Linux Kernel Runtime Guard (LKRG) 1.0

A talk by

Solar Designer <solar@openwall.com>
@solardiz

Openwall / @Openwall / https://www.openwall.com
CIQ / @CtrlIQ / https://ciq.com

Credits: Adam 'pi3' Zabrocki et al., Openwall, Binarly, CIQ/Rocky Linux

September 5, 2025
Berlin, Germany

What is LKRG

```
* Project of Adam 'pi3' Zabrocki
  + Brought under Openwall umbrella for its first public release in 2018
  + (Post-)detection of (and response to) kernel rootkits and exploits

* Linux kernel module that performs
  + Runtime integrity checking of the kernel and modules (including itself)   <- "code"
  + and of CPU flags and globals and the kernel's view of running processes   <- "data"
  + Sanity-checking of control flow (stack unwinding) and blocking some APIs
  + Response to (almost-)successful attacks and encrypted remote logging

* 20+ contributors total, 4 very active this release cycle (0.9.9 to 1.0.0)
  + Also users contribute testing beyond what we could perform ourselves

* Independent project with some corporate support
  + Remote logging research and initial implementation in 2022 by Binarly
  + Many general improvements, releases, packaging in 2023-2025 by CIQ
```

Openwall's most controversial project ever?

* Can't win against attack from same privilege level, they say?
* But what if you counter-attack?
* Compared to exploit or Core War, our goal is modest: detect and stop attack
* OTOH, exploits may know exactly what we do before we do it, so we randomize

> Core War is a programming game introduced in 1984 by D. G. Jones and A. K. Dewdney.
> In the game, two or more battle programs, known as warriors, compete for control of a
> virtual computer. These programs are written in an abstract assembly language

- Wikipedia

> Players write programs to eliminate all opponents in the memory [...]
> Core War News - 2025
> 16 Jun
> Congratulations to Inversed and Dave Hillis, who claimed first and second place in
> the Nano Core War Challenge

- Core War website

The landscape

* Kernel hardening patches
  + Early one-security-feature patches in mid-1990s, then compilations of those
  + Openwall's -ow patches for Linux 2.0.x since 1997 to 2.4.x maintained until 2010
  + grsecurity + PaX since 2001, by Brad Spengler (spender) and PaX Team (pipacs)
  + At first protecting the userland by the kernel, later also kernel self-protection

* Upstreaming efforts
  + Openwall's GSoC 2011 project with Vasiliy Kulikov, kernel-hardening mailing list
  + Kernel Self-Protection Project (KSPP) by Kees Cook et al. since 2015

* Tetragon eBPF-based Security Observability and Runtime Enforcement by Cilium, 2022+
  + Functionality and approach partially overlaps with LKRG, but implemented via eBPF

* LKRG is mostly orthogonal to and may co-exist with kernel hardening patches/changes
  + Some LKRG users are running it with hardened kernels (forks of old grsecurity)
  + LKRG may probably co-exist with and protect Tetragon, but this is trickier

Runtime integrity checking

This involves several activities

When LKRG is loaded:
* Generate a random key for SipHash
* Snapshot the initial state
  + Keyed hashes of CPU metadata, kernel image, and modules as they're seen in memory
  + Values of critical global variables such as SELinux "enabled" and "enforcing"
  + Shadow database of running tasks (processes and threads), including credentials

While LKRG runs:
* Track legitimate changes and update the snapshot (avoiding race conditions)
  + Hook kernel functions that make such changes, pause checking, update on success
* Validate current state against the snapshot (also avoiding race conditions)
  + At intervals, on random events, and most importantly just before it would matter
* On unexpected differences, log the event and trigger the configured response
  + log and accept or log only or kill task or panic the kernel (defaults vary)

Linux Kernel Runtime Guard (LKRG) 1.0

System (kernel) integrity checking (kINT)

This includes:
* CPU metadata: CPUs online, per-CPU WP SMEP SMAP, IDT, MSRs (e.g., syscall address)
* Kernel image in memory, modules loaded, module images in memory

* Some of these things are also checked more frequently (current CPU's flags)
* Some also have separate configuration knobs (SMEP, SMAP, MSRs)
  + For custom configuration, but normally these are updated as part of a profile

Legitimate updates include:
* Logical CPUs going offline and back online
* Kernel modules getting (un)loaded by legitimate root (task passes integrity check)
* Kernel or modules getting patched at runtime to optimize (not so) static branches

Questionable updates include:
* Hooking of kernel functions such as for profiling by legitimate root
  + We currently allow only ftrace (including k[ret]probes optimized to ftrace)

Task (process) integrity checking (pINT)

This includes, for each task (process or thread):

* Objective credentials (real_cred)
  + Pointer
  + Real, saved, effective, and filesystem UIDs/GIDs
  + User namespace (container)
* seccomp
  + Mode
  + First filter pointer

