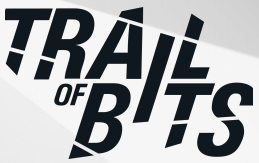

DFINITY Canister Sandbox

Security Assessment

July 7, 2022

Prepared for:
Robin Künzler
DFINITY

Prepared by: **Fredrik Dahlgren and Will Song**

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 80+ 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

Executive Summary

Engagement Overview

DFINITY engaged Trail of Bits to review the security of its canister sandbox. From April 18 to April 29, 2022, one consultant conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

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 target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered flaws that could impact system confidentiality, integrity, or availability. However, we need to assume that a malicious canister must obtain arbitrary code execution within a sandboxed process in order to exploit any of the issues identified during this review. This assumption should be taken into account when assessing the overall severity and difficulty of the issues we identified.

A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	2
Low	2
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2
Data Exposure	1
Data Validation	2
Patching	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-DFCS-4**
  The implementation relies on mandatory access controls through SELinux to restrict access to the rest of the system. However, the sandbox does not use Seccomp to restrict the system calls available to a sandboxed process. This increases the risk that a malicious canister could escalate its privileges and escape the sandbox using a vulnerable system call.

- **TOB-DFCS-5**
  When system-wide canister state changes are applied by the hypervisor, the implementation uses panicking functions like `assert` and `expect`. This allows a malicious canister that has already obtained arbitrary code execution within the sandboxed process to return a set of invalid state changes to the replica, causing the hypervisor to panic.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineers were associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Project Timeline

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

Date	Event
April 8, 2022	Pre-project kickoff call
April 25, 2022	Status update meeting #1
May 2, 2022	Delivery of report draft
May 2, 2022	Report readout meeting
July 7, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the DFINITY canister sandbox. Specifically, we sought to answer the following non-exhaustive list of questions:

- How is the sandbox process implemented?
- Could a canister that has gained arbitrary code execution in the context of a sandboxed process elevate its privileges or access other resources in the system?
- Could a user access sensitive application data owned by the replica from the sandboxed process?
- Are the replica's interprocess communication (IPC) endpoints resistant to malicious messages from a compromised sandboxed process?

Project Targets

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

Canister Sandbox

Repository	`dfinity/ic/rs/canister_sandbox`
Version	`f2568bece27d7cd40a2e774e4a39f3b84ee4e000`
Type	Rust
Platform	Linux

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:

- **Canister sandbox.** We began our review by reading through the existing documentation on the design and implementation of the canister sandbox. We then ran a number of static analysis tools on the codebase to identify low-hanging fruit and to ensure that the code follows Rust best practices. Finally, we performed an in-depth review of all the components of the sandbox, focusing particularly on identifying attack vectors against the replica process. We also implemented a fuzzing harness to fuzz the unsafe components of the IPC transport implementation.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this engagement, issues with running Tarpaulin and `llvm-cov` prevented us from obtaining reliable metrics on the unit test coverage of the codebase.

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
`cargo-afl`	An open-source fuzzing framework based on AFL++	Appendix C
`cargo-audit`	An open-source static analysis tool used to audit `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database	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 when editing or committing code and during build time	Appendix C
Tarpaulin	An open-source code coverage reporting tool that can be integrated with the Cargo build system on x86 Linux architectures	Appendix C

Areas of Focus

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

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of `unwrap` and `expect`
- Known vulnerable dependencies
- Poor unit and integration test coverage

- Issues related to the correctness of the unsafe parts of the IPC transport implementation

Test Results

The results of this focused testing are detailed below.

