HashEye

// SMART CONTRACT SECURITY AUDIT

Dfinity

Security assessment by HashEye · prepared for DFINITY

HASHEYE AUDITED

PROJECT	Dfinity
CLIENT	DFINITY
CATEGORY	Blockchain
PUBLISHED	September 1, 2022
REPORT ID	research-dfinity-2022-09-01-13z0vm

This report was produced under HashEye's layered review process — **automated detection**, **pattern correlation**, and **senior manual verification** — with every finding signed off by a human reviewer. Full findings detail and on-chain attestation are available on the report page at hasheye.io/audits/research-dfinity-2022-09-01-13z0vm.

DFINITY Service Nervous System Security Assessment December 1, 2022 Prepared for: Robin Künzler DFINITY Prepared by: Fredrik Dahlgren and Will Song

About HashEye Founded in 2012 and headquartered in New York, HashEye 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/hasheye-io/publications , with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye 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. HashEye 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 hasheye on Twitter and explore our public repositories at https://github.com/hasheye-io . To engage us directly, visit our "Contact" page at https://www.hasheye.io/contact , or email us at info@hasheye.io . HashEye, Inc. 228 Park Ave S #80688 New York, NY 10003 https://www.hasheye.io info@hasheye.io HashEye 1 DFINITY Security Assessment PUBLIC

 Test Coverage Disclaimer All activities undertaken by HashEye 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. HashEye 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. HashEye 2 DFINITY Security Assessment PUBLIC

Table of Contents

Executive Summary Engagement Overview DFINITY engaged HashEye to review the security of its Service Nervous System (SNS). From September 12 to September 23, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report. On February 9, 2023 , HashEye reviewed the fixes and mitigations implemented by DFINITY for the issues identified in this report. The result of this review is found in appendix A . Project Scope Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this

audit with full knowledge of the system, including access to all relevant source code and documentation. We performed static and dynamic testing of the target system, along with a manual review of the codebase. Summary of Findings The audit uncovered one significant flaw (TOB-DFSNS-1) that could impact system availability. A summary of this finding is provided in the “Notable Findings” section. Overall, we found the implementation of the SNS to be well documented, well structured, and written defensively in accordance with Rust best practices. The unit and integration test coverage of all canisters appears to be good, and there is ample negative testing of potential edge cases. However, because we chose to prioritize a manual code review, we were not able to obtain reliable test coverage data. Going forward, we recommend that DFINITY implement secure abstractions for common operations like caller authentication. Currently, each privileged system canister API must implement its own authentication mechanism, which means that there are multiple custom authentication checks throughout the system (and throughout most individual SNS canisters). Using a common API for such operations would make them easier to implement and review. The DFINITY codebase uses kcov to obtain test coverage data. Currently, this tool is supported on Linux, but not on MacOS. Since accurate code coverage data is invaluable when developing unit and integration tests, we also recommend that the team implement a way to easily obtain test coverage data on all supported development platforms. HashEye 5 DFINITY Security Assessment PUBLIC

EXPOSURE ANALYSIS Severity Count High 0 Medium 1 Low 2 Informational 4 Undetermined 0 CATEGORY BREAKDOWN Category Count Access Controls 1 Configuration 1 Data Validation 3 Denial of Service 1 Session Management 1 Notable Findings A significant flaw that impacts system confidentiality, integrity, or availability is detailed below. • Use of a custom SNS token transfer fee causes the creation of SNS neurons to fail (TOB-DFSNS-1) If the SNS token transfer fee is different from the default Internet Computer Protocol (ICP) transfer fee, the swap canister will fail to create new SNS neurons once the swap is committed. As a result, the SNS network governance will become nonoperational. This issue could have been identified through more comprehensive testing of the different SNS network configurations. HashEye 6 DFINITY Security Assessment PUBLIC

Project Summary Contact Information The following managers were associated with this project: Dan Guido , Account Manager Cara Pearson , Project Manager dan@hasheye.io cara.pearson@hasheye.io The following engineers were associated with this project: Fredrik Dahlgren , Consultant Will Song , Consultant fredrik.dahlgren@hasheye.io will.song@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event September 8, 2022 Pre-project kickoff call September 19, 2022 Status update meeting #1 October 11, 2022 Delivery of report draft; report readout meeting December 1, 2022 Delivery of final report February 22, 2023 Delivery of fix review HashEye 7 DFINITY Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the DFINITY SNS. Specifically, we sought to answer the following non-exhaustive list of questions: • Do the distinguished SNS canisters implement sufficient access controls for privileged methods? • Are canister global states handled in a way that prevents race conditions and reentrancy issues during inter-canister calls? • Are canister states rolled back correctly if an inter-canister call fails? • Are canister states preserved during upgrades? • Is the swap canister state machine implemented correctly? • Is the ICP and SNS token arithmetic implemented securely? HashEye 8 DFINITY Security Assessment PUBLIC

