DFINITY Internet Computer

Security Assessment

June 2, 2021

Prepared For:
Robin Kunzler | *DFINITY*
robin.kunzler@dfinity.org

Prepared By:
Fredrik Dahlgren | *Trail of Bits*
fredrik.dahlgren@trailofbits.com

Brad Larsen | *Trail of Bits*
brad.larsen@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Executive Summary

From March 29 to April 23, 2021 and from May 10 to May 31, 2021, DFINITY engaged Trail of Bits to perform a security assessment of the DFINITY Internet Computer and its software and dependencies. The assessment was conducted over 26 person-weeks by 4 engineers. Trail of Bits used techniques including a manual code review, static analysis (using Semgrep), and fuzz testing via cargo-fuzz and proptest; we also leveraged Cargo plug-ins such as cargo-audit, cargo-geiger, cargo-outdated, and cargo-deny.

We began the audit by familiarizing ourselves with the codebase and the extensive documentation provided by DFINITY. We then started reviewing the implementations of the cryptographic components included under `dfinity/rs/crypto`, as well as the canister execution environment, the WebAssembly runtime, the handling of ingress messages and routing, and the data validation performed by the HTTP endpoints. We also attempted to integrate the static analysis tools Mirai and Prusti into the codebase. However, because neither tool was mature enough to support the common Rust constructs used throughout the code, this attempt was unsuccessful.

We then shifted our focus to the Network Nervous System (NNS) and the ledger and governance canisters. As the ledger canister is a central part of the NNS, we sought to ensure that the implementation was robust and properly secured against external attackers. We began by focusing on the public API exposed by the ledger, ensuring that there were proper access controls on it. Additionally, we reviewed the code for synchronization issues and assessed the correctness of the archiving process.

The governance canister also exercises significant privileges, controlling voting, the proposal process, and neuron management. We focused on reviewing its management of the neuron life cycle, specifically the creation, splitting, and disbursement processes, and checked for synchronization issues related to inter-canister calls. When reviewing the proposal and voting implementations, we looked for ways in which a user could gain an unfair advantage.

During the fifth week, we concluded our review of the cryptographic components under `dfinity/rs/crypto` and pivoted to the use of cryptography throughout the system, seeking to ensure that the defined primitives were used correctly and securely.

During the sixth and seventh weeks, we assessed the registry and cycles minting canisters. We reviewed the access controls on the former canister's API, traced registry update-related state transitions, and reviewed its interactions with the registry client. We assessed the cycles minting canister's interactions with the ledger, looking for synchronization issues and reviewing the use and release of locks. We also analyzed the

Rust dependencies of the crates under `dfinity/rs`, using a number of Cargo plug-ins; the results of this review are detailed in Appendix C.

In the seventh week of the audit, Trail of Bits reviewed the ledger hardware wallet developed by Zondax. We also began our assessment of the peer-to-peer and gossip protocols, focusing on denial-of-service attack vectors. In the last week, in addition to concluding that protocol review, we revisited the governance and ledger canisters and reviewed the genesis token canister; specifically, we looked for privacy/access control issues and reviewed the genesis token canister's interactions with the governance canister.

Trail of Bits identified 34 issues ranging from high to informational severity. The three high-severity issues stemmed from the NNS canister implementations. The first issue, related to the archiving implementation in the ledger canister, could cause the ledger to forget balances during an upgrade if archiving has been enabled. The second, involving a lack of data validation in the disbursement of neurons, could enable a controlling principal to claim rewards multiple times. The final high-severity issue was related to the synchronization of the cycles minting canister's state and could cause a cycles minting canister to become perpetually locked.

The second and third high-severity issues are an indication of the inherent complexity of the Internet Computer, in which seemingly small issues can constitute high-severity vulnerabilities because of the system components' interoperability. To mitigate these types of issues, we suggest that DFINITY develop small self-contained components with a limited set of well-defined, well-documented interfaces for the Internet Computer.

We also recommend that DFINITY reduce the system's interdependencies to simplify the security properties of the codebase. For example, implementing the solution for the issue in TOB-DFINITY-021 (involving the locking of a canister's state) made it easier to reason about the security properties of the cycles minting canister without considering the ledger.

Finally, because of the recent public launch of the Internet Computer, parts of the codebase developed very rapidly during the assessment. Rapid changes make it difficult for both developers and auditors to assess the security of the code and can impede the process of keeping security requirements and corresponding invariants up to date. As a result, we recommend performing additional security assessments once the codebase has stabilized.

Project Dashboard

Application Summary

Name	DFINITY Internet Computer

Versions	`dfinity` `ad036a2 (May 21)` `4b59b78 (May 11)` `42ce95e (April 13)` `e8988f7 (April 9)` `c696e34 (March 9)`
	`infra` `77a2b41`
	`agent-js` `1fdba2f`
	`agent-rs` `e77bcd2`
	`candid` `fd841ed`
	`Zondax/ledger-dfinity` `0a02ac8`
	`Zondax/ledger-dfinity-rs` `26ce194`

Types	Rust, C, and TypeScript
Platform	Linux

Engagement Summary

Dates	March 29–April 23 and May 10–May 31, 2021
Method	Full knowledge
Consultants Engaged	4
Level of Effort	26 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■■■■
Total Medium-Severity Issues	4	■■■■
Total Low-Severity Issues	15	■■■■■■■■■■■■■■■
Total Informational-Severity Issues	10	■■■■■■■■■■
Total Undetermined-Severity Issues	1	■
Total	34	

Category Breakdown

Access Controls	2	■■

Auditing and Logging	1	■
Cryptography	1	■
Data Exposure	1	■
Data Validation	22	■ ■
Denial of Service	2	■ ■
Error Reporting	3	■ ■ ■
Session Management	1	■
N/A	1	■
Total	34	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Sufficient.** Certain system canisters running on the NNS subnet expose privileged functionalities that must be protected by proper access controls. We reviewed the ledger, governance, registry, cycles minting, and genesis token canisters, with a focus on the proper implementation of access controls for each canister. We identified two issues (TOB-DFINITY-017 and TOB-DFINITY-037) related to the implementation of access controls in the NNS. Neither issue presents any threat to the system or the corresponding canisters.
Arithmetic	**Moderate.** Integer arithmetic issues are easy to overlook in Rust, simply because developers are likely to assume that they will be handled by the language. However, overflow and underflow checks are omitted from release builds by default, which means that these types of issues are prevalent in Rust codebases as well. The same is true of type casting issues; when implemented using as, type casting will silently truncate the input if it does not fit into the given output type. In total, we found four issues (TOB-DFINITY-002, TOB-DFINITY-006, TOB-DFINITY-007, and TOB-DFINITY-008) related to potential integer overflows and underflows in the codebase.
Centralization	**Moderate.** The Internet Computer is decentralized in the sense that it does not require trust in individual node operators. However, some services, notably the NNS system canisters, hold privileged roles in the system. A vulnerability in a system canister could lead to the compromise of the entire network. For example, we found that a maliciously crafted request could cause the cycles minting canister to become perpetually locked, which would ultimately cause all other canisters outside of the NNS to run out of cycles (TOB-DFINITY-021).
Upgradeability	**Strong.** Both the replica firmware and the system canisters support upgrades, and the upgrade mechanisms are defensively written. Since upgrades to these components must be proposed and approved, the system is protected against malicious upgrades.
Function Composition	**Strong.** The entire codebase is cleanly separated into

	well-defined modules and subsystems. It is generally easy to navigate the code, find implemented functionalities, and build and test individual components of the system.
Account Management	**Moderate.** The ledger canister serves as the source of truth for accounts and balances in the system. The code maturity of the ledger is evaluated separately here.
Specification	**Sufficient.** The code is well documented and easy to read, and security invariants are in many cases detailed in comments located above the relevant function or inlined into the code. There are also many unit tests that help describe the intended functionalities of the code. Finally, there is much documentation available in Notion, on Google Drive, and elsewhere online. However, on numerous occasions, we found that the code had developed beyond the design documents available on Google Drive. For example, the behavior detailed in TOB-DFINITY-013 is not described in the design documents. The challenge going forward, then, will be to keep the documentation current and complete as the underlying codebase rapidly develops.
Testing & Verification	**Moderate.** The code has many unit and integration tests, including property-based tests. However, certain of the issues detected during this assessment were not covered by the existing tests (such as those in TOB-DFINITY-028 and TOB-DFINITY-036). We suggest using property-based tests and fuzzing more extensively, particularly for code that deals directly with user-controllable input.

Code Maturity Evaluation of the Ledger Canister

Category Name	Description
Access Controls	**Strong.** We found no issues related to insufficient access controls.
Arithmetic	**Strong.** We found no issues related to faulty arithmetic or integer over- or underflows.
Centralization	**Moderate.** The ledger uses an archiving mechanism to off-load persistent storage from the main canister. When we first reviewed this mechanism, it was not yet fully implemented, and we found a number of issues (TOB-DFINITY-010, (TOB-DFINITY-011, and TOB-DFINITY-012).
Upgradeability	**Moderate.** When first reviewing the ledger canister, we found two issues related to the archiving mechanism's impact on canister upgrades (TOB-DFINITY-011 and TOB-DFINITY-012). However, because the archiving mechanism was not enabled at that time, the issues did not affect the overall functionality of the canister.
Function Composition	**Strong.** The codebase is separated into components and functions with clear purposes. We have no concerns in this area.
Account Management	**Moderate.** We identified one account management issue (TOB-DFINITY-013) regarding the removal of accounts with low balances from the ledger. This behavior appears to be undocumented and could be quite surprising to users.
Specification	**Moderate.** The ledger canister design document provides a good overview of the canister's functional requirements. However, certain sections (like the "Implementation" section detailing the exported methods and the "Feature" section on scalability) appear to be out of date. In general, though, we found the codebase to be very readable, well structured, and well documented.
Testing & Verification	**Strong.** The ledger canister implements an extensive test suite that aims to cover both its core functionalities (like sending transactions, archiving processes, and canister upgrades) and corner cases (like the handling of invalid transactions and invalid requests made to the archiving nodes).

Engagement Goals

The engagement was scoped to provide a security assessment of the DFINITY Internet Computer. The DFINITY hardware wallet developed by Zondax was added to the scope of the project after it had started.

Specifically, we sought to answer the following questions about the Internet Computer:

- Are the Internet Computer interfaces or front-end HTTP endpoints susceptible to a denial of service caused by the transmission of large packets or an excessive number of idle open connections (i.e., a slowloris attack)?
- Is user-provided data sufficiently validated by the Internet Computer HTTP endpoints?
- Is it possible to break the Wasm instrumentation to cause a denial of service against the Wasm execution environment or to execute code on the system without paying for cycles?
- Is it possible to crash the execution environment and cause a denial-of-service attack (e.g., by allocating large amounts of memory from inside a canister)?
- Is the canister system API properly secured against malicious canisters?
- Is it possible to use the upgrade mechanism to avoid the fees associated with the creation of a canister?
- Is it possible to cause panics or denial-of-service conditions in XNet, Set Rules, or the `StateMachine` by sending malformed data?
- Does the `candid` serialization format handle malformed input correctly?
- Does the `ArtifactManager` correctly handle the various state transitions between pools?
- Are cryptographic primitives properly wrapped in cryptographic trait implementations?
- Are the cryptographic traits throughout the codebase used correctly ?
- Do the system canisters implement proper access controls on the NNS subnet?
- Is it possible to steal funds from the ledger canister or to rewind its internal state?
- Is the ledger canister vulnerable to denial-of-service attacks?
- Is proper domain separation implemented for account IDs tracked by the ledger?
- Is it possible to illicitly gain funds by attacking the ICPT-cycle conversion rate?
- Is it possible to create neurons without staking the appropriate number of tokens?
- Is it possible to disrupt the governance process by creating numerous proposals?
- Are the neuron management APIs exposed by the governance canister properly secured?
- Are the neuron life-cycle management APIs properly protected against race conditions and other synchronization issues?
- Is it possible to game the proposal or voting process to create an unfair voting advantage for certain neuron controllers?

- Are voting rewards distributed fairly across neurons?
- Is the governance canister vulnerable to denial-of-service attacks?
- Do the ledger and cycles minting canisters properly take and release locks on the global state?
- Could a failure in the genesis token or ledger canister cause seed round investors or early token holders to lose their stakes?
- Do the peer-to-peer layer and gossip protocols correctly handle malformed input?

Additionally, we sought to answer the following questions about the DFINITY hardware wallet:

- Does the DFINITY hardware wallet use cryptography appropriately?
- Can private key material be leaked from the device?
- Is transaction input parsed correctly and validated securely?
- Are errors handled correctly?
- Are there proper exploit mitigations in place to protect the application against attacks?
- Are the TypeScript and Rust libraries written correctly?

Coverage

Internet Computer interfaces. We manually reviewed the Candid UI, `agent-js`, and the HTTP handler to ensure that the Internet Computer interfaces and front-end HTTP endpoints were not susceptible to denial-of-service or slowloris attacks. We also reviewed the use of CORS headers across the HTTP endpoints. Finally, we traced incoming requests through the components of the system to assess the implementation of message routing and data validation.

Consensus. We read through the existing consensus-related documentation and did not identify any immediate issues. We also manually reviewed the XNet subsystem for correctness. Lastly, we assessed the wrapping of cryptographic primitives (as detailed in the "Cryptography" section below); however, other priorities (primarily the privileged canisters and hardware wallet) kept us from performing a full review of the primitives' use cases within NI-DKG or the handling of signature verification and block validation.

Network Nervous System. We manually reviewed the ledger, governance, registry, cycles minting, and genesis canisters to ensure that they implemented proper access controls on all publicly exposed APIs.

Ledger canister. We read through the existing documentation on the ledger canister and then manually reviewed the canister codebase to ensure that its implementation matched the specification. As the ledger uses a read/write lock to protect the internal state of the canister, we reviewed the code for potential locking issues during inter-canister calls. We also looked for ways to steal funds, add invalid transactions, or roll back the state of the ledger. Since we found issues in the archiving implementation during our first review of the ledger, we focused on that mechanism during our re-review of the canister.

Governance canister. We first reviewed the extensive documentation on the governance system and extracted a number of security invariants from it. We then focused on ensuring that it would not be possible to steal tokens using the neuron management system. This portion of our review covered the processes of spawning new neurons; configuring, disbursing, splitting, and following neurons; making proposals; and registering new votes. We also looked for ways to create new neurons without staking funds. Additionally, we reviewed the proposal management and voting implementations to ensure that we could not game the voting system by spamming it with new proposals or by preventing other neurons from voting on new proposals. Finally, we reviewed the distribution of voting rewards, checking the process against that outlined in the specification.

Registry canister. We manually reviewed the public-facing methods exposed by the registry canister to ensure that internal state changes were implemented correctly. We also reviewed the interactions between the registry client and the registry for general code correctness.

Cycles minting canister. We manually reviewed the cycles minting canister, focusing on its interactions with the ledger canister. Since the cycles minting canister uses a read/write lock to protect the internal state of the canister, we also examined the code for locking and synchronization issues.

Genesis token canister. We manually reviewed the canister's general code correctness, seeking to identify any issues that could cause token holders to lose their stakes.

Cryptography. We reviewed the wrapping of various cryptography libraries inside cryptographic trait providers to ensure their correctness. This coverage primarily included `rs/crypto/internal/crypto_lib`, `rs/crypto/internal/crypto_service_provider`, `rs/crypto/sha256`, `rs/crypto/key_validation`, `rs/crypto/tree_hash`, and `rs/crypto/internal/src`. We dedicated some time to analyzing the codebase's use of the libraries but then shifted our focus to a re-review of the hardware wallet and NNS.

Execution environment. We manually reviewed the Wasm code validation and instrumentation to ensure that it would not be possible to avoid metering. We also implemented a fuzzer based on `cargo-fuzz` to extend the negative test coverage of the Wasm validation flow. (For details, see Appendix B.) We manually reviewed the canister-facing system API to ensure that it was secured against malicious canisters. We also reviewed the code responsible for canister memory management and life-cycle management, mainly looking for ways in which users could evade canister creation payments or execution/memory usage fees. Finally, we used Semgrep to look for potential integer overflows and unexpected integer truncation related to the use of `as`.

