HashEye

// SMART CONTRACT SECURITY AUDIT

Dfinity SNS Phase 2

Security assessment by HashEye · prepared for DFINITY

HASHEYE AUDITED

PROJECT	Dfinity SNS Phase 2
PERFORMED BY	HashEye Security Review Team - Sergey, Brian
RESEARCH	Sergey, Brian
CLIENT	DFINITY
CATEGORY	Blockchain
PUBLISHED	June 1, 2023
REPORT ID	research-dfinity-sns-phase-2-2023-06-01-5i1jaf

This report was produced by HashEye's security review team under the HashEye review process, combining source review, deployment-state checks, and senior manual verification. Full report detail is available at hasheye.io/audits/research-dfinity-sns-phase-2-2023-06-01-5i1jaf.

DFINITY ckBTC and BTC Integration Security Assessment October 6, 2023 Prepared for: Robin Künzler DFINITY Prepared by: Artur Cygan and Fredrik Dahlgren

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

Notices and Remarks Copyright and Distribution © 2023 by HashEye, Inc. All rights reserved. HashEye hereby asserts its right to be identified as the creator of this report in the United Kingdom. This report is considered by HashEye to be public information; it is licensed to DFINITY under the terms of the project statement of work and has been made public at DFINITY's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of HashEye. The sole canonical source for HashEye publications is the HashEye Publications page . Reports accessed through any source other than that page may have been modified and should not be considered authentic. 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 About HashEye 1 Notices and Remarks 2 Table of Contents 3 Executive Summary 4 Project Summary 6 Project Goals 7 Project Targets 8 Project Coverage 9 Codebase Maturity Evaluation 10 Summary of Findings 12 Detailed Findings 13 1. KYT canister is centralized on third-party provider 13 2. Risk of amount underflow when retrieving BTC 14 3. Minter's init and upgrade configs insufficiently validated 16 4. Inconsistent error logging in minter 18 5. KYT API keys are exposed 20 A. Vulnerability Categories 21 B. Code Maturity Categories 23 C. Code Quality Recommendations 25 HashEye 3 DFINITY Security Assessment PUBLIC

Executive Summary Engagement Overview DFINITY engaged HashEye to review the security of its ckBTC and BTC integration. One consultant conducted the review from June 5, 2023 to June 23, 2023, for a total of two and a half engineer-weeks of eff ort. Our testing eff orts focused on a review of the ckBTC and BTC integration. With full access to the source code and documentation, we performed static and dynamic testing of the canisters that the system consists of, using automated and manual processes. We mainly focused on the ckBTC minter canister, since it is the most critical piece of the review scope. Observations and Impact The ckBTC and BTC integration is a complex system largely due to the nature of the Bitcoin network it integrates with. The UTXO model and lack of finality gadget make the integration particularly challenging. The implemented system appears to be well thought out and tested, with the exception of the KYT canister, which has the most significant issues (TOB-DFBTC-1 , TOB-DFBTC-5). Recommendations Based on the codebase maturity evaluation and findings identified during the security review, HashEye 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. • Rethink the design of the KYT canister. The current centralized design of the KYT canister appears to be experimental and stands out from the otherwise highly decentralized system. The risk is largely associated with denial of service of transfers between ckBTC and BTC. We also noted comments from members of the Internet Computer community raising similar points. • Improve the documentation availability. We found the internal “Chain Key Bitcoin (ckBTC) - Design Specification” document to be well written and helpful during the code review. Making the document public and referencing it from the code would greatly improve the system’s transparency. The following tables provide the number of findings by severity and category. HashEye 4 DFINITY Security Assessment PUBLIC

EXPOSURE ANALYSIS Severity Count Low 2 Informational 3 CATEGORY BREAKDOWN Category Count Data Exposure 1 Data Validation 2 Denial of Service 1 Error Reporting 1 HashEye 5 DFINITY Security Assessment PUBLIC