Project Targets The engagement involved a review and testing of the swap canister and the SNS root, ledger, governance, and Wasm canisters, including their integration points with the Network Nervous System (NNS). We reviewed code from the two commits listed below. Service Nervous System Repository https://github.com/dfinity/ic Versions de76d8c75849681a050de64945b64cc43f448dbc d0364fcc58486a7750a3d3306f9d0e618acf6256 Type Rust Platform Linux HashEye 9 DFINITY Security Assessment PUBLIC

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following: • We used Clippy and Semgrep to identify potentially problematic uses of panicking functions and instances of unidiomatic Rust code. We then manually reviewed the code they identified. • We manually reviewed the access control checks implemented by the in-scope SNS canisters (the swap canister and the SNS ledger, root, governance, and Wasm canisters) to ensure that privileged methods can be called only by the expected caller. • We manually reviewed the correctness of the canister state machines; we focused on the way that state is managed during inter-canister calls, seeking to identify potential race conditions and reentrancy issues. We also reviewed the way in which the canister state is rolled back if an inter-canister call fails. • We manually reviewed the token amount and neuron maturity computations performed by the NNS governance canister, SNS swap canister, and SNS ledger, looking for over- and underflows, rounding issues, and unit discrepancies. • Because the swap canister is new, we devoted extra time to reviewing the swap canister state machine and its interactions with the NNS and SNS governance canisters and SNS root

canister. We focused on the canister's behavior during swap finalization. Coverage Limitations Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we focused on achieving thorough manual coverage of the SNS canisters; as a result, we did not have time to obtain reliable data on the system's test coverage. HashEye 10 DFINITY Security Assessment PUBLIC

Automated Testing HashEye 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 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 when editing or committing code and during build time Appendix C cargo-llvm-cov A Cargo subcommand for generating LLVM source-based code coverage 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 unwrap and expect • Poor unit and integration test coverage HashEye 11 DFINITY Security Assessment PUBLIC

Test Results The results of this focused testing are detailed below. SNS. We used Clippy, Semgrep, and cargo-llvm-cov to analyze the distinguished canisters (the SNS governance, ledger, root, swap, and Wasm canisters) that implement the core functionality of the system. Property Tool Result The project adheres to Rust best practices by fixing code quality issues reported by linters like Clippy. Clippy Passed The project's use of panicking functions like unwrap and expect is limited. Semgrep TOB-DFSNS-3 , TOB-DFSNS-7 All components of the codebase have sufficient test coverage. cargo-llvm-cov Further Investigation Required HashEye 12 DFINITY Security Assessment PUBLIC

Codebase Maturity Evaluation HashEye 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 All of the canisters represent non-negative integers as u64 values and use both checked and saturating arithmetic to prevent over- and underflows. The ledger canister uses the Tokens type to represent token amounts; this type implements checked arithmetic by default. Strong Auditing Each distinguished SNS canister implements sufficient logging and provides descriptive error messages. Strong Authentication / Access Controls A number of the distinguished SNS canisters export privileged methods that should be reachable only by a predefined set of canisters. Each canister enforces the correct access controls, but these authentication controls are dispersed across each canister implementation. This makes reviewing the canisters' access controls more difficult than it should be. Satisfactory Complexity Management The SNS is implemented by a number of separate canisters and relies on complex inter-canister interactions. This design increases the risk of race conditions and reentrancy issues during inter-canister calls. The code is otherwise well written and well structured, and both the NNS and SNS ledgers effectively use traits to prevent code duplication. Satisfactory HashEye 13 DFINITY Security Assessment PUBLIC

Configuration We found one SNS configuration issue (TOB-DFSNS-1); specifically, the use of a custom transfer fee instead of the default ICP transfer fee will cause the bootstrapping of the system to fail. This issue indicates that the SNS would benefit from more extensive testing of its custom system configuration options. Moderate Cryptography and Key Management The implementation does not contain any cryptography or key management functionality. Not Applicable Data Handling All SNS canisters perform sufficient validation of the system initialization parameters. We also found that all canister APIs implement sufficient input validation. Strong Decentralization Once the SNS network has been configured, there is no way for an attacker to gain an unfair advantage in a token swap or to manipulate SNS governance. Strong Documentation DFINITY provided ample documentation on the SNS. This includes both high-level documentation on the background of the system and more low-level documentation describing the implementation. Strong Maintenance All SNS canisters use the standard canister upgrade mechanism. Strong Memory Safety and Error Handling The implementation contains very little unsafe code, most of which is related to global state management. Errors are typically propagated to the caller along with descriptive error messages. Satisfactory Testing and Verification Because we prioritized achieving good manual coverage of the SNS canisters, we did not have time to obtain reliable data on the system's test coverage. Further Investigation Required HashEye 14 DFINITY Security Assessment PUBLIC

Summary of Findings The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity 1 Use of a custom transfer fee causes the creation of SNS neurons to fail Configuration Medium 2 Failure to ensure that all neurons have been created before

the transition to Normal mode Data Validation Informational 3 Unnecessary calls to unwrap in get_root_status Data Validation Informational 4 Erroneous controller check in SnsRootCanister::set_dapp_controllers Data Validation Low 5 Accounts with low balances are trimmed from the ICRC-1 ledger Session Management Informational 6 Potentially harmful remove_self_as_controller pattern Access Controls Informational 7 Use of panicking functions poses a risk to the ledger's archiving mechanism Denial of Service Low HashEye 15 DFINITY Security Assessment PUBLIC