Canister sandbox: A SELinux-based sandbox for the WebAssembly (Wasm) canister execution environment

Property	Tool	Result
Fuzzing does not reveal any memory unsafety issues in the IPC transport implementation.	`cargo-afl`	**Passed**
The project does not import vulnerable dependencies.	`cargo-audit`	TOB-DFCS-1
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	**Passed**
All components of the codebase have sufficient test coverage.	Tarpaulin	Further Investigation Required

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 implementation does not rely heavily on arithmetic; for the arithmetic that it does rely on, it includes thorough checks for overflows and underflows.	**Strong**
Auditing	The crate logs errors to `stderr` (which is then handled by `journald`). However, in certain cases, the system does not log or act upon malicious behavior by a compromised sandbox (TOB-DFCS-6). Other than this finding, we found no other issues related to auditing in the implementation.	**Satisfactory**
Complexity Management	The canister sandbox codebase is well structured and easy to navigate. Additionally, the implementation uses Rust traits to abstract away much of the underlying complexity and to share code between the sandbox and controller implementations. This makes the code more readable and also avoids unnecessary code duplication.	**Strong**
Configuration	The canister sandbox's SELinux policy restricts almost all interactions with the rest of the operating system, apart from a small set of operations that are required to launch the process, communicate with the replica, and compile and execute Wasm code obtained from the replica. We identified only one issue related to the SELinux policy: a canister that has broken out of the Wasm execution environment could spoof log messages on replica file descriptors (TOB-DFCS-3). However, interactions with the kernel are not restricted in the same way, and we recommend that the DFINITY	**Moderate**

	team strengthen the sandbox by implementing system call filtering using Seccomp-BPF (TOB-DFCS-4).	
Data Handling	The canister sandbox defines a very narrow remote procedure call (RPC) interface that allows the canister to communicate with the replica. This greatly reduces the attack surface exposed to the sandboxed process and makes it significantly easier to protect the replica process from a rogue canister. However, we did identify two issues related to the validation of data passed from the sandboxed process to the replica. These issues could allow a canister that had already gained arbitrary code execution within the context of the sandboxed process to either crash the replica process (TOB-DFCS-5) or cause the replica to miscalculate the number of cycles available to the canister (TOB-DFCS-6). (We note that these issues do not represent a serious risk, as the canister would have to gain code execution in the sandboxed process.)	**Satisfactory**
Documentation	The sandbox implementation is very well documented. Its detailed design document also outlines a threat model for the system, including various attack scenarios and corresponding mitigations.	**Strong**
Maintenance	We identified a number of indirect dependencies of the canister sandbox crate that have known vulnerabilities (TOB-DFCS-1). However, the DFINITY team regularly runs `cargo-audit` and was already aware of this issue.	**Satisfactory**
Memory Safety and Error Handling	The implementation contains a small number of unsafe blocks to convert raw file descriptors to Unix streams, to assist in process management using the `libc` crate, and to implement the IPC transport mechanism on top of `libc::recvmsg`. All unsafe blocks are short and easy to review. We did not find any issues related to unsafe code during our manual review. We also fuzzed the IPC transport implementation, which did not result in any issues. Additionally, the implementation contains a signal handler implemented in C. However, the handler is	**Strong**

	defensively written, and we did not identify any issues in the implementation.	
Testing and Verification	A large number of unit tests for both low-level functionality, like memory serialization and file descriptor transfers, and higher-level functionality, like memory and execution state management, are implemented for the crate. However, because of time constraints, we failed to obtain sufficient data on unit test coverage for the crate.	**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	The canister sandbox has vulnerable dependencies	Patching	Low
2	Complete environment of the replica is passed to the sandboxed process	Data Exposure	Informational
3	SELinux policy allows the sandbox process to write replica log messages	Configuration	Low
4	Canister sandbox system calls are not filtered using Seccomp	Configuration	Medium
5	Invalid system state changes cause the replica to panic	Data Validation	Medium
6	SandboxedExecutionController does not enforce memory size invariants	Data Validation	Informational

Detailed Findings

1. The canister sandbox has vulnerable dependencies

Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-DFCS-1
Target: Canister sandbox	

Description
The canister sandbox codebase uses the following vulnerable or unmaintained Rust dependencies. (All of the crates listed are indirect dependencies of the codebase.)

