Dfinity Candid

Security Assessment

November 30, 2023

Prepared for:
Raghavan Sundaravaradan
Dfinity

Prepared by: **Dominik Czarnota, Dominik Klemba, and Samuel Moelius**

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 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

Anders Helsing, Engineering Director, Application Security
anders.helsing@trailofbits.com

The following consultants were associated with this project:

Dominik Czarnota, Consultant
dominik.czarnota@trailofbits.com

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Dominik Klemba, Consultant
dominik.czarnota@trailofbits.com

Project Timeline

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

Date	Event
October 30, 2023	Pre-project kickoff call
November 3, 2023	Status update meeting #1
November 13, 2023	Delivery of report draft
November 13, 2023	Report readout meeting
November 30, 2023	Delivery of comprehensive report

Executive Summary

Engagement Overview

Dfinity engaged Trail of Bits to review the security of its `candid` and `ic-transport-types` packages, focusing on the Rust code to be integrated with Trust Wallet. Candid is an interface description language (IDL) used to develop canisters (programs) on the Internet Computer blockchain.

A team of three consultants conducted the review from October 30 to November 9, 2023, for a total of three engineer-weeks of effort. Our testing efforts focused on the code paths intended to be used in Trust Wallet integration, with the `candid` package's optional features disabled. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The `candid` library suffers from what appears to be a design flaw. Success of a deserialization operation requires certain other operations to have been performed within the same thread. Those other operations are required to populate a map used to reconstruct recursive data structures. If the necessary operations were performed within a different thread, deserialization of such types will fail.

The project appears to run Clippy regularly in CI, but not with the pedantic lints enabled. Addressing the warnings produced by the pedantic lints would appear to have a positive impact on the code.

The code has existing fuzz harnesses. However, one of them did not build at the start of the audit. After that harness was fixed, we found a crash by running it. These facts suggest that the fuzzing harnesses are not being run regularly.

Finally, while the project's test coverage appears adequate, we discovered that certain tests are checking for incorrect behavior.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Dfinity take the following steps:

- **Remediate the findings disclosed in this report, including the design flaw mentioned above (TOB-CANDID-12).** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Enable the pedantic Clippy lints on the CI/CD system (`-W clippy::pedantic` flag).** This would have a positive impact on the overall correctness and quality of the codebase.
- **Regularly fuzz the codebase with the existing harnesses and develop new ones.** One of this report's findings was found by running one of the existing harnesses. Developing new harnesses to improve code and functionality coverage will increase the likelihood of detecting bugs early.
- **Write comprehensive and readable unit tests for every function.** Methodically cover all possible inputs and outputs, including edge cases and potential errors, to ensure that all functions work as intended**.**

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	1
Low	2
Informational	10
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	3
Denial of Service	2
Error Reporting	2
Patching	2
Testing	2
Undefined Behavior	5

Project Goals

The engagement was scoped to provide a security assessment of the Dfinity's `candid` and `ic-transport-types` packages. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the implementation of Candid IDL correspond to its specification?
- Is the data encoded or decoded by Candid validated properly?
- Is it possible to bypass the limitations described in the specification (e.g., the maximum length of a vector of zero-sized-type elements)?
- Is it possible to cause a denial of service to an application that encodes or decodes data with Candid?
- Are the encoding, decoding, and other code paths resistant against stack overflows caused by recursive calls?

Project Targets

The engagement involved a review and testing of the `candid` and `ic-transport-types` Rust packages and their non-optional external dependencies. Our focus was on the code where the `candid` package's features are disabled, and on code paths that are intended to be used in the upcoming Trust Wallet integration.

Candid

Repository	https://github.com/dfinity/candid
Version	`0191a6d2accda807c4dea4aece4e6be14b687338`
Type	Rust
Platform	POSIX
Directories	`candid/rust/candid`, `candid/spec`

ic-transport-types

Repository	https://github.com/dfinity/agent-rs
Version	`daf0035cfdc6abdef271a1b9179c282fc3e8550d`
Type	Rust
Platform	POSIX
Directory	`agent-rs/ic-transport-types`

Project Coverage

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

- Manual review of the `candid` (`candid/rust/candid` directory) and `ic-transport-types` Rust packages, and external dependencies of the `candid` package
- Running and triaging results from static analysis tools
- Running and reviewing unit tests implemented in those packages
- Fuzzing Candid code paths and developing new fuzzing harnesses

Coverage Limitations

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

- Code paths that require certain `candid` crate features to be enabled
- The `candid-parser`'s code paths, since they were out of scope of the audit
- The formal verification specifications that are written in the Coq formal proof management system

Automated Testing

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

Test Harness Configuration

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

- Clippy: A collection of lints to catch common mistakes and improve Rust code
- `cargo-audit`: A Cargo subcommand to audit `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database
- `cargo-llvm-cov`: A Cargo subcommand for obtaining and analyzing LLVM source–based code coverage
- `test-fuzz`: A Cargo subcommand and a collection of Rust macros to automate certain tasks related to fuzzing with `afl.rs`

Fuzzing

We developed fuzzers for the targets listed in the table below. For each, the inputs to the function were considered completely arbitrary, even in cases where the inputs would have to pass additional checks made by the fuzz targets' callers. For this reason, one of the resulting findings is marked informational. Addressing that finding may not fix an outright vulnerability, but would provide defense in depth.

Fuzz Target	Type	Finding
`subtype_`	`test-fuzz`	TOB-CANDID-4
`IDLArgs::from_bytes`	existing `cargo-fuzz` target	TOB-CANDID-10
`Decode!` macro	existing `cargo-fuzz` target	None

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	We found that the arithmetic operators sometimes produced incorrect values. Also, the correctness of arithmetic operations could be tested more thoroughly.	**Moderate**
Auditing	The library returns sufficiently descriptive error messages in appropriate places. Logging was not considered since this is primarily the responsibility of the library users.	**Satisfactory**
Authentication / Access Controls	The category is not applicable to this review.	**Not Applicable**
Complexity Management	Clippy appears to be run regularly in CI, but not with the pedantic lints enabled. Acting on the warnings produced by Clippy's pedantic lints would appear to have a positive impact on the code.	**Satisfactory**
Configuration	The library is configured through `candid` crate features. We found a minor issue that one of the features required another, which was not specified in the Cargo.toml (as detailed in appendix C).	**Satisfactory**
Cryptography and Key Management	The category is not applicable to this review.	**Not Applicable**
Data Handling	The `candid` library suffers from an apparent design flaw in that certain operations are required to have been performed within the same thread for deserialization of a recursive type to succeed.	**Moderate**

Documentation	The Candid IDL has its own specification that describes its format, usages and limitations. On the other hand, the `candid` package lacks documentation on its features and when one should enable or disable them.	**Satisfactory**
Maintenance	The project uses GitHub actions for its linters, builds, and tests. However, we found that the existing fuzzing harnesses do not build and that the GitHub Actions versions are not pinned to commits.	**Moderate**
Memory Safety and Error Handling	The `candid` crate allows developers to act on encoding and decoding errors. However, some returned errors are not specialized types and so cannot be acted upon without parsing the error message. Additionally, while Candid uses a check to prevent excessively deep recursion causing stack overflows, it cannot guarantee that the check will work in all cases because it assumes that the largest function stack frame is 32KB. Additionally, Candid does not perform the recursion checks in its encoding code path.	**Moderate**
Testing and Verification	The project includes tests and formal verification done in Coq (which was out of scope for this audit). However, the existing tests do not cover all cases, and certain tests are oversimplified and test for incorrect behavior.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Unmaintained dependency in candid_parser	Patching	Informational
2	Insufficient linter use	Patching	Informational
3	Imprecise errors	Error Reporting	Informational
4	Unnecessary recursion	Denial of Service	Undetermined
5	The IDL allows for recursive cyclic types which should not be allowed	Data Validation	Undetermined
6	Stack overflow in encoding/serialization path	Data Validation	Medium
7	The fuzzing harnesses do not build	Testing	Informational
8	The float32/float64 infinite signs are displayed incorrectly	Testing	Informational
9	Incorrect arithmetic	Undefined Behavior	High
10	Inadequate recursion checks	Denial of Service	Low
11	TypeId::of functions could be optimized into a single function	Undefined Behavior	Informational
12	Deserialization correctness depends on the thread in which operations are performed	Undefined Behavior	Informational

13	External GitHub CI actions versions are not pinned	Undefined Behavior	**Low**
14	Inconsistent support for types in operators	Undefined Behavior	**Informational**
15	Recursion checks do not ensure stack frame size	Data Validation	**Informational**
16	Misleading error message	Error Reporting	**Informational**

Detailed Findings

1. Unmaintained dependency in candid_parser	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-CANDID-1
Target: `rust/candid_parser/Cargo.toml`	

Description

The `candid_parser` package relies on `serde_dhall`, whose `README` contains the following message:

> *STATUS*
>
> *I am no longer maintaining this project. I got it to support about 90% of the language but then lost faith in the usability of dhall for my purposes. I am willing to hand this over to someone who's excited about dhall and rust.*

Note that this finding is informational because `candid_parser` is outside of the audit's scope.