Detailed Findings 1. Use of a custom transfer fee causes the creation of SNS neurons to fail Severity: Medium Difficulty: Not Applicable Type: Configuration Finding ID: TOB-DFSNS-1 Target: sns/swap/src/swap.rs Description When a token swap is committed or aborted, the Swap::finalize method is invoked to transfer funds and (if the swap was successful) create new Service Nervous System (SNS) neurons. When a swap is committed, the method calls the Swap::sweep_sns method to transfer the SNS tokens due to each buyer to the buyer's SNS neuron subaccount in the SNS governance canister. The sweep_sns method takes a transfer fee that must be equal to the fee defined during the SNS network's configuration. However, Swap::finalize uses the default Internet Computer Protocol (ICP) transfer fee instead. let sweep_sns = self .sweep_sns(now_fn, DEFAULT_TRANSFER_FEE , sns_ledger) . await ; Figure 1.1: The DEFAULT_TRANSFER_FEE passed to Swap::sweep_sns in Swap::finalize is that used in ICP token transfers. The actual token transfer is handled by the ICRC1Client::transfer method, which calls the icrc1_transfer endpoint on the SNS ledger to transfer the funds to the governance canister. As shown in the implementation of icrc1_transfer in figure 1.2, if the SNS transfer fee is different from the default ICP transfer fee, this call will fail. async fn icrc1_transfer (arg: TransferArg) → Result <Nat, TransferError> { let block_idx = Access::with_ledger_mut(|ledger| { // ... <redacted> let tx = if &arg.to == ledger.minting_account() { // ... <redacted> } else if &from_account == ledger.minting_account() { // ... <redacted> } else { let expected_fee_tokens = ledger.transfer_fee(); HashEye 16 DFINITY Security Assessment PUBLIC

let expected_fee = Nat::from(expected_fee_tokens.get_e8s()); if arg.fee.is_some() && arg.fee.as_ref() ≠ Some (&expected_fee) { return Err (TransferError::BadFee { expected_fee }); } Transaction::transfer(from_account, arg.to, amount, expected_fee_tokens, created_at_time, arg.memo,) }; let (block_idx, _) = apply_transaction(ledger, tx, now)?; Ok (block_idx) })?; } Figure 1.2: SNS token transfers will fail if the transfer fee expected by the SNS ledger (i.e., that defined during configuration) is different from the default ICP transfer fee used by the swap canister. In that case, all SNS token transfers will fail, and no new SNS governance neurons will be created. Because subsequent calls to Swap::finalize will fail in the same way, the SNS will become stuck in PreInitializationSwap mode. Moreover, by the time the system is in that state, the swap canister will have already transferred all buyer ICP tokens to the SNS governance canister; that means that buyers will be unable to request a refund by using the error_refund_icp API. Recommendations Short term, pass the SNS token transfer fee as an argument to the swap canister during canister creation, and ensure that the swap canister uses the correct transfer fee in all SNS token transfers. Long term, implement integration tests to check the correctness of token swaps that use SNS token transfer fees different from the default ICP transfer fee. Additionally, consider whether the default fee should be removed from the ledger API, since the fee cannot vary. HashEye 17 DFINITY Security Assessment PUBLIC

2. Failure to ensure that all neurons have been created before the transition to Normal mode Severity: Informational Difficulty: Not Applicable Type: Data Validation Finding ID: TOB-DFSNS-2 Target: sns/swap/src/swap.rs Description The Swap::set_sns_governance_to_normal_mode_if_all_neurons_claimed method sets the SNS governance mode to Normal , which unlocks functionality like the payout of SNS neuron rewards. The method's name indicates that Normal mode is enabled only if all neurons have been claimed. However, the method checks only whether create_neuron.failure is 0; it fails to check whether create_neuron.skipped is also 0. async fn set_sns_governance_to_normal_mode_if_all_neurons_claimed (sns_governance_client: & mut impl SnsGovernanceClient, create_neuron: & SweepResult ,) → Option <SetModeCallResult> { let all_neurons_created = create_neuron.failure == 0 ; if !all_neurons_created { return None ; } Some (sns_governance_client .set_mode(SetMode { mode: governance ::Mode::Normal as i32 , }) . await .into(),) } Figure 2.1: The function checks the create_neuron.failure field when verifying that all new neurons have been successfully created; however, create_neuron.skipped must also be 0. The create_neuron.skipped field is set to the skipped value returned by Swap::investors_for_create_neuron , which will be greater than 0 if any transfers of SNS tokens to the governance canister have not yet finished. Such transfers could conceivably fail, in which case the mode should not be set to Normal . The HashEye 18 DFINITY Security Assessment PUBLIC

Swap::set_sns_governance_to_normal_mode_if_all_neurons_claimed method should account for that possibility. pub fn investors_for_create_neuron (& self) → (u32 , Vec <Investor>) { if self

