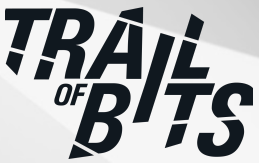

DFINITY Service Nervous System, Phase 2

Security Assessment

October 6, 2023

Prepared for:
Robin Künzler
DFINITY

Prepared by: **Artur Cygan and Fredrik Dahlgren**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

About Trail of Bits 1
Notices and Remarks 2
Table of Contents 3
Executive Summary 4
Project Summary 7
Project Targets 8
Project Goals 9
Project Coverage 10
Automated Testing 12
Codebase Maturity Evaluation 14
Summary of Findings 17
Detailed Findings 18
1. Uneven distribution of stake across neurons may impact SNS governance 18
2. Wrong error message returned from new_sale_ticket in Adopted state 20
3. Swap canister paging implementations panic on invalid ranges 23
4. The NNS governance canister always warns about missing neurons if a token swap fails 25
A. Vulnerability Categories 27
B. Code Maturity Categories 29
C. Automated Testing 31
D. Code Quality Recommendations 33

Executive Summary

Engagement Overview

DFINITY engaged Trail of Bits to review the security of the Internet Computer Service Nervous System (SNS). The SNS allows users to deploy applications controlled by a decentralized community on the Internet Computer.

One consultant conducted the review from June 5 to June 23, 2023, for a total of two and a half engineer-weeks of effort. Our testing efforts focused on access controls, inter-canister interactions, and token arithmetic across the NNS governance canister, SNS governance canister, SNS swap canister, and SNS ledger. We performed static and dynamic testing of the different canisters under review, with full access to both the documentation and source code.

Observations and Impact

We found the implementation of the SNS to be well written and to have few issues. Privileged APIs implement proper access controls and input validation, token arithmetic is implemented carefully to avoid overflow issues, and asynchronous canister interactions are written defensively to avoid common issues with asynchronous code interacting with the global state. Canister interactions are also carefully documented and modeled using TLA+ to ensure that edge cases and failure cases are well understood.

We did not find any serious issues with the implementation during this engagement. However, as part of the review, we identified a number of recommendations, each of which would improve the overall security posture of the system.

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Migrate all SNS canisters to use the more secure Rust CDK.** Versions 0.51 and above of the Internet Computer Rust CDK ensure that local variables are dropped correctly if a canister traps after a call to `await`. In particular, this ensures that locks on global resources are released correctly if the canister traps after an `await`. The SNS swap canister currently uses `dfn_core` as its CDK, which does not implement this feature. This makes it harder to reason about locks taken on the global state, and correspondingly increases the risk of the canister becoming perpetually locked

if the canister traps. For this reason, we recommend that DFINITY migrate all implementations based on `dfn_core` to use the Rust CDK instead.

- **Mirror API types using internal types that provide stronger compile-time guarantees.** Many canister APIs use types with optional fields to allow for easier evolution of the API over time. Often these fields are not in fact optional, which makes canister implementations more complex, as the canister needs to verify that the fields are actually present every time they are used. This also means that such fields are handled inconsistently throughout the codebase; sometimes functions panic if an optional field is missing, and sometimes they attempt to return a sane default or an error. We recommend that DFINITY define new internal types with a one-to-one correspondence with API types. This will allow API types to evolve over time and will provide type-level guarantees that non-optional fields are present and valid, which would make canister implementations easier to review and maintain.

- **Ensure that the build system makes it easy for developers to obtain accurate test coverage data.** DFINITY recently switched to using a Bazel-based build system. For reasons related to this migration, it is currently difficult for developers to obtain accurate test coverage reports for individual system components. This makes it very hard to ensure that all components are sufficiently covered by unit and integration tests or to investigate how test coverage for individual components develops or degrades over time. We recommend that DFINITY invest time to ensure that all developers have easy access to reliable test coverage reports to inform the development process.

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

EXPOSURE ANALYSIS

Severity	Count
Medium	1
Low	1
Informational	2

CATEGORY BREAKDOWN

Category	Count
Data Validation	2
Error Reporting	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Project Timeline

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

Date	Event
June 1, 2023	Pre-project kickoff call
June 12, 2023	Status update meeting #1
June 20, 2023	Status update meeting #2
June 27, 2023	Delivery of report draft
June 27, 2023	Report readout meeting
October 6, 2023	Delivery of comprehensive report

Project Targets

The engagement involved a review and testing of the following target.