Exploit Scenario

Eve learns of a vulnerability in `serde_dhall`. Because the package is unmaintained, the vulnerability persists. Eve exploits the vulnerability in `candid`, knowing that it relies on `serde_dhall`.

Recommendations

Short term, adopt one of the following three strategies:

- Deprecate support for the dhall format.
- Seek an alternative implementation of `serde_dhall`'s functionality. (We were unable to find one.)
- Take ownership of `serde_dhall`.

Taking one of these steps will eliminate `candid_parser`'s current reliance on an unmaintained dependency.

Long term, regularly run `cargo-audit` and `cargo upgrade --incompatible`. Doing so will help ensure that the project stays up to date with its dependencies.

2. Insufficient linter use

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-CANDID-2
Target: Various source files	

Description

The Candid project appears to run Clippy with only the default set of lints enabled (`clippy::all`). The maintainers should consider running additional lints, as many of them produce warnings when enabled for the project.

Running Clippy with `-W clippy::pedantic` produces several hundred warnings, indicating that enabling additional lints could have a positive impact on the correctness of the codebase. Examples of such warnings appear in figures 2.1 through 2.3.

```
warning: unnested or-patterns
   --> rust/candid/src/de.rs:721:13
    |
721 |             (TypeInner::Null, TypeInner::Opt(_)) | (TypeInner::Reserved,
TypeInner::Opt(_)) => {
    |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#unnested_or_patterns
    = note: `-W clippy::unnested-or-patterns` implied by `-W clippy::pedantic`
help: nest the patterns
    |
721 |             (TypeInner::Null | TypeInner::Reserved, TypeInner::Opt(_)) => {
    |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 2.1: Warning produced by the `clippy::unnested_or_patterns` lint

```
warning: calling `to_string` on `&&str`
  --> rust/candid/src/error.rs:53:26
   |
53 |                 message: err.to_string(),
   |                          ^^^^^^^^^^^^^^^ help: try dereferencing the receiver:
`(*err).to_string()`
   |
   = help: `&str` implements `ToString` through a slower blanket impl, but `str` has
a fast specialization of `ToString`
   = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#inefficient_to_string
```

```
   = note: `-W clippy::inefficient-to-string` implied by `-W clippy::pedantic`
```

Figure 2.2: Warning produced by the `clippy::inefficient_to_string` lint

```
warning: it is more concise to loop over references to containers instead of using
explicit iteration methods
   --> rust/candid/src/types/impls.rs:137:18
    |
137 |         for e in self.iter() {
    |                  ^^^^^^^^^^ help: to write this more concisely, try: `self`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#explicit_iter_loop
    = note: `-W clippy::explicit-iter-loop` implied by `-W clippy::pedantic`
```

Figure 2.3: Warning produced by the `clippy::explicit_iter_loop` lint

Exploit Scenario

Eve uncovers a bug in the `candid` package. The bug would have been caught by Dfinity had additional lints been enabled.

Recommendations

Short term, review all of the warnings currently generated by Clippy's pedantic lints. Address those in figures 2.1 through 2.3, and any others for which it makes sense to do so. Taking these steps will produce cleaner code, which in turn will reduce the likelihood that the code contains bugs.

Long term, take the following steps:

- Consider running Clippy with `-W clippy::pedantic` regularly. As demonstrated by the warnings in figures 2.1 through 2.3, the pedantic lints provide valuable suggestions.
- Regularly review Clippy lints that have been allowed to determine whether they should still be given such an exemption. Allowing a Clippy lint unnecessarily could cause bugs to be missed.

3. Imprecise errors

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-CANDID-3
Target: `candid/rust/candid/src/error.rs`	

Description

The `candid` package has a "catch all," `Custom` error variant that is overused. Using more precise error variants would make it easier for clients to know and act on errors that can occur in the `candid` package.

The `candid` package's error type appears in figure 3.1. Note that it contains only two specific error variants, `Binread` and `Subtype`. Errors that do not fall into one of these two categories must be returned as a `Custom` error.

```
15    #[derive(Debug, Error)]
16    pub enum Error {
17       #[error("binary parser error: {}", .0.get(0).map(|f| format!("{} at byte
offset {}", f.message, f.pos/2)).unwrap_or_else(|| "io error".to_string()))]
18       Binread(Vec<Label>),
19
20       #[error("Subtyping error: {0}")]
21       Subtype(String),
22
23       #[error(transparent)]
24       Custom(#[from] anyhow::Error),
25    }
```

Figure 3.1: The `candid` package's error type (`candid/rust/candid/src/error.rs#15–25`)

`Error` has an associated `msg` function that wraps a string in an `anyhow::Error`, and returns it in a `Custom` error. Example uses of the `msg` function appear in figure 3.2. As the figure shows, IO errors and errors related to integer parsing are turned into `Custom` errors using the `msg` function. It would be better to give such errors their own variants, similar to `Binread` and `Subtype`.

```
119    impl From<io::Error> for Error {
120       fn from(e: io::Error) -> Error {
121           Error::msg(format!("io error: {e}"))
122       }
123    }
124
125    impl From<binread::Error> for Error {
```

```
126         fn from(e: binread::Error) -> Error {
127             Error::Binread(get_binread_labels(&e))
128         }
129     }
130
131     impl From<ParseIntError> for Error {
132         fn from(e: ParseIntError) -> Error {
133             Error::msg(format!("ParseIntError: {e}"))
134         }
135     }
```

Figure 3.2: Example uses of `Error::msg` (candid/rust/candid/src/error.rs#119–135)

Exploit Scenario

Alice writes code that uses `candid` as a dependency. A catastrophic error occurs in `candid`. The error is bubbled up to Alice's code as a `Custom` error. Alice's code ignores the error, not knowing that it is one of the possible `Custom` errors.

Recommendations

Short term, use "catch all" error types (like `Custom`) sparingly. Prefer to use variants that communicate the type of the error that occurred. For example, the uses of `Error::msg` in figure 3.2 could be given their own error variants, analogous to `Error::Binread` in the same figure. Taking these steps will make it easier for clients to know the types of errors that can occur, and to act on them.

Long term, review all uses of `with_context`. This method adds information to an error, but as a string (i.e., unstructured data). It may be preferable to present such data to clients in struct fields.

4. Unnecessary recursion

Severity: **Undetermined**	Difficulty: **N/A**
Type: Denial of Service	Finding ID: TOB-CANDID-4
Target: `candid/rust/candid/src/types/subtype.rs`	

Description

The `candid` package contains a recursive function, `TypeEnv::rec_find_type` (figure 4.1), that could be rewritten to use iteration. Using recursion unnecessarily makes a program vulnerable to a stack overflow.

```
44    pub fn rec_find_type(&self, name: &str) -> Result<&Type> {
45        let t = self.find_type(name)?;
46        match t.as_ref() {
47            TypeInner::Var(id) => self.rec_find_type(id),
48            _ => Ok(t),
49        }
50    }
```

Figure 4.1: Definition of `TypeEnv::rec_find_type` (`candid/rust/candid/src/types/type_env.rs#44–50`)

Also note that the function does not protect against infinite loops/recursion. The function could do so by, for example, storing the hash of each name processed in a set, and consulting the set before iterating/recursing.

We observed stack overflows in `TypeEnv::rec_find_type` while fuzzing. However, we have not yet determined whether those stack overflows would be prevented by checks performed in `TypeEnv::rec_find_type`'s callers.

Exploit Scenario

Alice writes a program that uses the `candid` library. Eve discovers a bug that allows her to map a `candid` type variable to itself. Eve triggers the bug in Alice's program, causing it to overflow its stack and crash.

Recommendations

Short term, take the following steps:

- Rewrite `TypeEnv::rec_find_type` to use iteration instead of recursion.
- Add protection from infinite loops by, for example, storing the hash of each name processed in a set.

Taking these steps will help protect `candid` clients against denial-of-service attacks.

Long term, fuzz the `candid` package's functions regularly. This issue was found through fuzzing. Fuzzing regularly could help to expose similar issues.

5. The IDL allows for recursive cyclic types which should not be allowed

Severity: **Undetermined**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-CANDID-5
Target: `candid` IDL	

Description

While the Candid specification states that type cycles must be productive (figure 5.1), it is possible to create a record type with a cyclic reference to itself, thus making it a non-productive type.

An example of a self-referential type is shown in figure 5.2. Additionally, if one uses the `didc` tool to generate a random value of such a type, the tool crashes due to a stack overflow (as shown in figure 5.3), since it tries to create an infinitely long cycle of references.