.lifecycle() ≠ Lifecycle::Committed { return (self .neuron_recipes.len() as u32 , vec! []); } let mut investors = Vec ::new(); let mut skipped = 0 ; for recipe in self .neuron_recipes.iter() { if let Some (sns) = &recipe.sns { if sns.transfer_success_timestamp_seconds > 0 { if let Some (investor) = &recipe.investor { investors.push(investor.clone()); continue ; } else { println! ("{}WARNING: missing field 'investor'" , LOG_PREFIX); } } } else { println! ("{}WARNING: missing field 'sns'" , LOG_PREFIX); } skipped += 1 ; } (skipped , investors) } Figure 2.2: The skipped value will be greater than 0 if there are transfers of SNS tokens to the governance canister that have not yet finished. We did not find any ways to exploit this issue. Moreover, because the i nvestors_for_create_neuron method is called after the corresponding token swap has been completed, the transfers should finish successfully, given enough time. Hence, we set the severity of this issue to informational. Recommendations Short term, have the swap canister ensure that create_neuron.skipped is also 0 before updating the SNS governance mode to Normal . HashEye 19 DFINITY Security Assessment PUBLIC

3. Unnecessary calls to unwrap in get_root_status Severity: Informational Difficulty: Not Applicable Type: Data Validation Finding ID: TOB-DFSNS-3 Target: sns/root/src/lib.rs Description The get_root_status function is used to obtain the status of the root canister from the governance canister. The function calls unwrap on the result returned by the governance canister and then calls it again when decoding the returned status. This means that the function will panic if either of the two calls fails. However, the calling method, SnsRootCanister::get_sns_canisters_summary , wraps the result returned by get_root_status in a CanisterSummary . Because CanisterSummary will accept None for the canister status value, the get_root_status function could simply return an object of type Option<CanisterStatusResultV2> instead of CanisterStatusResultV2 . In that case, the function would not need to unwrap the result and would thus avoid a panic. async fn get_root_status (env: & impl Environment, governance_id: PrincipalId ,) → CanisterStatusResultV2 { let result = env .call_canister(CanisterId::new(governance_id).unwrap(), "get_root_canister_status" , Encode!(& ()).unwrap(),) . await .map_err(|err| { let code = err. 0. unwrap_or_default(); let msg = err. 1 ; format! ("Could not get root status from governance: {}: {}" , code, msg) }) .unwrap(); Decode! (&result, CanisterStatusResultV2) .unwrap() } Figure 3.1: The get_root_status function will panic if the call to the governance canister fails. HashEye 20 DFINITY Security Assessment PUBLIC

Indeed, this type is used by the get_owned_canister_summary function, which returns CanisterSummary::new_with_no_stat(canister_id) if it cannot obtain a canister status. Using this type rather than panicking in get_root_status would improve the error it reports when it fails to obtain the root canister status from the governance canister. Recommendations Short term, avoid panicking in get_root_status , and update the get_root_status function such that it returns None if the inter-canister call fails. HashEye 21 DFINITY Security Assessment PUBLIC

4. Erroneous controller check in SnsRootCanister::set_dapp_controllers Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-DFSNS-4 Target: sns/root/src/lib.rs Description The SnsRootCanister::set_dapp_controllers method is called to update the controller of each decentralized application canister. Before attempting the update, the method checks that the SNS root canister controls the application canister. The method verifies only that the call to management_canister_client.canister_status returned Ok(_) ; it fails to check that the root canister ID is in the set of controller IDs. let is_controllee = management_canister_client .canister_status(&dapp_canister_id.into()) . await .is_ok() ; assert! (is_controllee, "Operation aborted due to an error; no changes have been made: \ Unable to determine whether this canister (SNS root) is the controller \ of a registered dapp canister ({dapp_canister_id}). This may be due to \ the canister having been deleted, which may be due to it running out \ of cycles."); Figure 4.1: The management canister returns a Result that wraps a CanisterStatusResultV2 containing the list of controllers for the given canister. However, the error management that the method performs when making the update renders the check unnecessary; thus, the check can be removed. If a token swap is aborted, the swap canister calls the set_dapp_controllers API on the root canister to return control of the application canisters to a set of fallback controllers. If the check in SnsRootCanister::set_dapp_controllers were fixed rather than removed, the inter-canister call would cause the swap canister to panic if one of the application canisters had been deleted; in that case, control of the canisters would not be transferred from the SNS canister to the fallback controllers. Recommendations Short term, remove the check highlighted in figure 4.1 from the SnsRootCanister::set_dapp_controllers method. HashEye 22 DFINITY Security Assessment PUBLIC

5. Accounts with low balances are trimmed from the ICRC-1 ledger Severity: Informational Difficulty: Not Applicable Type: Session Management Finding ID: TOB-DFSNS-5 Target: rosetta-api/ledger_canister_core/src/ledger.rs Description If the number of ledger accounts with a non-zero balance reaches 28.1 million (the sum of the fixed values in figure 5.1), the ICRC-1 ledger will immediately burn the assets of the accounts with the lowest balances: Ledger::max_number_of_accounts() + Ledger::accounts_overflow_trim_quantity() Figure 5.1: The

