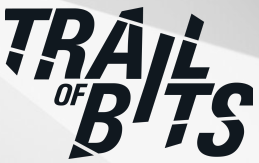

DFINITY Threshold ECDSA Integration and Bitcoin Canisters

Fix Review

September 5, 2022

Prepared for:
Robin Künzler
DFINITY

Prepared by: **Fredrik Dahlgren**

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.

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Table of Contents

Executive Summary

Engagement Overview

DFINITY engaged Trail of Bits to review the security of the integration of its ECDSA threshold signature scheme with its consensus protocol and Bitcoin canister and adapter. From May 2 to May 13, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

DFINITY contracted Trail of Bits to review the fixes implemented for issues identified in the original report. From September 1 to September 5, 2022, one consultant conducted a review of the client-provided source code.

Summary of Findings

The original audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the original findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	1
Informational	1
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	2
Denial of Service	2

Overview of Fix Review Results

DFINITY has sufficiently addressed all but one of the four issues described in the original audit report.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineer was associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Project Timeline

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

Date	Event
April 28, 2022	Kickoff call: ECDSA–consensus protocol integration
May 2, 2022	Kickoff call: ECDSA–Bitcoin integration
May 9, 2022	Status update meeting #1
May 16, 2022	Delivery of report draft
May 16, 2022	Report readout meeting
July 7, 2022	Delivery of final report
September 5, 2022	Delivery of fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material

Project Targets

The engagement involved a review of the fixes implemented in the targets listed below.

ECDSA Consensus Integration

Repository	`dfinity/ic/rs/consensus`
Version	`6febeeadc3dfefbb85bded262ff999ea8764e8a5`
Type	Rust
Platform	Linux

ECDSA Bitcoin Integration

Repository	`dfinity/ic/rs/bitcoin`
Version	`6febeeadc3dfefbb85bded262ff999ea8764e8a5`
Type	Rust
Platform	Linux

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Lack of validation of signed dealing against original dealing	Resolved
2	The ECDSA payload is not updated if a quadruple fails to complete	Resolved
3	Malicious canisters can exhaust the number of available quadruples	Unresolved
4	Aggregated signatures are dropped if their request IDs are not recognized	Resolved

Detailed Fix Review Results

1. Lack of validation of signed dealing against original dealing	
Status: **Resolved**	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-DFTECDSA-1
Target: `ic/rs/consensus/src/ecdsa/pre_signer.rs`	

Description

The `EcdsaPreSignerImpl::validate_dealing_support` method does not check that the content of signed dealings matches the original dealings. A malicious receiver could exploit this lack of validation by changing the requested height (that is, the `support.content.requested_height` field) or internal data (the `support.content.idk_dealing.internal_dealing_raw` field) before signing the ECDSA dealing. The resulting signature would not be flagged as invalid by the `validate_dealing_support` method but would result in an invalid *aggregated* signature.

After all nodes sign a dealing, the `EcdsaTranscriptBuilderImpl::build_transcript` method checks the signed dealings' content hashes before attempting to aggregate all the dealing support signatures to produce the final aggregated signature. The method logs a warning when the hashes do not agree, but does not otherwise act on signed dealings with different content.

```
let mut content_hash = BTreeSet::new();
for share in &support_shares {
    content_hash.insert(ic_crypto::crypto_hash(&share.content));
}
if content_hash.len() > 1 {
    warn!(
        self.log,
        "Unexpected multi share content: support_shares = {}, content_hash = {}",
        support_shares.len(),
        content_hash.len()
    );
    self.metrics.payload_errors_inc("invalid_content_hash");
}

if let Some(multi_sig) = self.crypto_aggregate_dealing_support(
    transcript_state.transcript_params,
```

```
    &support_shares,
) {
    transcript_state.add_completed_dealing(signed_dealing.content, multi_sig);
}
```

Figure 1.1: ic/rs/consensus/src/ecdsa/pre_signer.rs:1015-1034

The dealing content is added to the set of completed dealings along with the aggregated signature. When the node attempts to create a new transcript from the dealing, the aggregated signature is checked by `IDkgProtocol::create_transcript`. If a malicious receiver changes the content of a dealing before signing it, the resulting invalid aggregated signature would be rejected by this method. In such a case, the `EcdsaTranscriptBuilderImpl` methods `build_transcript` and `get_completed_transcript` would return `None` for the corresponding transcript ID. That is, neither the transcript nor the corresponding quadruple would be completed.