SNS

Repository	https://github.com/dfinity/ic
Version	`2867da6c18178ac79bc513a9c7cad59a09030655`
Type	Rust
Platform	Linux

Project Goals

The engagement was scoped to provide a security assessment of the SNS functionality on the Internet Computer. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are asynchronous SNS canister interactions vulnerable to race conditions or time-of-check/time-of-use (TOCTOU) issues?
- Is the SNS token swap vulnerable to double-spending attacks?
- Is the ticket-based payment protocol implemented correctly?
- Are all APIs that are part of the ticket-based payment system idempotent?
- Are swap canister lifecycle state transitions implemented correctly?
- Is it possible to bypass access controls or lifecycle checks when calling APIs on the swap canister?
- Are access controls enforced correctly by the SNS governance canister?
- Is it possible to bypass the restricted `PreInitializationSwap` mode though SNS governance?
- Does SNS neuron and proposals management properly take the SNS governance mode into account to ensure that the `PreInitializationSwap` mode is enforced correctly?
- Are community fund (recently renamed to neuron fund) contributions handled correctly if a token swap either succeeds or fails?
- Does the ICRC-1 ledger handle fees correctly?
- Does the ICRC-1 ledger handle large memo fields correctly?

Project Coverage

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

- **SNS swap canister:** We manually reviewed the lifecycle management to ensure that all state transitions are correct and that certain restricted APIs on the swap canister can be called only in the correct lifecycle state. We reviewed access controls and input validation performed by the canister to assess whether invariants around token swaps are maintained by the implementation. We also reviewed the ticket-based payment flow to assess whether each API is implemented correctly, and that all functions are idempotent. Finally, we reviewed the swap canister's interactions with other privileged SNS canisters, looking for race conditions or TOCTOU issues.

- **NNS governance canister:** We manually reviewed the parts of the NNS governance canister involved with SNS network creation. We focused on community fund (recently renamed to neuron fund) participation, ensuring that contributions are calculated correctly, looking for potential overflow issues or other edge cases. We also checked that the relevant APIs on the NNS governance canister implement correct access controls, and that community fund contributions are handled correctly when an SNS token swap fails.

- **SNS governance canister:** We reviewed neuron and proposal management, assessing whether each function implements the correct authorization and access control checks. In particular, we tried to find ways to bypass the restricted `PreInitializationSwap` mode using SNS governance. We reviewed whether all APIs check the governance canister mode to ensure that privileged APIs can be only called in `Normal` mode. Here, we focused on the handling of `GenericNervousSystemFunction` functions to assess whether they can be used to bypass the governance mode. Finally, we looked for race conditions and TOCTOU issues related to state management during asynchronous inter-canister calls.

- **ICRC-1 ledger:** We reviewed the APIs related to the ticket-based payment flow for correctness and idempotency. We also reviewed the new fee management implementation to detect issues related to correctness or arithmetic. Finally, we reviewed how the implementation handles large memo fields.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Because of time constraints, we did not perform a complete review of the SNS governance canister or the ICRC-1 ledger. Instead, we focused on security-critical paths in the SNS governance canister and changes introduced to the ICRC-1 ledger since our last review in September of 2022, like the ticket-based payment flow. We also did not manage to review the TLA+ models provided for the SNS during the engagement.
- Because of issues with the Bazel-based build system, we were not able to generate test coverage data for any of the SNS canisters.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
`bazel coverage`	A Bazel sub-command for generating test coverage reports	Appendix C
`cargo-audit`	A Cargo plugin for reviewing project dependencies for known vulnerabilities	Appendix C
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards	Appendix C

Areas of Focus

Our automated testing and verification work focused on detecting the following issues:

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of panicking functions, like `unwrap` and `expect`
- General issues with dependency management and known vulnerable dependencies
- Poor unit and integration test coverage

Test Results

The results of this focused testing are detailed below.

SNS: We ran Clippy on the swap canister, ICRC-1 ledger, and SNS governance canister implementations to identify issues related to general code quality and unidiomatic Rust. We also ran Semgrep using a number of custom rules to identify common issues related to

error handling and panicking functions. Finally, we used `cargo-audit` to identify any issues related to dependency management and known-vulnerable dependencies. This identified a number of dependencies of the `ic` monorepo with known vulnerabilities, but none of them are direct dependencies of the SNS.

