector_result.len() != expected_size {
        Err(format!("Wrong bit vector size. Expected {} bytes, found {} bytes", expected_size,
```

```

        raw_bit_vector_result.len()))?
    }

    raw_bit_vector_result.shrink_to_fit();
    Ok(raw_bit_vector_result)
}

fn bzip2_compress(bit_vector: &[u8]) -> Result<Vec<u8>, Error> {
    let mut compressor = BzEncoder::new(bit_vector, bzip2::Compression::best());
    let mut bzip_compressed = Vec::new();
    compressor.read_to_end(&mut bzip_compressed)?;

    Ok(bzip_compressed)
}

fn bzip2_decompress(compressed_bit_vector: &[u8]) -> Result<Vec<u8>, Error> {
    let mut uncompressed_bitvector = Vec::new();
    let mut decompressor = BzDecoder::new(compressed_bit_vector);

    decompressor.read_to_end(&mut uncompressed_bitvector)?; ←

    Ok(uncompressed_bitvector)
}

```

To reproduce this issue, modify the `sc_cert_custom_fields.py` test in the original repository and change the `BIT_VECTOR_BUF` value to `01` followed by the output of the following command:

```
dd if=/dev/zero bs=1G count=15 |bzip2 -9 | xxd -p | tr -d \n
```

where `01` indicates that `CompressionAlgorithm::Bzip2` is being used.

Also the custom bit vector config (`cmtCfg`) in the test must be adjusted to accommodate the new `BIT_VECTOR_BUF`.

The following output was obtained from the **proof-of-concept attack script**:

```
Verify vFieldElementCertificateFieldConfig / vBitVectorCertificateFieldConfig are correctly set
in scinfo for all SCs
```

```
Node 0 generates 4 block
```

```
epoch_number = 0, epoch_cum_tree_hash =
```

```
609586695e755c92dfb7bc03f331130046fb666f6270462d0a08365a1091ae20
```

```
Create raw cert with wrong field element for the referred SC2 (expecting failure)...
```

Send certificate failed with reason 16: bad-sc-cert-not-applicable

Create raw cert with good custom field elements for SC2...

cum = 609586695e755c92dfb7bc03f331130046fb666f6270462d0a08365a1091ae20

Check cert is in mempools

Create raw cert with bad custom field elements for SC1... (expecting failure)

Send certificate failed with reason 16: bad-sc-cert-not-applicable

Create raw cert with good custom field elements for SC1...

memory allocation of 4294967296 bytes failed ←

Unexpected exception caught during testing: [Errno 111] Connection refused ←

This is the output captured from the **node debug log**:

```
2021-06-16 17:12:43.985395 [140443970606848] dbg_log() - ##### [
Create raw cert with good custom field elements for SC1...] #####
2021-06-16 17:12:44.625181 [140443978999552] IsCertApplicableToState():1150 - called:
cert[cb59f2df96573005ce52e34be3ae9bcba3af5c0ddf059fdbc05a236257675a81],
scId[22a7e5e648ebee113bc4dcccdae5b6919c000e00c239bb692cd90d6056815ab3]
6919c000e00c239bb692cd90d6056815ab3]
2021-06-16 17:12:44.625240 [140443978999552] GetSidechain():695 - FetchedSidechain:
scId[22a7e5e648ebee113bc4dcccdae5b6919c000e00c239bb692cd90d6056815ab3]
2021-06-16 17:12:44.625270 [140443978999552] GetSidechain():695 - FetchedSidechain:
scId[22a7e5e648ebee113bc4dcccdae5b6919c000e00c239bb692cd90d6056815ab3]
2021-06-16 17:12:44.625288 [140443978999552] GetSidechain():695 - FetchedSidechain:
scId[22a7e5e648ebee113bc4dcccdae5b6919c000e00c239bb692cd90d6056815ab3]
2021-06-16 17:12:44.625305 [140443978999552] GetSidechain():695 - FetchedSidechain:
scId[22a7e5e648ebee113bc4dcccdae5b6919c000e00c239bb692cd90d6056815ab3]
2021-06-16 17:12:44.625339 [140443978999552] vRawData.size() 11252 > 12111
cfg.getMaxCompressedSizeBytes() ?
2021-06-16 17:12:44.625355 [140443978999552] decompressing, expected nBitVectorSizeBytes 129921
- The node crashes without logging anything else - ←
```

Recommendation

Enforce limits for the output size during the decompression process. Consider calling [read](#) in a loop instead of [read_to_end](#).

Status

This finding was correctly addressed in [PR #20](#), which includes new related test cases.

ZOO-003 Malicious sidechains can block withdrawals to the mainchain

Total Risk

High

Fixed



Impact

High

Likelihood

High

Location

zend_oo/src/coins.cpp

Description

Malicious sidechains can subvert the MBTR (Mainchain Backward Transfer Request) mechanism intended to safeguard users' funds.

Even though the MBTR mechanism is a safeguard intended to allow users to initiate funds withdrawals directly in the mainchain in a scenario where a malicious majority in the sidechain is censoring user's regular backward transfers, Coinspect found a malicious sidechain could subvert this protection mechanism implemented by the mainchain.

The MBTR mechanism is described in section 4.1.2.1 of the whitepaper:

4.1.2.1 Mainchain Managed Withdrawals

There might be cases when a user would want to request a backward transfer directly from the mainchain rather than creating a BT in the SC. **For instance, it would allow users to withdraw funds in case of a misbehaving (e.g., maliciously controlled sidechain that censors submission of backward transfers) or ceased sidechain. Hence, we introduced two additional mechanisms that allow users to make withdrawals directly in the mainchain: 1. Backward transfer request (BTR), and 2. Ceased sidechain withdrawal (CSW).** We consider each of them as a special type of transaction. Similar to withdrawal certificates, such operations are secured by SNARK proofs. The BTR is used to withdraw funds from an active sidechain if for some reason a user cannot create a backward transfer inside the sidechain. The idea is that all BTRs submitted to the mainchain will be synchronized to the sidechain and processed there to verify their legitimacy and include the corresponding backward transfers in the next WCert using the standard flow. Such processing can be enforced by the withdrawal certificate SNARK to force a maliciously controlled sidechain to process user's withdrawals. Importantly, the BTR does not lead to a direct coin transfer in the mainchain.

As explained above, MBTRs are enforced by mainchain Zendoo clients as a part of the SNARK verification circuit configured by the sidechain at creation time. Assuming the circuit is correct, a valid MBTR forces the SC to include a BWT in the next epoch certificate.

However, because the SC certificate issuers can set the fees for the sidechain related TXs, they can set the MBTR fee arbitrarily high to prevent MBTR requests from entering into the mempool. If the MBTR requests from the users of a sidechain are not accepted by mainnet nodes, then the mainchain does not have the opportunity to actually enforce the protection mechanisms. Even if the fees were not checked by the MC, the user would be required to have the necessary balance to pay for them.

The Zendoo mainchain client verifies the transaction's MBTR sidechain fee is above or equal to the current value for the target sidechain before accepting it to its mempool:

This is the code in `coins.cpp`:

```
CCoinsViewCache::IsScTxApplicableToState(  
  
/**  
 * Check that the Mainchain Backward Transfer Request amount is greater than or  
 * equal to the Sidechain Mainchain Backward Transfer Request fee.  
 */  
if (!CheckScMbtrFee(mbtr))  
{  
    LogPrintf("%s():%d - ERROR: Invalid tx[%s] :  
        MBTR fee [%s] cannot be less than SC MBTR fee [%s] for scId[%s]",  
        __func__, __LINE__, txHash.ToString(), FormatMoney(mbtr.scFee),  
        FormatMoney(GetActiveCertView(scId).mainchainBackwardTransferRequestScFee),  
        scId.ToString());  
    return CValidationState::Code::INVALID;  
  