Legitimate updates include (if existing task and/or control flow passes check):

* New task wake-up (start tracking it)
* Task exiting (stop tracking it, know its task_struct address and PID may be reused)
* SUID/SGID execve()
* set*id()
* prctl() PR_SET_SECCOMP
* Some filesystems (NFS, OverlayFS) temporarily overriding subjective credentials

Task (process) and control flow integrity checking (pINT and pCFI)

* When a task is about to be updated or when its credentials are about to be used
  + such as to open a file

  + Check that it (or the updating task) passes integrity check so far
  + Check that control flow up to this point looks sane

* Poor man's control flow integrity checking (pCFI)

  + Aimed to detect and stop invocation of certain kernel functions from ROP chains

  + We check stack frames to ensure the call chain corresponds to kernel addresses

  + We also do this in a few extra hooks that exist solely to perform this check
    + because those kernel functions were too useful in exploits, such as inode access

  + ROP skipping the hooked function's prologue (and entry hook) is a concern

Shadow task database

* We maintain our own database with our copies of critical data for each task we track

* The data structure(s) and locking conventions affect performance and scalability greatly

LKRG 0.0 (2018) to 0.7 (2000)
One RB tree guarded by one spinlock
original design by Adam

LKRG 0.8 (2000) to 0.9.9 (2024)
512-entry hash table of RB trees guarded by their corresponding 512 read-write locks
by Solar and Adam

LKRG 1.0.0 (2025)
512-entry hash table of RB trees guarded by RCU, lockless lookup (on Linux 4.2+)
by Sultan Alsawaf (CIQ)

usermodehelper blocking

* The kernel sometimes invokes userland programs on its own

  + Such as modprobe to load a kernel module on demand to expose some kernel CVE
  + or systemd-coredump to leak /etc/shadow via CVE-2025-4598

* On a more serious note, this functionality is in heavy use by existing distros
  + So blocking it completely is likely to break things (the system may fail to boot)

* However, usermodehelper is also used as a last step by many kernel exploits
  + Such as overwriting the string corresponding to the kernel.core_pattern sysctl
  + including for container escape

* We enforce a read-only allow list of known valid program pathnames by default
  + but also support complete blocking of usermodehelper as part of paranoid profile
    + which works OK if enabled on an already booted system

Configuration and self-protection

* Most LKRG configuration settings exist as both module parameters and sysctl
  + As an exception, remote logging currently only has module parameters

* For most features, we have pairs of *_validate and *_enforce settings
  + which control attack detection and response, respectively

* For simpler configuration, we also have profile_validate and profile_enforce
  + which adjust many other settings in those two categories to match a profile

* This runtime configuration, and other critical data, is kept on a read-only page
  + which is only set read-write temporarily when sysctl is used by legitimate root

* One of the sysctl's is named hide, which makes LKRG hide itself and its symbols

* LKRG frequently tests that a kretprobe hook works (detects disabling of kprobes)

Why remote logging

* Troubleshooting and post-mortem analyses of (non-)security incidents
  + System's local logs might be unavailable, incomplete, or tampered with
  + There's commonly no way to know the local logs are complete and intact
* Centralized processing for SIEM, EDR, ...
* Compliance

* Pre-existing remote logging solutions for Linux
  + Userspace: syslogd protocol, CORE-SDI's, rsyslog RELP, NXLog, systemd
  + Kernel: netconsole

Linux kernel netconsole

* A standard feature of the Linux kernel

* Can be built into the kernel image, but distros usually build as a module

* Sends kernel messages via syslog protocol over UDP
  + Unreliable and insecure transport - but simple and thus reliable setup

* Specialized and low-level enough that it:
  + Starts network logging early and doesn't fail even on a kernel panic
  + Requires Ethernet and manual specification of the target MAC address

* Intended to work over local Ethernet only, but also works over the Internet
  + Just specify the gateway's MAC address along with the target's IP address

* OK for one-off debug jobs, unreasonable and tough to use long-term at scale

Remote logging - initial design decisions

* Use libhydrogen to implement Noise protocol by Trevor Perrin et al., pattern "N"
  + Tiny embeddable library by Frank Denis of libsodium fame
  + Primitives designed by Dan Bernstein (DJB) et al.: X25519 ECDH, Gimli cipher
  + Noise (but with two-way patterns) is notably used in WireGuard and WhatsApp

* Use TCP (with UDP an option for later) and plaintext Noise message length headers

* Register a custom console
  + but don't send right from there to avoid deadlock and priority inversion issues
  + instead, merely queue a "work" to run from a pre-existing kworker thread

* Also, queue this "work" when LKRG knows it's just logged a message (redundant)

* In the queued "work":
  + Read from the kernel message buffer via the high-level almost-userspace API
  + Encrypt and send (if necessary, also [re]connect, resend what failed before)