```
Type definitions are mutually recursive, i.e., they can refer to themselves or each
other. However, every type cycle must be productive, i.e., go through a type
expression that is not just an identifier. A type definition that is vacuous, i.e.,
is only equal to itself, is not allowed.
```

Figure 5.1: Excerpt from the Candid IDL specification

```
type A = record { a : A };
service : { test : (A) -> (int32) query }
```

Figure 5.2: An example IDL file that defines a non-productive recursive type cycle

```
$ ./target/debug/didc random -d ./example.did -t '(A)'

thread 'main' has overflowed its stack
fatal runtime error: stack overflow
[1]    34219 abort      ./target/debug/didc random -d ./example.did -t '(A)'
```

Figure 5.3: Crash of the didc tool when we try to randomize a value of the A type defined in figure 5.2

Recommendations

Short term, change the Candid implementation so that it disallows the creation of record types with non-optional cycles to themselves (also consider longer cycles like A->B->C->A). The compiler could suggest that the cycle be fixed by specifying an optional reference, since it would be possible to create instances of such types. For example, the type from figure 5.2 could be fixed as: `type A = record { a : opt A };`.

6. Stack overflow in encoding/serialization path

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CANDID-6
Target: `candid` encoding	

Description

Encoding a long record cycle causes recursive calls that lead to a stack overflow, which causes the `Encode!` function to crash. This may lead to denial of service for programs that serialize user data using Candid into types that contain cycles. Figure 6.1 shows example test code that triggers this case.

Also note that Candid performs stack exhaustion checks via the `check_recursion!` macro, but while it is used in the `Decode!` functionality, it is missing from the `Encode!` path.

```
#[test]
fn test_recursive_type_encoding() {
    #[derive(Debug, CandidType, Deserialize)]
    struct A {
        a: Box<Option<A>>,
    }

    let mut data = A { a: Box::new(None) };
    for _ in 0..5000 {
        data = A { a: Box::new(Some(data)) };
    }

    Encode!(&data).unwrap();
}
```

Figure 6.1: Test that causes the Encode! function to crash due to a stack overflow

Running this test under a debug mode results in the following stack trace, shown in figure 6.2.

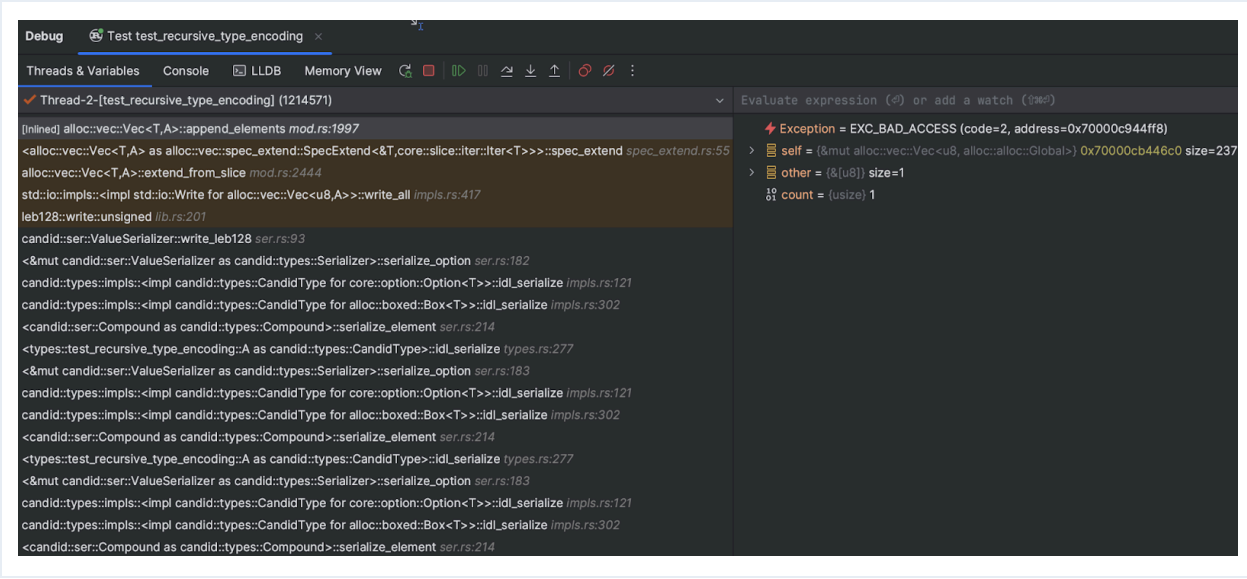

Figure 6.2: Screenshot from RustRover IDE when the stack overflow crash happens in the test from figure 6.1. Test run on MacOS.

Exploit Scenario

A web service uses Candid to serialize data sent as JSON into a record type with cyclic reference. An attacker leverages this fact and sends a JSON with a big linked structure causing the web service to crash.

Recommendations

Short term, fix the stack overflow crash in the Encode! functionality. This can be done either by adding the `check_recursion!` checks to the encoding code paths, or by changing the encoding algorithm to use loops instead of recursion (although this would not be trivial).

Long term, add the test from figure 6.1 to the Candid types tests and change it so it expects that the `Encode!` call errors out with a “Recursion limit exceeded at depth X” error (in case the code is fixed by adding the recursion checks mentioned in the short-term recommendation).

7. The fuzzing harnesses do not build

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-CANDID-7
Target: `candid/rust/candid/fuzz`	

Description

The fuzzing harnesses available in the `candid/rust/candid/fuzz` directory are not building in the audited codebase. One of the reasons for this is that the `candid-fuzz` crate refers to a non-existent feature called `parser` of the `candid` crate (figure 7.1).

```
features = ["parser"]
```

Figure 7.1: `candid-fuzz`'s reference to the non-existent `candid` `parser` feature (`candid/rust/candid/fuzz/Cargo.toml#18`)

Recommendations

Short term, fix the fuzzing harnesses so they build properly.

Long term, run the fuzzing harnesses regularly, at least before each new release of the Candid project.

8. The float32/float64 infinite signs are displayed incorrectly

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-CANDID-8
Target: `candid/rust/candid/src/pretty/candid.rs`	

Description

The Candid's `fmt::Debug` trait implementation for its `IDLValue` type displays the `+inf` and `-inf` values of its float types (`float32` and `float64`) by appending a ".0" to them, which seems unexpected and incorrect. This can be confirmed with the test from figure 8.1.

```
#[test]
fn test_floats_inf() {
    let f: f32 = "inf".parse().unwrap();
    // Note: A valid test after the bug is fixed should not include the ".0" part
    assert_eq!(format!("{:?}", IDLValue::Float32(f)), "inf.0 : float32");

    let f: f32 = "-inf".parse().unwrap();
    // Note: A valid test after the bug is fixed should not include the ".0" part
    assert_eq!(format!("{:?}", IDLValue::Float32(f)), "-inf.0 : float32");
}
```

Figure 8.1: An example test that shows how +inf and -inf floats are displayed

This issue is caused by the `f.trunc()` path in the `number_to_string` function (figure 8.2).

```
pub fn number_to_string(v: &IDLValue) -> String {
    match v {
        ...
        Float32(f) => {
            if f.trunc() == *f {
                format!("{f}.0")
            } else {
                f.to_string()
            }
        }
        Float64(f) => {
            if f.trunc() == *f {
                format!("{f}.0")
            } else {
                f.to_string()
            }
```

Figure 8.2: The `number_to_string` function (candid/rust/candid/src/pretty/candid.rs#L297–L310)

Additionally, the `didc` tool cannot encode special float values (`+inf`, `-inf`, and `NaN`), neither with the ".0" suffix nor without it, as shown in figure 8.3.

```
$ ./target/debug/didc encode -t '(float32)' '(inf : float32)'
error: parser error
   ┌─ candid arguments:1:2
   │
1  │ (inf : float32)
   │  ^^^ Unexpected token
   │
  = Expects one of "(", ")", "blob", "bool", "decimal", "float", "func",
    "hex", "null", "opt", "principal", "record", "service", "sign",
    "text", "variant", "vec"

error: invalid value '(inf : float32)' for '[ARGS]': Candid parser error:
Unrecognized token `Id("inf")` found at 1:4
Expected one of "(", ")", "blob", "bool", "decimal", "float", "func", "hex", "null",
"opt", "principal", "record", "service", "sign", "text", "variant" or "vec"

For more information, try '--help'.

$ ./target/debug/didc encode -t '(float32)' '(inf.0 : float32)'
error: parser error
   ┌─ candid arguments:1:2
   │
1  │ (inf.0 : float32)
   │  ^^^ Unexpected token
   │
  = Expects one of "(", ")", "blob", "bool", "decimal", "float", "func",
    "hex", "null", "opt", "principal", "record", "service", "sign",
    "text", "variant", "vec"
```

Figure 8.3: The didc tool cannot encode special float values.

Recommendations

Short term, fix the display of the `-inf` and `+inf` `float32` and `float64` values so that they do not include the ".0" suffix. Also, consider supporting those values as well as the `NaN` value in the `didc encode` tool.

Long term, extend the test suite to test for the cases described in this finding.

9. Incorrect arithmetic

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CANDID-9
Target: `candid` arithmetic operators	

Description

Certain arithmetic operators (minus, divide, modulo) for the `Nat` and `Int` types when the first argument of the operator is a built-in type are flawed. The operators produce values as though the operators' arguments are passed in an inverted order.

This issue can be seen in existing tests that check for the incorrect results for those operators (figure 9.1).

