Understanding Linux signals for IR

Taz Wake | Halkyn Consulting Ltd | Originally published 2026-06-16

Signals? What's this all about?

To most responders, Linux signals mean one thing: killing processes. That instinct is correct but incomplete. On Linux, *signals are a built-in mechanism for delivering asynchronous notifications to running processes*. In other words, they are a way for the kernel (or another program) to say "*stop what you're doing and handle this event right now*."

For someone coming from Windows, this can feel unusual because Windows applications don't typically receive these kinds of injected software interrupts (although Asynchronous Procedure Calls do exist). Instead, the Win32 model revolves around message queues, kernel objects, and structured exceptions, all of which require the application to actively wait for or poll system events.

Although this shouldn't be a surprise, Linux does it differently. Rather than a process checking for messages, the operating system can directly interrupt it. Understanding this difference makes the Linux kill command far more intuitive. It isn't really about "killing" a process, instead it is a way to send it a specific signal that the process may choose to handle, ignore, or act upon.

Signals, Signals, Signals

A primer on signals

A signal sends a message, typically a number or a "name" (or both), and a process can do one of three things with it: run the default action defined for that signal, usually termination; ignore it; or run a handler it has registered to catch it. The last case is the one that matters most here, because when a process registers a handler, the kernel records that decision, and that record is readable from outside the process.

Key point here - *the process can decide how to deal with the signal*.