Peer-to-peer layer. We manually reviewed the code related to the peer-to-peer and gossip protocol layers (`dfinity/rs/p2p`, `dfinity/rs/http_handler`, and `dfinity/rs/transport`), looking for denial-of-service attack vectors, data validation issues, and opportunities for remote code execution.

Dependencies. We used a number of Cargo plug-ins to review the dependencies of the crates under `dfinity/rs`. The results of this review are provided in Appendix C.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Identify the secrets in the codebase, remove them, and source them from a secret manager like `Vault`.** Additionally, use a tool such as `truffleHog` to identify any high-entropy strings that may actually be relevant secrets, and rebase the repositories to remove the secrets from their Git histories. TOB-DFINITY-001

❑ **Use `usize::checked_sub` to check for underflows in calculations of the actual function index in `validate_export_section`.** TOB-DFINITY-002

❑ **Consider limiting the number of connections that can be opened by a single user to prevent slowloris attacks.** Additionally, ensure that inactive connections cannot be kept alive. TOB-DFINITY-003

❑ **Add a check to the Wasm code validation to verify that the module does not export the "`canister counter_instructions`" symbol.** TOB-DFINITY-004

❑ **Ensure that the `additional_pages` argument provided to `ic0_stable_grow` is validated before calls to `SystemStateAccessor::stable_grow`.** TOB-DFINITY-005

❑ **Add a check to `available_memory` to ensure that `self.memory_allocation()` is greater than `self.memory_usage()` before the return value is computed.** TOB-DFINITY-006

❑ **Use `usize::checked_add` to check for overflows in calculations of the end of a slice in `valid_subslice`.** TOB-DFINITY-007

❑ **Use `usize::checked_add` to check for overflows in calculations of a new payload size in `RequestInPrep::extend_method_payload`.** TOB-DFINITY-008

❑ **Fix the computation of the cut-off index `split_off_ix` in `split_off_older_than`.** One solution would be to move the `break` statement to a separate `else` clause. TOB-DFINITY-009

❑ **Compute the split-off index and ensure that `split_off_older_than` updates the block vector `state.blocks.inner` to contain the newest blocks in the chain and returns the oldest blocks.** TOB-DFINITY-010

❑ **Serialize ledger canister balances together with the blockchain to ensure that balances are correctly restored after an upgrade.** TOB-DFINITY-011

❑ **Consider disabling the archiving mechanism until the issue detailed in TOB-DFINITY-012 has been properly addressed.** The easiest way to do this would be to pass in `None` for the `archive_canister` field in the `LedgerCanisterInitPayload` argument provided to `canister_init`. TOB-DFINITY-012

❑ **Ensure that the ledger account-trimming behavior is explained in both internal and external documentation.** TOB-DFINITY-013

❑ **Ensure that `disburse_amount_doms` is always less than or equal to `neuron.stake_doms()` and that the relevant cached balance is updated immediately after an asynchronous call to `LedgerCanister::transfer_funds` in `Governance::disburse_neuron`.** TOB-DFINITY-014

❑ **Review the codebase's use of panics.** TOB-DFINITY-015

❑ **Remove the restriction on reward amounts to allow for the disbursement of small rewards.** TOB-DFINITY-016

❑ **Use the same string constant for the method name in the definition of `AUTHZ_MAP` and the call to `check_authz_and_log`.** TOB-DFINITY-017

❑ **Update `get_value` so that it returns the right error code.** TOB-DFINITY-018

❑ **Ensure that when a new node is created, the registry canister saves the deserialized `NodeOperatorRecord` as a new variable, updates the node allowance, and creates a new registry update from the updated record.** TOB-DFINITY-019

❑ **Use a hash set rather than a vector to store keys in the call to `fold` when generating the node operator list.** TOB-DFINITY-020

❑ **Have the cycles minting canister check that the transaction amount is equal to or greater than the transaction fee and ensure that the transaction notification fails gracefully if it is not.** Ensure that a write lock cannot be taken on the global state when the canister is handling new top-ups. TOB-DFINITY-021

❑ **Switch the order of the two tests in the condition of the `while` statement in `utf8ndup`.** This will ensure that the pointer is not dereferenced if the input length is 0. TOB-DFINITY-023

❑ **Use randomized stack canaries to protect against active exploitation of a vulnerability.** TOB-DFINITY-024

❑ **Use a resource-intensive pseudorandom function (PRF) that cannot be easily guessed to extend the entropy source to a 32-byte value in `CanisterEnv::new`.** Suitable PRFs are `PBKDF2`, `scrypt`, and `argon2id`. TOB-DFINITY-025

❑ **Check that `outValLen` is at least 54 before reading and writing to offset 53 of the `outVal` buffer.** TOB-DFINITY-026

❑ **Fix the `TryFrom` trait implementations of gossip messages so that they will not panic when provided malformed data.** TOB-DFINITY-028

❑ **Add `getDeviceInfo` to the TypeScript interface.** TOB-DFINITY-032

❑ **Ensure that there is enough space for a null terminator in the output buffer of `base32_encode` and that the function returns an error if there is not.** TOB-DFINITY-033

❑ **Change the `create_port_file(PathBuf, u16)` function such that it returns a `Result` value when successful and returns an `Err` value instead of triggering a `panic!` upon a failure.** TOB-DFINITY-034

❑ **Audit all canister sandbox communication paths that use the `FrameDecoder::decode` function to verify that they will not handle untrusted input.** Consider reducing the maximum message size that can be handled by `FrameDecoder::decode` (perhaps to 2**16, or u16). A lower size limit would prevent a denial of service stemming from excessive memory pressure on the host system. Document the security requirements of the canister sandbox's RPC mechanism, clearly stating that it requires trusted access. TOB-DFINITY-035

❑ **Fix the `TryFrom` trait implementations and other conversion functions mentioned in finding TOB-DFINITY-036 so that they will not panic when provided malformed input.** TOB-DFINITY-036

❑ **If the lack of access controls in the GTC's method `get_account` is unintentional, consider restricting access to the method to only the principal of an account.** TOB-DFINITY-037

❑ **Determine whether `AccountState::transfer`'s behavior is expected and modify it if it is not.** TOB-DFINITY-038

Long Term

❑ **Consider enabling overflow checks in release builds.** Additionally, integrate fuzz testing into the test suite and integration pipeline to uncover unhandled corner cases. TOB-DFINITY-002

❑ **Consider using `Repr::checked_sub` (or perhaps `Repr::saturating_sub`) in the implementation of `std::ops::Sub` for `AmountOf<Unit, Repr>`.** TOB-DFINITY-006

❑ **Consider enabling overflow checks in release builds.** TOB-DFINITY-006, TOB-DFINITY-007, TOB-DFINITY-008

❑ **Use `Result` whenever possible.** TOB-DFINITY-015

❑ **Change the behavior of `is_authorized` such that the function returns `false` if a method name is not included in `AUTHZ_MAP`.** That way, `AUTHZ_MAP` will function as a proper allow list. TOB-DFINITY-017

❑ **Add unit tests to verify that all public methods return the correct error codes upon receipt of invalid input.** TOB-DFINITY-018

❑ **Consider whether it would make sense to enable the system API to lock and unlock the entire canister.** This would allow the execution environment to unlock a canister trapped in a locked state. TOB-DFINITY-021

❑ **Consider the effects of mandating that the ID of a canister's controller be different from the ID of the canister itself.** TOB-DFINITY-022

❑ **Consider using a higher-quality deterministic entropy source when creating new `CanisterEnv` instances.** TOB-DFINITY-025

❑ **Extend the fuzz testing to cover individual components of the codebase.** TOB-DFINITY-026

❑ **Avoid using panicking functions such as `Option::unwrap()` and `Result::unwrap()`.** Instead, use an explicit method of error handling, such as pattern matching. This type of method will leverage Rust's type system to identify places in which operations could fail. TOB-DFINITY-028, TOB-DFINITY-036

❑ **Develop a generic TypeScript ledger interface library that can be shared by all TypeScript interfaces to reduce the need for code reuse.** TOB-DFINITY-032

❑ **Perform threat modeling when designing new services.** This will help you identify trust boundaries and deployment security requirements of the system. TOB-DFINITY-035

Findings Summary

#	Title	Type	Severity
1	Hard-coded secrets in repositories	Data Exposure	Informational
2	Potential integer underflow in validate_export_section	Denial of Service	Low
3	Candid UI susceptible to slowloris attacks	Denial of Service	Low
4	Missing check for exported symbol during Wasm code validation	Data Validation	Informational
5	Weak input validation in ic0_stable_grow	Data Validation	Low
6	Potential integer underflow in CanisterState::available_memory	Data Validation	Informational
7	Potential integer overflow in valid_subslice	Data Validation	Low
8	Potential integer overflow in RequestInPrep::extend_method_payload	Data Validation	Low
9	Erroneous index calculation in split_off_older_than	Data Validation	Low
10	Helper function used in archiving returns new blocks and keeps old blocks in the ledger state	Data Validation	Medium
11	Ledger upgrade mechanism does not account for archived blocks	Data Validation	High
12	Calls to archive_blocks break block-height tracking	Data Validation	High
13	Accounts with low balances are trimmed from the ledger	Session Management	Informational
14	Controller can evade fees and claim extra rewards through neuron disbursement process	Data Validation	High
15	Inconsistent use of panics	Error Reporting	Informational

16	Rewards worth less than transaction fees are discarded during neuron disbursement	Data Validation	Low
17	The function check_caller_authz_and_log returns true if the function name is misspelled	Data Validation	Low
18	Registry method get_value returns the wrong error code for malformed messages	Error Reporting	Low
19	Registry canister fails to update node operator's node allowance when a new node is added	Data Validation	Low
20	The ic-admin subcommand SubCommand::GetNodeOperatorList reports duplicate and stale operator IDs	Data Validation	Low
21	Cycles minting canister can panic with the global state write-locked, thereby becoming perpetually unresponsive	Data Validation	High
22	A self-owned canister cannot be deleted	Data Validation	Informational
23	Potential out-of-bounds read in utf8ndup	Data Validation	Informational
24	Use of static stack canaries provides inadequate protection against adversarial attacks	Auditing and Logging	Low
25	Seeding via Unix epoch generates less entropy than expected	Cryptography	Medium
26	Out-of-bounds access in print_textual	Data Validation	Low
27	Missing data validation in deserialization of gossip protocol messages	Data Validation	Medium
28	Missing getDeviceInfo command in TypeScript interface	N/A	Informational
29	Potential missing null terminator in base32_encode	Data Validation	Informational
30	Unnecessary panics in the http_handler server's initialization	Error Reporting	Informational

31	Risk of denial of service caused by crafted RPC messages in canister_sandbox	Data Validation	Low
32	Widespread lack of data validation in conversion functions	Data Validation	Medium
33	Lack of access controls in the GTC's method get_account	Access Controls	Undetermined
34	Failed calls to GTC method donate_account cause permanent locking of the remaining neurons	Access Controls	Low

1. Hard-coded secrets in repositories

Severity: Informational Difficulty: High
Type: Data Exposure Finding ID: TOB-DFINITY-001
Target: `infra/monitoring/manifests/prometheus/nix/configuration.nix, git history`

Description
The repositories provided by DFINITY contain hard-coded secrets, such as the `service_key` value used in the PagerDuty incident response process.

The Git histories for the `dfinity` and `infra` repositories also contain hard-coded secrets, though it is unclear whether they have been invalidated or are otherwise non-critical.

Trail of Bits identified the following potentially significant hard-coded secrets:

- The PagerDuty `service_key`, currently located at `infra/monitoring/manifests/prometheus/nix/configuration.nix`
- The SSH keys for the IC-OS in the Git history of `infra/nix/overlays/infra.nix`
- The SSH keys in the Git history of `infra/cd/ssh/id_developer`
- The SSH keys in the Git history of `infra/os/ssh/id_developer`
- The SSH keys in the Git history of `infra/os/vagrant/id_developer`
- The `IC_HOST` password in the Git history of `dfinity/ic-os/hostos/ansible/testnet.yml`
- The same password at `ic-os/hostos/ubuntu/user-data_dell-g1_testnet`

Exploitation of this finding would require some form of internal access to the relevant repositories unless they were open source.

Exploit Scenario
Bob is a DFINITY developer with access to all relevant repositories of the Internet Computer. Eve, an attacker, gains access to Bob's account. As a result, Eve is able to exfiltrate both the source code and all hard-coded secrets from the database, which may lead to further compromise of the live production system.

Recommendations
Short term, identify the secrets in the codebase, remove them, and source them from a secret manager like `Vault`. Additionally, use a tool such as `truffleHog` to identify any high-entropy strings that may actually be relevant secrets, and rebase the repositories to remove the secrets from their Git histories.

2. Potential integer underflow in validate_export_section

Severity: Low
Type: Denial of Service
Target: `dfinity/rs/wasm_utils/src/validation.rs`

Difficulty: Medium
Finding ID: TOB-DFINITY-002

Description

The `validate_export_section` function fails to validate that the function index of an export entry is greater than or equal to the number of imports.

```
let actual_fn_index =
    *fn_index as usize - module.import_count(ImportCountType::Function);
let type_index =
    module.function_section().unwrap().entries()[actual_fn_index].type_ref();
```

Figure 2.1: The `validate_export_section` function fails to validate that `fn_index` is greater than the number of imported functions. (`dfinity/rs/wasm_utils/src/validation.rs`)

Since Rust does not compile code with overflow or underflow checks in release mode (see Rust RFC 560), the subtraction of the import number from the function index may underflow and result in a very large value. This would result in an index out-of-bounds issue, causing a panic.

We identified this issue through a manual review and then confirmed it using the Wasm code validation fuzzer. (For details, see Appendix B.)

Exploit Scenario

A malicious user deploys and starts a canister that has an invalid Wasm binary, causing the execution environment to panic.

Recommendations

Short term, use `usize::checked_sub` to check for underflows in calculations of the actual function index.

Long term, consider enabling overflow checks in release builds. Additionally, integrate fuzz testing into the test suite and integration pipeline to uncover unhandled corner cases such as this one.

3. Candid UI susceptible to slowloris attacks

Severity: Low
Type: Denial of Service
Target: `Candid UI`

Difficulty: High
Finding ID: TOB-DFINITY-003

Description
When the canister SDK is started, the canister bootstrap server in the Candid UI is susceptible to a slowloris denial-of-service attack.

When the local node and web server are run with the `dfx start` command, `localhost:8000` is susceptible to a slowloris attack; through this attack, a single machine could open over a thousand connections and maintain them via `keep-alive` requests. When we tested this locally by establishing 1,500 socket connections, the web server took over 60 seconds to respond.

This issue may also affect the HTTP handler node, as the node will accept many connections that can be maintained with `keep-alive` requests; however, during testing, we were not able to launch a slowloris attack that affected its availability from one machine.

Exploit Scenario
Bob operates an instance of the Candid UI. Eve, an attacker, opens many connections to the endpoint served by the Candid UI. The exhaustion of the open connections then causes the canister bootstrap server to become unresponsive.

Recommendations
Short term, consider limiting the number of connections that can be opened by a single user to prevent slowloris attacks. Additionally, ensure that inactive connections cannot be kept alive.

4. Missing check for exported symbol during Wasm code validation

Severity: Informational
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-004
Target: `dfinity/rs/wasm_utils/src/validation.rs`

Description

The `validate_export_section` function fails to check that the module does not export the symbol "`canister counter_instructions`." When the module is instrumented, a global symbol with that same name is added to it.

```
(module
  (func $run (export "run") (result i32)
    (global.get $counter)
  )
  (global $counter
    (export "canister counter_instructions")
    (mut i32) (i32.const 123)
  )
)
```

Figure 4.1: The `validate_export_section` function accepts the Wasm module as valid. During instrumentation, a second "`canister counter_instructions`" global symbol is injected into the compiled module, breaking the module.

If the module already contains a symbol named "`canister counter_instructions`," it will then have two exported symbols with the same name. Because the name of an exported symbol must be unique, the module will be rejected as invalid during compilation.