```
let x: $t = 1;
let value = <$res>::from(x + 1);
/* ... */
assert_eq!(x + value.clone(), 3);
assert_eq!(x - value.clone(), 1);   // 1-2 == 1
assert_eq!(x * value.clone(), 2);
assert_eq!(x / value.clone(), 2);   // 1/2 == 2
assert_eq!(x % value.clone(), 0);   // 1%2 == 0
```

Figure 9.1: Existing (passing) test case checking arithmetic of `Nat/Int` with flawed expected values (`candid/rust/candid/tests/number.rs#L98-L116`)

Exploit Scenario

A developer writes code that uses the flawed arithmetic in a financial operation, assuming it works correctly. An attacker finds that fact and exploits it to gain some funds.

Recommendations

Short term, fix the implementation of arithmetic operators and the relevant test assertions.

Long term, review test cases and expected values to mitigate human error. Where possible, attempt to automate tests so they do not rely on human-provided values.

10. Inadequate recursion checks

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-CANDID-10
Target: `candid/rust/candid/src/de.rs`	

Description

The de module uses a `check_recursion` macro (figure 10.1) to avoid overflowing the stack when deserializing. However, the macro is not used outside of the de module, so stack overflows can still occur.

```
 202    macro_rules! check_recursion {
 203        ($this:ident $($body:tt)*) => {
 204            $this.recursion_depth += 1;
 205            match stacker::remaining_stack() {
 206                Some(size) if size < 32768 => return
Err(Error::msg(format!("Recursion limit exceeded at depth {}",
$this.recursion_depth))),
 207                None if $this.recursion_depth > 512 => return
Err(Error::msg(format!("Recursion limit exceeded at depth {}. Cannot detect stack
size, use a conservative bound", $this.recursion_depth))),
 208                _ => (),
 209            }
 210            let __ret = { $this $($body)* };
 211            $this.recursion_depth -= 1;
 212            __ret
 213        };
 214    }
```

Figure 10.1: Definition of the `check_recursion` macro (`candid/rust/candid/src/de.rs#202–214`)

The issue is that function stack frame sizes differ. Thus, a function with a small stack frame could recurse N times and not overflow the stack. However, the use of that function could be followed by a call to a function with a larger stack frame that, when recursing a similar number of times, does overflow the stack. If the `check_recursion` macro is used only in the former function and not in the latter, a panic will result instead of an error.

For example, `IDLArgs`' `Debug` implementation (figure 10.2) is recursive. However, it does not use the `check_recursion` macro. Moreover, experiments suggest it has a larger stack frame size than `candid`'s deserialization functions. Thus, a program could deserialize an `IDLArgs` value and try to print its `Debug` representation, resulting in a panic.

```
250    impl fmt::Debug for IDLArgs {
251        fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
252            if self.args.len() == 1 {
253                write!(f, "({:?})", self.args[0])
254            } else {
255                let mut tup = f.debug_tuple("");
256                for arg in self.args.iter() {
257                    tup.field(arg);
258                }
259                tup.finish()
260            }
261        }
262    }
```

Figure 10.2: Recursive function not involved in deserialization (candid/rust/candid/src/pretty/candid.rs#250–262)

Appendix E gives an example test showing that the just described scenario is achievable.

Exploit Scenario

Alice writes a program that uses the `candid` library. Alice's program deserializes values from untrusted sources and prints their `Debug` representations when an error occurs. Eve uses the technique described above to crash Alice's program.

Recommendations

Short term, adapt `IDLArgs`' `Debug` and `Display` implementations to use the `check_recursion` macro. Similarly adapt other functions that are known to be recursive and that act on deserialized data. Doing so will eliminate potential denial-of-service vectors.

Long term, take the following steps:

- Fuzz the `candid` package's functions regularly. This issue was found by running one of the existing fuzzing harnesses. Fuzzing regularly could help to expose similar issues.
- Avoid recursion except where specifically necessary. Recursion can lead to stack overflows and can introduce denial-of-service vectors. (See also TOB-CANDID-4.)

11. TypeId::of functions could be optimized into a single function

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-CANDID-11
Target: `rust/candid/src/types/internal.rs`	

Description

The `candid` library uses the address of a function to identify a type. However, there is no guarantee that the compiler will not optimize those functions into a single function. Hence, the technique could stop working at any time.

The relevant code appears in figure 11.1. The general technique appears to have been introduced in rust-lang/rust issue #41875. However, several problems with the technique are pointed out in that issue, including the one in this finding's title.

```
15    impl TypeId {
16        pub fn of<T: ?Sized>() -> Self {
17            let name = std::any::type_name::<T>();
18            TypeId {
19                id: TypeId::of::<T> as usize,
20                name,
21            }
22        }
23    }
```

Figure 11.1: `TypeId` implementation (`candid/rust/candid/src/types/internal.rs#15–23`)

Note that the `TypeId` in figure 11.1 includes a `name` field, which the original proposal did not. The inclusion of the `name` field provides some protection against optimization. However, the compiler could, in principle, optimize out that field if it is unread. Even if the field is read, type names are meant only for diagnostic purposes, and there is no guarantee that they differ across types. Thus, the compiler could collapse `TypeId::of` for two types meant to be distinct.

Exploit Scenario

Alice writes a program that uses the `candid` library. A future version of the compiler collapses the `TypeId::of` functions for certain types that Alice's program uses. Alice's program no longer deserializes values correctly.

Recommendations

Short term, take the following steps:

- Conspicuously document that the `candid` library relies on an unsound `TypeId` implementation. Doing so will alert users of the possibility that the library may behave incorrectly on certain types.
- Develop thorough unit, property, and possibly fuzzing tests to check for distinct types whose `TypeId::of` functions are not distinguished by the compiler. Doing so could help alert you to types on which the `candid` library may behave incorrectly.

Long term, keep abreast of proposals to develop a non-static `TypeId`. If such a proposal is adopted, consider switching to it. Doing so could provide a solution that would not be undermined by the compiler.

References

- rust-lang/rust issue #41875: Tracking issue for non_static_type_id
- Pre-RFC: non-footgun non-static TypeId

12. Deserialization correctness depends on the thread in which operations are performed

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-CANDID-12
Target: `candid/rust/candid/src/types/{internal.rs, type_env.rs}`	

Description

The `candid` library uses a thread local `ENV` map to record the types that "knot" IDs map to. Because the map is thread local, it could fail to be populated in a thread that performs deserialization, causing deserialization of a knot to fail.

`ENV`'s declaration appears in figure 12.1. There is no public function to populate `ENV` outright. Rather, `ENV` is populated as a side effect of calling the trait method `CandidType::ty` (figures 12.2 and 12.3).

```
 629    thread_local! {
 630       static ENV: RefCell<HashMap<TypeId, Type>> =
RefCell::new(HashMap::new());
```

Figure 12.1: Declaration of ENV (candid/rust/candid/src/types/internal.rs#629–630)

```
 30    fn ty() -> Type {
 31       let id = Self::id();
 32       if let Some(t) = self::internal::find_type(&id) {
 33           match *t {
 34               TypeInner::Unknown => TypeInner::Knot(id).into(),
 35               _ => t,
 36           }
 37       } else {
 38           self::internal::env_add(id.clone(), TypeInner::Unknown.into());
 39           let t = Self::_ty();
 40           self::internal::env_add(id.clone(), t.clone());
 41           self::internal::env_id(id, t.clone());
 42           t
 43       }
 44    }
```

Figure 12.2: Trait method `CandidType::ty` (candid/rust/candid/src/types/mod.rs#34–48)