/**  
 * @brief Checks whether a Mainchain Backward Transfer Request output is still valid
```

```

*         based on sidechain current MBTR fee.
*
* @param mbtrOutput The Mainchain Backward Transfer Request output to be checked.
* @return true if mbtrOutput is still valid, false otherwise.
*/
bool CCoinsViewCache::CheckScMbtrFee(const CBwtRequestOut& mbtrOutput) const
{
    CScCertificateView certView = GetActiveCertView(mbtrOutput.scId);
    return mbtrOutput.scFee >= certView.mainchainBackwardTransferRequestScFee; ←
}

```

The sidechain related fees are not mentioned in the whitepaper (they are not included in the `wcert_sysdata`). Coinspect observed that these fees are part of the verification circuit as well as they are included in the proofs the same way the certificate quality is.

This was confirmed by Horizen's team, which explained that the sidechain logic will be responsible for defining minimum fees based on the processing cost of the operations on the sidechain side. The logic for enforcing the fees will be sidechain dependent, and the plan for the Latus sidechain model is to use an average of proof creation cost for a number of past epochs.

However, it is not clear (source code responsible for validating fee updates was not found during this review) if fees will have a maximum or if their value will be restrained at all, as this will depend on each sidechain implementation.

Because the mainchain Zendoo client does not enforce any limit to how fees are updated, it can not guarantee that the MBTR mechanism will allow users to claim funds stuck in a malicious sidechain.

Note the MBTR functionality is not mandatory for sidechains to implement, but if present, it would give a false sense of security to users.

Recommendation

Consider enforcing fee updates limitations on the mainchain Zendoo client side, for example only allow fees to be gradually increased. This would prevent malicious sidechains from gaming the fee updates restrictions that might be in place in the verification circuits. Also, this would allow users to note fees are being raised and give them time to withdraw their funds from a sidechain.

Alternatively, clearly document how the MBTR mechanism is limited by the sidechain fee updates.

Status

This issue has been **partially addressed**.

The Zendoo team will emphasize the importance of having the fees very-gradually increased in their documentation.

However, Zendoo decided not to enforce fee update limitations in the mainchain. Their main reason to reject this suggestion is that it would hinder flexibility in the sidechains design, which is one of the system's design goals, as they explained:

Horizen believes that it is the sidechains who must correctly implement their strategy to set fees for Mainchain Backward Transfer Requests and Forward Transfers, and therefore it is responsibility of each sidechain to implement them in a way to, for example, not grow in a non-gradual way. We would not put this enforcement on the mainchain also because there may be some sidechains that have a particular logic for which, for example, there is a stronger volatility on the computation price of the proofs on the sidechains side, and the mainchain would not be aware. One of the goals, for us, consists in leaving this flexibility.

ZOO-004 Sidechain certificates enable mainchain resource exhaustion attacks

Total Risk

High

Fixed



Impact

High

Likelihood

High

Location

zend_oo/src/main.cpp

Description

Attackers can abuse the sidechain fee system to flood the mainchain network with transactions that will be relayed by all nodes but are not guaranteed to be included in a future block, avoiding paying any cost for the use of node's resources and network bandwidth.

A sidechain certificate has the ability to remove many TXs in the MC mempool. By updating the SC fees, all those TXs related to the SC might become no longer valid and will be evicted from node's mempools by the `removeStaleTransactions` function as soon as the certificate is mined .

This creates the opportunity to spam the network with transactions that are relayed to the whole network but will never be mined, and thus the network will relay transactions at potentially no cost for the submitter.

In one possible attack scenario the attackers could create a sidechain, and submit a high priority transaction with a certificate having an updated FT fee. As soon as this TX is mined, the attacker can submit as many FTs with the old fee to all the nodes that have not processed this new block and these will be relayed to all the network until the new chain tip is updated to the new block. In that moment, all the transactions with the previous FT fee in the network will be either:

- a. Rejected upon receipt if they are received after the new block has been connected
- b. Evicted from the mempool as no longer applicable

Alternatively, depending on how the mempools are constituted, an attacker could start spamming the network with low priority TXs even before the certificate with the fee update is mined. If there are enough TXs in the mempools, the attackers will be sure their TXs will not be mined before the fee update certificate transaction is included in a block, propagated, and their spam transactions are evicted from all mempools.

Another possible scenario is a malicious miner taking advantage over the rest of the miners that will be slowed down while relaying and processing all the spam transactions that will be accepted and later invalidated without a chance of being mined.

These are node's debug.log excerpts from Coinspect's PoC for reproducing these scenarios:

a. When the outdated TXs are received after the new block has been connected in the node