MAX_ACCOUNTS value (28 million) is added to the ACCOUNTS_OVERFLOW_TRIM_QUANTITY value (100,000). The number of accounts trimmed by the ICRC-1 ledger in this way is indicated by the ACCOUNTS_OVERFLOW_TRIM_QUANTITY value (which is set to 100,000). let to_trim = if ledger.balances().store.len() ≥ ledger.max_number_of_accounts() + ledger.accounts_overflow_trim_quantity() { select_accounts_to_trim(ledger) } else { vec! [] }; for (balance, account) in to_trim { let burn_tx = L::Transaction::burn(account, balance, Some (now), None); burn_tx .apply(ledger.balances_mut()) .expect("failed to burn funds that must have existed"); let parent_hash = ledger.blockchain().last_hash; ledger .blockchain_mut() .add_block(L::Block::from_transaction(parent_hash, burn_tx, now)) .unwrap(); } Figure 5.2: If the number of accounts with a non-zero balance grows too large, accounts will be trimmed from the ledger. HashEye 23 DFINITY Security Assessment PUBLIC

This behavior could be very surprising to users and, to our knowledge, is not documented. (For reference, according to Glassnode , there are currently around 85 million Ethereum addresses with a non-zero balance.) If accounts with non-negligible balances are trimmed from the ledger, the practice could cause reputational issues for DFINITY. Moreover, the highlighted if statement condition is most likely wrong. If the total number of accounts tracked by the ledger reaches 28.1 million, the removal of 100,000 accounts will not decrease the number of accounts to less than Ledger::max_number_of_accounts() (which is set to 28 million). Recommendations Short term, update the if statement condition so that it compares the number of accounts to the Ledger::max_number_of_accounts() value. Long term, ensure that the account-trimming behavior is described in both internal and external documentation. HashEye 24 DFINITY Security Assessment PUBLIC

6. Potentially harmful remove_self_as_controller pattern Severity: Informational Difficulty: Not Applicable Type: Access Controls Finding ID: TOB-DFSNS-6 Target: rs/nns/sns-wasm/src/sns_wasm.rs Description During the setup of the governance canister, it is controlled by the SNS-WASM and root canisters. Then, when the SNS-WASM canister no longer needs to perform any privileged operations, its remove_self_as_controller function changes the governance canister's controllers to a predetermined set (via calls to the CanisterApi::set_controllers function). This set consists of only the root canister, so the function removes the SNS-WASM canister as a controller when making that change. This implementation works for the SNS-WASM canister because these controller sets are static and predetermined; however, the function implementation might not be suitable for reuse in a dynamic context. async fn add_controllers (canister_api: & impl CanisterApi, canisters: & SnsCanisterIds ,) → Result <(), String > { let this_canister_id = canister_api.local_canister_id().get(); let set_controllers_results = vec! [// Set Root as controller of Governance. canister_api .set_controllers(CanisterId::new(canisters.governance.unwrap()).unwrap(), vec! [this_canister_id, canisters.root.unwrap()],) . await .map_err(|e| { format! ("Unable to set Root as Governance canister controller: {}" , e) }), // ...]; join_errors_or_ok(set_controllers_results) } Figure 6.1: The initial setting of the controller set HashEye 25 DFINITY Security Assessment PUBLIC

async fn remove_self_as_controller (canister_api: & impl CanisterApi, canisters: & SnsCanisterIds ,) → Result <(), String > { let set_controllers_results = vec! [// Removing self, leaving root. canister_api .set_controllers(CanisterId::new(canisters.governance.unwrap()).unwrap(), vec! [canisters.root.unwrap()],) . await .map_err(|e| { format! ("Unable to remove SNS-WASM as Governance's controller: {}" , e) }), // ...]; join_errors_or_ok(set_controllers_results) } Figure 6.2: The use of a manual rather than dynamic controller-removal process Recommendations Short term, consider adding a remove_controller function to the CanisterApi and having that function call remove_self_as_controller . HashEye 26 DFINITY Security Assessment PUBLIC

7. Use of panicking functions poses a risk to the ledger's archiving mechanism Severity: Low Difficulty: Not Applicable Type: Denial of Service Finding ID: TOB-DFSNS-7 Target: rosetta-api/ledger_canister_core/src/archive.rs Description The ICRC-1 ledger calls the archive_blocks function to archive a prefix of the local blockchain. That function creates an ArchiveGuard that contains a write lock on the global Archive instance of the ledger state. let archive_arc = LA::with_ledger(|ledger| ledger.blockchain().archive.clone()); // NOTE: this guard will prevent another logical thread to start the archiving // process. let _archiving_guard = match ArchivingGuard::new(Arc::clone(&archive_arc)) { Ok (guard) ⇒ guard, Err (ArchivingGuardError::NoArchive) ⇒ { return ; // Archiving not enabled } Err (ArchivingGuardError::AlreadyArchiving) ⇒ { print::<LA>("[ledger] ledger is currently archiving, skipping archive_blocks()"); return ; } }; Figure 7.1: The archive_blocks function takes a write lock on the ledger archive. If a new archive node must be created, the ledger invokes the asynchronous create_and_initialize_node_canister function, which creates a new canister and installs the correct Wasm binary. When the function installs the archive node's Wasm code, it calls unwrap on the encoded arguments passed to it. Because this call is sandwiched between other