```
646     pub(crate) fn env_add(id: TypeId, t: Type) {
647         ENV.with(|e| drop(e.borrow_mut().insert(id, t)));
648     }
```

Figure 12.3: Function `env_add`, which is called by `CandidType::ty` (`candid/rust/candid/src/types/internal.rs#646–648`)

A thread can construct a type that implements `CandidType` without calling `CandidType::ty`. If that thread then tries to deserialize that type, the deserialization could fail. Appendix F gives an example demonstrating the problem.

Note: We were able to trigger the bug using `IDLDeserialize::get_value_with_type`, but not otherwise. That method is guarded by the `value` feature flag, and thus was out of scope of the audit. For that reason, we have given the finding informational severity. If there was a way to trigger the bug without requiring optional features, the finding's severity would be high.

Exploit Scenario

Alice writes a program that uses the `candid` library. Alice's program uses multiple threads and deserializes values from untrusted sources. Eve discovers a code path that requires Alice's program to perform deserialization without calling `CandidType::ty`. Eve uses the code path to crash Alice's program.

Recommendations

Short term, conspicuously document that the `candid` library is not thread safe. Moreover, users should not rely on data passed between threads, even when Rust's type system allows it. Doing so will reduce the risk of users misusing `candid`'s API.

Long term, eliminate the implicit requirement that `CandidType::ty` be called for deserialization to work correctly. Adopt an API that makes the population of `ENV` more explicit. For example, rather than have ENV be a thread local variable, consider storing it inside of a "context" object. When serializing or deserializing, a user could be required to pass such a context object. By making users responsible for managing such context objects and their respective ENV fields, users are less likely to be surprised by failures at inopportune times.

13. External GitHub CI actions versions are not pinned

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-CANDID-13
Target: GitHub Actions	

Description

The GitHub Actions pipelines used in `candid` and `ic-transport-types` (`agent-rs` repository) use several third-party actions versions that are not pinned to a commit but to a branch or a release tag that can be changed. These actions are part of the supply chain for CI/CD and can execute arbitrary code in the CI/CD pipelines. A security incident in any of the above GitHub accounts or organizations can lead to a compromise of the CI/CD pipelines and any artifacts they produce or any secrets they use.

The following actions are owned by GitHub organizations that might not be affiliated directly with the API/software they are managing:

- EmbarkStudios/cargo-deny-action@v1
- South-Paw/action-netlify-deploy@v1.0.4 [archived]
- actions-rs/cargo@v1 [archived]
- actions-rs/toolchain@v1 [archived]
- actions/cache@v2
- actions/checkout@master, v1, v2, v3
- actions/download-artifact@v3
- actions/setup-node@v3
- actions/upload-artifact@v3
- boa-dev/criterion-compare-action@master
- cachix/install-nix-action@v12
- unsplash/comment-on-pr@v1.2.0
- ructions/cargo@v1
- svenstaro/upload-release-action@v2

Note that we included GitHub actions from the actions organization owned by GitHub even though Dfinity already implicitly trusts GitHub by virtue of using their platform. However, if any of their repositories gets hacked, it may impact the Dfinity's CI builds.

Exploit Scenario

A private GitHub account with write permissions for one of the untrusted GitHub actions is taken over by social engineering. For example, a user uses an already-leaked password and

is convinced to send a 2FA code to the attacker. The attacker updates the GitHub actions and puts a backdoor in the release artifacts produced by those actions.

Recommendations

Short term, pin all external and third-party actions to a Git commit hash. Avoid pinning to a Git tag as these can be changed after creation. We also recommend using the pin-github-action tool to manage pinned actions. GitHub dependabot is capable of updating GitHub Actions that use commit hashes.

Long term, audit all pinned actions or replace them with a custom implementation. Also, consider updating archived GitHub actions to active ones.

14. Inconsistent support for types in operators

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-CANDID-14
Target: `candid/rust/candid/src/types/number.rs`	

Description

Only a subset of expected functions are implemented for the combination of `Nat` and `i32` types. As a result, the operators work only if `Nat` is exactly at the left side of the operator. As a result, certain code that could be assumed to compile, does not compile. This is demonstrated in figures 14.1–2, where the `test_i32_Nat` test function does not compile.

```
// This test compiles:
#[test]
fn test_Nat_i32() {
    let l : Nat = <Nat>::from(1);
    let r : i32 = 1;
    let _ = l + r;
}

// This test fails to compile:
#[test]
fn test_i32_Nat() {
    let l : i32 = 1;
    let r : Nat = <Nat>::from(1);
    let _ = l + r;
}
```

Figure 14.1: Two functions calculating sum of two variables of types i32 and Nat in two different orders

```
error[E0277]: cannot add `candid::Nat` to `i32`
  --> rust/candid/tests/playground.rs:74:15
   |
   |     let _ = l + r;
   |               ^ no implementation for `i32 + candid::Nat`
```

Figure 14.2: Compilation error from the code from figure 14.1

When an operator exists for two types, it is expected to work both ways; therefore, both tests are expected to pass. The plus operator shown in figure 14.1 is only an example, and the described issue occurs with other operators as well.

Recommendations

Short term, implement all operators for `Nat` and `i32`. Consider implementing them for other integer types, as support for only `i32` is inconsistent.

Long term, write methodically comprehensive tests for every function. Cover all edge cases and potential errors

15. Recursion checks do not ensure stack frame size

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-CANDID-15
Target: `candid/rust/candid/src/de.rs`	

Description

The `candid` package implements the `check_recursion` macro, which checks for recursion depth and whether the remaining stack size is lower than 32768 bytes. However, there is no guarantee that the stack frame of a function that will be called within the scope of the `check_recursion` macro will not be bigger than 32768 bytes. If this happens, the check would fail to fit its purpose.

The severity of this finding is informational because it is unlikely that a stack frame would exceed 32768 bytes. However, we still flag it since this is neither guaranteed nor checked by the code.

```
#[cfg(not(target_arch = "wasm32"))]
macro_rules! check_recursion {
    ($this:ident $($body:tt)*) => {
        $this.recursion_depth += 1;
        match stacker::remaining_stack() {
            Some(size) if size < 32768 => return Err(Error::msg(format!("Recursion
limit exceeded at depth {}", $this.recursion_depth))),
            None if $this.recursion_depth > 512 => return
Err(Error::msg(format!("Recursion limit exceeded at depth {}. Cannot detect stack
size, use a conservative bound", $this.recursion_depth))),
            _ => (),
        }
        let __ret = { $this $($body)* };
        $this.recursion_depth -= 1;
        __ret
    };
}
```

Figure 15.1: candid/rust/candid/src/de.rs#L201–L214

Recommendations

Long term, find a way to check the stack frame sizes of functions called within the `check_recursion` macro and ensure that the stack frame size is never greater than what `check_recursion` checks for. There exists a LLVM flag, `emit-stack-sizes`, that may help achieve this, although it is unstable. Additionally, document any meaningful

information around stack frame sizes so that future readers of the `check_recursion` macro have more information about its assurances.

16. Misleading error message

Severity: **Informational**	Difficulty: **N/A**
Type: Error Reporting	Finding ID: TOB-CANDID-16
Target: `ic-transport-types/src/{request_id.rs, request_id/error.rs}`	

Description

The length of a vector is checked to be exactly 32, but whenever it is not, a length parity error message is displayed. The message does not describe the error. Additionally, value 32 is a magic number and should be replaced with a named constant.

```
let vec = hex::decode(from).map_err(RequestIdFromStringError::FromHexError)?;
if vec.len() != 32 {
    return Err(RequestIdFromStringError::InvalidSize(vec.len()));
}
```

Figure 16.1: Vector length tested to be exactly 32 in `from_str` function and returning `RequestIdFromStringError::InvalidSize` with misleading message (see figure 16.2) whenever the length is not 32 (ic-transport-types/src/request_id.rs#L85–L87)

```
/// The string was not of a valid length.
#[error("Invalid string size: {0}. Must be even.")]
InvalidSize(usize),
```

Figure 16.2: Error message informing about incorrect string size with additional, unnecessary comment about number's parity (ic-transport-types/src/request_id/error.rs#L8–L10)

Recommendations

Short term, update the error message to accurately describe the issue and replace the value 32 with a named constant.

Long term, use error names that describe errors in detail; if length parity is a concern, add it to the error name instead of using more general naming like `InvalidSize`. Also, make sure that error messages do not contain any misleading text.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Non-Security-Related Findings

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

- **Remove or clarify the following statement in the Candid spec:**

  > *…a service of type T is* upgradable *to a version with another type T' if and only if T' is a structural subtype of T, written T' <: T.*

  Subsequent text makes clear that whether `T'` must be a structural subtype of T or vice versa depends on whether data of those types is outbound or inbound. But the above statement, as written, suggests that the former is always required.

- **Remove the following clause from the Candid spec:**

  > *where NI* is the <nat> sequence 1..|<datatypeNI>*|, respectively.*

  The notation `NI*` does not appear to be used in the spec.

- **Change "an" to "a" in the following three sentences in the Candid spec:**

  > *T maps an Candid type to a byte sequence representing that type.*
  >
  > *M maps an Candid value to a byte sequence representing that value.*
  >
  > *R maps an Candid value to the sequence of references contained in that value.*

- **Make `candid`'s `value` feature enable `candid`'s `printer` feature.** Building with just the former produces errors like in figure C.1.

