PURETENSOR
INTELLIGENCE DISTILLED

# The Harness Must Assume Its Commands Escape

About 111,000 Stub Processes, 1,055 Multiplexer Servers and 21 Sessions in the Real One

**Paper:** PT-R-2026-072 v1.0

**Author:** PureTensor Ltd

**Date:** 10 November 2026

**Status:** Published

## Abstract

On 10 November 2026 we were hardening an outbound-mail guard: a hook that reads a shell command line an agent is about to run and refuses it if bash would end up running our mail-sending script with a send or reply verb. To test it we ran generated commands under a real bash in a stub world, built from three environment settings: a scratch directory as the only entry on the search path, holding a stub that stands in for the mail script; a scratch home directory; and a private socket directory for the terminal multiplexer. The commands under test were, by design, the kind that reset or bypass the environment, and the wall between them and the real system was made of environment variables.

The commands got out three ways. Two of the escapes are timed and fall between 20:10 and 23:05 UTC, a window of 2 h 55 min; the time of the third is not recorded. An exported startup variable made the stub start itself again and again, a chain of about 111,000 processes by the incident record's count, lasting somewhere between 10 and 18 minutes until we killed it by hand. Cleanup that killed only the default multiplexer socket left 1,055 servers running on other sockets. A command that cleared the environment dropped the private socket directory and opened 21 sessions in the operator's real multiplexer; five of them launched the real send command line. None of that code ran, because the script's interpreter was resolved through a search path that still pointed at the stub directory, and the stand-in found there never ran Python. By the check the incident record gives of the sent folders (which we did not repeat), no mail was sent.

The damage was transient load, six missed metric scrapes and a critical alert that fired for about a minute and cleared itself. The margin that stopped a real send was an accident of how an interpreter was resolved. This is a small paper: one incident, one host, no controlled experiment. It states the three mechanisms, the fixes that landed in the repository at 23:19 UTC, and the rules we now apply to any harness that runs generated commands. The central one is to assume the commands escape, and to build the wall so that an escape costs nothing.

## 1. What We Were Testing, and Why It Needed a Real Shell

The guard is a pre-execution hook. It receives the text of a command an agent proposes to run and decides whether bash would, in the end, execute the mail script with a send or reply verb. Reading commands the way bash does is hard. The guard has to see through quoting layers, here-documents, wrapper programs, terminal-multiplexer command strings and variable indirection. We spent one day on it, with three code-review rounds, two independent QA passes by agents, two security reviews in that pull request and the engineer's own probing.

What made the guard testable at all was an oracle. We stopped arguing about what bash would do and ran it. Each generated command goes to a real bash in a stub world, and the guard is required to block exactly those commands for which the stub received a send. The stub world has three parts. The first is a scratch directory that is the only entry on the search path, holding a stub that records its argument vector in a marker file. The second is a scratch home directory. The third is a private socket directory for the terminal multiplexer, set through the variable the multiplexer reads to decide where its server socket lives. Real programs (tmux, env, bash and a set of common wrapper programs) stay available. Python is replaced by a stand-in that never runs Python (it re-executes a script named for the mail script, and otherwise exits with an error), and the fish, csh and tcsh shells, which are not installed, are replaced by scripts that run their command string with bash.

The test file in the repository held 92 tests before the review rounds and 278 at the end of the day. It was not the file that leaked. The escapes came from copies of the oracle harness kept in scratch space outside the repository: the engineer's probe script and the batteries of the two QA agents. The second QA pass built 22 generator families and produced 27,769 commands, of which 15,792 were real sends, and 15,719 of those sends were refused (counts taken from the pull request description; the batteries cannot be rerun). The first pass ran about 15,000 commands. Those figures matter here only as a measure of exposure: tens of thousands of generated commands, each with a chance to misbehave, were being run by a real shell on a real host.

The question of this paper is not about the guard. It is what happens when the commands under test are chosen to defeat environment-based isolation and the only isolation is environment-based.