Additionally, since signing requests are deterministically matched against quadruples, including quadruples that are not yet available, this issue could allow a single node to block the service of individual signing requests.

```
pub(crate) fn get_signing_requests<'a>(
    ecdsa_payload: &ecdsa::EcdsaPayload,
    sign_with_ecdsa_contexts: &'a BTreeMap<CallbackId, SignWithEcdsaContext>,
) -> BTreeMap<ecdsa::RequestId, &'a SignWithEcdsaContext> {
    let known_random_ids: BTreeSet<[u8; 32]> = ecdsa_payload
        .iter_request_ids()
        .map(|id| id.pseudo_random_id)
        .collect::<BTreeSet<_>>();
    let mut unassigned_quadruple_ids =
        ecdsa_payload.unassigned_quadruple_ids().collect::<Vec<_>>();
    // sort in reverse order (bigger to smaller).
    unassigned_quadruple_ids.sort_by(|a, b| b.cmp(a));
    let mut new_requests = BTreeMap::new();
    // The following iteration goes through contexts in the order
    // of their keys, which is the callback_id. Therefore we are
    // traversing the requests in the order they were created.
    for context in sign_with_ecdsa_contexts.values() {
        if known_random_ids.contains(context.pseudo_random_id.as_slice()) {
            continue;
        };
        if let Some(quadruple_id) = unassigned_quadruple_ids.pop() {
            let request_id = ecdsa::RequestId {
                quadruple_id,
                pseudo_random_id: context.pseudo_random_id,
            };
            new_requests.insert(request_id, context);
        } else {
            break;
        }
    }
```

```
    new_requests
}
```

Figure 1.2: ic/rs/consensus/src/ecdsa/payload_builder.rs:752-782

Fix Analysis

This issue has been fixed. Each dealing support now contains a hash of the original dealing. The `EcdsaPreSignerImpl::validate_dealing_support` method obtains the original dealing using the hash from the dealing support, and the `EcdsaPreSignerImpl::crypto_verify_dealing_support` method then verifies the signature share from the dealing support against the original dealing.

2. The ECDSA payload is not updated if a quadruple fails to complete

Status: **Resolved**	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-DFTECDSA-2
Target: Consensus integration	

Description

If a transcript fails to complete (as described in TOB-DFTECDSA-1), the corresponding quadruple, Q_i, will also fail to complete. This means that the quadruple ID for Q_i will remain in the `quadruples_in_creation` set until the key is reshared and the set is purged. (Currently, the key is reshared if a node joins or leaves the subnet, which is an uncommon occurrence.) Moreover, if a transcript and the corresponding Q_i fail to complete, so will the corresponding signing request, SR_i, as it is matched deterministically with Q_i.

```
let ecdsa_payload = ecdsa::EcdsaPayload {
    signature_agreements: ecdsa_payload.signature_agreements.clone(),
    ongoing_signatures: ecdsa_payload.ongoing_signatures.clone(),
    available_quadruples: if is_new_key_transcript {
        BTreeMap::new()
    } else {
        ecdsa_payload.available_quadruples.clone()
    },
    quadruples_in_creation: if is_new_key_transcript {
        BTreeMap::new()
    } else {
        ecdsa_payload.quadruples_in_creation.clone()
    },
    uid_generator: ecdsa_payload.uid_generator.clone(),
    idkg_transcripts: BTreeMap::new(),
    ongoing_xnet_reshares: if is_new_key_transcript {
        // This will clear the current ongoing reshares, and
        // the execution requests will be restarted with the
        // new key and different transcript IDs.
        BTreeMap::new()
    } else {
        ecdsa_payload.ongoing_xnet_reshares.clone()
    },
    xnet_reshare_agreements: ecdsa_payload.xnet_reshare_agreements.clone(),
};
```

Figure 2.1: The `quadruples_in_creation` set will be purged only when the key is reshared.