Project Summary Contact Information The following managers were associated with this project: Dan Guido , Account Manager Anne Marie Barry , Project Manager dan@hasheye.io annemarie.barry@hasheye.io The following engineers were associated with this project: Artur Cygan , Consultant Fredrik Dahlgren , Consultant artur.cygan@hasheye.io fredrik.dahlgren@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event June 2, 2023 Pre-project kickoff call June 12, 2023 Status update meeting #1 June 20, 2023 Status update meeting #2 June 22, 2022 Delivery of report draft and report readout meeting October 6, 2023 Delivery of comprehensive report HashEye 6 DFINITY Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the DFINITY ckBTC and BTC integration. Specifically, we sought to answer the following non-exhaustive list of questions: • Is it possible to double-spend funds? • Can malicious actors launch denial-of-service attacks against the service? • Are account access controls implemented correctly? • Is it possible to trick the minter into signing invalid transactions? • Is there any risk of arithmetic overflows, miscalculations, or rounding errors? • Is it possible to mint or burn ckBTC without spending BTC? HashEye 7 DFINITY Security Assessment PUBLIC

Project Targets The engagement involved a review and testing of the targets listed below. ckBTC Repository https://github.com/dfinity/ic Version 2867da6c18178ac79bc513a9c7cad59a09030655 Type Rust Platform Internet Computer bitcoin-canister Repository https://github.com/dfinity/bitcoin-canister Version 0cd235d98bde196fe7710fe319a84678b37ad093 Type Rust Platform Internet Computer HashEye 8 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 included the following: • A manual review of the code and documentation for the ckBTC canisters ○ We mainly focused on the minter canister, since it is the most critical component that processes Bitcoin transactions. • A best-eff ort review of the newest changes to the BTC integration canister 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: • Although we highlight the presence of the TLA+ specification as a positive sign of the project’s maturity, we did not review the specification itself. • We performed a best-eff ort code review of the latest additions to the BTC integration canisters; however, this work should not be treated as exhaustive. We did not have time to review the watchdog canister. HashEye 9 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. Defi ciencies 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 code performs many calculations on integers. We did not find any major issues with them; however, we observed some critical operations that could be coded more defensively against integer overflows/underflows (TOB-DFBTC-2). Satisfactory Auditing All significant operations and errors are logged. We observed some inconsistencies in the error messages that could impact system auditing and monitoring (TOB-DFBTC-4). Satisfactory Authentication / Access Controls The access controls rely on the ic_cdk -provided caller identity. The system computes subaccounts for deposits and withdrawals based on the identity. Its strength relies on strong hashing functions. We did not find any issues with this mechanism. Strong Complexity Management The system has substantial essential complexity stemming from the Bitcoin architecture. This complexity is generally well managed by the implementation. We found a few instances of accidental complexity, detailed in

appendix C . Overall, we found the code to be organized in reasonably sized modules and functions. Satisfactory Cryptography and Key Management ckBTC uses sophisticated cryptographic protocols, such as distributed key generation and a threshold signature scheme to sign Bitcoin transactions. However, we identified a flaw in the design of the KYT canister that stores API keys, which are subject to abuse Satisfactory HashEye 10 DFINITY Security Assessment PUBLIC

by malicious node operators (TOB-DFBTC-5). Data Handling Input data is consistently validated and represented using types that refl ect its semantics. We found two minor data validation issues, although they currently do not appear to introduce any risks (TOB-DFBTC-2 , TOB-DFBTC-3). Satisfactory Documentation The system is sufficiently documented. However, some of the documentation is private. The implementation is documented with code comments, but we noted some places where it could be improved and updated to reference the written documentation. Satisfactory Memory Safety and Error Handling Internet Computer containers compile to Wasm and run inside a virtual machine, thus greatly reducing the risk of memory safety issues. The Rust language provides primitives that facilitate error handling; we found that these primitives are extensively and correctly used throughout the codebase. However, we identified some parts of the code that could be simplified to handle fewer error cases (appendix C.1). Satisfactory Testing and Verification The system was implemented based on a detailed design document. The ckBTC and BTC integration code has extensive test suites that exercise many inputs. This is further strengthened by the use of property-based testing. The invariants of the system are stated in the code as comments and further enforced with dedicated functions. Although we did not review the TLA+ specification of ckBTC, we consider it a significant bonus that enhances our overall confidence in the system. Strong HashEye 11 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 KYT canister is centralized on third-party provider Denial of Service Low 2 Risk of amount underflow when retrieving BTC Data Validation Informational 3 Minter’s init and upgrade configs insufficiently validated Data Validation Informational 4 Inconsistent error logging in minter Error Reporting Informational 5 KYT API keys are exposed Data Exposure Low HashEye 12 DFINITY Security Assessment PUBLIC

