DFINITY Canister Sandbox

Fix Review

September 5, 2022

Prepared for:
Robin Künzler
DFINITY

Prepared by: **Fredrik Dahlgren**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 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.

When undertaking a fix review, Trail of Bits 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.

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 scope, timeline, test targets, and coverage are provided in the original audit report.

DFINITY contracted Trail of Bits to review the fixes implemented for issues identified in the original report. From September 1 to September 5, 2022, one consultant conducted a review of the client-provided source code.

Summary of Findings

The original audit uncovered flaws that could impact system confidentiality, integrity, or availability. However, we needed to assume that a malicious canister could obtain arbitrary code execution within a sandboxed process in order to exploit any of the issues identified during the 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

Overview of Fix Review Results

DFINITY has sufficiently addressed two of the six issues described in the original audit report.

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 engineer was associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@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
September 5, 2022	Delivery of fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided Jira issues, commits, and source code
- An automated review of the system's dependencies

Project Targets

The engagement involved a review of the fixes implemented in the following target.

Canister Sandbox

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

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	The canister sandbox has vulnerable dependencies	Resolved
2	Complete environment of the replica is passed to the sandboxed process	Unresolved
3	SELinux policy allows the sandbox process to write replica log messages	Unresolved
4	Canister sandbox system calls are not filtered using Seccomp	Unresolved
5	Invalid system state changes cause the replica to panic	Unresolved
6	SandboxedExecutionController does not enforce memory size invariants	Resolved

Detailed Fix Review Results

1. The canister sandbox has vulnerable dependencies	
Status: **Resolved**	
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.

Fix Analysis

This issue has been fixed. All vulnerable dependencies except `chrono` and `serde_cbor` have been updated. Since the implementation does not change the process environment, it is not vulnerable to the `chrono` issue. The `serde_cbor` crate is still an indirect dependency of the canister sandbox, but the vulnerable code path is not reachable.

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

Status: **Unresolved**	
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.

Fix Analysis

This issue has not been addressed, and the DFINITY team accepts the associated risk. Since all system environment variables are defined in a single location, it is easy to ensure that they do not contain sensitive data.

3. SELinux policy allows the sandbox process to write replica log messages	
Status: **Unresolved**	
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.

Fix Analysis

This issue has not been addressed, but the DFINITY team is planning to fix this issue at a later time by modifying the associated code so that log messages are relayed from the sandboxed process to the replica and by disabling access to `stdout` and `stderr`.

4. Canister sandbox system calls are not filtered using Seccomp	
Status: **Unresolved**	
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.

Fix Analysis

This issue has not been addressed. The DFINITY team is considering implementing Seccomp filtering using Seccomp-BPF at a later time, but there are a number of issues that need to be resolved before that is possible. Using Seccomp-BPF to filter system calls requires the replica to install a BPF filter for the sandboxed process. As the security model for the replica forbids processes to load BPF programs into the kernel, this is not currently an option. Using Seccomp-BPF also requires infrastructure to be able to monitor the process for potential false positives to ensure that the installed filter is not overly restrictive.

5. Invalid system state changes cause the replica to panic

Status: **Unresolved**	
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.

Fix Analysis

This issue has not been addressed. According to the DFINITY team, the general problem of protecting the replica against a compromised sandboxed process is difficult to solve completely and needs more thought before it can be addressed.

6. SandboxedExecutionController does not enforce memory size invariants	
Status: **Resolved**	
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.

Fix Analysis

This issue has been addressed. The `update_execution_state` method in the sandbox execution controller now validates the page map against the memory size reported by the canister. If a discrepancy is found, it is logged by the system.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

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

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.