The canister will never be notified that the signing request failed and will be left waiting indefinitely for the corresponding reply from the distributed signing service.

Fix Analysis

This issue has been fixed. It is now possible to set an expiry time for signing requests. If a request has expired, it is removed from the set of ongoing signing requests in the `get_signing_requests` function. Since signing requests are removed only once they have been paired with a quadruple, the corresponding quadruple will be removed as well. This ensures that the following invariants hold:

1. If a quadruple fails to complete, it will eventually be dropped by the payload builder when the corresponding signature request expires.

2. Quadruples are deterministically matched with signing requests, which ensures that a malicious user cannot influence how quadruples and signing requests are paired.

3. Malicious canisters can exhaust the number of available quadruples

Status: **Unresolved**	
Severity: **Undetermined**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-DFTECDSA-3
Target: Consensus integration	

Description

By requesting a large number of signatures, a canister (or set of canisters) could exhaust the number of available quadruples, preventing other signature requests from completing in a timely manner.

The ECDSA payload builder defaults to creating one extra quadruple in `create_data_payload` if there is no ECDSA configuration for the subnet in the registry.

```
let ecdsa_config = registry_client
    .get_ecdsa_config(subnet_id, summary_registry_version)?
    .unwrap_or(EcdsaConfig {
        quadruples_to_create_in_advance: 1, // default value
        ..EcdsaConfig::default()
    });
```

Figure 3.1: `ic/rs/consensus/src/ecdsa/payload_builder.rs:400-405`

Signing requests are serviced by the system in the order in which they are made (as determined by their `CallbackID` values). If a canister (or set of canisters) makes a large number of signing requests, the system would be overwhelmed and would take a long time to recover.

This issue is partly mitigated by the fee that is charged for signing requests. However, we believe that the financial ramifications of this problem could outweigh the fees paid by attackers. For example, the type of denial-of-service attack described in this finding could be devastating for a DeFi application that is sensitive to small price fluctuations in the Bitcoin market.

Since the ECDSA threshold signature service is not yet deployed on the Internet Computer, it is unclear how the service will be used in practice, making the severity of this issue difficult to determine. Therefore, the severity of this issue is marked as undetermined.

Fix Analysis

This issue has not been addressed, and the DFINITY team accepts the associated risk. The DFINITY team agrees with our assessment that this issue is difficult to mitigate completely and that developers should be made aware of the system limitations described in this finding.

4. Aggregated signatures are dropped if their request IDs are not recognized

Status: **Resolved**	
Severity: **Informational**	Difficulty: **N/A**
Type: Denial of Service	Finding ID: TOB-DFTECDSA-4
Target: `ic/rs/consensus/src/ecdsa/payload_builder.rs`	

Description

The `update_signature_agreements` function populates the set of completed signatures in the ECDSA payload. The function aggregates the completed signatures from the ECDSA pool by calling `EcdsaSignatureBuilderImpl::get_completed_signatures`. However, if a signature's associated signing request ID is not in the set of ongoing signatures, `update_signature_agreements` simply drops the signature.

```
for (request_id, signature) in builder.get_completed_signatures(
    chain,
    ecdsa_pool.deref()
) {
    if payload.ongoing_signatures.remove(&request_id).is_none() {
        warn!(
            log,
            "ECDSA signing request {:?} is not found in
            payload but we have a signature for it",
            request_id
        );
    } else {
        payload
            .signature_agreements
            .insert(request_id, ecdsa::CompletedSignature::Unreported(signature));
    }
}
```

Figure 4.1: `ic/rs/consensus/src/ecdsa/payload_builder.rs:817-830`

Barring an implementation error, this should not happen under normal circumstances.

Fix Analysis

This issue has been fixed. The `update_signature_agreements` function has been rewritten; it now attempts to aggregate only signatures corresponding to ongoing signing requests in `payload.ongoing_signatures`. If the signature is completed successfully, the corresponding request ID is moved from `payload.ongoing_signatures` to `payload.signature_agreements`.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

B. 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.