Detailed Findings 1. KYT canister is centralized on third-party provider Severity: Low Difficulty: High Type: Denial of Service Finding ID: TOB-DFBTC-1 Target: bitcoin/ckbtc/kyt Description The KYT canister relies entirely on the third-party provider Chainalysis to clear Bitcoin transactions and is tightly integrated with it, creating a single point of failure in an otherwise highly decentralized system. It is possible to upgrade the KYT canister to a mode that clears all transactions, but this likely requires manual intervention. Exploit Scenario Chainalysis is compromised and marks all UTXOs as tainted, eff ectively denying the ckBTC to/from BTC transfer service for all users. Recommendations Short term, document this limitation and run monitoring to detect whether there is a risk that Chainalysis will become unreliable. Long term, add additional KYT providers and cross-check the results. This will ensure that a single provider cannot launch a denial-of-service attack against the KYT canister, and that the team is alerted if any anomalies arise. HashEye 13 DFINITY Security Assessment PUBLIC

2. Risk of amount underflow when retrieving BTC Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-DFBTC-2 Target: bitcoin/ckbtc/minter Description The retrieve_btc call deducts the kyt_fee from the requested withdrawal amount. The arithmetic operation (figure 6.1) is not checked for underflow, and there is no prior explicit check that would guarantee that kyt_fee is greater than or equal to args.amount . The args.amount value is guaranteed to be at least retrieve_btc_min_amount (figure 6.2). This implies that an underflow will not occur if retrieve_btc_min_amount is greater than or equal to kyc_fee . This is a sane assumption; however, the condition is not ensured by the minter code and relies entirely on the correct init /upgrade value configuration, which is subject to human error. let request = RetrieveBtcRequest { // NB. We charge the KYT fee from the retrieve amount. amount: args .amount - kyt_fee , address: parsed_address , block_index, received_at: ic_cdk ::api::time(), kyt_provider: Some (kyt_provider), }; Figure 2.1: (bitcoin/ckbtc/minter/src/updates/retrieve_btc.rs:178-185) let (min_amount, btc_network) = read_state(|s| (s.retrieve_btc_min_amount, s.btc_network)); if args.amount < min_amount { return Err (RetrieveBtcError::AmountTooLow(min_amount)); } Figure 2.2: (bitcoin/ckbtc/minter/src/updates/retrieve_btc.rs:122-125) Exploit Scenario The canister is upgraded with a new kyt_fee value that is larger than the current retrieve_btc_min_amount value and enables the amount to underflow. Recommendations Short term, use a checked_sub to compute the amount and provide a comment explaining why the inequality holds. HashEye 14 DFINITY Security Assessment PUBLIC

Long term, perform checked arithmetic for all critical operations, and add comments to the code explaining why the corresponding invariant holds. HashEye 15 DFINITY Security Assessment PUBLIC