```
info: running `cargo check --no-default-features --features value` on candid (7/9)
    Checking candid v0.9.100
(/Users/sam.moelius/gh/trailofbits/audit-dfinity-candid/rust/candid)
error[E0277]: `IDLValue` doesn't implement `std::fmt::Display`
   --> rust/candid/src/types/value.rs:263:41
    |
263 |                         "type mismatch: {self} can not be of type {t}"
    |                                         ^^^^^^ `IDLValue` cannot be formatted
with the default formatter
    |
    = help: the trait `std::fmt::Display` is not implemented for `IDLValue`
    = note: in format strings you may be able to use `{:?}` (or {:#?} for
pretty-print) instead
```

```
    = note: this error originates in the macro `$crate::__export::format_args` which
comes from the expansion of the macro `format` (in Nightly builds, run with -Z
macro-backtrace for more info)

error[E0277]: `IDLValue` doesn't implement `std::fmt::Display`
   --> rust/candid/src/types/value.rs:269:37
    |
269 |                     "type mismatch: {self} cannot be of type {t}"
    |                                     ^^^^^^ `IDLValue` cannot be formatted with
the default formatter
    |
    = help: the trait `std::fmt::Display` is not implemented for `IDLValue`
    = note: in format strings you may be able to use `{:?}` (or {:#?} for
pretty-print) instead
    = note: this error originates in the macro `$crate::__export::format_args` which
comes from the expansion of the macro `format` (in Nightly builds, run with -Z
macro-backtrace for more info)
```

Figure C.1: Error messages produce when one build with `--no-default-features --features value` (candid/rust/candid/fuzz/Cargo.toml#18)

Note that problems like these can be discovered with cargo-hack.

- **Move the maximum header length (500) to a constant since it is hard-coded in the following two places:**

```
impl<'de> IDLDeserialize<'de> {
    /// Create a new deserializer with IDL binary message.
    pub fn new(bytes: &'de [u8]) -> Result<Self> {
        let de = Deserializer::from_bytes(bytes).with_context(|| {
            if bytes.len() <= 500 {
                format!("Cannot parse header {}", &hex::encode(bytes))
        ...
}

pub fn new_with_config(bytes: &'de [u8], config: Config) -> Result<Self> {
    let mut de = Deserializer::from_bytes(bytes).with_context(|| {
        if config.full_error_message || bytes.len() <= 500 {
            format!("Cannot parse header {}", &hex::encode(bytes))
    ...
}
```

Figure C.2: Hard-coded maximum header length for the header to be printed (candid/rust/candid/src/de.rs#L32–L43)

Note that the value determines the maximum number of header bytes printed when an error occurs.

- **If you create a common "utilities" crate, define the `idl_hash` function there, and use it throughout the codebase.** Currently, the function is defined in both

`candid` and `candid_derive`. The former includes a comment to "remember to update the same function in `candid_derive`". Defining the function once in a common crate would eliminate the need for code duplication.

- **Remove the commented-out debug lines from the codebase.** These lines should not be included in the file, as it appears that their only purpose is debugging.

```
248    //println!("{}", __export_service());
249    //assert!(false);
```

Figure C.3: Unnecessary comments (candid/rust/candid/tests/types.rs#248–249)

- **Eliminate the call to `drop` in figure C.4.** The call is unnecessary.

```
646    pub(crate) fn env_add(id: TypeId, t: Type) {
647        ENV.with(|e| drop(e.borrow_mut().insert(id, t)));
648    }
```

Figure C.4: Unnecessary call to `drop` (candid/rust/candid/src/types/internal.rs#646–648)

- **Eliminate the call to `as_bytes` in figure C.5.** The call is unnecessary.

```
334    self.fields
335        .push((Sha256::digest(key.as_bytes()).into(), hash));
```

Figure C.5: Unnecessary call to `as_bytes` (agent-rs/ic-transport-types/src/request_id.rs#334–335)

- **Rewrite the code in figure C.6 as in figure C.7.** The C.7 version is more succinct, but has the same effect.

```
264    let mut s = self.serialize_struct(name, 1)?;
265    SerializeStruct::serialize_field(&mut s, variant, value)?;
266    SerializeStruct::end(s)
```

Figure C.6: Code that could be written more succinctly (agent-rs/ic-transport-types/src/request_id.rs#264–266)

```
264    let mut s = self.serialize_struct(name, 1)?;
265    s.serialize_field(variant, value)?;
266    s.end()
```

Figure C.7: More succinct version of the code in figure C.6 (agent-rs/ic-transport-types/src/request_id.rs#264–266)

- **Replace the magic value in figure C.8 with a named constant.**

```
65    let mut signable = Vec::with_capacity(43);
```

Figure C.8: It is not clear from where number 43 comes from (agent-rs/ic-transport-types/src/request_id.rs#65)

D. Patch Extending Arithmetic Operator Tests

This appendix includes a patch that we developed during the audit to test for certain cases related to finding TOB-CANDID-9. We recommend adding this patch to the `candid` repository to increase the test coverage of its code. The patch can be applied after saving it to a file with the following command executed in the repository directory:

```
git am <patch-file>
```

Additionally, this appendix describes certain recommendations to further improve the tests so that they would cover more areas of the code.

Arithmetic operator check
The following patch adds additional test cases for arithmetic operators for `Nat` and `Int` structures. We explicitly test the comparison operators (`>`, `<`, `<=`, `>=`, `==`, `!=`), the compound assignment operators (`+=`, `-=`, `*=`, `/=`, `%=`) and similar operators to ensure that all of them work correctly and to increase the overall coverage of the tests. Also, since some operators and conversion constructor are implemented for `Nat` and `i32`, those types are added in a few places.

Additionally, to avoid human errors (like testing for incorrect behavior or results), we modified the `random_operator_test` macro to compute the expected results based on arithmetic on basic types instead of having hard-coded results. The random number generator we used is initialized with a constant seed for reproducibility.

While we modified the `operators` test function to include more test cases, we recommend splitting it into multiple test functions so that each function tests either one operator or a group of operators. As an example, we developed the `conversion_constructors` test function that tests exactly one functionality.

Errors (like overflows, underflows, dividing by zero, and conversion errors) are not covered by the patch and deserve additional tests to be tested more thoroughly.

In the future, we recommend taking the following actions:

- Explicitly test every function and operator.
- Test with all supported types, if possible.
- In addition to manually generated tests, develop algorithmically generated, random tests (i.e., property tests).
- Test all edge cases and exceptions.
- Create small test cases that check single functionalities instead of many at once.

The full patch is shown below:

```
From fc3787ce7685a8887a76cec2bcdd8f4ec4e4b8df Mon Sep 17 00:00:00 2001
From: Dominik Klemba <Dominik.Klemba@trailofbits.com>
Date: Fri, 3 Nov 2023 05:43:25 +0100
Subject: [PATCH] Additional test cases for numbers

This commit adds new tests to `tests/number.rs` test file.
Significant new tests:
- Conversion between types (Nat, Int, i32).
- Operator tests for Nat and i32.
- Comparison (ge, le) tests.
- Operator tests for Nat and Int.
- Compound assignment operators tests.

This commit also fixes incorrect expected results in existing test cases.
Those tests don't cover all cases. Negative numbers and
handling overflows or dividing by zero should be tested more.
operators() test function should be split into more tests.
---
 rust/candid/tests/number.rs | 256 ++++++++++++++++++++++++++++++++----
 1 file changed, 234 insertions(+), 22 deletions(-)

diff --git a/rust/candid/tests/number.rs b/rust/candid/tests/number.rs
index 48baffa..0286214 100644
--- a/rust/candid/tests/number.rs
+++ b/rust/candid/tests/number.rs
@@ -1,6 +1,7 @@
 use candid::types::leb128::*;
 use candid::{Int, Nat};
 use num_traits::cast::ToPrimitive;
+use rand::{rngs::StdRng, Rng, SeedableRng};

 #[test]
 fn test_numbers() {
@@ -93,32 +94,243 @@ fn random_u64() {
 #[allow(clippy::cmp_owned)]
 #[test]
 fn operators() {
-    macro_rules! test_type {
+    macro_rules! basic_test_1 {
+        // A handwritten test, confirming correctness of basic mathematical operations.
+        // Operators have to work for Nat with i32.
         ($res: ty, $( $t: ty )*) => ($(
-            let x: $t = 1;
-            let value = <$res>::from(x + 1);
-
-            assert_eq!(value.clone() + x, 3);
-            assert_eq!(value.clone() - x, 1);
-            assert_eq!(value.clone() * x, 2);
-            assert_eq!(value.clone() / x, 2);
-            assert_eq!(value.clone() % x, 0);
-
-            assert_eq!(x + value.clone(), 3);
-            assert_eq!(x - value.clone(), 1);
-            assert_eq!(x * value.clone(), 2);
-            assert_eq!(x / value.clone(), 2);
-            assert_eq!(x % value.clone(), 0);
+            let x: $t = 1;
+            let value = <$res>::from(x + 1);
+
+            assert_eq!(value.clone() + x, 3);
+            assert_eq!(value.clone() - x, 1);
+            assert_eq!(value.clone() * x, 2);
+            assert_eq!(value.clone() / x, 2);
+            assert_eq!(value.clone() % x, 0);
+
+            assert_eq!(x + value.clone(), 3);
+            assert_eq!(x - value.clone(), 1);
+            assert_eq!(x * value.clone(), 2);
+            assert_eq!(x / value.clone(), 2);
+            assert_eq!(x % value.clone(), 0);
+
```