```
2021-06-22 23:24:53.725776 [140656009799424] ProcessMessage() - received: tx (305 bytes) peer=4
2021-06-22 23:24:53.725820 [140656009799424] checkTxSemanticValidity():194 -
tx=f4573645912db17edf2315b3947f65f05a03dc87696baf5032ab536dfadc230d
2021-06-22 23:24:53.725868 [140656009799424] GetSidechain():695 - FetchedSidechain:
scId[0179b8f3e52efd8f305d91ba5829f1ab527fd1cc285fe1ff5397ffdeafe0ed8a]
2021-06-22 23:24:53.725978 [140656009799424] IsScTxApplicableToState():1320 - ERROR: Invalid
tx[f4573645912db17edf2315b3947f65f05a03dc87696baf5032ab536dfadc230d] to
scId[0179b8f3e52efd8f305d91ba5829f1ab527fd1cc285fe1ff5397ffdeafe0ed8a]: FT amount [11.00] must
be
greater than SC FT fee [11.00]ERROR: AcceptTxToMemoryPool():1520 - ERROR: sc-related tx
[f4573645912db17edf2315b3947f65f05a03dc87696baf5032ab536dfadc230d] is not applicable:
ret_code[0x10]
```

```
2021-06-22 23:24:53.726003 [140656009799424]
f4573645912db17edf2315b3947f65f05a03dc87696baf5032ab536dfadc230d from peer=4 /zen:2.1.0beta4/
was not accepted into the memory pool: bad-sc-tx-not-applicable
2021-06-22 23:24:53.726017 [140656009799424] sending: reject (61 bytes) peer=4
```

b. When outdated TXs are in the mempool and the new block is received

```
node1/regtest/debug.log:2021-06-22 23:24:53.552443 [140656144017152]
removeStaleTransactions():813 - removed 0 certs and 98 txes
```

```
node2/regtest/debug.log:2021-06-22 23:24:53.717313 [139831912949504]
removeStaleTransactions():813 - removed 0 certs and 98 txes
```