We also attempted to obtain test coverage data using the `bazel coverage` sub-command. However, obtaining accurate test coverage reports for individual components of the system is currently not supported by the build system used by DFINITY, and we were not able to achieve this during the engagement.

Property	Tool	Result
All components of the codebase have sufficient test coverage.	Bazel `coverage`	**Further Investigation Required**
The project does not depend on any libraries with known vulnerabilities.	`cargo-audit`	Appendix C
The project adheres to Rust best practices by fixing code quality issues reported by linters such as Clippy.	Clippy	**Passed**
The project's use of panicking functions such as `unwrap` and `expect` is limited.	Semgrep	**Passed**

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The SNS governance and swap canisters represent non-negative integers as `u64` values and mostly use either checked or saturating arithmetic to prevent overflows. If standard arithmetic operations are used, comments are included to explain why the operations are safe. However, some of the arithmetic in the SNS governance and swap canisters could be simplified by defining amounts using custom types that implement either checked or saturating arithmetic by default. The ICRC-1 ledger uses the `Tokens` type to represent token amounts, which does implement checked arithmetic by default.	**Satisfactory**
Auditing	The reviewed SNS canisters all implement sufficient logging and provide descriptive error messages. This makes it easier to investigate incidents and track unexpected events.	**Satisfactory**
Authentication / Access Controls	All of the reviewed canisters expose privileged methods that should be callable only by a restricted set of principals. Access controls are implemented correctly throughout the reviewed canisters, but the actual checks are sometimes difficult to find, which makes reviewing the canisters difficult. This is particularly true for the SNS governance canister, where fine-grained access controls can be defined for each neuron.	**Moderate**
Complexity Management	The implementation of the SNS is broken up into a number of distinct canisters, each with a small and well-defined set of responsibilities. This makes	**Satisfactory**

	development and code review easier, but it also increases the number of inter-canister interactions. This increases the risk of race conditions and reentrancy issues during inter-canister calls. It also means that a small issue in one component may cause cascading, and more serious, failures in different components of the system. That being said, we did not identify any issues related to inter-canister interactions during this review.	
Configuration	When a new SNS is created and opened, configuration options are supplied as part of the corresponding NNS proposals. The SNS configuration options are validated by end users during the proposal process, as well as by the relevant SNS canisters during canister creation. The SNS configuration can also be updated through SNS proposals. These are heavily restricted until the SNS governance canister has entered normal mode to ensure the integrity of the network.	**Satisfactory**
Cryptography and Key Management	The reviewed canisters do not implement any cryptography.	**Not Considered**
Data Handling	All of the reviewed canisters implement careful input validation for both trusted and untrusted inputs.	**Satisfactory**
Decentralization	The SNS provides decentralized governance for applications deployed on the Internet Computer. The creation of a new network is governed by an NNS proposal, and updates to a deployed SNS are governed by SNS proposals, both of which require a majority vote to pass. We found no ways for an attacker to influence the creation of a new SNS, the SNS token swap, or the SNS governance function.	**Strong**
Documentation	The SNS implementation is well documented. As part of the review, we were provided access to ample high-level documentation describing system design goals and canister interactions. We also found the codebase to be well documented, with comments explaining low-level details and particular design choices.	**Strong**

Maintenance	All SNS canisters use the standard canister upgrade mechanism, which allows for timely upgrades if issues are identified. Using `cargo-audit`, we identified a number of dependencies of the `ic` monorepo with known vulnerabilities (appendix C). None of them are direct dependencies of the SNS.	**Moderate**
Memory Safety and Error Handling	The reviewed canisters contain very little unsafe code, all of which is related to global state management. Errors are generally logged and propagated to the caller along with descriptive error messages. In cases where panicking functions like `unwrap` and `expect` are used, it is typically clear why the call should not fail. However, since many mandatory fields on types representing messages and canister state are optional to allow for updates, panicking functions often have to be used extensively, which makes the code more difficult to read.	**Moderate**
Testing and Verification	The reviewed canisters are all covered by extensive test suites, implementing positive and negative unit tests, integration tests, and property tests for a number of invariants of the implementation. Parts of the system have also been modeled using TLA+, which increases confidence in the system design. However, the current Bazel-based build system makes it difficult to obtain test coverage data for the reviewed canisters, which means that it is impossible to determine whether there are any gaps in the current test coverage of the implementation.	**Further Investigation Required**