Recommendations

Short term, add a check to the Wasm code validation to verify that the module does not export the "`canister counter_instructions`" symbol.

5. Weak input validation in ic0_stable_grow

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-005
Target: `dfinity/rs/system_api/src/lib.rs`

Description

The system API's `ic0_stable_grow` function can be used to increase a canister's stable memory. The function does this by calling `SystemStateAccessor::stable_grow`. When this call returns, the function calls the private system API's `SystemApi::update_available_memory` function, which increases the amount of memory available to the canister by the number of pages allocated to it.

```
fn ic0_stable_grow(&mut self, additional_pages: u32) -> HypervisorResult<i32> {
    match &self.api_type {
        ApiType::Start {} => Err(self.error_for("ic0_stable_grow")),
        _ => {
            let native_memory_grow_res =
                self.system_state_accessor.stable_grow(additional_pages)?;
            self.update_available_memory(native_memory_grow_res, additional_pages)
        }
    }
}
```

Figure 5.1: After the amount of stable memory has been updated, the requested number of additional pages is checked against the amount of memory available to the canister. (`dfinity/rs/system_api/src/lib.rs`)

Since the requested number of additional memory pages is validated in `SystemApiImpl::update_available_memory` *after* the amount of stable memory has been updated, the `ic0_stable_grow` API could be used to increase the memory pressure on the node.

```
fn update_available_memory(
    &mut self,
    native_memory_grow_res: i32,
    additional_pages: u32,
) -> HypervisorResult<i32> {
    let additional_bytes =
        ic_replicated_state::num_bytes_from(NumWasmPages::from(additional_pages));
    if native_memory_grow_res != -1 {
        if additional_bytes > self.available_memory {
            return Err(HypervisorError::OutOfMemory);
        }
        self.available_memory -= additional_bytes;
    }
    Ok(native_memory_grow_res)
}
```

Figure 5.2: The number of additional pages is validated in `SystemApiImpl::update_available_memory`. (`dfinity/rs/system_api/src/lib.rs`)

The `SystemStateAccessorDirect` implementation of the `SystemStateAccessor` trait uses a `StableMemory` instance backed by a `Vec<u8>` to represent the state of the stable memory. Since `StableMemory::grow` simply calls `Vec<u8>::resize` to increase the stable memory, the implementation will panic if the node runs out of memory.

Exploit Scenario
A malicious user deploys a number of canisters, each of which attempts to allocate a large amount of memory to itself. Since the amount of memory allocated to them is validated only after the reallocation is performed, the canisters exhaust the memory available to the node. This causes instability and could result in a runtime panic.

Recommendations
Short term, ensure that the `additional_pages` argument provided to `ic0_stable_grow` is validated before calls to `SystemStateAccessor::stable_grow`.

6. Potential integer underflow in CanisterState::available_memory

Severity: Informational
Difficulty: N/A
Type: Data Validation
Finding ID: TOB-DFINITY-006
Target: `dfinity/rs/replicated_state/src/canister_state.rs`

Description
The `CanisterState::available_memory` function is used to determine how much memory a canister can use before exceeding the amount of memory allocated to it.

```
pub fn available_memory(&self) -> NumBytes {
    self.memory_allocation() - self.memory_usage()
}
```

Figure 6.1: If the function is called while the canister is using more memory than was allocated to it, the calculation of its available memory could result in an underflow. (`dfinity/rs/replicated_state/src/canister_state.rs`)

A canister's allocated memory is initialized when the canister is deployed (either installed or upgraded); however, it may increase its memory usage over time (by calling `ic0_stable_grow` to increase the amount of its stable memory, for example). The execution environment (in particular the system API implementation) keeps track of the amount of allocated memory to ensure that it is always less than the amount of available memory.

However, in some cases (in `ic0_stable_grow`, for example), a canister's current memory usage may exceed the amount of memory allocated to it. If `CanisterState::available_memory` were called in such a case, the computation above would result in an underflow and return a large value.

Recommendations
Short term, add a check to `available_memory` to ensure that `self.memory_allocation()` is greater than `self.memory_usage()` before the return value is computed.

Long term, consider using `Repr::checked_sub` (or perhaps `Repr::saturating_sub`) in the implementation of `std::ops::Sub` for `AmountOf<Unit, Repr>`. Also consider enabling overflow checks in release builds.

7. Potential integer overflow in valid_subslice

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-007
Target: `dfinity/rs/system_api/src/lib.rs`

Description

The helper function `valid_subslice` is used to check that a given offset and size actually correspond to a subslice of an existing slice. If so, the subslice is returned to the caller. However, the calculation of `src + len` could cause an overflow when run on a system using a 32-bit architecture, since `usize` is only 32 bits.

```
pub(crate) fn valid_subslice<'a>(
    ctx: &str,
    src: u32,
    len: u32,
    slice: &'a [u8],
) -> HypervisorResult<&'a [u8]> {
    let len = len as usize;
    let src = src as usize;
    if slice.len() < src + len {
        return Err(ContractViolation(format!(
            "{}: src={} + length={} exceeds the slice size={}",
            ctx,
            src,
            len,
            slice.len()
        )));
    }
    Ok(&slice[src..src + len])
}
```

Figure 7.1: The `src + len` operation could cause an overflow when run on a system using a 32-bit architecture. An overflow would cause a panic upon creation of the subslice `&slice[src..src + len]`. (`dfinity/rs/system_api/src/lib.rs`)

If the addition resulted in an overflow, it would mean that the sum of `src + len` was less than `src`; this would cause the function to panic when creating the subslice `&slice[src..src + len]`.

Recommendations

Short term, use `usize::checked_add` to check for overflows in calculations of the end of a slice.

Long term, consider enabling overflow checks in release builds.

8. Potential integer overflow in RequestInPrep::extend_method_payload

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-008
Target: `dfinity/rs/system_api/src/lib.rs`

Description
When `ic0_call_data_append` is called, the given heap data is used to extend the method payload in `RequestInPrep::extend_method_payload`. The function checks that the sum of the payload size and the size of the extension is less than the maximum inter-subnet message size. However, the `current_size` + `size` operation could result in an overflow when run on a system using a 32-bit architecture, since `usize` is only 32 bits.

```
pub(crate) fn extend_method_payload(
    &mut self,
    src: u32,
    size: u32,
    heap: &[u8],
) -> HypervisorResult<()> {

    [...]

    if current_size + size as usize > max_size_intra_subnet.get() as usize {
        Err(
            [...]
        )
    } else {
        let data = valid_subslice("ic0.call_data_append", src, size, heap)?;
        self.method_payload.extend_from_slice(data);
        Ok(())
    }
}
```

Figure 8.1: The `current_size` + `size` operation in `RequestInPrep::extend_method_payload` could cause an overflow when run on a system using a 32-bit architecture. (`dfinity/rs/system_api/src/lib.rs`)

An overflow would most likely cause a panic in `valid_subslice`. (See TOB-DFINITY-007.)

Recommendations
Short term, use `usize::checked_add` to check for overflows in calculations of a new payload size.

Long term, consider enabling overflow checks in release builds.

9. Erroneous index calculation in split_off_older_than

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-009
Target: `dfinity/rs/rosetta-api/canister/src/main.rs`

Description

The `split_off_older_than` function is called to split blocks off from the ledger for archiving. It takes an `std::time::Duration` argument as input and should return a vector containing all blocks in which `now.duration_since(block.timestamp().into()).unwrap()` is greater than the duration value.

To find the cut-off point in a chain of transactions, the function loops through the blocks in `state.blocks.inner` and checks the blocks' timestamps to see whether they are older than the age indicated by the duration. When the function finds the first block older than the given duration, it designates that block's index as the chain's cut-off point.

```
let now = dfn_core::api::now();
let mut split_off_ix = None;
for (ix, block) in state.blocks.inner.iter().enumerate() {
    let elapsed: std::time::Duration = now
        .duration_since(block.timestamp().into())
        .unwrap();
    dfn_core::api::print(format!("block {} elapsed {:?}", ix, elapsed));
    if elapsed > split_off_age {
        split_off_ix = Some(ix);
        break;
    }
}
```

Figure 9.1: The `split_off_ix` is the index of the first block that is older than the given duration. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

However, since blocks in the chain are sorted from oldest to newest, if any transaction is older than the age indicated by the duration, the first block will be too. Thus, the cut-off index will always be either `None` (if all transactions are "younger" than the age indicated by the duration) or `Some(0)` (if any transaction is older than that age).

```
pub fn add_block(&mut self, block: Block) -> Result<BlockHeight, String> {
    [...]
    self.inner.push(block);
    Ok(self.last_block_index())
}
```

Figure 9.2: New blocks are added to the chain by `BlockChain::add_block`. It follows that blocks are sorted based on their timestamps, from oldest to newest. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

This issue, like that in TOB-DFINITY-010, means that in practice, the entire ledger is always archived whenever the `archive_blocks` method is called on the ledger canister.

Recommendations
Short term, fix the computation of the cut-off index, `split_off_ix`. One solution would be to move the `break` statement to a separate `else` clause.

10. Helper function used in archiving returns new blocks and keeps old blocks in the ledger state

Severity: Medium
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-010
Target: `dfinity/rs/rosetta-api/canister/src/main.rs`

Description

The ledger canister API's `archive_blocks` function is used to archive blocks older than a given age. The age is passed to the helper function `split_off_older_than`, which should prune blocks older than that age from the ledger state and return a new vector of blocks to be archived.

```
let mut blocks_to_archive: VecDeque<RawBlock> = split_off_older_than(age)
    .iter()
    .map(|block| block.encode())
    .collect();
```

Figure 10.1: The function `split_off_older_than` is used to obtain a vector of blocks to be archived. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

The `split_off_older_than` function computes a cut-off index, `split_off_ix`, for the vector that contains the blocks in the ledger state. It then returns the second half of the chain (i.e., the blocks after the cut-off index).

```
pub fn split_off_older_than(split_off_age: std::time::Duration) -> Vec<Block> {
    // Compute the cut-off index split_off_ix.
    [...]

    split_off_ix
        .map(|ix| state.blocks.inner.split_off(ix))
        .unwrap_or_default()
}
```

Figure 10.2: Since `Vec::split_off` returns the second half of the vector (which is split by the given index), `split_off_older_than` will return the newer blocks, leaving older blocks in the ledger state. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

However, `Vec::split_off` returns a vector containing all elements of the original vector that have indices greater than or equal to the given index. Since new blocks are added to the end of the `blocks.inner` vector, `split_off_older_than` will actually return all of the blocks that are "younger" than the given age, leaving the older blocks in `blocks.inner`.

Recommendations

Short term, compute the split-off index and ensure that `split_off_older_than` updates the block vector `state.blocks.inner` to contain the newest blocks in the chain and returns the oldest blocks.

11. Ledger upgrade mechanism does not account for archived blocks

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-DFINITY-011
Target: `dfinity/rs/rosetta-api/canister/src/main.rs`

Description

When the ledger canister is upgraded, the ledger blockchain is serialized to stable memory. After the upgrade, the blocks are deserialized, and the transactions from the blocks are used to restore the ledger balances.

```
pub fn encode(&self) -> Vec<u8> {
    let mut bytes = Vec::new();
    bytes.write_u32::<LittleEndian>(Self::VERSION).unwrap();
    self.blocks.encode(&mut bytes).unwrap();
    bytes
}
```

Figure 11.1: The `State::encode` function serializes the blocks in the state but not the corresponding balances. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

The ledger balances are not serialized to stable memory when the canister is upgraded, which could cause the ledger to forget the balances during an upgrade.

This would occur, for example, if some of the blocks in the ledger were archived. An archived block is off-loaded from the ledger blockchain to an archive node and deleted from the ledger canister. Depending on the age provided to the `archive_blocks` method, certain balances in the `State::balances` hashmap may no longer be reflected in the ledger blockchain. If the ledger canister is upgraded, these balances will not be added to the `State::balances` hashmap, which will cause the `account_balance` method to return incorrect values for certain accounts.

```
fn account_balance(account: PrincipalId, sub_account: Option<[u8; 32]>) -> ICPTs {
    let id = account_identifier(account, sub_account).expect("Account creation failed");
    STATE.read().unwrap().balances.account_balance(&id)
}
```

Figure 11.2: The `account_balance` method in the ledger canister returns the balance of a given account. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

Exploit Scenario

The ledger canister API's `archive_blocks` method is called, causing blocks to be archived and removed from the ledger state. Later, when the ledger canister is upgraded, the balances of all serialized blocks are restored to the ledger, effectively reverting the balances of certain accounts to their former values. As a result, some users lose funds, while others may be able to engage in double-spending.

Recommendations
Short term, serialize ledger canister balances together with the blockchain to ensure that balances are correctly restored after an upgrade.

12. Calls to archive_blocks break block-height tracking

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-DFINITY-012
Target: `dfinity/rs/rosetta-api/canister/src/lib.rs`

Description
The ledger state uses the length of the `self.blocks.inner` vector to track blocks' height. However, if `archive_blocks` is called on the ledger canister, some (or all) of the blocks will be split off from `self.blocks.inner`, and the length of the vector will be updated. This will break the internal assumptions on the current block height.

One method affected by this issue is the ledger canister's `tip_of_chain`, which is used by the Rosetta client to obtain the current block height. This method delegates the call to the `BlockChain::last_block_index` function, which is implemented as follows:

```
pub fn last_block_index(&self) -> BlockHeight {
    self.inner.len() as u64 - 1
}
```

Figure 12.1: The `tip_of_chain` method simply returns the index of the last element in the `BlockChain::inner` block vector. (`dfinity/rs/rosetta-api/canister/src/lib.rs`)

The `tip_of_chain` method will, therefore, return different block heights before and after a call to `archive_blocks`. If the entire blockchain is archived (which is currently the default behavior because of `archive_blocks` implementation issues), the subtraction in `BlockChain::last_block_index` will cause an underflow, and the function will return `u64::MAX`.

Another such method is `block`, which is used by the Rosetta client to retrieve new blocks from the ledger. This method takes a block height as an argument and delegates the call to `BlockChain::get`.

```
pub fn get(&self, height: BlockHeight) -> Option<&Block> {
    self.inner.get(usize::try_from(height).unwrap())
}
```

Figure 12.2: The `block` method takes a block height, which is used as an index for the vector `BlockChain::inner`. (`dfinity/rs/rosetta-api/canister/src/lib.rs`)

When `archive_blocks` works as expected, it deletes a prefix of the vector `self.inner` on the `BlockChain`; this means that the `block` method will return different values before and after a call to `archive_blocks`.

Exploit Scenario
A malicious user transfers funds to a victim canister at block height *N* and then calls `notify` to inform the victim canister that a payment has been made.

After a call to `archive_blocks`, the transaction's block height changes to *N - M*. The malicious user can then call `notify` again with the new block height, *N - M*, falsely signaling to the victim canister that a second payment has been made. Since the ledger uses block heights to track notifications, the call to `notify` succeeds.

```
  change_notification_state(block_height, true)
      .expect("There is already an outstanding notification");
```

Figure 12.3: Since the ledger tracks notifications using block heights, a malicious user could call `notify` a second time for the same transaction after `archive_blocks` has been called. (`dfinity/rs/rosetta-api/canister/src/lib.rs`)

Recommendations

Short term, since this issue affects a large portion of the ledger canister, consider disabling the archiving mechanism until the issue has been properly addressed. The easiest way to do this would be to pass in `None` for the `archive_canister` field in the `LedgerCanisterInitPayload` argument provided to `canister_init`.