```
node3/regtest/debug.log:2021-06-22 23:24:53.831225 [140202202875648]
removeStaleTransactions():813 - removed 0 certs and 98 txes
```

In order to be able to spam the MC for all blocks, attackers can create many sidechains with their corresponding epochs expiring in different blocks in the mainchain.

It is worth noting Coinspect verified that those nodes relaying the outdated transactions do not get banned by the nodes receiving the transactions. This is correct because it prevents this issue from being abused to ban and isolate nodes from the network.

Recommendation

In order to prevent this issue, Coinspect recommends further evaluating the logic used to determine if SC related transactions should be accepted to a Zendo mainchain node's mempool.

One possible solution for this specific issue could be to restrict the SC fee updates to a certain range each epoch and enforce this in the MC node code (restricting this to the verification circuit is not enough in the evil sidechain scenario). For example, only allow the fee to be increased or decreased by a 1% each epoch. Then, this would allow the node to only accept and relay SC related TXs with fees that make them minable during the current and the following epoch, thus preventing the attack.

Even though Coinspect focused on the fee update mechanism and how it affects the mempool for this finding description, it is also recommended to analyze if this issue could be exploited through other vectors involving other sidechain events and transactions.

Status

This finding was correctly addressed in [PR #165](#), which includes new related test cases.

ZOO-005 Ceased sidechains enable mainchain resource exhaustion attacks

Total Risk

High

Fixed



Impact

High

Likelihood

High

Location

zend_oo/src/txmempool.cpp

Description

Attackers can abuse CSW (Ceased Sidechain Withdrawal) transactions to flood the mainchain network with transactions that will be relayed by all nodes but are not guaranteed to be included in a future block, avoiding paying any cost for the use of nodes' resources and network bandwidth.

When a sidechain is ceased (after no certificate is submitted for the duration of the configured withdrawal epoch) users can submit CSW (Ceased Sidechain Withdrawal) transactions to the mainchain to claim their funds. In order to accept a CSW to the mempool, the Zendo client verifies that the current sidechain balance, including the effect of other CSW transactions already in the mempool, is enough to fulfil the request, or the transaction is rejected.

However, because the CSW proof verification circuit is defined by the sidechain when it is created, it is possible to define a sidechain with an ill-behaved circuit which allows generating proofs for users to claim more funds than actually available to the mainchain.

In that scenario, it is possible for conflicting CSW proofs to be created which can be used, when mined in a block, to invalidate many other legitimate proofs which were already accepted in the mempool and relayed to the whole network. All those TXs related to the SC might become no longer valid and will be evicted from nodes' mempools by the `removeOutOfScBalanceCsw` function as soon as the conflicting transaction is mined.

This creates the opportunity to spam the network with transactions that are relayed to the whole network but will never be mined, and thus the network will relay transactions at potentially no cost for the submitter.

The attack scenarios are similar to the ones previously discussed in [ZOO-004 Mainchain TX flooding network denial of service](#).

This is the output from Coinspect's PoC for reproducing this specific scenario:

```
Created SC id: 1f126e457487acda6352f7aa58995c88a882c700857a70589b96213939f004f6
```