```
-             assert!(value.clone() < <$res>::from(x + 2));
-             assert!(<$res>::from(x + 2) > value.clone());
-             assert!(x < <$res>::from(x + 2));
-             assert!(<$res>::from(x + 2) > x);
+             let x1: $t = 1;
+             let x2: $t = 2;
+             let value : $res = <$res>::from(x1 + 1);
+
+             assert_eq!(value.clone() + x1, 3);
+             assert_eq!(value.clone() - x1, 1);
+             assert_eq!(value.clone() * x2, 4);
+             assert_eq!(value.clone() / x2, 1);
+             assert_eq!(value.clone() % x1, 0);
+
+             // To check != on a basic example
+             assert_ne!(value.clone() + x1, 2);
+             assert_ne!(value.clone() - x1, 2);
+             assert_ne!(value.clone() * x2, 0);
+             assert_ne!(value.clone() / x2, 2);
+             assert_ne!(value.clone() % x1, 1);
+
+             let mut value_copy = value.clone();
+             value_copy += x1;
+             assert_eq!(value_copy, 3);
+             value_copy = value.clone();
+             value_copy -= x1;
+             assert_eq!(value_copy, 1);
+             value_copy = value.clone();
+             value_copy *= x2;
+             assert_eq!(value_copy, 4);
+             value_copy = value.clone() * 3;
+             value_copy /= x2;
+             assert_eq!(value_copy, 3);
+             value_copy = value.clone();
+             value_copy %= x1;
+             assert_eq!(value_copy, 0);
+
+             assert!(value.clone() < <$res>::from(x1 + 2));
+             assert!(value.clone() <= <$res>::from(x2));
+             assert!(<$res>::from(x1 + 2) > value.clone());
+             assert!(<$res>::from(x2) >= value.clone());
+
+             let y: $t = 2;
+             assert_eq!(value.clone(), y);
+         )*)
+     }
+     macro_rules! basic_test_2 {
+         // A handwritten test, confirming correctness of basic mathematical operations.
+         // Operators don't have to work for Nat with i32.
+         ($res: ty, $( $t: ty )*) => ($(
+             let x1: $t = 1;
+             let x3: $t = 3;
+             let x4: $t = 4;
+             let value : $res = <$res>::from(x1 + 1);
+
+             assert_eq!(x1 + value.clone(), 3);
+             assert_eq!(x3 - value.clone(), 1);
+             assert_eq!(x3 * value.clone(), 6);
+             assert_eq!(x4 / value.clone(), 2);
+             assert_eq!(x3 % value.clone(), 1);
```

```
+
+            assert!(x1 < <$res>::from(x1 + 1));
+            assert!(x3 <= <$res>::from(x3));
+            assert!(<$res>::from(x1 + 1) > x1);
+            assert!(<$res>::from(x3) >= x3);
+
+            let y: $t = 2;
+            assert_eq!(y, value.clone());
+        )*)
+    }
+    macro_rules! basic_test_for_structures {
+        // A handwritten test, confirming correctness of basic mathematical operations.
+        // Works when the second argument is a structure.
+        ($res: ty, $( $t: ty )*) => ($(
+            let r1 : $t = <$t>::from(1);
+            let l1 : $res = <$res>::from(r1.clone());
+            let r2 : $t = <$t>::from(2);
+            let l2 : $res = <$res>::from(r2.clone());
+
+            assert_eq!(l1 , r1);
+            assert!(l1 == r1);
+            assert_ne!(l1 , r2);
+            assert!(l1 != r2);
+            assert!(l1 <= r1);
+            assert!(l1 < r2);
+            assert!(l1 <= r2);
+            assert!(r1 >= l1);
+            assert!(r2 > l1);
+            assert!(r2 >= l1);
+
+            assert_eq!(l1.clone() + r1.clone(), 2);
+            assert_eq!(l2.clone() - r1.clone(), 1);
+            assert_eq!(l2.clone() * r2.clone(), 4);
+            assert_eq!(l2.clone() / r1.clone(), 2);
+            assert_eq!(<$res>::from(5) % r2.clone(), 1);
+
+            let mut m = <$res>::from(0);
+            m += r1.clone();
+            assert_eq!(m, 1);
+            m *= <$t>::from(8);
+            assert_eq!(m, 8);
+            m -= r2.clone();
+            assert_eq!(m, 6);
+            m /= r2.clone();
+            assert_eq!(m, 3);
+            m %= r2.clone();
+            assert_eq!(m, r1);
         )*)
     }
+    macro_rules! random_operator_test {
+        // Random test for operators, runs for 1000 random pairs of numbers.
+        ($res:ty, $t:ty, $n_min:expr, $n_max:expr) => {
+            let seed_bytes : [u8; 32] = [1u8; 32];
+            let mut rng = StdRng::from_seed(seed_bytes);

-    test_type!( Nat, usize u8 u16 u32 u64 u128 );
-    test_type!( Int, usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 );
+            for _ in 0..1000 {
+                let a: $t = rng.gen_range($n_min..$n_max);
+                let b: $t = rng.gen_range($n_min..$n_max);
```

```
+                let ra : $res = <$res>::from(a);
+                let rb : $res = <$res>::from(b);
+
+                if a < b {
+                    assert!(ra < rb);
+                    assert!(ra < b);
+                    assert!(a < rb);
+                } else if a > b {
+                    assert!(ra > rb);
+                    assert!(ra > b);
+                    assert!(a > rb);
+                } else {
+                    assert!(ra <= rb);
+                    assert!(ra <= b);
+                    assert!(a <= rb);
+                    assert!(ra >= rb);
+                    assert!(ra >= b);
+                    assert!(a >= rb);
+                    assert_eq!(ra, rb);
+                    assert_eq!(ra, b);
+                    assert_eq!(a, rb);
+                }
+
+                assert_eq!(ra.clone() + rb.clone(), a.clone() + b.clone());
+                if a >= b || std::any::TypeId::of::<$res>() == std::any::TypeId::of::<Int>() {
+                    assert_eq!(ra.clone() - rb.clone(), a.clone() - b.clone());
+                    assert_eq!(ra.clone() - b.clone(), a.clone() - b.clone());
+                    assert_eq!(a.clone() - rb.clone(), a.clone() - b.clone());
+                } else {
+                    assert_eq!(rb.clone() - ra.clone(), b.clone() - a.clone());
+                    assert_eq!(rb.clone() - a.clone(), b.clone() - a.clone());
+                    assert_eq!(b.clone() - ra.clone(), b.clone() - a.clone());
+                }
+                assert_eq!(ra.clone() * rb.clone(), a.clone() * b.clone());
+                assert_eq!(ra.clone() * b.clone(), a.clone() * b.clone());
+                assert_eq!(a.clone() * rb.clone(), a.clone() * b.clone());
+                if b != 0 {
+                    assert_eq!(ra.clone() / rb.clone(), a.clone() / b.clone());
+                    assert_eq!(ra.clone() / b.clone(), a.clone() / b.clone());
+                    assert_eq!(a.clone() / rb.clone(), a.clone() / b.clone());
+                    assert_eq!(ra.clone() % rb.clone(), a.clone() % b.clone());
+                    assert_eq!(ra.clone() % b.clone(), a.clone() % b.clone());
+                    assert_eq!(a.clone() % rb.clone(), a.clone() % b.clone());
+                }
+
+                let mut value_copy : $res = ra.clone();
+                value_copy += rb.clone();
+                assert_eq!(value_copy, a.clone() + b.clone());
+                if a >= b || std::any::TypeId::of::<$res>() == std::any::TypeId::of::<Int>() {
+                    value_copy = ra.clone();
+                    value_copy -= rb.clone();
+                    assert_eq!(value_copy, a.clone() - b.clone());
+                }
+                value_copy = ra.clone();
+                value_copy *= rb.clone();
+                assert_eq!(value_copy, a.clone() * b.clone());
+                if b != 0 {
+                    value_copy = ra.clone();
+                    value_copy /= rb.clone();
+                    assert_eq!(value_copy, a.clone() / b.clone());
```

```
+                    value_copy = ra.clone();
+                    value_copy %= rb.clone();
+                    assert_eq!(value_copy, a.clone() % b.clone());
+                }
+
+                value_copy = ra.clone();
+                value_copy += b.clone();
+                assert_eq!(value_copy, a.clone() + b.clone());
+                if a >= b || std::any::TypeId::of::<$res>() == std::any::TypeId::of::<Int>() {
+                    value_copy = ra.clone();
+                    value_copy -= b.clone();
+                    assert_eq!(value_copy, a.clone() - b.clone());
+                }
+                value_copy = ra.clone();
+                value_copy *= b.clone();
+                assert_eq!(value_copy, a.clone() * b.clone());
+                if b != 0 {
+                    value_copy = ra.clone();
+                    value_copy /= b.clone();
+                    assert_eq!(value_copy, a.clone() / b.clone());
+                    value_copy = ra.clone();
+                    value_copy %= b.clone();
+                    assert_eq!(value_copy, a.clone() % b.clone());
+                }
+            }
+        };
+    }
+
+    basic_test_2!( Nat, usize u8 u16 u32 u64 u128 );
+    basic_test_2!( Int, usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 );
+    basic_test_1!( Nat, usize u8 u16 u32 u64 u128 i32 ); // additionally i32
+    basic_test_1!( Int, usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 );
+    basic_test_for_structures!( Nat, Nat );
+    basic_test_for_structures!( Int, i32 Int );
+    random_operator_test!(Int, i64, -10, 10); // to generate collisions
+    random_operator_test!(Int, i64, -1_0000_000, 1_000_000);
+    random_operator_test!(Nat, u64, 0, 10); // to generate collisions
+    random_operator_test!(Nat, u64, 0, 1_000_000);
+}
+
+#[allow(clippy::cmp_owned)]
+#[test]
+fn conversion_constructors() {
+    macro_rules! from_basic_types {
+        ($res: ty, $( $t: ty )*) => ($(
+            let r : $t = 1;
+            let _ = <$res>::from(r.clone());
+        )*)
+    }
+    macro_rules! from_structures {
+        ($res: ty, $( $t: ty )*) => ($(
+            let r : $t = <$t>::from(1);
+            let _ = <$res>::from(r.clone());
+        )*)
+    }
+    from_basic_types!( Nat, usize u8 u16 u32 u64 u128 i32 );
+    from_basic_types!( Int, usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 );
+    from_structures!( Nat, i32 Nat );
+    from_structures!( Int, i32 Nat Int );
 }
```