Dependency	Version	ID	Description
`chrono`	0.4.19	RUSTSEC-2020-0159	Potential segfault in `localtime_r` invocations
`regex`	1.5.4	RUSTSEC-2022-0013	Regexes with large repetitions on empty sub-expressions take a very long time to parse
`thread_local`	1.0.1	RUSTSEC-2022-0006	Data race in `Iter` and `IterMut`
`serde_cbor`	0.11.2	RUSTSEC-2021-0127	`serde_cbor` is unmaintained

Other than `chrono`, all of the vulnerable dependencies can simply be updated to their newest versions to fix the vulnerabilities. The `chrono` crate issue has not been mitigated and remains problematic. A specific sequence of calls must occur to trigger the vulnerability, which is discussed in this GitHub thread in the `chrono` repository.

Exploit Scenario
An attacker exploits a known vulnerability in one of the canister sandbox dependencies and gains arbitrary code execution within the sandbox.

Recommendations
Short term, update all dependencies to their newest versions. Monitor the referenced GitHub thread regarding the `chrono` crate segfault issue.

Long term, run `cargo-audit` as part of the CI/CD pipeline and ensure that the team is alerted to any vulnerable dependencies that are detected.

2. Complete environment of the replica is passed to the sandboxed process

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-DFCS-2
Target: `canister_sandbox/common/src/process.rs`	

Description

When the `spawn_socketed_process` function spawns a new sandboxed process, the call to the `Command::spawn` method passes the entire environment of the replica to the sandboxed process.

```
pub fn spawn_socketed_process(
    exec_path: &str,
    argv: &[String],
    socket: RawFd,
) -> std::io::Result<Child> {
    let mut cmd = Command::new(exec_path);
    cmd.args(argv);

    // In case of Command we inherit the current process's environment. This should
    // particularly include things such as Rust backtrace flags. It might be
    // advisable to filter/configure that (in case there might be information in
    // env that the sandbox process should not be privy to).

    // The following block duplicates sock_sandbox fd under fd 3, errors are
    // handled.
    unsafe {
        cmd.pre_exec(move || {
            let fd = libc::dup2(socket, 3);

            if fd != 3 {
                return Err(std::io::Error::last_os_error());
            }
            Ok(())
        })
    };

    let child_handle = cmd.spawn()?;

    Ok(child_handle)
}
```

Figure 2.1: `canister_sandbox/common/src/process.rs:17-46`

The DFINITY team does not use environment variables for sensitive information. However, sharing the environment with the sandbox introduces a latent risk that system configuration data or other sensitive data could be leaked to the sandboxed process in the future.

Exploit Scenario

A malicious canister gains arbitrary code execution within a sandboxed process. Since the environment of the replica was leaked to the sandbox when the process was created, the canister gains information about the system that it is running on and learns sensitive information passed as environment variables to the replica, making further efforts to compromise the system easier.

Recommendations

Short term, add code that filters the environment passed to the sandboxed process (e.g., `Command::env_clear` or `Command::env_remove`) to ensure that no sensitive information is leaked if the sandbox is compromised.

3. SELinux policy allows the sandbox process to write replica log messages	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-DFCS-3
Target: `guestos/rootfs/prep/ic-node/ic-node.te`	

Description

When a new sandboxed process is spawned using `Command::spawn`, the process's `stdin`, `stdout`, and `stderr` file descriptors are inherited from the parent process. The SELinux policy for the canister sandbox currently allows sandboxed processes to read from and write to all file descriptors inherited from the replica (the file descriptors created by `init` when the replica is started, as well as the file descriptor used for interprocess RPC). As a result, a compromised sandbox could spoof log messages to the replica's `stdout` or `stderr`.

```
# Allow to use the logging file descriptor inherited from init.
# This should actually not be allowed, logs should be routed through
# replica.
allow ic_canister_sandbox_t init_t : fd { use };
allow ic_canister_sandbox_t init_t : unix_stream_socket { read write };
```

Figure 3.1: `guestos/rootfs/prep/ic-node/ic-node.te:312-316`

Additionally, sandboxed processes' read and write access to files with the `tmpfs_t` context appears to be overly broad, but considering the fact that sandboxed processes are not allowed to open files, we did not see any way to exploit this.