Summary of Findings

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

ID	Title	Type	Severity
1	Uneven distribution of stake across neurons may impact SNS governance	Data Validation	Low
2	Wrong error message returned from new_sale_ticket in Adopted state	Error Reporting	Medium
3	Swap canister paging implementations panic on invalid ranges	Error Reporting	Informational
4	The NNS governance canister always warns about missing neurons if a token swap fails	Data Validation	Informational

Detailed Findings

1. Uneven distribution of stake across neurons may impact SNS governance

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DFSNSR-1
Target: `sns/swap/src/swap.rs`	

Description

When a user stakes ICP to obtain SNS tokens, the tokens are distributed according to a vesting schedule defined by the `NeuronBasketConstructionParameters` type. The SNS tokens are distributed as a number of SNS neurons with staggered dissolve delays to ensure that all neurons do not dissolve at the same time. Because of how this vesting schedule is determined, the entire staked amount may vest immediately in exceptional cases.

The number of neurons obtained and the amount of time it takes for each neuron to dissolve are both determined by the proposal to open the SNS swap. These two parameters are then used by the `apportion_approximately_equally` function to determine the dissolve delays for the individual neurons.

```
pub fn apportion_approximately_equally(total: u64, len: u64) -> Vec<u64> {
    assert!(len > 0, "len must be greater than zero");
    let quotient = total.saturating_div(len);
    let remainder = total % len;

    let mut result = vec![quotient; len as usize];
    *result.first_mut().unwrap() += remainder;

    result
}
```

Figure 1.1: Here, `len` represents the number of neurons distributed to a single user, and `total` is the number of SNS tokens staked by the same user. (`sns/swap/src/swap.rs:214-223`)

Since the number of staked SNS tokens (`total` in figure 1.1) is given as SNS e8s, it will generally be much greater than the number of SNS neurons obtained (`len` in figure 1.1). However, this is not guaranteed by the implementation, since these numbers are set by the originator of the proposal. If the SNS token amount could be similar in size to the number of neurons, most (or even all) neurons would dissolve immediately. For example, eight SNS

e8s distributed over 10 SNS neurons would result in the vesting schedule `[8, 0, 0, …, 0]`, where the entire amount (8 SNS e8s) is vested immediately. This would then affect governance of the SNS adversely, since dissolved neurons are not able to participate.

Exploit Scenario

A group of users misconfigures the parameters for an SNS token swap. This goes undetected, and the proposal to open the token swap is passed. When the token swap completes, all of the distributed SNS neurons dissolve immediately, making governance of the network impossible.

Recommendations

Short term, ensure that the remainder in `apportion_approximately_equally` is distributed evenly over the first `total % len` neurons. This would ensure that the distributed amounts differ by, at most, one.

Long term, document implicit assumptions made for the utility functions that each component relies on. Ensure that each assumption is properly tested using unit or property testing.

2. Wrong error message returned from new_sale_ticket in Adopted state

Severity: **Medium**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-DFSNSR-3
Target: `sns/swap/src/gen/ic_sns_swap.pb.v1.rs`	

Description

If the token swap canister is in the lifecycle state `Adopted`, then the proposal to open the swap has been adopted, but the token swap has not yet been opened. If a user calls `new_sale_ticket` to obtain a new ticket from the swap canister in this state, the method will return a `NewSaleTicketResponse::err_sale_closed()` error, erroneously signaling that the sale is closed.