Two signals are exempt from all of the above. **SIGKILL** (9) and **SIGSTOP** (19) cannot be caught, blocked, or ignored. SIGKILL terminates a process unconditionally; SIGSTOP suspends it. The process gets no say in either, and that cuts both ways. You can always halt or kill a target regardless of how it was written, and an attacker can always freeze your monitoring agent with the same guarantee. Much of what follows relies on those two facts.

```
root@siftworkstation:~: # kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

The signal numbers fall into two ranges. Standard signals occupy 1–31 and carry fixed meanings: **SIGHUP**, **SIGINT**, **SIGTERM**, etc. Real-time signals occupy the upper range and have no predefined purpose; a program may use them however it sees fit, which is exactly what makes them attractive as a covert trigger.

That upper range has a caveat worth noting. The kernel provides 33 real-time signals, numbered 32 to 64. However, glibc's threading implementation reserves the first two for internal use and adjusts the application-visible **SIGRTMIN** upwards. This moves it to 34 on *glibc*, and 35 under *musl*. The practical consequence is that real-time signals should always be referenced with the **SIGRTMIN+n** notation rather than a hardcoded integer, because the same logical signal lands on a different number depending on the C library the binary was built against. Tooling that hardcodes will eventually be wrong on a host that doesn't match its assumptions. This carries over to different architectures as well. The numbers above hold on x86-64 and ARM, which cover most of what you will triage, but signal numbers are architecture-specific, so SPARC, MIPS, and Alpha differ. Read the names, not the integers, and confirm with **kill -l** on the host in front of you.

Decoding Signals

Signals (or at least signal handlers) are often stored in a bitmasked form. This is a useful bash "helper" function that will decode the signal bitmasks.

You can test this by running

This will return a list of signals that your current shell session has handlers (SigCgt) for.

Process state checking

Before you look at what signals a process handles, look at the state the process is already in. Checking process status is a good, quick-win, and will often be enough here. There are two ways to do this, either ps or reading the proc system. Examples could be:

You can narrow it down further by filtering for processes which are in a *stopped* or *stopped under a tracer* state.

Alternatively, you can read the process status in full with

Both approaches give you the data you need, but show it in slightly different ways.

The states you actually care about during an investigation are a short list. **R** is running or runnable. **S** is interruptible sleep, which is the normal resting state for most processes, waiting on an event. **D** is uninterruptible sleep, almost always blocked in a kernel I/O path. **T** is stopped. **t** is stopped specifically under a tracer. **Z** is a zombie, terminated but not yet reaped by its parent. Most of what you see will be **S** and **R**; the others are where attention is warranted.

Two of these are worth dwelling on.

T means something sent the process a **SIGSTOP** or **SIGTSTP,** and it is suspended but still resident. This means the process isn't dead; it is just frozen. There are benign reasons: job control, a paused batch job, a developer's debugger. There is also a hostile one that we might care a LOT about. An attacker can suspend a monitoring agent or logging daemon with SIGSTOP, and because the process never exits, nothing restarts it. It simply stops doing its job, quietly, until a matching SIGCONT. Processes with a T status are not common, and if it is a security-relevant process, it is definitely worth investigating further.

t (lower case t), and its companion field TracerPid, tell you about tracing. TracerPid in /proc/[pid]/status holds the PID of the process currently ptrace-attached to this one, or 0 if there is none. If it isn't zero, this is a clear indicator that something is attached to the process. Legitimate reasons would be a debugger or an strace session. However, it might also be an implant or an attacker process injecting into the victim. A reasonably old anti-forensics technique was an attacker adding a self-trace, because Linux systems only permit one tracer at a time, which means the self-tracer blocks the investigator's debugger.

A practical aside on the **D** state message. A process in uninterruptible sleep will not respond to signals at all, including SIGKILL, until the kernel operation it is blocked on completes. This is the usual explanation for "*I sent kill -9 and nothing happened*." It is normally a stuck I/O or a hung network mount rather than malice, but it is worth recognising on sight so you don't misread an unkillable process as something cleverer than a blocked syscall.

Hunting example - finding active tracers

The bigger picture - signals in IR

The process state we've just looked at tells you what a process is doing. Disposition tells you what it has decided about the signals it might receive, and the kernel records that decision, where you can read it. The /proc/[pid]/status fields **SigBlk**, **SigIgn**, and **SigCgt** are bitmasks of the signals a process is blocking, ignoring, and catching with a handler

Decode SigCgt with the helper from earlier and you are looking at the process's own handlers. Most of it is mundane, e.g. a shell catches SIGCHLD, a daemon catches SIGHUP to reload and so on. The value is in the entries that have no innocent explanation. A process catching SIGTRAP is very often resisting a debugger. A process catching a real-time signal which it has no functional reason to use is worth a deeper check: it could be a control channel hiding in plain sight, waiting for a specific kill to switch behaviour, activate, or wipe. This is often seen with rootkits.

That is the same mechanism viewed from the attacker's side. Because the two uncatchable signals give a guaranteed result, SIGSTOP is a quiet way to neutralise a defender, suspend a monitoring agent or logging daemon, and because the process never exits, nothing restarts it; it simply stops working until a matching SIGCONT. And a caught signal makes a serviceable command interface for an implant that would rather not open a socket.

We mentioned rootkits previously. The Diamorphine LKM rootkit uses the kill command as its entire control channel: sending signal 31 toggles hiding a process, signal 63 toggles the module's own visibility, and signal 64 grants root to the caller. **kill -64 0**, then by an **id** comand showing **uid=0(root)** is the whole privilege escalation step. Sneakily, this is often going to be missed by checking SigCgt. Diamorphine intercepts these signals by hooking the kill syscall in the kernel, so they are consumed in the syscall path and never delivered to a userland handler, so nothing is registered, nothing appears in any mask, and detection is annoyingly hard.

Signals in DFIR

Signals are also a tool you can turn the other way during live response. The same that an attacker uses to silence an agent lets you halt a suspect process *without killing it*. That means no exit, no handler-driven cleanup, no lost memory image, so you can capture it before it can react. Consider something like this:

This workflow, freeze, capture, then decide, rather than killing first and losing everything the process was holding, allows you to understand the process before acting. To be clear, there are two caveats to be aware of here: the freeze itself changes process state, and a sufficiently aware implant can watch for it, and a SIGCONT handler will fire on resume, so neither step is truly silent. Keep good notes and record your actions/decisions just in case something goes wrong.

Conclusion

Signals, and the kill command, do a lot more than simply cause a process to exit, although some signals will do that. When we investigate systems during IR, we need to check the process statistics to confirm what it is doing and what signals may have been triggered, but we should also be aware of the impact that individual signals have. Attackers can use signals to stealthily block security tools or interact with rootkits. A good way to find a lot of this data is by checking the /proc/[pid]/ filesystem and looking into how the memory objects are structured.

If you are interested in learning more about investigating Linux systems, then have a look at https://sans.org/for577 - the Linux Incident Response class from SANS Digital Forensics and Incident Response

---

About Halkyn Consulting

Halkyn Consulting Ltd is a UK-based security and risk management consultancy providing 24/7 incident response, digital forensics, threat hunting, virtual security leadership (vCISO) and security compliance support to organisations worldwide. We deliver clear, practical, vendor-independent advice based on real-world threats — no hype, no upsell.

More free security resources are available at www.halkynconsulting.co.uk/security-resources.

Web: www.halkynconsulting.co.uk | **Email:** info@halkynconsulting.co.uk | **Tel:** +44 1244 940858