Exploit Scenario

A malicious canister gains arbitrary code execution within a sandboxed process. By writing fake log messages to the replica's `stderr` file descriptor, the canister makes it look like the replica has other issues, masking the compromise and making incident response more difficult.

Recommendations

Short term, change the SELinux policy to disallow sandboxed processes from reading from and writing to the inherited file descriptors `stdin`, `stdout`, and `stderr`.

4. Canister sandbox system calls are not filtered using Seccomp	
Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-DFCS-4
Target: Canister sandbox	

Description
Seccomp provides a framework to filter outgoing system calls. Using Seccomp, a process can limit the type of system calls available to it, thereby limiting the available attack surface of the kernel.

The current implementation of the canister sandbox does not use Seccomp; instead, it relies on mandatory access controls (via SELinux) to restrict the system calls available to a sandboxed process. While SELinux is useful for restricting access to files, directories, and other processes, Seccomp provides more fine-grained control over kernel system calls and their arguments. For this reason, Seccomp (in particular, Seccomp-BPF) is a useful complement to SELinux in restricting a sandboxed process's access to the system.

Exploit Scenario
A malicious canister gains arbitrary code execution within a sandboxed process. By exploiting a vulnerability in the kernel, it is able to break out of the sandbox and execute arbitrary code on the node.

Recommendations
Long term, consider using Seccomp-BPF to restrict the system calls available to a sandboxed process. Extra care must be taken when the canister sandbox (or any of its dependencies) is updated to ensure that the set of system calls invoked during normal execution has not changed.

5. Invalid system state changes cause the replica to panic

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DFCS-5
Target: `system_api/src/sandbox_safe_system_state.rs`	

Description

When a sandboxed process has completed an execution request, the hypervisor calls `SystemStateChanges::apply_changes` (in `Hypervisor::execute`) to apply the system state changes to the global canister system state.

```
pub fn apply_changes(self, system_state: &mut SystemState) {
    // Verify total cycle change is not positive and update cycles balance.
    assert!(self.cycle_change_is_valid(
        system_state.canister_id == CYCLES_MINTING_CANISTER_ID
    ));
    self.cycles_balance_change
        .apply_ref(system_state.balance_mut());

    // Observe consumed cycles.
    system_state
        .canister_metrics
        .consumed_cycles_since_replica_started +=
        NominalCycles::from_cycles(self.cycles_consumed);

    // Verify we don't accept more cycles than are available from each call
    // context and update each call context balance
    if !self.call_context_balance_taken.is_empty() {
        let call_context_manager = system_state.call_context_manager_mut().unwrap();
        for (context_id, amount_taken) in &self.call_context_balance_taken {
            let call_context = call_context_manager
                .call_context_mut(*context_id)
                .expect("Canister accepted cycles from invalid call context");
            call_context
                .withdraw_cycles(*amount_taken)
                .expect("Canister accepted more cycles than available ...");
        }
    }

    // Push outgoing messages.
    for msg in self.requests {
        system_state
            .push_output_request(msg)
            .expect("Unable to send new request");
    }
```

```
    // Verify new certified data isn't too long and set it.
    if let Some(certified_data) = self.new_certified_data.as_ref() {
        assert!(certified_data.len() <= CERTIFIED_DATA_MAX_LENGTH as usize);
        system_state.certified_data = certified_data.clone();
    }

    // Verify callback ids and register new callbacks.
    for update in self.callback_updates {
        match update {
            CallbackUpdate::Register(expected_id, callback) => {
                let id = system_state
                    .call_context_manager_mut()
                    .unwrap()
                    .register_callback(callback);
                assert_eq!(id, expected_id);
            }
            CallbackUpdate::Unregister(callback_id) => {
                let _callback = system_state
                    .call_context_manager_mut()
                    .unwrap()
                    .unregister_callback(callback_id)
                    .expect("Tried to unregister callback with an id ...");
            }
        }
    }
}
```