The implementation of `new_sale_ticket` uses the partial ordering defined on the `Lifecycle` enum to determine which error to return to the user if the current state is not `Open`.

```
pub fn new_sale_ticket(
    &mut self,
    request: &NewSaleTicketRequest,
    caller: PrincipalId,
    time: u64,
) -> NewSaleTicketResponse {
    if self.lifecycle() < Lifecycle::Open {
        return NewSaleTicketResponse::err_sale_not_open();
    }
    if self.lifecycle() > Lifecycle::Open {
        return NewSaleTicketResponse::err_sale_closed();
    }

    // ...
}
```

Figure 3.1: The derived partial order on `Lifecycle` is used to determine the swap canister state. (`sns/swap/src/swap.rs:1944-1949`)

The implementation of `PartialOrd` for `Lifecycle` is automatically derived from the `Lifecycle` enum definition.

```
pub enum Lifecycle {
    /// The canister is incorrectly configured. Not a real lifecycle state.
    Unspecified = 0,
    /// In PENDING state, the canister is correctly initialized. Once SNS
    /// tokens have been transferred to the swap canister's account on
    /// the SNS ledger, a call to `open` with valid parameters will start
    /// the swap.
    Pending = 1,
    /// In ADOPTED state, the proposal to start decentralization sale
    /// has been adopted, and the sale can be opened after a delay
    /// specified by params.sale_delay_seconds.
    Adopted = 5,
    /// In OPEN state, prospective buyers can register for the token
    /// swap. The swap will be committed when the target (max) ICP has
    /// been reached or the swap's due date/time occurs, whichever
    /// happens first.
    Open = 2,
    /// In COMMITTED state the token price has been determined; on a call to
    /// finalize`, buyers receive their SNS neurons and the SNS governance canister
    /// receives the ICP.
    Committed = 3,
    /// In ABORTED state the token swap has been aborted, e.g., because the due
    /// date/time occurred before the minimum (reserve) amount of ICP has been
    /// retrieved. On a call to `finalize`, participants get their ICP refunded.
    Aborted = 4,
}
```

Figure 3.2: The partial order for lifecycle states is derived from the standard ordering of the corresponding discriminants. (sns/swap/src/gen/ic_sns_swap.pb.v1.rs:2122-2147)

According to the documentation for the `std::cmp::PartialOrd` trait, the derived order is given by the order induced by the corresponding discriminants. This means that `Adopted` > `Open` since the corresponding discriminant for `Adopted` (5) is greater than the discriminant of `Open` (2). It follows that `new_sale_ticket` will return the wrong error message when the swap canister is in `Adopted` state.

The same issue is present in the implementation of `get_open_ticket`.

Exploit Scenario

Alice uses a scripted front-end application to participate in the token swap. When Alice starts the application, the swap proposal has been adopted, but has not yet been opened. When the application attempts to obtain a new ticket from the swap canister to initiate Alice's first purchase, the swap canister returns an error message indicating that the sale is already closed. This causes the application to shut down, which prevents Alice from participating in the token sale.

Recommendations

Short term, add a custom implementation of the `PartialOrd` trait for the `Lifecycle` enum, ensuring that `B` ≥ `A` if and only if `B` is reachable from `A`.

Long term, make sure that all derived traits are covered by property tests to ensure that they work as expected.

3. Swap canister paging implementations panic on invalid ranges

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-DFSNSR-4
Target: `sns/swap/src/swap.rs`	

Description

The swap canister implements a number of APIs to list swap participants and neuron recipes. All of them implement paging using an offset and a limit to allow callers to request the entire set of participants over a sequence of calls.

However, the paging implementations in `list_community_fund_participants` and `list_sns_neuron_recipes` both use the caller-supplied offsets and limits to index into a vector of elements. The upper bound for the range (end) is bounded by the length of the vector, but if the caller-supplied offset falls outside of the vector, the implementation will still panic.

```
pub fn list_community_fund_participants(
    &self,
    request: &ListCommunityFundParticipantsRequest,
) -> ListCommunityFundParticipantsResponse {
    let ListCommunityFundParticipantsRequest { limit, offset } = request;
    let offset = offset.unwrap_or_default() as usize;
    let limit = limit
        .unwrap_or(DEFAULT_LIST_COMMUNITY_FUND_PARTICIPANTS_LIMIT) // use default
        .min(LIST_COMMUNITY_FUND_PARTICIPANTS_LIMIT_CAP) // cap
        as usize;

    let end = (offset + limit).min(self.cf_participants.len());
    let cf_participants = self.cf_participants[offset..end].to_vec();

    ListCommunityFundParticipantsResponse { cf_participants }
}
```