## 2. The Incident Timeline

All three escapes happened on one four-GPU compute node. They were sequential and of different lengths, not simultaneous. The sessions in the real multiplexer opened between 20:10 and 22:38 UTC. The self-starting chain ran until about 23:05 UTC. The time of the server leak is not recorded. The hardening commit landed at 23:19 UTC, 14 minutes after the chain ended.

| Event | Start (UTC) | End (UTC) |
|---|---|---|
| 21 sessions opened in the real multiplexer | 20:10 | 22:38 |
| Self-starting stub chain, work-log version | 22:55 | 23:05 |
| Self-starting stub chain, pull request version | 22:47 | 23:05 |
| Critical alert fired and cleared | 22:55 | about 22:56 |
| Hardening commit | 23:19 | not applicable |

*Table 1. Timeline of 10 November 2026. The two versions of the chain's start come from two texts written within minutes of each other; the monitoring record (Section 3) favours the earlier one.*

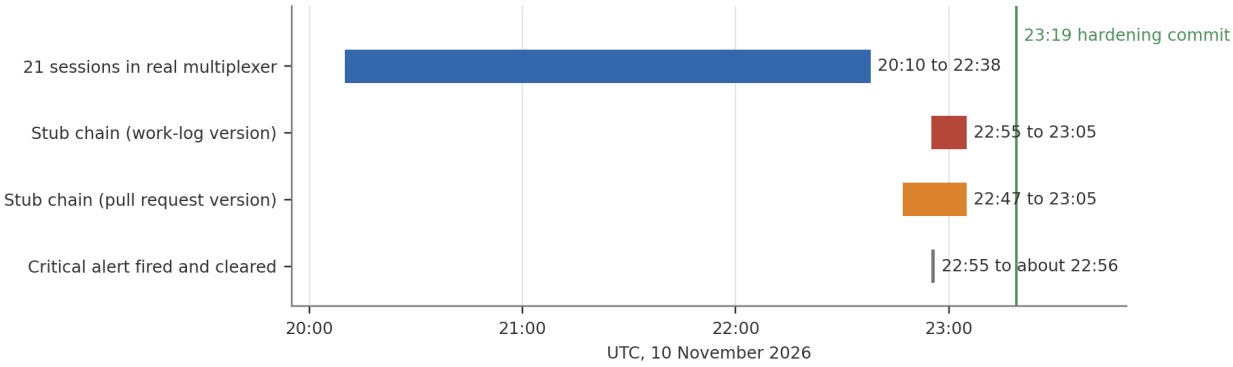



*Figure 1. The incident on 10 November 2026, UTC. The two bars for the self-starting chain are the two recorded versions of its start.*

The chain's duration is stated three ways in our records. The work-log entry gives 22:55 to 23:05, which is 10 minutes. The pull request text gives about 22:47 to 23:05, which is 18 minutes. The lesson note and the end-of-day report say about 15 minutes. The monitoring record shows the node's metrics exporter missing scrapes from 22:49, before the work-log start, so 22:55 is at best the time the engineer noticed the chain and the earlier start is better supported; the true figure is therefore nearer 18 minutes than 10. We still say "10 to 18 minutes", because no instrument sampled the chain itself.

## 3. Escape One: A Stub That Started Itself

One probe case exported BASH_ENV with a value that was a command substitution invoking the stub. A non-interactive bash expands BASH_ENV and runs the result before anything else. The stub was itself a bash script. So each time the stub started, it first expanded BASH_ENV, which started the stub, which expanded BASH_ENV, and so on. Every hop is a new process, so this is an unbounded chain and not a fork bomb with fan-out: one process at a time, but with nothing to end it.

The size was about 111,000 processes. That figure is the incident record's count, written in the work log at 23:10 UTC and repeated in the pull request text and the end-of-day report. The processes no longer exist, nothing recorded the peak, and a comment in the harness code puts the chain at about 100,000 processes. We treat both as approximate. The host process count of about 3,970 that the end-of-day report gives was a snapshot taken after the chain had been killed, so it is not the peak and must not be read as one.

What the chain cost was six missed scrapes by the node's metrics exporter, from 22:49 to 22:54, and one critical alert, a stalled-materialisation alert on our agent memory, which was pending from 22:50, fired at 22:55 and resolved itself after about a minute. The alert fired on the scrape gap; the materialiser itself kept running throughout. The end-of-day report notes that the alert may have paged. The end-of-day report gives a host load average of 18 before cleanup and about 9 and falling after it, and about 3,970 processes before cleanup against about 1,810 after. We could not attribute fleet-wide metrics to a single node, so we use none of them.

The case was invisible to the oracle. The stub was working: it did record calls, and a harness that only inspects the marker file sees many calls and no failure. A test can pass while the machine underneath it is on fire.

The stub's first line was a shebang naming bash with no flags. Bash has a privileged mode, switched on with the -p flag, in which it ignores BASH_ENV and does not import functions from the environment. Adding the flag to the stub's shebang closes this escape.