Figure 5.1: system_api/src/sandbox_safe_system_state.rs:99-157

The `apply_changes` method uses `assert` and `expect` to ensure that system state invariants involving cycle balances, call contexts, and callback updates are upheld. By sending a WebAssembly (Wasm) execution output with invalid system state changes, a compromised sandboxed process could use this to cause the replica to panic.

Exploit Scenario

A malicious canister gains arbitrary code execution within a sandboxed process. The canister sends a Wasm execution output message containing invalid state changes to the replica, which causes the replica process to panic, crashing the entire subnet.

Recommendations

Short term, revise `SystemStateChanges::apply_changes` so that it returns an error if the system state changes from a sandboxed process are found to be invalid.

Long term, audit the codebase for the use of panicking functions and macros like `assert`, `unreachable`, `unwrap`, or `expect` in code that validates data from untrusted sources.

6. SandboxedExecutionController does not enforce memory size invariants

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DFCS-6
Target: `replica_controller/src/sandboxed_execution_controller.rs`	

Description

When a sandboxed process has completed an execution request, the execution state is updated by the `SandboxedExecutionController::process` method with the data from the execution output.

```
// Unless execution trapped, commit state (applying execution state
// changes, returning system state changes to caller).
let system_state_changes = if exec_output.wasm.wasm_result.is_ok() {
    if let Some(state_modifications) = exec_output.state {
        // TODO: If a canister has broken out of wasm then it might have allocated
        // more wasm or stable memory than allowed. We should add an additional
        // check here that the canister is still within its allowed memory usage.
        execution_state
            .wasm_memory
            .page_map
            .deserialize_delta(state_modifications.wasm_memory.page_delta);
        execution_state.wasm_memory.size = state_modifications.wasm_memory.size;
        execution_state.wasm_memory.sandbox_memory = SandboxMemory::synced(
            wrap_remote_memory(&sandbox_process, next_wasm_memory_id),
        );

        execution_state
            .stable_memory
            .page_map
            .deserialize_delta(state_modifications.stable_memory.page_delta);
        execution_state.stable_memory.size = state_modifications.stable_memory.size;
        execution_state.stable_memory.sandbox_memory = SandboxMemory::synced(
            wrap_remote_memory(&sandbox_process, next_stable_memory_id),
        );
        // ... <redacted>

        state_modifications.system_state_changes
    } else {
        SystemStateChanges::default()
    }
} else {
    SystemStateChanges::default()
};
```

Figure 6.1: `replica_controller/src/sandboxed_execution_controller.rs:663–700`

However, the code does not validate the Wasm and stable memory sizes against the corresponding page maps. This means that a compromised sandbox could report a Wasm or stable memory size of `0` along with a non-empty page map. Since these memory sizes are used to calculate the total memory used by the canister in `ExecutionState::memory_usage`, this lack of validation could allow the canister to use up cycles normally reserved for memory use.

```
pub fn memory_usage(&self) -> NumBytes {
    // We use 8 bytes per global.
    let globals_size_bytes = 8 * self.exported_globals.len() as u64;
    let wasm_binary_size_bytes = self.wasm_binary.binary.len() as u64;
    num_bytes_try_from(self.wasm_memory.size)
        .expect("could not convert from wasm memory number of pages to bytes")
        + num_bytes_try_from(self.stable_memory.size)
            .expect("could not convert from stable memory number of pages to bytes")
        + NumBytes::from(globals_size_bytes)
        + NumBytes::from(wasm_binary_size_bytes)
}
```