Figure 4.1: `Swap::list_community_fund_participants` may index out of bounds. (sns/swap/src/swap.rs:2407-2422)

```
pub fn list_sns_neuron_recipes(
    &self,
    request: ListSnsNeuronRecipesRequest,
) -> ListSnsNeuronRecipesResponse {
    let limit = request
        .limit
        .unwrap_or(DEFAULT_LIST_SNS_NEURON_RECIPES_LIMIT)
        .min(DEFAULT_LIST_SNS_NEURON_RECIPES_LIMIT) as usize;

    let start_at = request.offset.unwrap_or_default() as usize;
    let end = (start_at + limit).min(self.neuron_recipes.len());

    let sns_neuron_recipes = self.neuron_recipes[start_at..end].to_vec();

    ListSnsNeuronRecipesResponse { sns_neuron_recipes }
}
```

Figure 4.2: `Swap::list_sns_neuron_recipes` may index out of bounds. (sns/swap/src/swap.rs:2456-2471)

Recommendations

Short term, check the caller-provided offset against the size of the underlying vector and return an empty list or an error if the given offset would cause the function to index out of bounds.

Long term, use property testing to discover panicking edge cases in the implementation.

4. The NNS governance canister always warns about missing neurons if a token swap fails

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-DFSNSR-5
Target: `sns/swap/src/swap.rs`	

Description

If a token swap fails, the contribution from the community fund (recently renamed to neuron fund) is refunded to the contributing neurons by the NNS governance canister. This logic is implemented by the `refund_community_fund_maturity` function. If the function fails to refund the contributed maturity to one or more neurons, these neurons are returned from the function, and the missing neurons are logged to `stdout`.

```
let settlement_result = match &request_type {
    settle_community_fund_participation::Result::Committed(committed) => {
        committed
            .mint_to_sns_governance(proposal_data, &*self.ledger)
            .await
    }

    settle_community_fund_participation::Result::Aborted(_aborted) => {
        let missing_neurons = refund_community_fund_maturity(
            &mut self.proto.neurons,
            &proposal_data.cf_participants,
        );
        println!(
            "{}WARN: Neurons are missing from Governance when attempting to refund \
            community fund participation in an SNS Sale. Missing Neurons: {:?}",
            LOG_PREFIX, missing_neurons
        );
        Ok(())
    }
};
```

Figure 5.1: The list of missing neurons is always logged by the NNS governance canister. (nns/governance/src/governance.rs:6763-6782)

However, this log message is emitted even if the list of neurons returned from the `refund_community_fund_maturity` function is empty, which means that the NNS

governance canister will always claim that there are missing neurons, even when all neurons have been accounted for by the `refund_community_fund_maturity` function.

Recommendations

Short term, check whether the list of missing neurons returned from `refund_community_fund_maturity` is non-empty before emitting the log message.

Long term, review log messages to ensure that they are correct.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Bazel Coverage

The `bazel coverage` sub-command can be used to generate test coverage reports for repositories using a Bazel-based build system. Bazel can be installed using the wrapper command `bazelisk`. To run Bazel to obtain test coverage data for a given target, run `bazel coverage --combined_report=lcov <TEST TARGET>`.

Since DFINITY recently switched to using Bazel, the team still has issues with generating test coverage data using the new build system. For this reason, we were not able to obtain accurate test coverage reports during the engagement.

cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

Running `cargo-audit` on the codebase identified a number of dependencies with known vulnerabilities. None of them are direct dependencies of the SNS canisters under review, but we still recommend that the DFINITY team investigate and update them.

Dependency	Version	ID	Description
`rocksdb`	0.15.0	RUSTSEC-2022-0046	Out-of-bounds read when opening multiple column families with TTL
`time`	0.1.45	RUSTSEC-2020-0071	Potential segfault in the time crate
`ansi_term`	0.11.0, 0.12.1	RUSTSEC-2021-0139	`ansi_term` is unmaintained
`difference`	2.0.0	RUSTSEC-2020-0095	`difference` is unmaintained
`libusb`	0.3.0	RUSTSEC-2016-0004	`libusb` is unmaintained; use `rusb` instead
`mach`	0.3.2	RUSTSEC-2020-0168	`mach` is unmaintained
`net2`	0.2.39	RUSTSEC-2020-0016	`net2` crate has been deprecated; use `socket2` instead
`serde_cbor`	0.11.2	RUSTSEC-2021-0127	`serde_cbor` is unmaintained

`term`	0.6.1	RUSTSEC-2018-0015	`term` is looking for a new maintainer
`wee_alloc`	0.4.5	RUSTSEC-2022-0054	`wee_alloc` is unmaintained
`atty`	0.2.14	RUSTSEC-2021-0145	Potential unaligned read
`borsh`	0.10.3	RUSTSEC-2023-0033	Parsing `borsh` messages with ZST which are not-copy/clone is unsound

Table C.1: Project dependencies with known vulnerabilities

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

Running Clippy on the codebase did not identify any issues.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a rule set hosted on the Semgrep registry.