Remote logging - initial implementation

* Released in LKRG 0.9.8 on February 28, 2024 just in time for a talk on the topic
  + Along with builds for Rocky Enterprise Linux 8.9 and 9.3 via SIG/Security

* Logs not only messages generated by LKRG, but also all other kernel messages

* The sending component is in the LKRG kernel module itself

* Communication is over a TCP socket opened write-only
  + Limits LKRG's remote attack surface - great

* Write-only is possible due to use of the "N" pattern
  + Precludes implementation of forward secrecy - not great

* The receiving and logging counterpart is in a userspace daemon, lkrg-logger

* There are also additional userspace utilities, lkrg-keygen and lkrg-logctl

Rootkit detection effectiveness

Master's Thesis of Juho Junnila, entitled "Effectiveness of Linux Rootkit Detection Tools", shows LKRG as the most effective kernel rootkit detector (of those tested):

* When LKRG is loaded before the rootkit, it detected 8 out of 9 kernel rootkits tested:
  + Diamorphine, Honey Pot Bears, LilyOfTheValley, Nuk3 Gh0st, Puszek, Reptile, Rootfoo Linux Rootkit, Sutekh
* There were no false positives
* The one undetected "rootkit" was actually a keylogger-only module, not a full rootkit

* No other tested rootkit detector was anywhere close to LKRG's effectiveness at this
  + AIDE, OSSEC, and Rootkit Hunter detected 2 out of 9 each, and Chkrootkit detected none

* However, those other tools detected some userspace rootkits, which LKRG doesn't try to
* AIDE plus LKRG is shown to be most effective, detecting 14 out of 15 rootkits total

Exploit detection, prevention, and bypasses

* There are few exploits we or the user community got working and tested directly
  + but of those we/they did, LKRG tends to detect and stop exploits in time
  + albeit sometimes close to the last step, e.g. one for CVE-2024-1086 at usermodehelper
  + Exceptions are exploits that target the userland, such as for DirtyCOW and DirtyPipe

* The bypasses we are aware of were implemented as PoC on top of pre-existing exploits
  + implying that those exploits were originally stopped by LKRG

* A series of bypasses by Ilya Matveychikov in 2018-2019, which we were responding to
  + on top of Andrey Konovalov's exploit for CVE-2017-1000112

* A bypass by Alexander Popov in 2021 on top of his own exploit for CVE-2021-26708
  + Finds and patches LKRG

* A common theme is that LKRG could do better by hiding its and the kernel's specifics
  + Our existing non-default hide sysctl actually breaks Alexander's bypass

Linux Kernel Runtime Guard (LKRG) 1.0

Portability and maintenance challenge

* We support a wide range of kernel versions/branches spanning more than a decade
  + LKRG 1.0.0 has been tested with kernels from RHEL/CentOS 7's 3.10.0-1160 to 6.17-rc4
* We support 4 CPU architectures: x86-64, 32-bit x86, AArch64 (ARM64), and 32-bit ARM

* The extent to which LKRG integrates into the kernel may exceed that of a rootkit

* We hook many kernel functions, including some non-exported ones
  + Function inlining and duplication for specialization by compiler are concerns

* We look up addresses of some non-exported kernel symbols

* With some of the addresses, we call into non-exported kernel functions
  + Control flow integrity enforcement by the CPU or compiler is a concern

* Yet we manage

Linux Kernel Runtime Guard (LKRG) 1.0

Intel CET IBT and KCFI (clang CFI) bypass

New in 1.0:

```
 *) Support (or rather be compatible with the kernel's use of) Intel CET IBT
    (CONFIG_X86_KERNEL_IBT) and/or KCFI (CONFIG_CFI_CLANG) for now on x86_64

 * When Intel CET IBT is in use, we can't call non-exported kernel functions
 * indirectly via pointers because there's generally no ENDBR64 instruction
 * at their start.  However, when CET SS is not in use, like it currently is
 * not by the kernel, we can bypass IBT by using RET as our indirect branch
 * instruction (RET is exempt from IBT and is assumed to be protected by SS).
 * We cannot use the NOTRACK prefix for this purpose because the kernel does
 * not enable its support in the CPU, but this wouldn't make much difference.
 *
 * We use macros to generate wrapper functions for making CET IBT compatible
 * indirect calls via PUSH/RET, one wrapper function per function pointer.
```

Typical bugs/issues so far

* Typical issues are mismatch in assumptions between kernel and LKRG, often races
  + e.g., we overlooked the introduction of SECCOMP_MODE_DEAD, now fixed in 1.0

* Typical impact is false positives
  + e.g., when a task was dying but not yet dead, we'd see SECCOMP_MODE_DEAD as "corruption"