13. Accounts with low balances are trimmed from the ledger

Severity: Informational
Difficulty: N/A
Type: Session Management
Finding ID: TOB-DFINITY-013
Target: `dfinity/rs/rosetta-api/canister/src/lib.rs`

Description
If the number of ledger accounts with a non-zero balance reaches the `Ledger::maximum_number_of_accounts` value (which defaults to 50 million), the ledger will burn the assets of the accounts with the lowest balances; the number of accounts trimmed is indicated by `Ledger::accounts_overflow_trim_quantity` (which defaults to 100,000).

```
let to_trim = if self.balances.store.len() > self.maximum_number_of_accounts {
    self.balances
        .select_accounts_to_trim(self.accounts_overflow_trim_quantity)
} else {
    vec![]
};

for (balance, account) in to_trim {
    let payment = Transfer::Burn {
        from: account,
        amount: balance,
    };
    self.balances.add_payment(&payment);
    self.blocks
        .add_payment(message, payment, created_at, timestamp)
        .unwrap();
}
```

Figure 13.1: If the number of balances stored by the ledger exceeds the `Ledger::maximum_number_of_accounts` value, balances will be trimmed from the ledger. (`dfinity/rs/rosetta-api/canister/src/lib.rs`)

This behavior could be very surprising to users and, to our knowledge, is not documented.

For reference, according to blockchain.com, there are currently around 55 million Ethereum addresses with a non-zero balance.

Recommendations
Short term, ensure that the account-trimming behavior is explained in both internal and external documentation.

14. Controller can evade fees and claim extra rewards through neuron disbursement process

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-014
Target: `dfinity/rs/nns/governance/src/governance.rs`

Description

Neuron disbursement is implemented by `Governance::disburse_neuron` through the following asynchronous calls to the ledger canister:

1. A call to transfer the disbursement to a certain account
2. A call to transfer the rewards associated with a neuron's maturity to the chosen account
3. A call to burn the accumulated fees (e.g., those charged for rejected proposals)

The respective amounts are calculated as follows:

```
let disburse_amount_doms = disburse
    .amount
    .as_ref()
    .map_or(neuron.stake_doms(), |a| a.doms);
let rewards_amount_doms = neuron.maturity_doms_equivalent;
let fees_amount_doms = neuron.neuron_fees_doms;
```

Figure 14.1: The disbursement amount may be provided by the method payload; if no amount is provided, the value of the cached stake will be used by default. (`dfinity/rs/nns/governance/src/governance.rs`)

The transfers are implemented as follows:

```
// Transfer 1 - Disburse to the chosen account.
let block_height = self
    .ledger
    .transfer_funds(
        disburse_amount_doms - transaction_fee_doms,
        transaction_fee_doms,
        Some(from_subaccount),
        disburse.to_account.clone().unwrap().into(),
        self.env.now(),
    )
    .await?;

// Transfer 2 - Transfer the accumulated maturity.
if rewards_amount_doms > transaction_fee_doms {
    let _ = self
        .ledger
        .transfer_funds(
            rewards_amount_doms,
            0, // Minting transfer don't pay a fee.
            None,
            disburse.to_account.clone().unwrap().into(),
            self.env.now(),
        )
        .await?;
}

// Transfer 3 - Burn the fees.
if fees_amount_doms > transaction_fee_doms {
    let _ = self
        .ledger
        .transfer_funds(
            fees_amount_doms,
            0, // Burning transfers don't pay a fee.
            Some(from_subaccount),
            AccountIdentifier::new(
                ic_base_types::PrincipalId::from(GOVERNANCE_CANISTER_ID),
                None,
           ),
            self.env.now(),
        )
        .await?;
}
```

Figure 14.2: When the third transfer is initiated, there is no guarantee that there will still be enough funds in the account to cover the accrued fees. (`dfinity/rs/nns/governance/src/governance.rs`)

Finally, the cached balances of the neuron are updated.

```
    if disburse_amount_doms >= neuron.cached_neuron_stake_doms {
        neuron.cached_neuron_stake_doms = 0;
    } else {
        neuron.cached_neuron_stake_doms -= disburse_amount_doms;
    }
    neuron.maturity_doms_equivalent = 0;
    neuron.neuron_fees_doms = 0;
```

Figure 14.3: If any of the transfers fail, `Governance::disburse_neuron` will exit early, and the cached balances will not be updated. (`dfinity/rs/nns/governance/src/governance.rs`)

Consider the following scenario: A malicious neuron controller stakes `M` doms and then submits proposals that are subsequently rejected. Since the stake is cached by the governance canister, `neuron.cached_neuron_stake_doms` will be equal to `M`. Then, because the neuron fee amount increases each time a proposal is rejected, `neuron.neuron_fees_doms` will be equal to `N` > `0`. Lastly, since the balance on the ledger is not updated to reflect each rejected proposal, it will also be `M` doms.

When the neuron is dissolved, the neuron controller should be able to reclaim only `M - N` doms, plus any maturity rewards. However, if the controller disburses `M` doms held by the neuron, the first two transfers will succeed, and the ledger will transfer `M` doms to the chosen account. When the governance canister attempts to burn the fees charged for the rejected proposals, there will be no funds left in the account, and the transfer will fail.

Moreover, the first two transfers will succeed before the cached reward amount (`neuron.maturity_doms_equivalent`) is updated. As a result, the neuron controller may be able to disburse the neuron multiple times and claim a maturity reward each time.

Exploit Scenario
A malicious neuron controller creates a new neuron and stakes 10 ICPTs. She then creates proposals that are subsequently rejected, incurring neuron fees in the form of 9 ICPTs. Assume that the total rewards over the lifetime of the neuron are `R > 0`. When the neuron is dissolved, the controller disburses 2 ICPTs from the stake. The first and second transfers (of the disbursement and the maturity rewards) are both successful, sending 2 ICPTs and `R` ICPTs, respectively, to the neuron controller's account. Because there are only 8 ICPTs left in the account, the final transfer—the fee burning transaction—fails.

Since each transfer moves 2 ICPTs from the neuron's account to the account specified by the neuron controller, this process can be repeated five times, resulting in total ICPT rewards of `10 + 5R`.

By transferring more funds to the neuron's account, the controller can repeat this attack. This means that it is possible to claim neuron rewards an unlimited number of times.

Recommendations
Short term, ensure that `disburse_amount_doms` is always less than or equal to `neuron.stake_doms()` and that the relevant cached balance is updated immediately after an asynchronous call to `LedgerCanister::transfer_funds`.

15. Inconsistent use of panics

Severity: Informational
Type: Error Reporting
Target: `dfinity/rs`
Difficulty: N/A
Finding ID: TOB-DFINITY-015

Description
The Rust codebase handles errors in one of two ways: by returning a `Result` with a custom error enum to be handled at a later time or by immediately panicking.

```
_ => Err(CryptoError::InvalidArgument {
    message: format!(
        "Cannot generate key pair for unsupported algorithm: {:?}",
        algorithm_id
    ),
}),
```

Figure 15.1: The `Err` constructor for `Result` (`dfinity/rs/crypto/internal/crypto_service_provider/src/keygen/mod.rs`)

```
fn write_secret_keys_to_disk(sks_data_file: &Path, secret_keys: &SecretKeys) {
    let mut tmp_data_file = sks_data_file.to_owned();
    tmp_data_file.set_file_name(TEMP_SKS_DATA_FILENAME);
    let sks_proto = ProtoSecretKeyStore::secret_keys_to_sks_proto(secret_keys);
    let mut buf = Vec::new();
    match sks_proto.encode(&mut buf) {
        Ok(_) => match fs::write(&tmp_data_file, &buf) {
            Ok(_) => {
                fs::rename(&tmp_data_file, sks_data_file)
                    .expect("Could not update SKS file.")
            }
            Err(err) => panic!("IO error {}", err),
        },
        Err(err) => panic!("Error serializing data: {}", err),
    }
}
```

Figure 15.2: The code panics in response to input/output (IO) errors. (`dfinity/rs/crypto/internal/crypto_service_provider/src/secret_key_store/proto_store.rs`)

IO errors happen occasionally and should be handled gracefully. Currently, the functions that panic are not all labeled as such. (The function `store_secret_key_or_panic` in `keygen/mod.rs` is an example of one that is correctly labeled.)

Recommendations
Short term, review the codebase's use of panics.

Long term, use `Result` whenever possible.

16. Rewards worth less than transaction fees are discarded during neuron disbursement

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-016
Target: `dfinity/rs/nns/governance/src/governance.rs`

Description
When a neuron is disbursed, the maturity reward is discarded if it is worth less than the transaction fee.

```
// Transfer 2 - Transfer the accumulated maturity by minting into the
// chosen account, but only if the value exceeds the cost of a transaction fee
// as the ledger doesn't support ledger transfers for an amount less than the
// transaction fee.
if rewards_amount_doms > transaction_fee_doms {
    let _ = self
        .ledger
        .transfer_funds(
            rewards_amount_doms,
            0, // Minting transfer don't pay a fee.
            None,
            disburse.to_account.clone().unwrap().into(),
            self.env.now(),
        )
        .await?;
}
```

Figure 16.1: According to the comment, small rewards are discarded because the ledger does not support transfers of amounts lower than the related transaction fees. (`dfinity/rs/nns/governance/src/governance.rs`)

A comment in the code claims that the ledger does not support transfers of amounts lower than the associated transaction fees. However, this is true only for burning, not for minting or ordinary transfers.

```
let transfer = if from == minting_acc {
    assert_eq!(fee, ICPTs::ZERO, "Fee for minting should be zero");
    assert_ne!(
        to, minting_acc,
        "It is illegal to mint to a minting_account"
    );
    Transfer::Mint { to, amount }
} else if to == minting_acc {
    assert_eq!(fee, ICPTs::ZERO, "Fee for burning should be zero");
    if amount < MIN_BURN_AMOUNT {
        panic!("Burns lower than {} are not allowed", MIN_BURN_AMOUNT);
    }
    Transfer::Burn { from, amount }
} else {
    [...]
};
let (height, _) = add_payment(memo, transfer, block_height);
```

Figure 16.2: There is no lower limit on minting in the implementation of `send` in the ledger canister. (`dfinity/rs/rosetta-api/canister/src/main.rs`)

This behavior causes neuron controllers to lose rewards and does not appear to be necessitated by the ledger implementation.

Recommendations

Short term, remove the restriction on reward amounts to allow the disbursement of small rewards.

17. The function check_caller_authz_and_log returns true if the function name is misspelled

Severity: Low
Difficulty: N/A
Type: Data Validation
Finding ID: TOB-DFINITY-017
Target: `dfinity/rs/registry/canister/canister.rs`

Description

The NNS canisters use the function `check_authz_and_log` to control access to privileged canister methods. It is typically called with the method name, given as a string literal, as follows:

```
check_caller_authz_and_log("bless_replica_version", LOG_PREFIX)
```

Figure 17.1: The method name, given as a string literal, is used to call *`check_caller_authz_and_log`. (`dfinity/rs/registry/canister/canister.rs`)*

The IDs of public methods' authorized principals are passed to a canister upon initialization as part of the `RegistryCanisterInitPayload` argument. The list of `MethodAuthzInfo` instances is used to build the `AUTHZ_MAP`, which is used to control access to the canister's public API.

```
MethodAuthzInfo {
    method_name: "bless_replica_version".to_string(),
    principal_ids: vec![ic_nns_constants::GOVERNANCE_CANISTER_ID
        .get()
        .to_vec()],
},
```

Figure 17.2: In `MethodAuthzInfo`, the method name is passed to the canister as a string literal upon initialization. (`dfinity/rs/registry/src/init.rs`)

The actual authorization check is delegated to the function `is_authorized`, which obtains the list of authorized principals from `AUTHZ_MAP` and then checks whether the caller is authorized to call the given method.

```
match authz.get(method_name) {
    // Test whether the (string representation) of the principal is registered
    // as having access to this method.
    Some(method_authz) => {
        let authorized = method_authz.principal_ids.contains(&principal_id.to_vec());
        if !authorized {
            println!(
                "Principal: {} not authorized to access method: {}",
                principal_id, method_name
            );
        }
```

```
            authorized
        }
        // If a method is not in access control, then by default it is authorized
        None => true,
    }
```

Figure 17.3: If the method name is misspelled, the `is_authorized` function will return `true`.
(`dfinity/rs/registry/canister/canister.rs`)

If the method name is misspelled in either the definition of `AUTHZ_MAP` or the call to `check_authz_and_log`, then `authz.get(method_name)` will return `None`, and `is_authorized` will return `true`.

Exploit Scenario
A DFINITY developer adds a privileged method to a system canister or refactors an existing privileged method but misspells the method name parameter passed to `check_authz_and_log`. As a result, the method can be called by any canister, which could allow a node or neuron operator to compromise the NNS.

Recommendations
Short term, use the same string constant for the method name in the definition of `AUTHZ_MAP` and the call to `check_authz_and_log`.

Long term, change the behavior of `is_authorized` such that the function returns `false` if a method name is not included in `AUTHZ_MAP`. That way, `AUTHZ_MAP` will function as a proper allow list.

18. Registry method get_value returns the wrong error code for malformed messages

Severity: Low
Type: Error Reporting
Target: `dfinity/rs/registry/canister/canister.rs`

Difficulty: N/A
Finding ID: TOB-DFINITY-018

Description

When encountering a malformed message, the registry method `get_value` returns `Code::KeyNotPresent` rather than the expected error code, `Code::MalformedMessage`.

```
fn get_value() {
    let response_pb = match deserialize_get_value_request(arg_data()) {
        Ok((key, version_opt)) => {
            [...]
        }
        Err(error) => RegistryGetValueResponse {
            error: Some(RegistryError {
                code: Code::KeyNotPresent as i32,
                key: Vec::<u8>::default(),
                reason: error.to_string(),
            }),
            version: 0,
            value: Vec::<u8>::default(),
        },
    };
    [...]
}
```

Figure 18.1: The method `get_value` returns `Code::KeyNotPresent` upon encountering malformed input.

Recommendations

Short term, update `get_value` so that it returns the right error code.

Long term, add unit tests to verify that all public methods return the correct error codes upon receipt of invalid input.

19. Registry canister fails to update node operator's node allowance when a new node is added

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-019
Target: `dfinity/rs/registry/canister/src/mutations/do_add_node.rs`

Description

New nodes are added by the function `Registry::add_node`. To update the node allowance of the corresponding node operator, a new registry update is created as follows:

```
[...]

decode_registry_value::<NodeOperatorRecord>(node_operator_record.clone()).node_allowance -=
    1;
let update_node_operator_record = update(
    node_operator_key.as_bytes().to_vec(),
    encode_or_panic(node_operator_record),
);

[...]
```

Figure 19.1: The encoded node operator record is decoded and updated, but it is not saved.

The `node_operator_record` is a serialized `NodeOperatorRecord` instance. This is deserialized into a new `NodeOperatorRecord` value, which is then updated. However, the newly created record is immediately discarded, and the old serialized record is used to update the registry. This means that the node allowance in the registry is never updated.

A node operator could take advantage of this issue, along with the incorrectly implemented `remove_node` registry method (which increases the node allowance by one when a node is removed) to arbitrarily increase his or her node allowance.

Exploit Scenario

A malicious node provider calls `add_node` on the registry multiple times to add a large number of new nodes to the Internet Computer.

Note, though, that it does not appear that this action would threaten any particular subnet; this is because only the governance canister can add a new node to a subnet in response to an accepted proposal.

Recommendations

Short term, ensure that when a new node is created, the canister saves the deserialized `NodeOperatorRecord` as a new variable, updates the node allowance, and creates a new registry update from the updated record.