asynchronous inter-canister calls, the ledger state will have been persisted to disk between the placement of the lock on the archive and the call to unwrap . This means that if the unwrapping operation fails, the archive will remain locked indefinitely. HashEye 27 DFINITY Security Assessment PUBLIC

let () = spawn::install_code::<Rt>(node_canister_id, Wasm::archive_wasm().into_owned(), Encode!(&Rt::id(), &node_block_height_offset, & Some (node_max_memory_size_bytes)) .unwrap() ,) . await .map_err(|(reject_code, message)| { FailedToArchiveBlocks(format! ("install_code failed; reject_code={}, message={}" , reject_code, message)) })?; Figure 7.2: When a canister's state is locked, the canister should avoid calling panicking functions after making asynchronous inter-canister calls. Normally, this issue would constitute a severe risk to the availability of the ledger canister's archiving functionality. However, because the call to Encode is very unlikely to fail, we set the severity of this issue to low. Calls to panicking functions in situations like this one, in which a canister's state is persisted when part of it is locked, make the implementation more difficult to understand. To determine whether such a call is safe, the reader (whether a developer or external reviewer) must identify the exact conditions under which the function will panic, which is typically nontrivial. Exploit Scenario A DFINITY developer refactors the implementation of Encode and inadvertently introduces an error into it. Later, when the ledger calls create_and_initialize_node_canister , the error causes the call to Encode to fail. As a result, the ledger archive remains in a locked state indefinitely. Recommendations Short term, remove the call to unwrap from the create_and_initialize_node_canister function, and have the function return an error if the call to the Encode macro fails. Long term, review the system canisters to ensure that they do not call panicking functions after making asynchronous inter-canister calls. Alternatively, ensure that the documentation covers the calls to panicking functions and explains why they are safe. HashEye 28 DFINITY Security Assessment PUBLIC

Summary of Recommendations HashEye recommends that DFINITY address the findings detailed in this report and take the following additional steps to further strengthen the overall security posture of the SNS: • Use a custom type for token amounts in the swap canister. The swap canister uses the u64 type to represent token amounts. Although we did not identify any issues related to the swap canister's token arithmetic, we recommend that DFINITY instead use a dedicated type that uses checked arithmetic by default (as it does in the ledger implementation). This would provide stronger protection against over- and underflows in token-related calculations. • Implement a unified mechanism for system canister access controls. It is very common for the system canisters to expose privileged APIs that should be callable only by a small predefined set of canisters. However, the access control mechanisms used to prevent unauthorized calls are usually unique to each canister. We recommend that the DFINITY team implement a common access control API for the NNS and SNS canisters. A unified mechanism would make the access controls easier to review, which would reduce the risk of implementation mistakes. • Implement tooling for measuring unit and integration test coverage on all supported platforms. The codebase uses kcov to obtain data on the unit and integration test coverage of all Rust crates. Because the tool is currently supported only on Linux, it is difficult and time consuming to obtain accurate code coverage data when developing on MacOS. Enabling developers to obtain accurate code coverage results on their platform of choice typically leads to a tighter feedback loop during development, along with better test coverage (and thus fewer bugs). For that reason, we recommend that DFINITY make it easier for developers to obtain local test coverage data for individual crates on all supported development platforms. HashEye 29 DFINITY Security Assessment PUBLIC

A. Fix Review Results When undertaking a fix review, HashEye reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system. On February 9, 2023 , HashEye reviewed the fixes and mitigations implemented by the DFINITY team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue. In summary, DFINITY has resolved five of the issues described in this report. The remaining two unresolved issues are only informational and do not constitute a threat to the system. For additional information, please see the Detailed Fix Review Results below. ID Title Severity Status 1 Use of a custom transfer fee causes the creation of SNS neurons to fail Medium Resolved 2 Failure to ensure that all neurons have been created before the transition to Normal mode Informational Resolved 3 Unnecessary calls to unwrap in get_root_status Informational Resolved 4 Erroneous controller check in SnsRootCanister::set_dapp_controllers Low Resolved 5 Accounts with low balances are trimmed from the ICRC-1 ledger Informational Unresolved 6 Potentially harmful remove_self_as_controller pattern Informational Unresolved 7 Use of panicking functions poses a risk to the ledger's archiving mechanism Low Resolved HashEye 30 DFINITY Security Assessment PUBLIC