* The worst near-miss (or near-hit?) was an ABI mismatch (we set a wrong register)
  + User-triggerable Oops (read via a near-NULL pointer), exposed/found/fixed in 2020

* Most input would be trusted under a non-compromised kernel
  + but under our threat model we treat it as partially untrusted to add defenses
  + Even if we fail to process such input safely, we don't make things worse than before

* Some input is actually untrusted even under a non-compromised kernel
  + We don't process it a lot, but we recognize that there's risk of vulnerabilities

Continuous Integration

* Currently have 24 CI jobs setup in GitHub Actions, mostly by Vitaly Chikunov

* 11 build+boot tests (we run QEMU, covering the 4 supported architectures)
  + Ubuntu from 18.04 to 25.10
  + Kernels up to Fedora's builds of latest unreleased mainline

* 11 build-only tests (some are cross-builds)
  + Includes CentOS 7 (boot testing on this is currently manual)

* 2 source code quality checks

* Every commit and pull request is tested, also works in personal forks of the repo

* This has helped A LOT, but there's room for improvement

Adoption in distros and products

* LKRG packages exist in:
  + ALT Linux, Arch Linux, Astra Linux, Gentoo, Guix, NixOS, Rocky Linux, Whonix, and Yocto

* Many of these are maintained, a few are not (are of older versions)

* With support by CIQ, we maintain LKRG in the Rocky Linux SIG/Security yum/dnf repository
  + Also usable on other Enterprise Linux distributions (RHEL, AlmaLinux, etc.)
  + Red Hat's stable kABI and "weak-modules" allow same build to work within a minor version
  + Update to LKRG 1.0 for 9.6 and 8.10 has been built and tested and is pending publication

In Rocky Linux from CIQ - Hardened (RLC-H), LKRG is enabled out of the box and is supported by CIQ. Moreover, it is signed to be part of the UEFI Secure Boot chain starting with keys pre-installed in off-the-shelf hardware.

Performance impact

* Adding up to performance impact are:
  + Relatively infrequent kernel integrity checks, which are somewhat time-consuming
  + Relatively quick but frequent task credentials and control flow checks
  + Keeping track of legitimate changes
  + Function hooking overhead

Numerous benchmarks were run to estimate LKRG's performance impact, which was found to vary greatly between different workloads. Some are not impacted at all (such as compute-only userspace workloads), while others are impacted significantly. A run of the Phoronix Test Suite against LKRG 0.8 shows overall performance impact at 2.5%, as the geometric mean of 58 individual test results. Some extra optimizations contributed by CIQ to LKRG for 1.0 may have reduced the performance impact slightly, as seen in individual tests we ran. OTOH, we are planning to add more defenses after 1.0, and some may have extra performance cost.

We are in contact with Michael Larabel from Phoronix again, and we expect official Phoronix benchmarks of 1.0 to be available in a few weeks.

LKRG 1.0

* List of major changes from LKRG 0.9.9 to 1.0.0 is too long for this slide (see CHANGES)
* In short, 1.0.0:
  + adds support for Linux 6.13+ (tested to 6.17-rc4)
  + adds support for forward-edge CFI (Intel CET IBT, KCFI)
  + reduces performance overhead (lockless task lookups, fewer and some lighter kprobes)
  + shrinks the codebase by ~2500 lines

```
$ git diff --shortstat v0.9.9..v1.0.0
 144 files changed, 2279 insertions(+), 4700 deletions(-)

$ git shortlog -sn v0.9.9..v1.0.0
    99  Solar Designer
    30  Sultan Alsawaf
     6  Vitaly Chikunov
```

Adam 'pi3' Zabrocki also remains active with the project this release cycle and going forward

Potential future

* Evolution towards even greater maturity
  + Cleaner source tree and code (adopt Linux kernel coding style)

* Improved self-protection (more read-only, etc.)
* Hiding own and the kernel's build specifics and load addresses

* Kernel attack surface reduction (such as reduced exposure of user and net namespaces)

* Detection and prevention of userspace attacks
* General anomaly detection in combination with remote logging/analysis

* Moving protection to higher privilege level (hypervisor, TEE)
  + However, sync/locking with the kernel may keep the kernel the leader

* Protect eBPF, and protect from it
  + Integrity checking, allow list of eBPF program hashes (fuzzy to support systemd)

Contact information and credits

e-mail
Solar Designer <solar@openwall.com>

Twitter
@solardiz @Openwall @CtrlIQ

websites
https://lkrg.org https://www.openwall.com https://ciq.com

LKRG is due to Adam 'pi3' Zabrocki and contributors (including me and others)

LKRG 1.0 release, Rocky Linux integration, and this talk are due to my work at
CIQ, the primary corporate sponsor of Rocky Linux https://ciq.com

LKRG remote logging research and initial implementation have been sponsored by
Binarly software supply chain security platform https://binarly.io