20. The ic-admin subcommand SubCommand::GetNodeOperatorList reports duplicate and stale operator IDs

Severity: Low
Difficulty: N/A
Type: Data Validation
Finding ID: TOB-DFINITY-020
Target: `dfinity/rs/registry/client/src/client.rs`

Description

The `ic-admin` subcommand `SubCommand::GetNodeOperatorList` is implemented using the function `RegistryClientImpl::get_key_family`. This function is used to obtain a list of all keys with a given prefix from the cached registry state.

```
SubCommand::GetNodeOperatorList => {
    [...]

    let keys = registry_client
        .get_key_family(
            NODE_OPERATOR_RECORD_KEY_PREFIX,
            registry_client.get_latest_version(),
        )
        .unwrap();

    println!();
    for key in keys {
        let node_operator_id = key.strip_prefix(NODE_OPERATOR_RECORD_KEY_PREFIX).unwrap();
        println!("{}", node_operator_id);
    }
}
```

Figure 20.1: `RegistryClient::get_key_family` is used to obtain a list of all node operators. (`dfinity/rs/registry/admin/src/main.rs`)

The implementation of `RegistryClientImpl::get_key_family` obtains a reference to the cached state (which contains a sorted list with an entry for each registry update). For each entry with a key matching the given prefix, it updates the resultant list, adding the key if the entry corresponds to an update and removing the last key if the entry corresponds to a deletion.

```
// 1. Skip all entries up to the first_match_index
// 2. Filter out all versions newer than the one we are interested in
// 3. Only consider the subsequence that starts with the given prefix
let res = cache_state
    .records
    .iter()
    .skip(first_match_index) // (1)
    .filter(|r| r.version <= version) // (2)
    .take_while(|r| r.key.starts_with(key_prefix)) // (3)
    .fold(vec![], |mut acc, r| {
        // is_set iff the key is set in this transport record.
        let is_set = r.value.is_some();
        // if this record indicates a removal for the last key in the list, we need to
        // remove that key.
        if !is_set && acc.last().map(|k| k == &r.key).unwrap_or(false) {
            acc.pop();
        } else if is_set {
            acc.push(r.key.clone());
        }
        acc
    });
```

Figure 20.2: The function `RegistryClientImpl::get_key_family` cannot handle multiple updates made to the same key. (`dfinity/rs/registry/client/src/client.rs`)

However, if a key has been updated multiple times, there will be multiple entries for it in the cached state. The `RegistryClientImpl::get_key_family` function cannot handle such scenarios and instead adds each entry to the resultant list. Similarly, if a key has been updated multiple times and then deleted, `RegistryClientImpl::get_key_family` will remove the last entry related to that key from the list, leaving stale entries in the list.

Since node operator records are updated each time the operator adds a new node to the network, the lists returned by `RegistryClientImpl::get_key_family` typically contain multiple entries for each node operator. Moreover, if DFINITY introduced a functionality to remove a node operator from the registry, the list could potentially contain stale operator IDs as well.

Recommendations
Short term, use a hash set rather than a vector to store keys in the call to `fold`.

21. Cycles minting canister can panic with the global state write-locked, thereby becoming perpetually unresponsive

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-021
Target: `dfinity/rs/rosetta-api/cycles_minting_canister/src/main.rs`

Description

When the cycles minting canister receives a transaction notification, it checks the memo to determine which type of operation is required (`MEMO_CREATE_CANISTER` or `MEMO_TOP_UP_CANISTER`). In both cases, the canister performs the update (either creating a new canister or topping up an existing canister); it then calls `burn_or_refund` to handle the transferred funds, which are burned if the operation was successful and refunded if not.

```
  if tn.memo == MEMO_CREATE_CANISTER {
      [...]

      let cycles = IcptsToCycles {
          xdr_permyriad_per_icp,
          cycles_per_xdr: STATE.read().unwrap().cycles_per_xdr,
      }
      .to_cycles(tn.amount);

      let res = create_canister(controller, cycles).await;

      let refund_block = burn_or_refund(
          res.is_ok(),
          CREATE_CANISTER_REFUND_FEE,
          &tn,
          &ledger_canister_id,
      )
      .await?;

     [...]

     Ok(TransactionNotificationResult::encode(res)?)
  } else if tn.memo == MEMO_TOP_UP_CANISTER {
      [...]

      let cycles = IcptsToCycles {
          xdr_permyriad_per_icp,
          cycles_per_xdr: STATE.write().unwrap().cycles_per_xdr,
      }
      .to_cycles(tn.amount);

      let res = deposit_cycles(canister_id, cycles).await;

      let refund_block = burn_or_refund(
          res.is_ok(),
          TOP_UP_CANISTER_REFUND_FEE,
          &tn,
          &ledger_canister_id,
      )
      .await?;

      [...]

      Ok(TransactionNotificationResult::encode(res)?)
  }
```

Figure 21.1: If the transaction is a top-up, the function mistakenly takes a write lock on the global state. It is persisted during the asynchronous call to `deposit_cycles`. If the canister panics in `burn_or_refund`, the lock will never be released, and the canister will become perpetually unresponsive. (`dfinity/rs/rosetta-api/cycles_minting_canister/src/main.rs`)

However, if the transferred amount is less than the transaction fee, the call to `burn_or_refund` will cause a panic.

```
async fn burn_or_refund(
    is_ok: bool,
    extra_fee: ICPTs,
    tn: &TransactionNotification,
    ledger_canister_id: &CanisterId,
) -> Result<Option<BlockHeight>, String> {
    if is_ok {
        burn_and_log(
            &tn,
            (tn.amount - TRANSACTION_FEE).unwrap(),
            &ledger_canister_id,
        )
        .await;
        Ok(None)
    } else {
        refund(&tn, &ledger_canister_id, extra_fee).await
    }
}
```

Figure 21.2: If the transferred amount is less than the transaction fee, the call to unwrap will cause a panic. (`dfinity/rs/rosetta-api/cycles_minting_canister/src/main.rs`)

If the transaction is a top-up, a write lock is mistakenly taken on the global state. (See figure 21.1.) Since the canister state is saved during the asynchronous call to `deposit_cycles` (which calls the root canister), the locked state is persisted. If the canister panics in `burn_or_refund`, the lock will never be released, and the canister will become perpetually unresponsive.

If the transaction requires that a new canister be created, a read lock is taken on the global state when the transferred funds are converted to cycles. The asynchronous call to `create_canister` will return an error if the number of cycles is less than the canister creation fee (which defaults to 1 trillion cycles). If the call succeeds, the canister will panic in `burn_or_refund`, and, like in the above scenario, the lock will never be released. In that case, it will be possible to take other read locks on the global state, but it will be impossible to obtain a write lock; this means that the global state will effectively be rendered immutable.

Moreover, since the execution environment will automatically reject the transaction notification if the canister panics, the ledger canister will reset the notification state for the transaction to `false`; this will occur regardless of whether a canister is being topped up or created.

```
let on_reject = move || {
    // discards error which is better than a panic in a callback
    let _ = change_notification_state(block_height, false);
    api::reject(&format!(
        "Notification failed with message '{}'",
        api::reject_message()
    ));
};
```

Figure 21.3: If the notification is rejected, the notification state of the transaction will be reset to `false`, enabling a malicious user to provide another notification of the same transaction to the cycles minting canister. (`dfinity/rs/rosetta-api/src/main.rs`)

If the canister creation process succeeds, a malicious user will be able to notify the cycles minting canister of the same transaction multiple times. (Note that this scenario is extremely unlikely; while the transferred amount has to be less than the transaction fee to trigger a panic, the corresponding number of cycles has to be greater than or equal to the canister creation fee.)

Exploit Scenario

A malicious user tops up a canister with 100 doms. As a result, a write lock is placed on the global state and persisted to the canister state, trapping the cycles minting canister. Any subsequent attempt to obtain a read or write lock on the global state will fail, making it impossible to mint new cycles and eventually causing all application canisters to run out of cycles.

Recommendations

Short term, have the canister check that the transaction amount is equal to or greater than the transaction fee and ensure that the call fails gracefully if it is not. Ensure that a write lock cannot be taken on the global state when the canister is handling new top-ups.

Long term, consider whether it would make sense to enable the system API to lock and unlock the entire canister. This would allow the execution environment to unlock a canister trapped in a locked state.

22. A self-owned canister cannot be deleted

Severity: Informational | Difficulty: Low
Type: Data Validation | Finding ID: TOB-DFINITY-022
Target: `dfinity/rs/execution_environment/src/canister_manager.rs`

Description

It is possible to use the root canister method `Ic00Method::SetController` to set the controller of a canister to the canister itself.

```
pub(crate) fn set_controller(
    &self,
    sender: PrincipalId,
    canister_id: CanisterId,
    new_controller: PrincipalId,
    state: &mut ReplicatedState,
) -> Result<(), CanisterManagerError> {
    let canister = state
        .canister_state(&canister_id)
        .ok_or_else(|| CanisterManagerError::CanisterNotFound(canister_id))?;

    self.validate_controller(&canister, &sender)?;

    let mut canister = state.take_canister_state(&canister_id).unwrap();
    canister.system_state.controller = new_controller;
    state.put_canister_state(canister);

    Ok(())
}
```

Figure 22.1: It is possible to use `Ic00Method::SetController` to set a canister's controller to the canister itself. (`dfinity/rs/execution_environment/src/canister_manager.rs`)

A canister can be deleted only by its controller, through the `Ic00Method::DeleteCanister` method; however, a canister cannot delete itself, so a canister controlled by itself cannot be deleted.

```
pub(crate) fn delete_canister(
    &self,
    sender: PrincipalId,
    canister_id_to_delete: CanisterId,
    state: &mut ReplicatedState,
) -> Result<(), CanisterManagerError> {
    if let Ok(canister_id) = CanisterId::try_from(sender) {
        if canister_id == canister_id_to_delete {
            // A canister cannot delete itself.
            return Err(CanisterManagerError::DeleteCanisterSelf(canister_id));
        }
    }
    let canister_to_delete = self.validate_canister_exists(state, canister_id_to_delete)?;

    // Validate the request is from the controller.
    self.validate_controller(&canister_to_delete, &sender)?;

    [...]
}
```

Figure 22.2: A self-owned canister cannot be deleted through `Ic00Method::DeleteCanister`, since a canister cannot delete itself.
(`dfinity/rs/execution_environment/src/canister_manager.rs`)

Recommendations

Long term, consider the effects of mandating that the ID of a canister's controller be different from the ID of the canister itself.

23. Potential out-of-bounds read in utf8ndup

Severity: Informational
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-023
Target: `deps/ledger-zxlib/include/utf8.h`

Description

The function `utf8ndup` will read one byte out of bounds if the input length is 0 or if the input is not null-terminated.

```
void *utf8ndup(const void *src, size_t n) {
    const char *s = (const char *)src;
    char *c = utf8_null;
    size_t bytes = 0;

    // Find the end of the string or stop when n is reached
    while ('\0' != s[bytes] && bytes < n) {
        bytes++;
    }

    [...]
```

Figure 23.1: If the input length, n, is 0, or if the input is not null-terminated, the function `utf8ndup` will read one byte out of bounds. (`deps/ledger-zxlib/include/utf8.h`)

This issue occurs a second time in the same function, on line 623.

Recommendations

Short term, switch the order of the two tests in the condition of the `while` statement. This will ensure that the pointer is not dereferenced if the input length is 0.

24. Use of static stack canaries provides inadequate protection against adversarial attacks

Severity: Low
Difficulty: High
Type: Auditing and Logging
Finding ID: TOB-DFINITY-024
Target: `deps/ledger-zxlib/src/zxmacros.c` and `deps/ledger-zxlib/src/zbuffer.c`

Description
The application uses hard-coded stack canaries to protect the stack space from stack buffer overflows.

```
#define CHECK_APP_CANARY() check_app_canary();
#define APP_STACK_CANARY_MAGIC 0xDEAD0031
extern unsigned int app_stack_canary;
```

Figure 24.1: The stack canary `APP_STACK_CANARY_MAGIC` is defined as the constant `0xDEAD0031`. (`deps/ledger-zxlib/include/zxmacros.h`)

```
void check_app_canary() {
#if defined (TARGET_NANOS) || defined(TARGET_NANOX)
    if (app_stack_canary != APP_STACK_CANARY_MAGIC) handle_stack_overflow();
#endif
}
```

Figure 24.2: The function `check_app_canary` simply checks the stack canary against the hard-coded value. (`deps/ledger-zxlib/src/zxmacros.c`)

This strategy is typically sufficient to protect against unexpected memory issues but will not prevent the exploitation of potential vulnerabilities. An adversary could simply dump the firmware of the device and then read the stack canary.

The same type of issue is present in `deps/ledger-zxlib/src/zbuffer.c`, which also defines a hard-coded canary, `CANARY_EXPECTED`.

```
#define CANARY_EXPECTED 0x987def82u
```

Figure 24.3: The `CANARY_EXPECTED` canary is defined as the constant `0x987DEF82`. (`deps/ledger-zxlib/src/zbuffer.c`)

```
zbuffer_error_e zb_check_canary() {
#if defined (TARGET_NANOS) || defined(TARGET_NANOX)
    CHECK_APP_CANARY();
    if (zbuffer_internal.size != 0) {
        // allocated
        uint32_t *zb_canary =
            (uint32_t * )(zbuffer_internal.ptr + zbuffer_internal.size + 4);
        if (*zb_canary != CANARY_EXPECTED) {
            handle_stack_overflow();
        }
    }
#endif
    return zb_no_error;
}
```

Figure 24.4: The function `zb_check_canary` simply checks the canary against the hard-coded value. (`deps/ledger-zxlib/src/zbuffer.c`)

Recommendations

Short term, use randomized stack canaries to protect against active exploitation of a vulnerability.

25. Seeding via Unix epoch generates less entropy than expected

Severity: Medium
Difficulty: Low
Type: Cryptography
Finding ID: TOB-DFINITY-025
Target: `nns/governance/canister/canister.rs` and `nns/gtc/src/init.rs`

Description

When a new `CanisterEnv` is created, its internal pseudorandom number generator (PRNG) is seeded twice with the epoch in nanoseconds, a 16-byte value, to meet the 32-byte seed requirement. The current Unix epoch time in nanoseconds is a 61-bit value. Assuming that an attacker is able to guess the first 21 bits of the epoch's 31 bits, equating to a ~17-minute window, the PRNG gains only 40 bits of entropy. This amount is easily within range of brute-forcing.

```
fn new() -> Self {
    CanisterEnv {
        // Seed the PRNG with the current time.
        //
        // This is safe since all replicas are guaranteed to see the same result of time()
        // and it isn't easily predictable from the outside.
        //
        // Using raw_rand from the ic00 api is an asynchronous call so can't really be
        // used to generate random numbers for most cases. It could be used to seed
        // the PRNG, but that wouldn't help much since after inception the pseudo-random
        // numbers could be predicted.
        rng: {
            let now_nanos = now()
                .duration_since(SystemTime::UNIX_EPOCH)
                .unwrap()
                .as_nanos();
            let mut seed = [0u8; 32];
            seed[..16].copy_from_slice(&now_nanos.to_be_bytes());
            seed[16..32].copy_from_slice(&now_nanos.to_be_bytes());
            StdRng::from_seed(seed)
        },
    }
}
```

*Figure 25.1: The `CanisterEnv` PRNG is seeded with the current Unix time in nanoseconds, 1621259542 * 10^9, which is a 61-bit value. (`nns/governance/canister/canister.rs`)*

This RNG is used indirectly whenever the associated `Environment` trait is used and directly when a new neuron ID (8 bytes) or a new subaccount (32 bytes) is generated.

The same entropy issue is present in `GenesisTokenCanisterInitPayloadBuilder::get_rng`. Here, the RNG is used to generate a random dissolve delay for neurons created by the genesis token canister (GTC). This dissolve delay is also used to generate the IDs of neuron subaccounts belonging to both seed-round investors and early contributors.

Exploit Scenario
An attacker is able to guess roughly when the replicas will seed their RNGs and to brute-force the seed. With little difficulty, the attacker can then use publicly available blockchain data to determine the order of the calls to the RNG, gaining access to the IDs of the neurons and their subaccounts. DFINITY has expressed concern over this possibility.

Recommendations
Short term, use a resource-intensive pseudorandom function (PRF) that cannot be easily guessed to extend this entropy source to a 32-byte value. Suitable PRFs are `PBKDF2`, `scrypt`, and `argon2id`.

Long term, consider using a higher-quality deterministic entropy source.