## 4. Escape Two: 1,055 Multiplexer Servers

The batteries started multiplexer servers on named sockets and on explicit socket paths inside the generated commands. Their cleanup ran the multiplexer's kill-server command against the default socket of the private directory only. A server on a different named or explicit socket is a different server, so the cleanup never reached them.

We found 1,055 of them still running. The count comes from the engineer's cleanup listing: the servers were identified by their working directory, which lay inside the scratch worlds, listed first and then stopped. It cannot be re-verified now. The count may include servers from both batteries and the probe, and the split is not recorded. The cleanup also found three stale sockets and two leftover watch processes, and a check of the two other compute hosts found no leftover servers.

This leak is quiet in a particular way. A multiplexer server is a long-lived process that survives the test that started it. It is invisible to the test's pass or fail, and invisible to a subprocess timeout, which kills only the direct child. The server is not the child. It daemonises, and the commands that arrive later come from it.

There was also an organisational cause. Three harness copies (the engineer's and the two agents') each carried its own cleanup, and none was the hardened one in the repository. We did not examine whether any of the 1,055 servers ran a stub call after the test that created it had finished.

## 5. Escape Three: The Environment Reset That Reached the Real Server

Battery commands of the form env -i PATH=... tmux new -d (command) clear the whole environment and set only the search path. That removed two of the three things the wall was built from: the socket-directory variable and the home variable. With the private socket-directory variable gone, the multiplexer fell back to its default socket, which is the operator's real server. With the home variable gone, a tilde in the command resolved to the real home directory. Only the search path survived, and it still pointed at the stub directory.

The result was 21 new sessions in the operator's real multiplexer, opened between 20:10 and 22:38 UTC. The operator's own two sessions were not touched. We killed all 21, and the server was left holding the operator's two sessions only. The numbering of the sessions carries no information about their count, so we use only the 21 listed.

Five of the 21 sessions launched the real mail script with a send verb and an identifier argument, because with the home variable reset the tilde in the command resolved to the real script. This is the part of the incident that needs care in the telling. The working summary said that five sessions "executed the real mail script". That is true of the file and false of its code. The interpreter the script names resolved, through the stub search path, to the stand-in, which re-executed the script in a loop, and the real interpreter never ran a line of it. Five sessions launched the real send command line, and none of it ran.

On whether any mail went out, we rely on the incident record's check, recorded in the work log, the pull request and the report: the sent folders of three mailboxes held only the day's routine messages, and no bounce arrived. We did not open the mailboxes ourselves.

This escape matters more than the other two. They cost CPU and a minute of alert. This one was a path to a real side effect, an outgoing message, and it was stopped by an accident of search-path resolution and not by any control. The wall was one environment variable deep, and the commands under test were chosen to remove environment variables.

# 6. One Soft Wall, Three Holes

| Escape | What was left behind | The fix that closes it |
|---|---|---|
| Stub re-ran itself through BASH_ENV | about 111,000 processes (incident record's count) | bash -p on the stub |
| Servers on other sockets survived cleanup | 1,055 servers | own socket directory per command, kill-server on whatever appears, process-group kill |
| env -i reached the real server | 21 sessions, 5 launching the real send line, 0 lines of its code run | never run an environment reset next to a home path or the multiplexer |

*Table 2. What each escape left behind and the one change that closes it.*

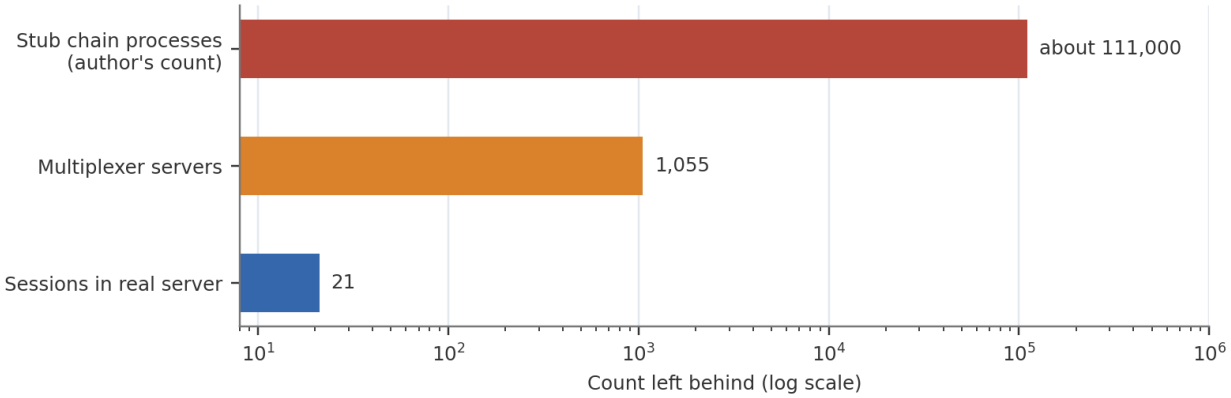



*Figure 2. What each escape left behind, on a logarithmic axis: about 111,000 processes, 1,055 servers, 21 sessions. The 111,000 is the incident record's count.*