Figure 6.2: `replicated_state/src/canister_state/execution_state.rs:411–421`

Canister memory usage affects how much the cycles account manager charges the canister for resource allocation. If the canister uses best-effort memory allocation, the implementation calls through to `ExecutionState::memory_usage` to compute how much memory the canister is using.

```
pub fn charge_canister_for_resource_allocation_and_usage(
    &self,
    log: &ReplicaLogger,
    canister: &mut CanisterState,
    duration_between_blocks: Duration,
) -> Result<(), CanisterOutOfCyclesError> {
    let bytes_to_charge = match canister.memory_allocation() {
        // The canister has explicitly asked for a memory allocation.
        MemoryAllocation::Reserved(bytes) => bytes,
        // The canister uses best-effort memory allocation.
        MemoryAllocation::BestEffort => canister.memory_usage(self.own_subnet_type),
    };
    if let Err(err) = self.charge_for_memory(
        &mut canister.system_state,
        bytes_to_charge,
        duration_between_blocks,
    ) {
        // ... <redacted>
    }

    // ... <redacted>
}
```

Figure 6.3: `cycles_account_manager/src/lib.rs:671–714`

Thus, if a sandboxed process reports a lower memory usage, the cycles account manager will charge the canister less than it should.

It is unclear whether this represents expected behavior when a canister breaks out of the Wasm execution environment. Clearly, if the canister is able to execute arbitrary code in the context of a sandboxed process, then the replica has lost all ability to meter and restrict canister execution, which means that accounting for canister cycle and memory use is largely meaningless.

Exploit Scenario

A malicious canister gains arbitrary code execution within a sandboxed process. The canister reports the wrong memory sizes back to the replica with the execution output. This causes the cycles account manager to miscalculate the remaining available cycles for the canister in the `charge_canister_for_resource_allocation_and_usage` method.

Recommendations

Short term, document this behavior and ensure that implicitly trusting the canister output could not adversely affect the replica or other canisters running on the system.

Consider enforcing the correct invariants for memory allocations reported by a sandboxed process. The following invariant should always hold for Wasm and stable memory:

```
page_map_size <= memory.size <= MAX_SIZE
```

`page_map_size` could be computed as `memory.page_map.num_host_pages() * PAGE_SIZE`.

Summary of Recommendations

The DFINITY canister sandbox is a work in progress with multiple planned iterations, but the current iteration already provides strong security guarantees against malicious canisters deployed on the network.

In addition to addressing the findings detailed in this report, Trail of Bits recommends that DFINITY consider deploying cgroups to restrict sandboxed processes' use of resources. This will further strengthen the canister sandbox against attackers.

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 for the various automated analysis tools used during this audit.

cargo-afl

The Cargo plugin `cargo-afl` can be installed with the command `cargo install cargo-afl`. To build the fuzzing harness with thread-safe instrumentation and with the address and thread sanitizer enabled, write the following command in the `fuzz_transport` root directory:

```
AFL_USE_ASAN=1 AFL_USE_TSAN=1 AFL_LLVM_THREADSAFE_INST=1 \
cargo afl build
```

To run the fuzzer, create two directories, `corpus` and `output`, in the `fuzz_transport` root directory, and add some small (or empty) files to the `corpus` directory. (This will constitute the initial inputs for the fuzzer.) Start the fuzzer by executing the following command in the `fuzz_transport` root directory:

```
cargo afl fuzz -i corpus -o output target/debuf/fuzz_transport
```