26. Out-of-bounds access in print_textual

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFINITY-026
Target: `app/src/parser.c`

Description
The function `print_textual` is used to convert a principal ID to a base-32 textual representation. The function verifies that `outValLen` is at least 37 bytes and then calls `pageString` to copy at most 37 bytes of the ID's textual representation to the output buffer `outVal`. It then removes any trailing dashes from a number of fixed positions.

```
__Z_INLINE parser_error_t print_textual(sender_t *sender,
                                        char *outVal, uint16_t outValLen,
                                        uint8_t pageIdx, uint8_t *pageCount) {
    [...]

    if (outValLen < 37) { return parser_unexpected_buffer_end; }
    outValLen = 37;

    pageString(outVal, outValLen, buffer, pageIdx, pageCount);

    // Remove trailing dashes
    if (outVal[17] == '-') outVal[17] = ' ';
    if (outVal[35] == '-') outVal[35] = ' ';
    if (outVal[53] == '-') outVal[53] = ' ';

    return parser_ok;
}
```

Figure 26.1: The function `print_textual` fails to check that `outValLen` is at least 54 before reading `outVal[53]`. (`app/src/parser.c`)

However, the function does not check whether the buffer is at least 54 bytes long before reading or writing to offset 53 of `outVal`. This means that the function could read and write out of bounds.

The function `print_textual` is reachable from `parser_validate`, which passes two 40-byte buffers for the key and value to `parser_getItem`.

```
char tmpKey[40];
char tmpVal[40];

for (uint8_t idx = 0; idx < numItems; idx++) {
    uint8_t pageCount = 0;
    CHECK_PARSER_ERR(parser_getItem(ctx, idx,
                                    tmpKey, sizeof(tmpKey),
                                    tmpVal, sizeof(tmpVal),
                                    0, &pageCount))
}
```

Figure 26.2: The `parser_validate` function calls `parser_getItem` with a 40-byte buffer for `tmpVal`. Depending on the index, this buffer may be passed as `outVal` to `parser_textual`, causing the function to access the buffer out of bounds. (`app/src/parser.c`)

This request is handled by `parser_getItemTokenTransfer` or `parser_getItemTransactionStateRead`, depending on the transaction type. If the index, `idx`, is 1 and the application is running in expert mode (i.e., `app_mode_expert` returns `true`), the `tmpVal` buffer will be passed to `print_textual`. This will cause an out-of-bounds read (and could cause an out-of-bounds write if `*(tmpVal + 53) == '-'`).

This same issue is present in `addr_getItem` (defined in `app/src/addr.c`). However, we could not identify any vulnerable paths to this function.

Recommendations

Short term, check that `outValLen` is at least 54 before reading and writing to offset 53 of the `outVal` buffer.

Long term, extend the fuzz testing to cover individual components of the codebase.

27. Missing data validation in deserialization of gossip protocol messages

Severity: Medium
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-DFINITY-028
Target: `dfinity/rs/p2p/gossip_protocol.rs`

Description

The gossip protocol implemented in the `ic_p2p` crate uses several message types in its communications. The messages are deserialized from network data in the peer-to-peer layer. However, because of the lack of data validation checks, a maliciously crafted or otherwise malformed message could be sent to a node. This could cause the node to panic, resulting in a denial of service.

A similar data validation issue is described in TOB-DFINITY-036.

```
/// `P2PEventHandlerImpl` implements the `AsyncTransportEventHandler` trait.
#[async_trait]
impl AsyncTransportEventHandler for P2PEventHandlerImpl {
    /// The method sends the given message on the flow associated with the given
    /// flow ID.
    async fn send_message(&self, flow: FlowId, message: TransportPayload)
            -> Result<(), SendError> {
        let gossip_message = <pb::GossipMessage as ProtoProxy<GossipMessage>>::proxy_decode(
            &message.0,
        )
        .map_err(|e| {
            trace!(self.log, "Deserialization failed {}", e);
            SendError::DeserializationFailed
        })?;
        [...]
    }
}
```

Figure 27.1: The send_message function of `P2PEventHandlerImpl` deserializes messages from the network using the `ProtoProxy<GossipMessage>::proxy_decode(&[u8])` method, which ultimately uses `TryFrom` trait implementations. (`dfinity/rs/p2p/src/event_handler.rs`)

```
/// A chunk request can be converted into a `pb::GossipChunkRequest`.
impl TryFrom<pb::GossipChunkRequest> for GossipChunkRequest {
    type Error = ProxyDecodeError;
    /// The function attempts to convert the given Protobuf chunk request into a
    /// GossipChunkRequest.
    fn try_from(gossip_chunk_request: pb::GossipChunkRequest)
            -> Result<Self, Self::Error> {
        Ok(Self {
            artifact_id: deserialize(&gossip_chunk_request.artifact_id).unwrap(),
            chunk_id: ChunkId::from(gossip_chunk_request.chunk_id),
        })
    }
}

/// A `pb::GossipChunk` can be converted into an artifact chunk.
impl TryFrom<pb::GossipChunk> for GossipChunk {
    type Error = ProxyDecodeError;
    /// The function attempts to convert a Protobuf chunk into a GossipChunk.
    fn try_from(gossip_chunk: pb::GossipChunk) -> Result<Self, Self::Error> {
        let response = try_from_option_field(
            gossip_chunk.response,
            "GossipChunk.response")?;
        let chunk_id = ChunkId::from(gossip_chunk.chunk_id);
        Ok(Self {
            artifact_id: deserialize(&gossip_chunk.artifact_id).unwrap(),
            chunk_id,
            artifact_chunk: match response {
                Response::Chunk(c) => Ok(add_chunk_id(c.try_into()?, chunk_id)),
                Response::Error(_e) => Err(P2PError {
                    p2p_error_code: P2PErrorCode::NotFound,
                }),
            },
        })
    }
}
```

Figure 27.2: Two `TryFrom` trait implementations of types used in `GossipMessage` are missing data validation checks and panic when they receive bad data. (`dfinity/rs/p2p/src/gossip_protocol.rs`)

```
#[cfg(test)]
pub mod tests {
    use super::*;
    use proptest::prelude::*;
    use ic_protobuf::proxy::ProtoProxy;

    proptest! {
        #[test]
        fn test_gossip_message(data: Vec<u8>) {
            let _req: GossipMessage = match pb::GossipMessage::proxy_decode(&data) {
                Ok(req) => req,
                Err(_e) => return Ok(())
            };
        }
    }
}

thread 'gossip_protocol::tests::test_gossip_message' panicked at 'Test failed: called
`Result::unwrap()` on an `Err` value: Io(Kind(UnexpectedEof)); minimal failing input: data
= [165, 1, 0, 0, 0, 0, 18, 0]
```

```
        successes: 413788
        local rejects: 0
        global rejects: 0
', p2p/src/gossip_protocol.rs:638:5

```

Figure 27.3: A new `proptest` test case that reveals a panic in the deserialization process (added to `dfinity/rs/p2p/src/gossip_protocol.rs`)

Exploit Scenario
An attacker runs a malicious node that sends malformed gossip protocol messages to peer nodes and evades detection by honest nodes. This causes the peer nodes to crash, resulting in a denial of service.

Recommendations
Short term, fix the `TryFrom` trait implementations of gossip messages so that they will not panic when provided malformed data.

Long term, avoid using panicking functions such as `Option::unwrap()` and `Result::unwrap()`. Instead, use an explicit method of error handling, such as pattern matching. This type of method will leverage Rust's type system to identify places in which operations could fail.

28. Missing getDeviceInfo command in TypeScript interface

Severity: Informational
Type: N/A
Target: `ledger-dfinity/js/src/index.ts`

Difficulty: N/A
Finding ID: TOB-DFINITY-032

Description
The TypeScript library includes the `getAppInfo` command but not the `getDeviceInfo` command. Because these commands appear in the generic Rust interface, developers may expect both of them to be included in the TypeScript library.

```
async getAppInfo(): Promise<ResponseAppInfo> {
  return this.transport.send(0xb0, 0x01, 0, 0).then(response => {
    // ...
  }, processErrorResponse);
}
```

Figure 28.1: The `getAppInfo` command in `index.ts`

Recommendations
Short term, add `getDeviceInfo` to the TypeScript interface.

Long term, develop a generic TypeScript ledger interface library that can be shared by all TypeScript interfaces to reduce the need for code reuse.

29. Potential missing null terminator in base32_encode

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-DFINITY-033
Target: `ledger-dfinity/app/src/base32.c`

Description

The function `base32_encode` returns the same value regardless of whether there is enough space to write a null terminator to the output string. This type of behavior is likely to cause errors that can be hard to diagnose.

```
uint32_t base32_encode(const uint8_t *data, unsigned int length,
                       char *result, uint32_t bufSize) {

    [...]

    if (count < bufSize) {
        result[count] = '\000';
    }
    return count;
}
```

Figure 29.1: The function `base32_encode` does not signal an error to the caller if there is no space left for the null terminator. (`Ledger-dfinity/app/src/base32.c`)

We set the severity of this issue to informational because we could not find any instances in the application in which the output buffer was too small to hold both the base-32-encoded output and the null terminator.

Exploit Scenario

The application is refactored, and new functionality is added to it. Under certain circumstances, a user can provide base-32-encoded input to the application; if there is not enough space for the null terminator, the application will crash.

Recommendations

Short term, ensure that there is enough space for a null terminator in the output buffer and that the function returns an error if there is not.

30. Unnecessary panics in the http_handler server's initialization

Severity: Informational
Difficulty: Undetermined
Type: Error Reporting
Finding ID: TOB-DFINITY-034
Target: `dfinity/rs/http_handler/src/lib.rs`

Description

The `create_port_file(PathBuf, u16)` function, which is used in the initialization of the HTTP handler server, panics in several places (figure 30.1). However, the only function that calls `create_port_file` returns a `Result` value (figure 30.2). This error-handling behavior is not accurately reflected in either the function's type or the codebase's documentation.

```
fn create_port_file(path: PathBuf, port: u16) {
    // Figure out which port was assigned; write it to a temporary
    // file; and then rename the file to `path`.  We write to a
    // temporary file first to ensure that the write is atomic.  We
    // create the temporary file in the same directory as `path` as
    // `rename` between file systems in case of different
    // directories can fail.
    let dir = path.parent().unwrap_or_else(|| {
        panic!(
            "Could not get parent directory of port report file {}",
            path.display()
        )
    });
    let mut port_file = NamedTempFile::new_in(dir)
        .unwrap_or_else(|err| panic!("Could not open temporary port report file: {}", err));
    port_file
        .write_all(format!("{}", port).as_bytes())
        .unwrap_or_else(|err| {
            panic!(
                "Could not write to temporary port report file {}: {}",
                path.display(),
                err
            )
        });
    [...]
}
```

Figure 30.1: The `create_port_file` function (`dfinity/rs/http_handler/src/lib.rs`)

```
/// Creates HTTP server, binds to HTTP port and handles HTTP requests forever.
/// This ***async*** function ***never*** returns unless binding to the HTTP
/// port fails.
/// The function spawns a tokio task per connection.
pub async fn start_server(
    metrics_registry: MetricsRegistry,
    http_handler_owned: HttpHandler,
) -> Result<(), Error>
```

Figure 30.2: The single caller of `create_port_file` returns a `Result` value.
(`dfinity/rs/http_handler/src/lib.rs`)

Recommendations

Short term, change the `create_port_file(PathBuf, u16)` function such that it returns a `Result` value when successful and returns an `Err` value instead of triggering a `panic!` upon a failure.

31. Risk of denial of service caused by crafted RPC messages in canister_sandbox

Severity: Low
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-DFINITY-035
Target: `dfinity/rs/canister_sandbox`

Description

The `FrameDecoder::decode` function deserializes a length-prefixed object from a provided `BytesMut` (effectively a byte array buffer). The implementation is a state machine with two states (`FrameDecoderState::NoLength` and `FrameDecoderState::Length(u32)`). The source of the function is shown in figure 31.1.

```
enum FrameDecoderState {
    NoLength,
    Length(u32),
}

impl<Message: DeserializeOwned + Clone> FrameDecoder<Message> {
    #[allow(clippy::new_without_default)]
    pub fn new() -> Self {
        FrameDecoder {
            state: FrameDecoderState::NoLength,
            phantom: PhantomData,
        }
    }

    /// Tries to extract one message from buffer given. The buffer is
    /// consumed as far as possible and resized to adequate size
    /// if more data is needed. Returns one frame if it can be parsed
    /// from given buffer.
    /// This is to be called repeatedly, interleaved with filling the
    /// buffer with more data as needed.
    pub fn decode(&mut self, data: &mut BytesMut) -> Option<Message> {
        loop {
            match &self.state {
                FrameDecoderState::NoLength => {
                    if data.len() < 4 {
                        data.reserve(4);
                        return None;
                    } else {
                        let size = data.get_u32();
                        self.state = FrameDecoderState::Length(size);
                    }
                }
                FrameDecoderState::Length(size) => {
                    let size: usize = *size as usize;
                    if data.len() < size {
                        data.reserve(size);
                        return None;
                    } else {
                        let frame = data.split_to(size);
                        self.state = FrameDecoderState::NoLength;
                        let frame = frame.clone();
                        let mut deserializer = Deserializer::from_slice(&frame);
                        let value: Result<Message, _> =
                            serde::de::Deserialize::deserialize(&mut deserializer);

                        if value.is_err() {
                            continue;
                        }
                        let value = value.unwrap();
                        return Some(value);
                    }
                }
            }
        }
    }
}
```

Figure 31.1: The `FrameDecoder::decode` function
(`dfinity/rs/canister_sandbox/common/src/frame_decoder.rs`)

When decoding a message, the state machine begins in the `FrameDecoderState::NoLength` state and extracts the message size (as a `u32` value). Its state then changes to `FrameDecoderState::Length(u32)`, which takes the extracted size value as an argument.

While the state machine continues executing in the `FrameDecoderState::Length(u32)` state, the capacity of the byte array buffer is resized to at least the extracted size. In an extreme case, the buffer could be allocated 4 GB (2**32 bytes) of memory.

Currently, the `FrameDecoder::decode` function appears to be called only from the `socket_read_demux` function, defined in `dfinity/rs/canister_sandbox/common/src/transport.rs`. This function reads data from a socket and is used in `dfinity/rs/canister_sandbox/src/main.rs` and `dfinity/rs/canister_sandbox/common/src/process.rs` to deal with RPC messages.

Exploit Scenario
An attacker who is able to send crafted messages to a socket read by the canister sandbox sends a small message with a sizable byte-length prefix. This causes the canister sandbox process to allocate 4 GB to the buffer, exhausting the host machine's memory and causing a denial of service.

Recommendations

Short term, audit all canister sandbox communication paths that use the `FrameDecoder::decode` function to verify that they will not handle untrusted input. Consider reducing the maximum message size that can be handled by `FrameDecoder::decode` (perhaps to 2**16, or u16). A lower size limit would prevent a denial of service stemming from excessive memory pressure on the host system. Document the security requirements of the canister sandbox's RPC mechanism, clearly stating that it requires trusted access.

Long term, perform threat modeling when designing new services. This will help you identify trust boundaries and deployment security requirements of the system.

32. Widespread lack of data validation in conversion functions

Severity: Medium
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-DFINITY-036
Target: Multiple files in `dfinity/rs`

Description

Many `TryFrom` trait implementations and other conversion functions use `Result::unwrap()` on values that can be user-controlled or on error values. When `unwrap` is called on one such value, the process will panic, which could cause a denial of service.