```
 fn check(num: &str, int_hex: &str, nat_hex: &str) {
--
2.34.1
```

Figure D.1: Patch extending tests for arithmetic operators

E. Test Demonstrating Recursion Checks' Inadequacy

This appendix contains a test to demonstrate the inadequacy of `candid`'s recursion checks, described in TOB-CANDID-10. On our machine, running this test with `RUST_MIN_STACK=6000000` produces a stack overflow. Intuitively, the functions involved in deserialization have small enough stack frames that the stack does not exceed its allocation. However, the functions related to `Debug` formatting have larger stack frames and cause the stack to exceed its allocation. Note that using `Display` formatting instead of `Debug` formatting produces the same result.

This program was tested on a Digital Ocean instance running Ubuntu 22.04.3 LTS. It was compiled with rustc 1.73.0 (cc66ad468 2023-10-03).

```
#[test]
fn stack_overflow() {
    if std::env::var("RUST_MIN_STACK").is_err() {
        <_ as std::io::Write>::write(
            &mut std::io::stderr(),
            "Try setting `RUST_MIN_STACK=6000000`.\n".as_bytes(),
        )
        .unwrap();
    }
    match IDLArgs::from_bytes(CRASH) {
        Ok(v) => {
            // With `RUST_MIN_STACK` set as above, commenting out the next line
            // causes the stack overflow to go away.
            let _ = format!("{:?}", v);
        }
        Err(e) => {
            dbg!(e);
        }
    };
}

#[cfg(test)]
static CRASH: &[u8] = &[
    68, 73, 68, 76, 1, 110, 0, 43, 0, 122, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
```

```
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
];
```

Figure E.1: Test demonstrating that `check_recursion` is not used in all places where it is needed

F. Test Demonstrating that Candid is Not Thread Safe

This appendix contains a test to demonstrate that deserialization can fail if certain operations are not performed in the right threads (see TOB-CANDID-12).

The test constructs a `Type` without calling `CandidType::ty`, to prevent the main thread's `ENV` map from being populated. Then, the test calls `encode_one` in a separate thread, which sends the resulting bytes back to the main thread over a channel. Finally, when the main thread tries to deserialize those bytes, a panic results because the main thread's `ENV` map has not been populated.

As mentioned in TOB-CANDID-12, we were able to trigger the bug using `IDLDeserialize::get_value_with_type` (yellow in figure F.1), but not otherwise.

```
#[test]
fn test_deserialize_with_type() {
    use candid::types::{Field, Label, TypeId};
    use std::rc::Rc;
    #[derive(CandidType, Eq, PartialEq, serde::Deserialize)]
    enum A {
        A,
        B(B),
    }
    #[derive(CandidType, Eq, PartialEq, serde::Deserialize)]
    struct B {
        a: Box<A>,
    }
    let ty = Type(Rc::new(TypeInner::Record(vec![Field {
        id: Rc::new(Label::Named("a".to_owned())),
        ty: Type(Rc::new(TypeInner::Knot(TypeId::of::<A>()))),
    }])));

    // Uncommenting either of the next two lines makes the panic go away.
    // dbg!(A::ty());
    // dbg!(B::ty());
    dbg!(&ty);

    // Changing the next `true` to `false` makes the panic go away.
    let bytes = if true {
        let (tx, rx) = std::sync::mpsc::channel();
        std::thread::spawn(move || {
            tx.send(candid::encode_one(&B { a: Box::new(A::A) }).unwrap())
                .unwrap();
        });
        rx.recv().unwrap()
    } else {
        candid::encode_one(&B { a: Box::new(A::A) }).unwrap()
    };
```

```
    let mut deserializer = candid::de::IDLDeserialize::new(&bytes).unwrap();

    // The next call tries to unwrap `None`.
    let result = deserializer.get_value_with_type(&candid::TypeEnv::new(), &ty);

    let _ = dbg!(result);
}
```

Figure F.1: Test demonstrating that deserialization success depends on what operations are performed in which threads

G. Fix Review Results

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.

On April 11, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Dfinity team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 16 issues described in this report, Dfinity has resolved four issues, has partially resolved one issue, and has not resolved the remaining eleven issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Unmaintained dependency in candid_parser	Unresolved
2	Insufficient linter use	Partially Resolved
3	Imprecise errors	Unresolved
4	Unnecessary recursion	Unresolved
5	The IDL allows for recursive cyclic types which should not be allowed	Unresolved
6	Stack overflow in encoding/serialization path	Unresolved
7	The fuzzing harnesses do not build	Resolved
8	The float32/float64 infinite signs are displayed incorrectly	Resolved
9	Incorrect arithmetic	Resolved

10	Inadequate recursion checks	**Unresolved**
11	TypeId::of functions could be optimized into a single function	**Unresolved**
12	Deserialization correctness depends on the thread in which operations are performed	**Unresolved**
13	External GitHub CI actions versions are not pinned	**Unresolved**
14	Inconsistent support for types in operators	**Resolved**
15	Recursion checks do not ensure stack frame size	**Unresolved**
16	Misleading error message	**Unresolved**

Detailed Fix Review Results

TOB-CANDID-1: Unmaintained dependency in candid_parser
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Will deprecate later this year*

TOB-CANDID-2: Insufficient linter use
Partially resolved in Clippy PR#492 because Clippy reported additional warnings and potential fixes that can be addressed.

The client provided the following context for this finding's fix status:

> *Not all linter suggestions fit our use case. We fixed the ones that can be applied.*

TOB-CANDID-3: Imprecise errors
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk.*

TOB-CANDID-4: Unnecessary recursion
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. The compiler already excludes the infinite loop case.*

TOB-CANDID-5: The IDL allows for recursive cyclic types which should not be allowed
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. The decoder can handle the empty record type, such as type A = record {A} just fine. didc random is not used in production.*

TOB-CANDID-6: Stack overflow in encoding/serialization path
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk*

TOB-CANDID-7: The fuzzing harnesses do not build
Resolved in PR#489, which updates the candid crate feature to include fix fuzzers. After that change, we are able to build and run harnesses (`parser` and `type_decoder`).

TOB-CANDID-8: The float32/float64 infinite signs are displayed incorrectly
Resolved in 9bf9379 Release candid 0.10. A check to confirm if values are finite numbers (`is_finite()`) has been added before appending a suffix. Infinite values (both, negative and positive) are no longer followed by ".0".

However, we still recommend including the test from figure 8.1 in the codebase (with the ".0" part removed, as suggested in the test comment).

TOB-CANDID-9: Incorrect arithmetic
Resolved in fix ops for u64 (op) Nat/Int (PR#490).

Both the arithmetic issue and tests have been fixed so that correct results are returned and verified. However, we still recommend extending the tests as detailed in D. Patch Extending Arithmetic Operator Tests.

The operators that we originally identified as working incorrectly are no longer affected by this issue.

TOB-CANDID-10: Inadequate recursion checks
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk*

TOB-CANDID-11: TypeId::of functions could be optimized into a single function
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk*

TOB-CANDID-12: Deserialization correctness depends on the thread in which operations are performed
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix.*
>
> *`IDLDeserialize::get_value_with_type` is guarded by a feature flag, and not used in production*

TOB-CANDID-13: External GitHub CI actions versions are not pinned
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk*

TOB-CANDID-14: Inconsistent support for types in operators
Resolved in ed5e921 Release candid 0.10. To ensure consistent behavior, support for problematic pairs of types has been removed. This means that operators now function in a consistent way, regardless of the order in which numbers are used, and do not work with previously problematic types at all.

TOB-CANDID-15: Recursion checks do not ensure stack frame size
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk*

TOB-CANDID-16: Misleading error message
Unresolved. The client accepted the risk and provided the following context for this finding's fix status:

> *Won't fix. Low risk*

H. Fix Review 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.