```
Let 2 epochs pass by...
```

```
==> certificate for epoch 0
```

```
15a57eec202275a66df329621b0c2b95b41f95e07b1a6d5879cd5909bc91e971
```

```
==> certificate for epoch 1
```

```
e3ae8e7e09262b38086f480504e8e07f51f09846424604e1de1aee8af8bdd0b31
```

```
Node0 generates 1 block confirming last cert
```

```
Let SC cease...
```

```
Node0 generates 9 blocks
```

```
Splitting network 0:1-2-3
```

```
Creating 10 CSW withdrawing sc_balance/10 each...
```

```
Active Cert Data Hash: ----->
```

```
16e6298e066db1a0dd39fb0e45ad9bce56246a864a5d291ce31c3abee1b4193e
```

```
Ceasing Cum Sc Tx Commitment Tree: ----->
```

```
32da6a97d50fb58309465f3feaa241d3caf37819424bbd8598fd47498f646835
```

```
Node 0 generates 1 block with CSW for full SC balance
```

```
! sent eb07457b118577e54a3a8140b51228e5053d0e2b51eff642a10e09ac0b0714e8
```

```
SPAMMING NETWORK WITH SOON TO BE INVALID CSWs...
```

```
! sent ee587d665340560bca99c50d6c97e7ceb608e5308be131695b85f02526cb4e0e
```

```
! sent b0dbed3949ce707550709c87ed3d5419a7314469fa2365e41b517948e92e2d6c
```

```
! sent 3d671f11d11a028e5e8ce2e37997ca6ffa2f1149554b3523c90c6538c2232771
! sent 92713fd07838090c50662742c975f984c27727f61f9b8a2872a94d9d2dd2ffc0
! sent 1a320acfd8a18bb7e82b2a992f86d45fc9713ab37bdca92fc09dd68b5b10b40f
! sent e2deee147286c13a133a0f0e066dd79d5037bdd64fdb6c3eb92029340e691f0f
! sent 4290bb35ab602366cb9fb48ea21c1bbc6f800a81adecdadfae956dfffd1ca3284
! sent 52e1fe189841b0e3a79ef7f2a91a5fff63fe570ec413416dd5069ed10fc1f00f6
! sent 9259db499e9b577a7c08802fec4b45829066f4002848736361a451042b9481d9
! sent b91f2df8cdd395f06350737dc1f0f2979d651263d28bf903587fad43ec23c7dd
```

Node 0 publishes block with CSW to node 1 to invalidate all previously relayed TXs
Node3 mempool has 0 tx

It is worth noting Coinspect verified that those nodes relaying the outdated transactions do not get banned by the nodes receiving the transactions. This is correct because it prevents this issue from being abused to ban and isolate nodes from the network.

Recommendation

In order to prevent this issue, Coinspect recommends further evaluating the logic used to determine if SC related transactions should be accepted to a Zendo mainchain node's mempool.

One possible solution for this specific issue could be to restrict the number of pending CSW transactions in the mempool for each sidechain. This would prevent the relaying of massive quantities of transactions that are not guaranteed to be included in a block.

Even though Coinspect focused on the CSW mechanism and how it affects the mempool for this finding description, it is also recommended to analyze if this issue could be exploited through other vectors involving other sidechain events and transactions.

Status

This finding was correctly addressed in [PR #168](#), which includes new related test cases.

7. Appendix I

During September 2021, Coinspect reviewed the following modifications performed to the Zendoo client repository as per the client's request:

- ForwardTransfer: mcReturnAddress field added.
https://github.com/HorizenOfficial/zend_oo/pull/180
- Sidechains integration step4 low prio thr pause
https://github.com/HorizenOfficial/zend_oo/pull/147
- Increased block size up to 4M; txes are allowed to occupy only a subs...
https://github.com/HorizenOfficial/zend_oo/pull/149
- Added optional parameter 'constant' field element to the generation a...
https://github.com/HorizenOfficial/zend_oo/pull/179
- backport of zen pr #379
https://github.com/HorizenOfficial/zend_oo/pull/170
- Websocket write optimization
https://github.com/HorizenOfficial/zend_oo/pull/171
- Changed segment size from 2^{17} to 2^{18}
https://github.com/HorizenOfficial/zend_oo/pull/169
- Fix for certificates handling in GetBlockTemplate
https://github.com/HorizenOfficial/zend_oo/pull/173
- Removed the duplicated 'scTxCommitment' field in 'getblock' response
https://github.com/HorizenOfficial/zend_oo/pull/175
- Added strict deserialization functions and size checks in sidechaintype.cpp to fix an arbitrary data appending to CscProof and CScvKey objects issue found by the Zendoo team:
 - https://github.com/HorizenOfficial/zend_oo/blob/sidechains_integration_code_review/src/sc/sidechaintypes.cpp#L307
 - https://github.com/HorizenOfficial/zend_oo/blob/sidechains_integration_code_review/src/sc/sidechaintypes.cpp#L394
 - https://github.com/HorizenOfficial/zend_oo/blob/sidechains_integration_code_review/src/gtest/test_libzendoo.cpp#L1932
 - <https://github.com/HorizenOfficial/zendoo-cctp-lib/pull/28>
 - <https://github.com/HorizenOfficial/zendoo-mc-cryptolib/pull/52>

No security issues were identified.

8. Disclaimer

The information presented in this document is provided "as is" and without warranty. Source code reviews are a "point in time" analysis and as such it is possible that something in the code could have changed since the tasks reflected in this report were executed. This report should not be considered a perfect representation of the risks threatening the analyzed system.