See TOB-DFINITY-028 for details on a similar issue.

```
impl TryFrom<(PrincipalId, InstallCodeArgs)> for InstallCodeContext {
    type Error = InstallCodeContextError;

    fn try_from(input: (PrincipalId, InstallCodeArgs)) -> Result<Self, Self::Error> {
        let (sender, args) = input;
        let canister_id = CanisterId::new(args.canister_id).map_err(|err| {
            InstallCodeContextError::InvalidCanisterId(format!(
                "Converting canister id {} failed with {}",
                args.canister_id, err
            ))
        })?;
        let compute_allocation = match args.compute_allocation {
            Some(ca) => Some(ComputeAllocation::try_from(ca.0.to_u64().unwrap())?),
            None => None,
        };
        let memory_allocation = match args.memory_allocation {
            Some(ma) => Some(MemoryAllocation::try_from(NumBytes::from(
                ma.0.to_u64().unwrap(),
            ))?),
            None => None,
        };
        let query_allocation = match args.query_allocation {
            Some(qa) => QueryAllocation::try_from(qa.0.to_u64().unwrap())?,
            None => QueryAllocation::default(),
        };

        Ok(InstallCodeContext {
            sender,
            mode: args.mode,
            canister_id,
            wasm_module: args.wasm_module,
            arg: args.arg,
            compute_allocation,
            memory_allocation,
            query_allocation,
        })
    }
}
```

Figure 32.1: The conversion code for `InstallCodeContext` (`dfinity/rs/types/types/src/lib.rs`)

```
impl TryFrom<pb::ProvisionalWhitelist> for ProvisionalWhitelist {
    type Error = ProxyDecodeError;

    fn try_from(src: pb::ProvisionalWhitelist) -> Result<Self, Self::Error> {
        if src.list_type == pb::provisional_whitelist::ListType::All as i32 {
            Ok(Self::All)
        } else if src.list_type == pb::provisional_whitelist::ListType::Set as i32 {
            Ok(Self::Set(
                src.set
                    .into_iter()
                    .map(|id| PrincipalId::try_from(id).unwrap())
                    .collect(),
            ))
        } else {
            Err(ProxyDecodeError::ValueOutOfRange {
                typ: "ProvisionalWhitelist::ListType",
                err: format!(
                    "{} is not one of the expected variants of ListType.",
                    src.list_type
                ),
            })
        }
    }
}
```

Figure 32.2: The conversion code for `ProvisionalWhitelist` (`dfinity/rs/registry/provisional_whitelist/src/lib.rs`)

```
impl TryFrom<pb::CanisterId> for CanisterId {
    type Error = PrincipalIdBlobParseError;

    fn try_from(id: pb::CanisterId) -> Result<Self, Self::Error> {
        // All fields in Protobuf definition are required hence they are encoded in
        // `Option`.  We simply treat them as required here though.
        let principal_id = PrincipalId::try_from(id.principal_id.unwrap())?;
        Ok(CanisterId(principal_id))
    }
}
```

Figure 32.3: The conversion code for `CanisterId`
(`dfinity/rs/types/base_types/src/canister_id.rs`)

```
impl TryFrom<pb::CanisterIdRange> for CanisterIdRange {
    type Error = ProxyDecodeError;

    fn try_from(src: pb::CanisterIdRange) -> Result<Self, Self::Error> {
        Ok(Self {
            start: CanisterId::try_from(src.start_canister_id.unwrap())?,
            end: CanisterId::try_from(src.end_canister_id.unwrap())?,
        })
    }
}
```

Figure 32.4: The conversion code for `CanisterIdRange`
(`dfinity/rs/registry/routing_table/src/proto.rs`)

```
impl TryFrom<pb::RoutingTable> for RoutingTable {
    type Error = ProxyDecodeError;

    fn try_from(src: pb::RoutingTable) -> Result<Self, Self::Error> {
        let mut map = BTreeMap::new();
        for entry in src.entries {
            let range = try_from_option_field(entry.range, "RoutingTable::Entry::range")?;
            let subnet_id = subnet_id_try_from_protobuf(entry.subnet_id.unwrap())?;
            if let Some(prev_subnet_id) = map.insert(range, subnet_id) {
                return Err(ProxyDecodeError::DuplicateEntry {
                    key: format!("{:?}", range),
                    v1: prev_subnet_id.to_string(),
                    v2: subnet_id.to_string(),
                });
            }
        }
        Ok(Self(map))
    }
}
```

Figure 32.5: The conversion code for `RoutingTable`
(`dfinity/rs/registry/routing_table/src/proto.rs`*)*

```
impl TryFrom<pb_metadata::NetworkTopology> for NetworkTopology {
    type Error = ProxyDecodeError;
    fn try_from(item: pb_metadata::NetworkTopology) -> Result<Self, Self::Error> {
        let mut subnets = BTreeMap::<SubnetId, SubnetTopology>::new();
        for entry in item.subnets {
            subnets.insert(
                subnet_id_try_from_protobuf(try_from_option_field(
                    entry.subnet_id,
                    "NetworkTopology::subnets::K",
                )?)?,
                try_from_option_field(
                    entry.subnet_topology,
                    "NetworkTopology::subnets::V")?,
            );
        }
        // NetworkTopology.nns_subnet_id will be removed in the following PR
        // Currently, initialise nns_subnet_id with dummy value in case not found
        let nns_subnet_id =
            match try_from_option_field(
                item.nns_subnet_id,
                "NetworkTopology::nns_subnet_id") {
            Ok(subnet_id) => subnet_id_try_from_protobuf(subnet_id).unwrap(),
            Err(_) => SubnetId::new(PrincipalId::new_anonymous()),
        };
        Ok(Self {
            subnets,
            routing_table: try_from_option_field(
                item.routing_table,
                "NetworkTopology::routing_table",
            )?,
            nns_subnet_id,
        })
    }
}
```

Figure 32.6: The conversion code for `NetworkTopology`
(`dfinity/rs/replicated_state/src/metadata_state.rs`)

```
/// From its protobuf definition convert to a SubnetId.  Normally, we would
/// use `impl TryFrom<pb::SubnetId> for SubnetId` here however we cannot as
/// both `Id` and `pb::SubnetId` are defined in other crates.
pub fn subnet_id_try_from_protobuf(
    value: pb::SubnetId,
) -> Result<SubnetId, PrincipalIdBlobParseError> {
    // All fields in Protobuf definition are required hence they are encoded in
    // `Option`.  We simply treat them as required here though.
    let principal_id = PrincipalId::try_from(value.principal_id.unwrap())?;
    Ok(SubnetId::from(principal_id))
}
```

Figure 32.7: The conversion code for `SubnetId` (`dfinity/rs/types/base_types/src/lib.rs`)

Exploit Scenario
An attacker crafts and sends messages that cannot be deserialized. This causes the process to crash, resulting in a denial of service (and may not be detected by honest nodes).

Recommendations
Short term, fix the `TryFrom` trait implementations and other conversion functions mentioned in this finding so that they will not panic when provided malformed input.

Long term, avoid using panicking functions such as `Option::unwrap()` and `Result::unwrap()`. Instead, use an explicit method of error handling, such as pattern matching. This type of method will leverage Rust's type system to identify places in which operations could fail.

33. Lack of access controls in the GTC's method get_account

Severity: Undetermined
Difficulty: Low
Type: Access Controls
Finding ID: TOB-DFINITY-037
Target: `dfinity/rs/nns/gtc/canister/canister.rs`

Description
The GTC's `get_account` method returns the `AccountState` of a given address. The account state contains information including the IDs of neurons created on the account's behalf and the total value of the account.

Since the GTC address is derived from a public key, a user could obtain a list of the neurons controlled by a given public key, including the total value of their stakes.

If the user did not know the neuron IDs of a given public key, he or she could query the governance canister using the `get_neuron_info` method to obtain more information on the neurons (like their votes on recent proposals). However, it would not be possible to use this public method to map an arbitrary neuron ID to its controlling principal.

Exploit Scenario
A user interested in mapping neuron-voting behavior uses the GTC's `get_account` method to obtain a list of the IDs of neurons controlled by known public keys. The user then calls `get_account` to obtain information on the neurons' voting behavior.

Recommendations
Short term, if the lack of access controls is unintentional, consider restricting access to the GTC's `get_account` method to only the principal of an account.

34. Failed calls to GTC method donate_account cause permanent locking of the remaining neurons

Severity: Low
Difficulty: High
Type: Access Controls
Finding ID: TOB-DFINITY-038
Target: `dfinity/rs/nns/gtc/src/lib.rs`

Description

The implementation of the GTC method `donate_account` calls `AccountState::transfer` to transfer the neurons in an account to a predefined custodian account.

```
let neuron_ids = self.neuron_ids.clone();

for neuron_id in neuron_ids {
    let result = GovernanceCanister::transfer_gtc_neuron(
        neuron_id.clone(),
        custodian_neuron_id.clone(),
    )
    .await;

    self.neuron_ids.retain(|id| id != &neuron_id);

    let mut donated_neuron = TransferredNeuron {
        neuron_id: Some(neuron_id),
        timestamp_seconds: now_secs(),
        error: None,
    };

    match result {
        Ok(_) => self.successfully_transferred_neurons.push(donated_neuron),
        Err(e) => {
            donated_neuron.error = Some(e.to_string());
            self.failed_transferred_neurons.push(donated_neuron)
        }
    }
}
```

Figure 34.1: As part of the neuron transfer process, neuron IDs are purged from `self.neuron_ids`. However, there are no checks on the results of inter-canister calls to `transfer_gtc_neuron`. This means that if a call to transfer a neuron fails, it will no longer be possible to claim or donate the neuron. (`dfinity/rs/nns/gtc/src/lib.rs`)

If an inter-canister call fails, the neuron's ID will still be purged from the list of neurons in the account. This means that it will not be possible to call `donate_account` a second time to transfer the remaining neurons; similarly, it will not be possible to call `claim_neurons` to claim the remaining neurons.

Exploit Scenario

Numerous initial controllers decide to donate their accounts. However, an issue in the governance canister causes the calls to `transfer_gtc_neuron` to fail. As a result, a large portion of the voting power becomes perpetually locked, which could cause further issues for the IC governance system.

Recommendations

Short term, determine whether `AccountState::transfer`’s behavior is expected and modify it if it is not.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

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

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

B. Fuzzing

During the first week of the assessment, Trail of Bits used fuzz testing to obtain better negative test coverage of the Wasm code validation.

Wasm Code Validation Fuzzer

Trail of Bits developed a fuzzer to increase the negative test coverage of the Wasm code validation. The fuzzer is based on `cargo fuzz`, which uses libFuzzer for test case generation. (libFuzzer is an in-process, coverage-guided evolutionary fuzzing engine integrated into Clang.) The fuzzer covers the Wasm code validation implemented in `dfinity/rs/wasm_util/src/validation.rs`.

To obtain code coverage data, we used `cargo fuzz`'s built-in source-based code coverage feature.

Function	Lines Covered	Percentage Covered
`validate_data_section`	23 (23)	100.00%
`validate_segment`	11 (13)	81.82%
`can_compile`	6 (6)	100.00%
`validate_wasm_binary`	8 (8)	100.00%
`get_valid_exported_functions`	58 (58)	100.00%
`validate_export_section`	50 (58)	84.00%
`validate_function_signature`	25 (25)	100.00%
`validate_import_section`	40 (40)	100.00%
`get_valid_system_apis`	479 (479)	100.00%

Running the Fuzzer

To build and run the fuzzer, write `cargo fuzz run validate_wasm_binary`. You can use `--` on the command line to pass arguments (like the number of jobs, maximum runtime, or path to the input corpus) directly to libFuzzer. As a corpus, we used the test suite from the GitHub `wasm3/wasm-core-testsuite` repository, together with the `wasm_util` crate unit tests.

C. Third-Party Dependencies

Trail of Bits audited the third-party Rust dependencies of all crates in `dfinity/rs`, working from commit `42ce95e` of the repository. To perform this audit, we used the following Cargo plug-ins:

1. `cargo-audit`: To audit third-party dependencies for known vulnerabilities
2. `cargo-geiger`: To audit third-party dependencies for unsafe code
3. `cargo-outdated`: To look for outdated third-party dependencies
4. `cargo-deny`: To audit third-party licenses to ensure that they do not conflict with the project license

Known Vulnerabilities and Advisories

The table below details the results of running `cargo audit --deny warnings` on the crates in `dfinity/rs`. The red rows indicate vulnerabilities identified by `cargo audit`, while the pink rows indicate vulnerabilities included in `dfinity/rs/ignored-vulnerabilities.nix`. The yellow rows represent warnings about unsound code and unmaintained, deprecated, or yanked dependencies.

ID	Crate	Issue
RUSTSEC-2020-0146	`generic-array` 0.12.3	`arr!` macro erases lifetimes
RUSTSEC-2019-0017	`once_cell` 1.4.0-alpha.0	Panic during initialization of `Lazy<T>` might trigger undefined behavior
RUSTSEC-2021-0013	`raw-cpuid` 8.1.2	Soundness issues in `raw-cpuid`
RUSTSEC-2020-0041	`sized-chunks` 0.6.2	Multiple soundness issues in `Chunk` and `InlineArray`
RUSTSEC-2020-0031	`tiny_http` 0.7.0	HTTP Request smuggling through malformed Transfer Encoding headers
RUSTSEC-2020-0096	`im` 15.0.0	`TreeFocus` lacks bounds on its `Send` and `Sync` traits
RUSTSEC-2020-0080	`miow` 0.2.1	`miow` invalidly assumes the memory layout of `std::net::SocketAddr`
RUSTSEC-2020-0078	`net2` 0.2.34	`net2` invalidly assumes the

		memory layout of `std::net::SocketAddr`
RUSTSEC-2020-0070	`lock_api` 0.3.4 and 0.4.1	Some `lock_api` lock guard objects can cause data races
RUSTSEC-2020-0029	`rgb` 0.8.18	Allows viewing and modifying arbitrary structs as bytes
RUSTSEC-2020-0027	`traitobject` 0.1.0	`traitobject` assumes the layout of fat pointers
RUSTSEC-2019-0036	`failure` 0.1.8	Type confusion if `__private_get_type_id__` is overriden [*sic*]
RUSTSEC-2020-0095	`difference` 2.0.0	Unmaintained
RUSTSEC-2020-0077	`memmap` 0.7.0	Unmaintained
RUSTSEC-2020-0056	`stdweb` 0.4.20	Unmaintained
RUSTSEC-2020-0053	`dirs` 1.0.5 and 2.0.2	Unmaintained (use `dirs-next`)
RUSTSEC-2020-0036	`failure` 0.1.8	Deprecated
RUSTSEC-2020-0016	`net2` 0.2.34	Deprecated (use `socket2`)
RUSTSEC-2018-0015	`term` 0.5.2 and 0.6.1	Unmaintained
-	crossbeam-queue 0.2.2	Yanked
-	`miow` 0.2.1	Yanked
-	net2 0.2.34	Yanked
-	`pin-project-lite` 0.1.7 and 0.2.4	Yanked

The current build uses outdated versions of all of the vulnerable crates identified by `cargo-audit` (i.e., those listed in the red and pink rows).

Unsafe Third-Party Code

We used the Cargo plug-in `cargo-geiger` to identify unsafe Rust code used in third-party crates. Because of build problems and the verbose nature of `cargo-geiger` output, we decided to focus on sample crates from `dfinity/rs`.

The tables below list the third-party crates that contribute the most unsafe expressions to the build. In each table, the middle column lists the number of unsafe expressions local to the crate, and the last column shows the number of unsafe expressions in the crate and all of its dependencies.

`crypto`		
Dependency	**Number of Unsafe Expressions (Crate-Specific)**	**Number of Unsafe Expressions (Aggregate)**
ed25519-dalek 1.0.1	0	1117
prometheus 0.9.0	339	339
arrayvec 0.5.1	301	301
ff 0.5.2	0	274
prost 0.7.0	202	202

`process_manager`		
Dependency	**Number of Unsafe Expressions (Crate-Specific)**	**Number of Unsafe Expressions (Aggregate)**
memchr 2.3.3	1823	1823
net2 0.2.34	332	332
mio-uds 0.6.8	202	202
signal-hook-registry 1.3.0	109	109
futures-channel 0.3.13	83	83

`governance`		
Dependency	**Number of Unsafe Expressions (Crate-Specific)**	**Number of Unsafe Expressions (Aggregate)**
csv 1.1.3	6	4897
clap 3.0.0	0	269

prost 0.7.0	202	202
async-trait 0.1.42	0	45
rand_core 0.5.1	22	22

`ledger`		
Dependency	**Number of Unsafe Expressions (Crate-Specific)**	**Number of Unsafe Expressions (Aggregate)**
prost 0.7.0	202	202
byteorder 1.4.2	193	193
crc32fast 1.2.0	108	108
lazy_static 1.4.0	7	7

`registry`		
Dependency	**Number of Unsafe Expressions (Crate-Specific)**	**Number of Unsafe Expressions (Aggregate)**
prost 0.7.0	202	202

Outdated Dependencies

The Cargo plug-in `cargo-outdated` can be used to audit a codebase for outdated dependencies (i.e., dependencies for which more recent published versions are available). The plug-in found that updates are available for 111 of the `dfinity/rs` crates' direct dependencies.