We ran the fuzzer for a few hours but did not encounter any crashes.

```
use afl::fuzz;
use std::sync::Arc;
use std::io::Write;
use std::os::unix::net::UnixStream;
use byteorder::{BigEndian, WriteBytesExt};

use once_cell::sync::Lazy;
use std::sync::Mutex;

use ic_canister_sandbox_common::transport::socket_read_messages;
use ic_canister_sandbox_common::protocol::transport::SandboxToController;

// Create channel and receiving thread lazily.
static SENDER: Lazy<Mutex<UnixStream>> = Lazy::new(|| {
    let (sender, receiver) = UnixStream::pair().unwrap();
    let receiver = Arc::new(receiver);
    std::thread::spawn(move || {
        loop {
            eprintln!("RECEIVER: starting message loop");
            socket_read_messages(
                |_: SandboxToController| { /* Ignore decoded messages. */ },
                receiver.clone()
            );
        }
```

```
    });
    Mutex::new(sender)
});

fn main() {
    fuzz!(|data: &[u8]| {
        let mut size_buffer = Vec::new();
        let size = data.len() & 0xFFFF;
        size_buffer.write_u32::<BigEndian>(size as u32).unwrap();

        // Send size and data over the channel.
        let mut guard = SENDER.lock().unwrap();
        guard.write_all(&size_buffer)
            .expect("SENDER: failed to write size to socket");
        guard.write_all(&data[..size])
            .expect("SENDER: failed to write data to socket");
        guard.flush()
            .expect("SENDER: failed to flush socket");
    });
}
```

Figure C.1: An AFL-based fuzzing harness for the IPC transport implementation

cargo-audit

The Cargo plugin `cargo-audit` can be installed using the command `cargo install cargo-audit`. Invoking `cargo audit` in the root directory of the project runs the tool.

We ran `cargo-audit` in the `dfinity/ic/rs` directory, filtering out the dependencies for the `canister_sandbox` crate. This run identified a number of dependencies with known vulnerabilities and resulted in one issue (TOB-DFCS-1).

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 `canister_sandbox` crate. This run did not generate any warnings.

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 either 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 `canister_sandbox` crate. Since support for Rust in Semgrep is still experimental, we focused on locating 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.2: `panic-in-function-returning-result.yaml`

```
rules:
- id: unwrap-outside-test
  patterns:
  - pattern: $RESULT.unwrap()
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.unwrap()
            ...
        }
    "
  message: Calling `unwrap` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.3: `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.4: `expect-outside-test.yaml`

- The use of the `as` keyword, which may silently truncate integers during casting (for example, casting `data.len()` into a u32, may 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
```

Figure C.5: `length-to-smaller-integer.yaml`

- Unexpected comparisons before subtractions (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 {

            } 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.6: `switched-underflow-guard.yaml`

This run did not result in any issues or code quality recommendations.

Tarpaulin

Tarpaulin can be installed using Cargo by running `cargo install cargo-tarpaulin`. To execute Tarpaulin on Linux, simply run the command `cargo tarpaulin` in the crate root directory.

Note that we failed to obtain unit test coverage data using Tarpaulin because of a compilation issue related to the `bitvec` crate. We also tried to use `llvm-cov` to generate coverage data but were unsuccessful.

D. Code Quality Recommendations

The following is a list of findings that were not identified as immediate security issues but may warrant further investigation.

- In the `spawn_socketed_process` function (in `replica_controller2/src/launch_as_process.rs`), the path can be created as `CString::new(exec_path)` directly, since `&str` implements `Into<Vec<u8>>`.

- The SELinux policy for the node (`ic-os/guestos/rootfs/prep/ic-node/ic-node.te`) contains multiple lines that allow the canister sandbox to execute process memory.

```
allow ic_canister_sandbox_t ic_canister_sandbox_t:process {
    execmem fork getsched
};

# ... <redacted>

allow ic_canister_sandbox_t self : process { execmem };
```

Figure D.1: The SELinux policy grants `execmem` to `ic_canister_sandbox_t` twice.

  This could cause issues if the policy is refactored in the future.

- The call to `buf.reserve(MIN_READ_SIZE)` in the `socket_read_messages` function (in `common/src/transport.rs`) will panic on 32-bit platforms if the message size passed from the sandbox is greater than `usize::MAX - MIN_READ_SIZE`. (This is not a security issue since the codebase targets only x86_64.)