The common cause is plain. The only things between a generated command and the real system were environment variables: the search path, the home directory and the socket directory. The commands under test were, by construction, the kind that reset or bypass the environment, or that start programs the harness does not own. A multiplexer starts servers of its own and runs commands from them later; a shell may export a startup hook. A harness that tests an adversary-shaped command set behind a soft wall will be breached by exactly the cases it is good at generating. The same structural lesson, that a gate which can only pass says nothing about what it was meant to contain, appeared in an earlier post of ours on a green merge gate that hid dropped code; we do not retell it here.

# 7. What These Numbers Will Not Carry

This is one incident on one host on one day, written up by the engineer who caused it. There are no rates and no confidence intervals. The process, server and session counts are from the engineer's cleanup and not from a monitoring system; the test, timing and command counts come from the repository or the pull request text. The work-log entry, the pull request, the end-of-day report and the lesson note were written by the same agent within minutes of each other, so their agreement is not independent confirmation.

Several things are not established. The true peak of the process chain is unknown. The exact start of the chain is unknown; 22:47 is better supported than 22:55, but neither was measured. We do not know whether the surviving servers ran any stub call after their tests ended. The 111,000 figure cannot be remeasured. The window of 2 h 55 min describes the two timed escapes together, not any single escape, and the third escape has no recorded time. "No mail was sent" is the incident record's check of the sent folders. An independent verification pass

planned that evening did not run, so we cannot say whether it would have caught the leak.

What survives is narrower. The three mechanisms are deterministic and are documented in the harness code comments. The nuance that five sessions launched the send line and none of it ran follows from how the interpreter was resolved. The fixes are in the repository history, and the test count is taken from the repository at the final commit. What does not survive is any claim that the rules below are sufficient: they are post hoc, and we did not run the hardened harness against a hostile case. Of the 11 additional tests collected in that round (267 to 278), the commit message attributes the rise to the guard fixes; only one new test function targets the harness, and it checks only the skip rule in the fourth item below. The rest are not new harness tests.

## 8. What We Changed

All of this landed in the repository's test file in the final commit of the day, at 23:19 UTC, together with the guard fixes of that round. The file then held 278 tests, up from 267 at the start of that round, and runs in about 57 seconds (the pull request's figure, not re-timed).

First, the stub runs under bash -p, so BASH_ENV and imported functions are ignored. Second, every command runs in its own session, and after each run, on success as well as on timeout, the whole process group is killed; the old code killed only the direct child on timeout. Third, the harness stops a multiplexer server whenever the command may have started one, whatever socket it used: each command gets its own socket directory, a kill-server is run against it if anything appears there, and the caller's multiplexer variable is never passed through. The environment given to bash is a fixed whitelist of a few variables.

Fourth, a command that resets or replaces the environment and also mentions a home path, the home variable or tmux is never run. It is judged by the guard alone. Fifth, ad-hoc harnesses outside the repository are not protected by any of this, so the rule is to copy the run function from the repository file and not to write a new one.

Two operating habits came with it. After any such battery we look at the host: total process count, the number of multiplexer servers and the session list on the real socket. And when cleaning up, we identify leftovers by working directory, list first and then kill, because a pkill pattern typed on a command line matches the shell that is running it; the pattern belongs in a script file.

The guard itself is a pre-execution filter and was always meant to be one layer. Further hardening of the sender is planned.

## 9. Methods and Artefacts

Sources read: the hook-hardening pull request description, the harness test file at its final version and its history (six commits that day), the hardening commit message, the work-log entry written at 23:10 UTC, the end-of-day report and the lesson note in agent memory. The test count is taken from the repository at the final commit. Tools: bash 5 with the -p flag, a tmux 3.x multiplexer, pytest and Python, on a compute node with four GPUs. Retained: the repository history and the work-log entry. The harness copies that leaked and the processes themselves are gone, so no raw process listing exists.

*PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction. This paper is PT-R-2026-072. The incident took place on 10 November 2026 between 20:10 and 23:05 UTC; the repository history and the work-log entry are retained, while the processes and the leaked harness copies are not.*