Detailed Fix Review Results TOB-DFSNS-1: Use of a custom transfer fee causes the creation of SNS neurons to fail Resolved in commit 3a6bb3b740 . The Swap canister takes the SNS transfer fee as

input during initialization, and the SNS transfer fee is used when SNS tokens are transferred from the Swap canister to individual buyers. TOB-DFSNS-2: Failure to ensure that all neurons have been created before the transition to Normal mode Resolved in commit eda98c86b2 . The Swap canister now takes a lock when finalize_swap is called, which ensures that the method can be called only sequentially. This also changes the semantics of the skipped field, since the result of each transfer will always be available before the lock is released. If transfer_success_timestamp_seconds is greater than 0, it must mean that the transfer actually completed successfully in a previous call to finalize_swap . (If not, the timestamp would have been reset to 0 before the lock was released.) Thus, the value of the skipped field will be increased only for completed transfers, and there are no longer any outstanding transactions that may fail after the call to finalize_swap has completed. TOB-DFSNS-3: Unnecessary calls to unwrap in get_root_status Resolved in commit 08807e6c49 . The get_root_status function has been rewritten to return a Result<CanisterStatusResultV2, String> , which is appropriately handled by the caller. TOB-DFSNS-4: Erroneous controller check in SnsRootCanister::set_dapp_controllers Resolved in commit 04b55d0e24 . The set_dapp_controllers method now correctly checks that each decentralized application canister is controlled by the root canister before attempting to update the set of controllers. TOB-DFSNS-5: Accounts with low balances are trimmed from the ICRC-1 ledger Unresolved. The ledger still deletes accounts with low balances when the number of accounts reaches a given threshold. The DFINITY team has explained that this is mainly a safeguard against denial-of-service attacks on the ledger. Since balances could be obtained from the blockchain, the balance of any deleted account could still be recovered. The team currently accepts the risk, but is considering addressing the issue at a later point. TOB-DFSNS-6: Potentially harmful remove_self_as_controller pattern Unresolved. As this is only an informational finding and the proposed fix would potentially make the SNS slower by adding additional update calls, the DFINITY team have chosen not to address the issue. HashEye 31 DFINITY Security Assessment PUBLIC

TOB-DFSNS-7: Use of panicking functions poses a risk to the ledger’s archiving mechanism Resolved in commit 611ac9e2d7 . The create_and_initialize_node_canister function now returns an error if the call to Encode fails. HashEye 32 DFINITY Security Assessment PUBLIC

B. Vulnerability Categories The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document. Vulnerability Categories Category Description Access Controls Insufficient authorization or assessment of rights Auditing and Logging Insufficient auditing of actions or logging of problems Authentication Improper identification of users Configuration Misconfigured servers, devices, or software components Cryptography A breach of system confidentiality or integrity Data Exposure Exposure of sensitive information Data Validation Improper reliance on the structure or values of data Denial of Service A system failure with an availability impact Error Reporting Insecure or insufficient reporting of error conditions Patching Use of an outdated software package or library Session Management Improper identification of authenticated users Testing Insufficient test methodology or test coverage Timing Race conditions or other order-of-operations flaws Undefined Behavior Undefined behavior triggered within the system HashEye 33 DFINITY Security Assessment PUBLIC

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. HashEye 34 DFINITY Security Assessment PUBLIC

C. 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 HashEye 35 DFINITY Security Assessment PUBLIC

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. HashEye 36 DFINITY Security Assessment PUBLIC

D. Automated Testing This section describes the setup of the automated analysis tools used during this audit. 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. We ran Clippy on the crates that implement the SNS swap, ledger, root, and governance canisters. This run did not generate any findings or code quality issues. Semgrep Semgrep can be installed using pip by running python3 -m pip install semgrep . To run Semgrep on a codebase, simply 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. We ran a number of custom Semgrep rules on the crates that implement the SNS swap, ledger, root, and governance canisters. Because support for Rust in Semgrep is still experimental, we focused on identifying the following small set of issues: • The use of panicking functions like assert , unreachable , unwrap , and expect in production code (that is, outside unit tests) rules : - id : panic-in-function-returning-result patterns : - pattern-inside : | fn $FUNC(...) → Result<$T> { ... } - pattern-either : - pattern : $EXPR.unwrap() - pattern : $EXPR.expect(...) message : | `expect` or `unwrap` called in function returning a `Result`. languages : [rust] severity : WARNING Figure C.1: panic-in-function-returning-result.yaml rules : - id : unwrap-outside-test patterns : - pattern : $RESULT.unwrap() HashEye 37 DFINITY Security Assessment PUBLIC

- pattern-not-inside : " #[test] fn $TEST() { ... $RESULT.unwrap() ... } " message : Calling `unwrap` outside unit test languages : [rust] severity : WARNING Figure C.2: unwrap-outside-test.yaml rules : - id : expect-outside-test patterns : - pattern : $RESULT.expect(...) - pattern-not-inside : " #[test] fn $TEST() { ... $RESULT.expect(...) ... } " message : Calling `expect` outside unit test languages : [rust] severity : WARNING Figure C.3: expect-outside-test.yaml • The use of the as keyword in casting, which can silently truncate integers (for example, casting data.len() to a u32 can truncate the input length on 64-bit systems) rules : - id : length-to-smaller-integer pattern-either : - pattern : $VAR.len() as u32 - pattern : $VAR.len() as i32 - pattern : $VAR.len() as u16 - pattern : $VAR.len() as i16 - pattern : $VAR.len() as u8 - pattern : $VAR.len() as i8 message : | Casting `usize` length to smaller integer size silently drops high bits on 64-bit platforms languages : [rust] severity : WARNING HashEye 38 DFINITY Security Assessment PUBLIC