3. Minter’s init and upgrade configs insuciently validated Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-DFBTC-3 Target: bitcoin/ckbtc/minter Description The init and upgrade arguments are described by the InitArgs and UpdateArgs structures, which are used to initialize and modify the minter’s state. There is very little validation of these values, which could lead to a pathological state as described in the previous finding. For instance, the ecdsa_key_name can be configured to be empty, kyt_fee can be larger than retrieve_btc_min_amount , and retrieve_btc_min_amount and max_time_in_queue_nanos are unbounded in the u64 range (figure 3.1). impl CkBtcMinterState { pub fn reinit (& mut self , InitArgs { btc_network, ecdsa_key_name, retrieve_btc_min_amount, ledger_id, max_time_in_queue_nanos, min_confirmations, mode, kyt_fee, kyt_principal, }: InitArgs ,) { self .btc_network = btc_network; self .ecdsa_key_name = ecdsa_key_name; self .retrieve_btc_min_amount = retrieve_btc_min_amount; self .ledger_id = ledger_id; self .max_time_in_queue_nanos = max_time_in_queue_nanos; self .mode = mode; self .kyt_principal = kyt_principal; if let Some (kyt_fee) = kyt_fee { self .kyt_fee = kyt_fee; } if let Some (min_confirmations) = min_confirmations { self .min_confirmations = min_confirmations; } HashEye 16 DFINITY Security Assessment PUBLIC

} ... } Figure 3.1: (bitcoin/ckbtc/minter/src/state.rs#L322-L350) Exploit Scenario The canister is upgraded with a value that breaks an assumption in the code, which leads to state corruption. Recommendations Short term, include additional validation of configuration values where applicable and ensure that the implementation is covered by unit tests. Long term, always validate all data provided by the system's users. HashEye 17 DFINITY Security Assessment PUBLIC

4. Inconsistent error logging in minter Severity: Informational Difficulty: High Type: Error Reporting Finding ID: TOB-DFBTC-4 Target: bitcoin/ckbtc/minter Description We found log messages produced by the minter to be inconsistent (figures 4.1 versus 4.2) and sometimes ambiguous (figure 4.2). Making log messages more consistent would simplify auditing and monitoring of the system. let block_index = client .transfer(TransferArg { from_subaccount: None , to, fee: None , created_at_time: None , memo: Some (Memo::from(txid.to_vec())), amount: Nat ::from(amount), }) . await .map_err(|e| UpdateBalanceError::TemporarilyUnavailable(e. 1))??; Figure 4.1: The error code (e.0) is ignored. (bitcoin/ckbtc/minter/src/updates/update_balance.rs#L302-L312) let result = client .transfer(TransferArg { from_subaccount: Some (from_subaccount), to: Account { owner: minter , subaccount: None , }, fee: None , created_at_time: None , memo: None , amount: Nat ::from(amount), }) . await .map_err(|(code , msg)| { RetrieveBtcError::TemporarilyUnavailable(format! ("cannot enqueue a burn transaction: {} (reject_code = {})" , msg, code)) })?; HashEye 18 DFINITY Security Assessment PUBLIC

Figure 4.1: The error code is included in the log message. (bitcoin/ckbtc/minter/src/updates/retrieve_btc.rs#L236-L254) log!(P1, "Minted {} {token_name} for account {caller_account} with value {}" , DisplayOutpoint(&utxo.outpoint), DisplayAmount(utxo.value),); Figure 4.2: Without checking the code, it is unclear whether the value refers to the actual minted amount or the utxo.value . (bitcoin/ckbtc/minter/src/updates/update_balance.rs#L199-L204) Recommendations Short term, review the log messages produced by the minter and change them to follow a consistent pattern. Long term, add instructions to the development guidelines on how to structure log messages. HashEye 19 DFINITY Security Assessment PUBLIC

5. KYT API keys are exposed Severity: Low Difficulty: Medium Type: Data Exposure Finding ID: TOB-DFBTC-5 Target: bitcoin/ckbtc/kyt Description The KYT API keys reside in canister memory, which is replicated across the network. The Internet Computer assumes that part of the network can be malicious (Byzantine fault tolerance) and the network should continue to function without disruption even if it is. This is not the case for the KYT canister, as a rogue node could abuse the API keys to drain the funds on the associated Chainalysis accounts and cause a denial of service. This further weakens the decentralization and reliability of the KYT service, as the risk of having the funds drained without remuneration will discourage maintainers from providing access to Chainalysis accounts. Exploit Scenario A malicious node operator reads the Chainalysis API keys from the canister memory and uses them anonymously to drain the funds intended to ensure the correct functioning of the KYT canister. Recommendations Short term, document this limitation and provide a risk assessment. Long term, find a way to hold node operators accountable for misuse of the KYT API keys. HashEye 20 DFINITY Security Assessment PUBLIC