Third-Party Licenses

The Cargo plug-in `cargo-deny` can be used to audit third-party licenses to ensure that they do not conflict with each other or with the larger project's license. The table below lists the licenses of the dependencies of the crates under `dfinity/rs`. The green rows indicate licenses that are considered free by the Free Software Foundation. (For more detail, see the SPDX license list and GNU's resource on licenses.) Of the dependencies, 143 of them lack a specific license that could be parsed by `cargo-deny`.

License	Dependencies
Apache 2.0	456
Apache 2.0 with LLVM-exception	24
BSD 0-Clause	1
BSD 2-Clause	3
BSD 3-Clause	10
BSL 1.0	2
CC0 1.0	4
CDDL 1.0	1
ISC	10
MIT	548
MPL 2.0	10
MPL 2.0+	3
NCSA	1
No license	143
Unlicense	11
Zlib	1

D. Non-Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

General

- The code occasionally uses anonymous tuples as return types. This practice is essentially equivalent to creating a tuple struct of the same arity for storage but provides less type safety. It is also best to use normal structs rather than tuple structs whenever possible, as they provide greater type safety and adhere more closely to idiomatic Rust code.

Governance Canister

- The `Neuron` field `dissolve_state` is an optional `DissolveState`. However, it can never be `None` and should not be optional.
- The `Neuron` field `aging_since_timestamp_seconds` should be an `Option<u64>`. (The value `u64::MAX` is used to denote `None`.) This would safeguard against underflows.
- The `ProposalInfo` fields `executed_timestamp_seconds`, `failed_timestamp_seconds`, and `reward_event_round` should be optional or changed to Rust enums. (The value 0 is currently used to denote `None`.)
- Consider making the `Governance` field `wait_for_quiet_threshold_seconds` a Rust enum. (Currently, 0 is used to indicate "wait forever.")

Registry Canister

- The function `Registry::do_create_subnet` should use the implementation of the trait `Into::<SubnetRecord>` for `CreateSubnetPayload` to create new subnet records.
- The function `decode_certified_deltas` (in `dfinity/rs/registry/common/src/certification.rs`) should use `String::from_utf8` and return an error if the key-decoding process fails. Instead, it uses `String::from_utf8_lossy`, which could lead to registry collisions.

Root Canister API

- The `Ic00Method::DepositFunds` and `Ic00Method::DepositCycles` methods essentially implement the same functionality. One of these methods should be deprecated.

Zondax Hardware Wallet

- The content size check for `call` requests is commented out. This check should be enabled to ensure that only valid data is passed to the application (`app/src/parser_impl.c`, lines 279–280).
- The functions `_readEnvelope` and `readContent` both pass the global variable `parser_tx_obj` as an argument, but `readProtobuf` uses `parser_tx_obj` directly. This use is error-prone and may break if the code is refactored (`app/src/parser_impl.c`, line 258).
- The function `readProtobuf` copies the resultant request to `parser_tx_obj` before checking whether the value was parsed correctly. To ensure that invalid data is not used by mistake (after the code is refactored, for example), the output status should be checked before the resultant data is copied (`app/src/parser_impl.c`, line 258).
- The condition `displayIdx < 0` is redundant since `displayIdx` is unsigned (`app/src/parser.c`, lines 188, 208, and 244).
- The function `app_main` uses unlabeled constants in error handling, which makes the code more difficult to understand and audit (`app/src/common/app_main.c`, lines 231–237).
- The condition `length < 0` is redundant since the `length` variable is unsigned (`app/src/base32.c`, line 25).
- The codebase is not compiled with `-Wall` or `-Wextra`. Enabling these compilation flags reveals a number of code quality issues that should be fixed.
- In `base32_encode`, on line 37, it is redundant to compute the `bitwise AND` of `data[next++]` with 0xFF; since `data` is a pointer to `unsigned char`, `data[next++]` will always be less than 256.
- Implicit assumptions throughout the codebase are poorly documented. For example, the functions `base32_encode` and `crypto_principalToTextual` will return the same value regardless of whether there is enough space to write a null terminator to the output string. This type of behavior is likely to cause errors that can be hard to diagnose and should at least be documented.
- The CLA field of the ADPU transport layer expects a magic constant for the wallet to accept commands and work correctly. The TypeScript library defines a CLA constant to be used in `js/src/common.ts`. However, for `ledger-dfinity-rs/src/dfinity.rs`, the CLA is a magic constant provided to the `DfinityApp::new` constructor by `ledger-dfinity-rs/src/lib.rs`. While this is unlikely to cause issues, moving the CLA constant to `dfinity.rs` instead of passing it in via `lib.rs` would help prevent future errors and centralize the error-handling process.

Ledger Canister

- Since `Archive::nodes` may be empty, consider using `checked_sub` in `Archive::last_node_index` (line 320 in `dfinity/rs/rosetta-api/ledger_canister/src/archive.rs`) to avoid a potential underflow if no archive node has been created.
- Consider using `checked_add` in `append_blocks` (line 70 in `dfinity/rs/rosetta-api/ledger_canister/src/archive_node.rs`) to ensure that computations of the total size do not cause an overflow.

Genesis Token Canister

- The field `genesis_timestamp_seconds` in `GenesisTokenCanisterInitPayloadBuilder` is initialized to 0 by default in `GenesisTokenCanisterInitPayloadBuilder::new`. In `canister_init`, the GTC checks whether the value is 0 and updates it by calling `now` to obtain the current Unix timestamp in seconds. The value is treated as optional (with 0 indicating `None`), which would be clearer if the type of `genesis_timestamp_seconds` were converted to `Option<u64>`.
- The method `AccountState::transfer` expects to receive a custodian neuron ID as input. However, the type of the `custodian_neuron_id` parameter is `Option<NeuronId>`, and the parameter is unwrapped immediately before it is used. The signature should take a `NeuronId` instead. Moreover, since core GTC functionality depends on the receipt of this neuron ID, the types of `Gtc::donate_account_recipient_neuron_id` and `Gtc::forward_all_unclaimed_accounts_recipient_neuron_id` should be changed to `NeuronId`; this change would also prevent runtime panics caused by the incorrect initialization of the canister.

E. Fix Log

After the initial assessment, the DFINITY team provided Trail of Bits with a detailed fix review document and links to the corresponding Jira issues and Git pull requests. The Trail of Bits audit team reviewed each fix to ensure that the underlying issue was correctly addressed.

#	Title	Severity	Status
1	Hard-coded secrets in repositories	Informational	Fixed
2	Potential integer underflow in validate_export_section	Low	Fixed
3	Candid UI susceptible to slowloris attacks	Low	Not fixed
4	Missing check for exported symbol during Wasm code validation	Informational	Fixed
5	Weak input validation in ic0_stable_grow	Low	Fixed
6	Potential integer underflow in CanisterState::available_memory	Informational	Fixed
7	Potential integer overflow in valid_subslice	Low	Not fixed
8	Potential integer overflow in RequestInPrep::extend_method_payload	Low	Fixed
9	Erroneous index calculation in split_off_older_than	Low	Fixed
10	Helper function used in archiving returns new blocks and keeps old blocks in the ledger state	Medium	Fixed
11	Ledger upgrade mechanism does not account for archived blocks	High	Fixed
12	Calls to archive_blocks break block-height tracking	High	Fixed
13	Accounts with low balances are trimmed from the ledger	Informational	Not fixed

14	Controller can evade fees and claim extra rewards through neuron disbursement process	High	Fixed
15	Inconsistent use of panics	Informational	Fixed
16	Rewards worth less than transaction fees are discarded during neuron disbursement	Low	Not fixed
17	The function check_caller_authz_and_log returns true if the function name is misspelled	Low	Fixed
18	Registry method get_value returns the wrong error code for malformed messages	Low	Fixed
19	Registry canister fails to update node operator's node allowance when a new node is added	Low	Fixed
20	The ic-admin subcommand SubCommand::GetNodeOperatorList reports duplicate and stale operator IDs	Low	Fixed
21	Cycles minting canister can panic with the global state write-locked, thereby becoming perpetually unresponsive	High	Fixed
22	A self-owned canister cannot be deleted	Informational	Not fixed
23	Potential out-of-bounds read in utf8ndup	Informational	Fixed
24	Use of static stack canaries provides inadequate protection against adversarial attacks	Low	Partially fixed
25	Seeding via Unix epoch generates less entropy than expected	Medium	Not fixed
26	Out-of-bounds access in print_textual	Low	Fixed
27	Missing data validation in deserialization of gossip protocol messages	Medium	Fixed
28	Missing getDeviceInfo command in TypeScript interface	Informational	Not fixed

29	Potential missing null terminator in base32_encode	Informational	Fixed
30	Unnecessary panics in the http_handler server's initialization	Informational	Not fixed
31	Risk of denial of service caused by crafted RPC messages in canister_sandbox	Low	Not fixed
32	Widespread lack of data validation in conversion functions	Medium	Fixed
33	Lack of access controls in the GTC's method get_account	Undetermined	Not fixed
34	Failed calls to GTC method donate_account cause permanent locking of the remaining neurons	Low	Not fixed

For additional information for each fix, please refer to the detailed fix log below.

Detailed Fix Log

Finding 1: Hardcoded secrets in repositories
Fixed. The DFINITY team has reviewed the list of potentially sensitive secrets in the finding description, and it claims to have rotated the keys that are still in use.

Finding 2: Potential integer underflow in validate_export_section
Fixed. The function `validate_export_section` now explicitly checks that the function index is greater than the size of the import section and returns an error if this is not the case.

Finding 3: Candid UI susceptible to slowloris attacks
Not fixed. According to the DFINITY team, this issue is restricted to the local development environment and does not present a security issue to the production environment.

Finding 4: Missing check for exported symbol during Wasm code validation
Fixed. The function `validate_export_section` now checks each exported symbol name against a list of reserved symbols, which includes "`canister_counter_instructions`."

Finding 5: Weak input validation in ic0_stable_grow
Fixed. The number of requested pages is now validated against the current memory usage in `MemoryUsage::increase_usage` before `SystemStateAccessor::stable_grow` is called.

Finding 6: Potential integer underflow in CanisterState::available_memory
Fixed. The team replaced the function `CanisterState::available_memory`, and the canister now uses the `MemoryUsage` type to track canister memory usage. Because this type checks the number of bytes available to the canister before the memory is grown (in `MemoryUsage::increase_usage`), the issue appears to be resolved.

Finding 7: Potential integer overflow in valid_subslice
Not fixed. According to the DFINITY team, the issue will be addressed as part of a larger refactor implementing support for 64-bit architectures.

Finding 8: Potential integer overflow in RequestInPrep::extend_method_payload
Fixed. The function `RequestInPrep::extend_method_payload` now converts `size` and `current_size` to `u64` before checking the new size against `max_size_intra_subnet.get()`.

Finding 9: Erroneous index calculation in split_off_older_than
Fixed. This issue was addressed as part of a larger rewrite of the ledger canister archiving functionality.

Finding 10: Helper function used in archiving returns new blocks and keeps old blocks in the ledger state
Fixed. This issue was addressed as part of a larger rewrite of the ledger canister archiving functionality.

Finding 11: Ledger upgrade mechanism does not account for archived blocks
Fixed. This issue was addressed as part of a larger rewrite of the ledger canister archiving functionality.

Finding 12: Calls to archive_blocks break block-height tracking
Fixed. This issue was addressed as part of a larger rewrite of the ledger canister archiving functionality.

Finding 13: Accounts with low balances are trimmed from the ledger
Not fixed. According to the DFINITY team, this is not currently an issue because the number of active accounts is much smaller than the threshold (which defaults to 50 million).

Finding 14: Controller can evade fees and claim extra rewards through neuron disbursement process
Fixed. The DFINITY team reordered the three transfers performed by the function `Governance::disburse_neuron` so that the system burns the fees before transferring any funds to the user. The cached neuron stake is now updated and checked between each transfer.

Finding 15: Inconsistent use of panics
Fixed. The DFINITY team addressed all the examples provided in the finding. The team now also provides clearer guidelines for developers on when it is acceptable to use unrecoverable errors, which will help prevent future issues of this type.

Finding 16: Rewards worth less than transaction fees are discarded during neuron disbursement
Not fixed. According to the DFINITY team, this is currently the intended behavior, as the ledger canister will not accept minting transactions with an amount less than the transaction fee. However, we found no such restriction in the current implementation of the ledger canister. In particular, the call to `Ledger::transfer_funds` to mint the rewards sets the transaction fee to 0, and the `send` method on the ledger canister also verifies that the received fee for minting transactions is 0.

Finding 17: The function check_caller_authz_and_log returns true if the function name is misspelled
Fixed. The team removed the function, and public methods now rely on the `check_caller_is_*` methods to implement access controls. However, the `is_authorized` function still remains in `access_controls.rs`. This function represents a potential risk

because it returns `true` by default. Because the function appears to be unused, we recommend that it be removed as well.

Finding 18: Registry method get_value returns the wrong error code for malformed messages
Fixed. The registry method `get_value` now returns the correct error code when encountering a malformed message.

Finding 19: Registry canister fails to update node operator's node allowance when a new node is added
Fixed. The function `Registry::do_add_node` now saves the decoded node operator record to a local variable before it is decremented. The decremented value is then used to update the node operator allowance.

Finding 20: The ic-admin subcommand SubCommand::GetNodeOperatorList reports duplicate and stale operator IDs
Fixed. The function `RegistryClientImpl::get_key_family` now ensures that each key is added to the resultant set only once.

Finding 21: Cycles minting canister can panic with the global state write-locked, thereby becoming perpetually unresponsive
Fixed. The team removed the call to `unwrap` in `burn_or_refund`, and the implementation now uses `dfn_core::api::call_with_cleanup`, which will clean up any local state if the call fails.

Finding 22: A self-owned canister cannot be deleted
Not fixed. This is intended behavior and is already used in production for a number of canisters to provide assurance that they will not be removed by the controlling entity.

Finding 23: Potential out-of-bounds read in utf8ndup
Fixed. The Zondax team deleted the corresponding source file, as it was not in use.

Finding 24: Use of static stack canaries provides inadequate protection against adversarial attacks
Partially fixed. The stack canary used in `zb_check_canary` is now randomized using `cx_rng`, but the stack canary used by `check_app_canary` is still constant. According to Zondax, this second canary is initialized by the system and cannot be updated by the developer.

Finding 25: Seeding via Unix epoch generates less entropy than expected
Not fixed. According to the DFINITY team, this is not a security issue because keeping neuron IDs private is not a design goal of the system.

Finding 26: Out-of-bounds access in print_textual

Fixed. The function `print_textual` now allocates a temporary stack buffer that is guaranteed to be large enough to hold the entire base-32 textual representation of the input.

Finding 27: Missing data validation in deserialization of gossip protocol messages
Fixed. Calls to `unwrap` in the gossip protocol parsers have been replaced by early returns using the `?` operator.

Finding 28: Missing getDeviceInfo command in TypeScript interface
Not fixed. The finding is only a recommendation to make the API more uniform across different languages.

Finding 29: Potential missing null terminator in base32_encode
Fixed. The function `base32_encode` now returns `-1` if the buffer is not large enough to hold the entire base-32-encoded output, including the null terminator. This return value is then checked for in `crypto_principalToTextual`, which is the only call site of the function `base32_encode`.

Finding 30: Unnecessary panics in the http_handler server's initialization
Not fixed. The DFINITY team does not consider this a security issue.

Finding 31: Risk of denial of service caused by crafted RPC messages in canister_sandbox
Not fixed. According to the DFINITY team, the code is currently unused and will be revisited and refactored at a later date.

Finding 32: Widespread lack of data validation in conversion functions
Fixed. The `TryFrom` implementations referred to in the issue now return an error instead of panicking on invalid input.

Finding 33: Lack of access controls in the GTC's method get_account
Not fixed. According to the DFINITY team, this is intended behavior and not a security issue.

Finding 34: Failed calls to GTC method donate_account cause permanent locking of the remaining neurons
Not fixed. According to the DFINITY team, this is intended behavior and not a security issue.