Running Semgrep on the codebase did not identify any issues.

D. Code Quality Recommendations

The following section contains code quality recommendations that do not have any immediate security implications.

- **Update the comment for the sale neuron memo range.** The constant `SALE_NEURON_MEMO_RANGE_END` is not used by the swap canister. The comment for the memo range may cause readers to assume that memos will always be in this range, but this is not enforced by the implementation.

```
/// Range of allowed memos for neurons distributed via an SNS sale. This
/// range is used to choose the memos of Sale neurons, and to enforce that
/// other memos (e.g. for Airdrop neurons) do not conflict with the memos of
/// Sale neurons.
pub const SALE_NEURON_MEMO_RANGE_START: u64 = 1_000_000;
pub const SALE_NEURON_MEMO_RANGE_END: u64 = 10_000_000;
```

Figure D.1: `SALE_NEURON_MEMO_RANGE_END` (sns/swap/src/swap.rs)

- **Update the unit used in the log message.** The following log message claims that the neuron age is given as seconds, but in fact, the number is given in nanoseconds.

```
log!(
    INFO,
    "Purging {} open tickets because they are older than {} seconds
        (number of open tickets: {})",
    to_purge.len(),
    max_age_in_nanoseconds,
    tickets.borrow().len(),
);
```

Figure D.2: `Swap::purge_old_tickets` (sns/swap/src/swap.rs)

- **Update the comment for `FIRST_PRINCIPAL_BYTES`.** The inequality in the comment describing the constant `FIRST_PRINCIPAL_BYTES` is reversed and should be fixed.

```
/// The principal with all bytes set to zero. The main property
/// of this principal is that for any principal p != FIRST_PRINCIPAL_BYTES
/// then p.as_slice() < FIRST_PRINCIPAL_BYTES.as_slice().
pub const FIRST_PRINCIPAL_BYTES: [u8; PrincipalId::MAX_LENGTH_IN_BYTES] =
    [0; PrincipalId::MAX_LENGTH_IN_BYTES];
```

Figure D.3: `FIRST_PRINCIPAL_BYTES` (sns/swap/src/swap.rs)

- **Update the comment for `Swap::can_open`.** The `Swap::can_open` method is used to check whether the swap can be opened in the `Adopted` state. However, the name of the method and the corresponding comment are both misleading, as the swap could also be opened immediately from a `Pending` state. However, in this case, the method will return `false`.

```
/// Returns true if the swap can be opened at the specified
/// timestamp, and false otherwise.
pub fn can_open(&self, now_seconds: u64) -> bool {
    if self.lifecycle() != Lifecycle::Adopted {
        return false;
    }
    self.sale_opening_due(now_seconds)
}
```

Figure D.4: `Swap::can_open` (sns/swap/src/swap.rs)

- **Remove the `ValidGovernanceProto::validate_canister_id_field` method.** The `ValidGovernanceProto::validate_canister_id_field` (in `sns/governance/src/governance.rs`) is used by the SNS governance canister to validate SNS canister IDs. The method does nothing and always returns `Ok`. It should be removed so that it does not falsely indicate that validation has occurred.

- **Avoid using `as` when casting from larger to smaller integer types.** The `Governance::check_neuron_population_can_grow` method (in `sns/governance/src/governance.rs`) uses `as` to cast from `u64` to `usize` (which is 32 bits on 32-bit targets like `wasm32-unknown-unknown`). Avoid using `as` when casting from larger to smaller integer types, since this can cause truncation issues if the developer is not careful.

```
let max_number_of_neurons = self
    .nervous_system_parameters_or_panic()
    .max_number_of_neurons
    .expect("NervousSystemParameters must have max_number_of_neurons")
    as usize;
```

Figure D.5: `Governance::check_neuron_population_can_grow` (sns/governance/src/governance.rs)