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 Undefi ned behavior triggered within the system HashEye 21 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. Diffi culty 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 22 DFINITY Security Assessment PUBLIC

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 defi ned 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. HashEye 23 DFINITY Security Assessment PUBLIC

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 24 DFINITY Security Assessment PUBLIC

C. Code Quality Recommendations The following section contains code quality recommendations that do not have any immediate security implications. 1. Change the init_ecdsa_public_key function to return the key. At the moment, the key has to be fetched again from the state, which requires an additional unsafe expect call that leads to unnecessarily complex code. This pattern is used in multiple places in the minter’s code. init_ecdsa_public_key(). await ; let main_account = Account { owner: ic_cdk ::id(), subaccount: None , }; let main_address = match state::read_state(|s| { s.ecdsa_public_key .clone() .map(|key| account_to_bitcoin_address(&key, &main_account)) }) { Some (address) ⇒ address, None ⇒ { ic_cdk::trap("unreachable: have retrieve BTC requests but the ECDSA key is not initialized" ,); } }; Figure C.1: An example of repeated key presence checking (bitcoin/ckbtc/minter/src/lifecycle/init.rs) 2. Update the documentation comment for mode in minter’s InitArgs . The current comment appears to be left over from the old is_read_only Boolean flag. pub struct InitArgs { ... /// Flag that indicates if the minter is in read-only mode. # [serde(default)] pub mode: Mode , ... } Figure C.2: Outdated comment (bitcoin/ckbtc/minter/src/lifecycle/init.rs) 3. Update the broken documentation link in this comment . HashEye 25 DFINITY Security Assessment PUBLIC

4. Remove the extra space between sentences in this code comment and this one . 5. The sighash and encode_sighash_data functions should take input instead of index . This change will eliminate the need to perform bounds checks, which will simplify the code. 6. The redundant check in submit_pending_requests can be removed. The first check is already performed during the second check with the can_form_a_batch function. if state::read_state(|s| s.pending_retrieve_btc_requests.is_empty()) { return ; } // We make requests if we have old requests in the queue or if have enough // requests to fill a batch. if !state::read_state(|s| s.can_form_a_batch(MIN_PENDING_REQUESTS, ic_cdk::api::time())) { return ; } Figure C.3: The first

check is redundant. (bitcoin/ckbtc/minter/src/lib.rs#L212-L220) 8. The key_derivation function is deprecated and should be replaced with public_key_derivation . 9. Document all the data types and functions. For instance, the variants below are undocumented, but their semantics are not immediately obvious. pub enum UtxoStatus { ValueTooSmall(Utxo), Tainted(Utxo), Checked(Utxo), Minted { block_index: u64 , minted_amount: u64 , utxo: Utxo , }, } Figure C.4: Undocumented data type (bitcoin/ckbtc/minter/src/updates/update_balance.rs#L31C1-L40) 10. Avoid using unwrap in the production code. For instance, it is not clear why the use of unwrap below is safe; it should be replaced with an expect that explains why. let hrp = hrp(network); bech32::encode(hrp, data, bech32::Variant::Bech32) .unwrap() Figure C.5: Unexplained error type unwrap (bitcoin/ckbtc/minter/src/address.rs#L136-L137)

11. The type representing txid should be more precise. There is a type mismatch between the minter code and ic_btc_interace that forces error handling that should be impossible (figure C.6). let txid = TryInto::<[u8 ; 32]>::try_into(utxo.outpoint.txid.as_ref()) .unwrap_or_else(|_| panic! ("BUG: UTXO ID {:?} is not 32 bytes long" , utxo.outpoint.txid)); Figure C.6: Type mismatch for the data with the same semantics (bitcoin/ckbtc/minter/src/management.rs#L274-L275)