Figure C.4: length-to-smaller-integer.yaml • Unexpected comparisons before subtraction (for example, ensuring that x < y before subtracting y from x), which may indicate errors in the code rules : - id : switched-underflow-guard pattern-either : - patterns : - pattern-inside : | if $Y > $X { ... } - pattern-not-inside : | if $Y > $X { } else { ... } - pattern : $X - $Y - patterns : - pattern-inside : | if $Y ≥ $X { ... } - pattern-not-inside : | if $Y ≥ $X { } else { ... } - pattern : $X - $Y - patterns : - pattern-inside : | if $Y < $X { ... } - pattern-not-inside : | if $Y < $X { } else { ... } - pattern : $Y - $X - patterns : - pattern-inside : | if $Y ⩽ $X { ... } - pattern-not-inside : | if $Y ⩽ $X { HashEye 39 DFINITY Security Assessment PUBLIC

} else { ... } - pattern : $X - $Y - patterns : - pattern-inside : | if $Y > $X { } else { ... } - pattern : $Y - $X - patterns : - pattern-inside : | if $Y ≥ $X { } else { ... } - pattern : $Y - $X - patterns : - pattern-inside : | if $Y < $X { } else { ... } - pattern : $X - $Y - patterns : - pattern-inside : | if $Y ⩽ $X { } else { ... } - pattern : $X - $Y - patterns : - pattern : | if $X < $Y { } ... message : Potentially switched comparison in if-statement condition languages : [rust] severity : WARNING Figure C.5: switched-underflow-guard.yaml This run led us to identify the issues detailed in TOB-DFSNS-3 and TOB-DFSNS-7 , as well as a number of the code quality findings provided in appendix D . HashEye 40 DFINITY Security Assessment PUBLIC

cargo-llvm-cov The cargo-llvm-cov Cargo plugin is used to generate LLVM source - based code coverage data. The plugin can be installed via the command cargo install cargo-llvm-cov . To run the plugin, simply run the command cargo llvm-cov in the crate root directory. Because we

prioritized achieving good manual coverage of the SNS canisters, we did not have time to obtain reliable data on the system’s test coverage. HashEye 41 DFINITY Security Assessment PUBLIC

E. Code Quality Findings Swap Canister • The Swap::sns_token_e8s method (in sns/swap/src/swap.rs) uses 0 to represent None . This method should either return an Option or panic to indicate an implementation error. pub fn sns_token_e8s (& self) → u64 { if let Some (params) = & self .params { params.sns_token_e8s } else { 0 } } Figure D.1: The sns_token_e8s function uses 0 to signal an error. • The Swap::cf_total_icp_e8s method (in sns/swap/src/swap.rs) should use saturating or checked addition to calculate token amounts. Alternatively, it could use the Tokens type (defined in ic_ledger_core) to represent tokens. pub fn cf_total_icp_e8s (& self) → u64 { // TODO: use saturating add... self .cf_participants .iter() .map(|x| x.participant_total_icp_e8s()) .sum() } Figure D.2: The Swap::cf_total_icp_e8s method should use saturating or checked addition. • The Swap::can_commit method (in sns/swap/src/swap.rs) can be simplified. The code in figure D.3 can be changed to the simpler code in figure D.4. if !(self .swap_due(now_seconds) || self .icp_target_reached()) { return false ; } true Figure D.3: The Swap::can_commit method can be simplified. self .swap_due(now_seconds) || self .icp_target_reached() Figure D.4: This code is equivalent to the if statement in figure D.3. HashEye 42 DFINITY Security Assessment PUBLIC

• The Params::validate function (in sns/swap/src/swap.rs) returns an error if validation fails. However, if self.max_icp_e8s * self.min_participants exceeds 64 bits, the function will panic; it should probably return an error in those cases as well. • The Params::validate function should verify that self.min_participant_icp_e8s ⩾ DEFAULT_TRANSFER_FEE . This will ensure that buyers cannot obtain tokens without paying for them in TransferableAmount::transfer_helper . Governance Canister • The governance canister’s use of &PrincipalId and PrincipalId is inconsistent, as PrincipalId implements Copy . In particular, the set_mode , is_swap_canister , and validate_canister_id_field functions have owned PrincipalID arguments, while other functions that take PrincipalId take references instead. • In a similar vein, the claim_neuron function accepts an owned NeuronId argument, but all other uses of NeuronID are references. • There should be code comments explaining why the canister’s expect and unwrap operations are infallible. SNS-WASM Canister • There should be code comments explaining why the canister’s expect and unwrap operations are infallible. Root Canister • The SnsRootCanister::register_dapp_canister operation should fail if the registered canister passed to the function is a distinguished SNS canister. However, the function fails to check whether the registered canister’s ID is equal to the swap canister ID. Ledger Canister • The Balances::debi t and Balances::credit methods (defined in rosetta-api/ledger_core/src/balances.rs) are used to define internal functionality and should not be public. HashEye 43 DFINITY Security Assessment PUBLIC