Linux Kernel Self-Protection

KEES COOK

Kees Cook has been working with Free Software since 1994, and has been a Debian Developer since 2007. He is currently employed as a software developer by Google, working on Android, Brillo, and Chrome OS. From 2006 through 2011 he worked for Canonical as the Ubuntu Security Team's Tech Lead and remains on the Ubuntu Technical Board. Before that, he worked as the lead sysadmin at OSDL before it became the Linux Foundation. He has written various utilities, including GOPchop and Sendpage, and contributes randomly to other projects, including fun chunks of code in OpenSSH, Inkscape, Wine, MPlayer, and Wireshark. Lately, he's been spending most of his time focused on security features in the Linux kernel. kees@outflux.net

Recent focus on defending the Linux kernel from attack has resulted in many fundamental self-protections being brought into the upstream releases across a wide spectrum of kernel internals. Getting these defenses deployed into the real world means there are fewer chances for attackers to gain a foothold on systems.

Linux systems have seen significant improvements in security over the last decade. Containers (with various combinations of namespaces) and mandatory access control policies (like SELinux) keep walls between groups of processes; privileged processes try to use only fine-grained capabilities; risky processes confine themselves with seccomp; execution chains are cryptographically integrity-checked, and the list goes on. This reduction in the attack surface of user space has resulted in more attention being given to attacks against the Linux kernel itself. Because the kernel is the mediator for all the mentioned security systems, successful exploitation of a flaw in the kernel means all these protections go out the window.

Much recent work has involved providing the Linux kernel with better self-protection. Although much of the prior security work in the kernel was designed to protect user space from user space, the Kernel Self-Protection Project [1] focuses instead on protecting the kernel from user space. Many of the ideas and technologies in this project come from the large PaX and grsecurity (https://grsecurity.net) patches, while others originate from academic research papers and similar sources. Ultimately, there are two fundamental principles: eliminate classes of bugs and remove exploitation methods.

Fixing security bugs is important, but there are always more to be found. With the average lifetime of security bugs being five years [2], kernel development needs to be aimed at eliminating entire classes of bugs instead of playing whack-a-mole. Poor design patterns that lead to bugs can be exterminated by changing APIs or data structures.

Removing exploitation methods is fundamentally about creating a hostile environment for an attack. The kernel already runs smoothly day-to-day, but when it hits unexpected situations, it needs to deal with them gracefully. These situations tend not to affect the regular operation of the kernel, but leaving them unaddressed makes exploitation easier.

Even redesigning kernel internals so that the criticality of flaws is reduced has a significant impact on security. If a bug causes a system to reboot instead of give full control to an attacker, this is an improvement. The downtime will be annoying, but it sure beats going weeks not realizing a system was backdoored and then having to perform extensive post-intrusion forensics.

There has been a steady stream of improvements making their way into the kernel, but the last three years have seen a number of significant (or at least interesting) protections added or improved. There isn't room to cover everything in this article, but what follows are highlights spanning a range of areas.

The self-protection technologies in the Linux kernel can be roughly separated into two categories: probabilistic and deterministic. Understanding the differences between these categories helps us evaluate their utility for a given system or organization's threat model. After defining what needs to be protected against, it's easier to digest what actually addresses the risks.

Probabilistic protections derive their strength from some system state being unknown to an attacker. They tend to be weaker than deterministic protections since information exposures can defeat them. However, they still have very practical real-world value. They tend to be pragmatic defenses, geared toward giving an advantage (even if small) to a defender.

Deterministic protections derive their strength from some system state that always blocks an attacker. Since these protections are generally enforced by architectural characteristics of the system, they cannot be bypassed just by knowing some secret. In order to disable the protection, an attacker would need to already have control over the system.

Probabilistic Protections

Two familiar examples of probabilistic protections, present in user space too, are the stack canary and Address Space Layout Randomization (ASLR). The stack canary is used to detect the common flaw of a stack buffer overflow in an effort to kill this entire class of bug. The protection, however, depends on the secrecy of the canary value in memory. If this is exposed, the protection can be bypassed by including the canary in the overflow. Similarly, ASLR raises the bar for attackers since they can no longer easily predict where targets are in memory. If the ASLR offset is exposed, then the memory layout becomes predictable again.

The Linux kernel has used a stack canary for a very long time. Recent improvements in the compiler (since GCC v4.9) have allowed for wider coverage of the stack canary protection, with `-fstack-protector-strong`, available in Linux since v3.14 when the kernel build configuration option `CONFIG_CC_STACKPROTECTOR_STRONG` was enabled.

ASLR in the kernel (KASLR) is a contentious issue since there have been a large number of ways to locally expose the offset. However, KASLR isn't limited to just randomizing the position of the kernel code. Improvements have been made to randomize the location of otherwise fixed data allocation positions as well.

KASLR still raises the bar for attackers, especially on systems that run without exposing user space, for example on protocol-only systems like routers, access points, or similar. An attacker facing KASLR risks crashing or rebooting their target if they make a mistake, which leads to very noticeable events from the perspective of the defender.

KASLR of the kernel code itself is controlled by `CONFIG_RANDOMIZE_BASE` and was introduced on x86 in Linux v3.14, arm64 in v4.6, and MIPS in v4.7. Other architectures are expected to gain the feature soon. In the further future, in an effort to address the weakness to exposures, the hope is to reorganize the kernel code at boot instead of just shifting it in memory by a single offset. KASLR of kernel memory is still being worked on, and is similarly architecture-specific. `CONFIG_RANDOMIZE_MEMORY` exists for x86_64 since Linux v4.8, and much of the same effect is already present on arm64 since v4.6.

Another place for randomization in the kernel is the order of the kernel's heap memory layout (not just its base offset). The introduction of `CONFIG_SLAB_FREELIST_RANDOM` in v4.7 (for the SLAB allocator) and v4.8 (for the SLUB allocator) makes it harder for attackers to build heap-spraying attacks. With this protection, an attacker has less control over the relationship between sequential memory allocations (they're less likely to be adjacent). If enough memory is allocated, though, the effect of this protection is diminished. Like KASLR, it raises the bar, if only a little.

Deterministic Protections

Two familiar examples of deterministic protections, present in user space too, are read-only memory and bounds-checking. The read-only memory flag, enforced by the CPU over designated segments of memory, will block any write attempts made within the marked regions. For an attacker trying to redirect execution flow, the less writable memory there is, the less opportunity they have to make changes to the kernel after they have found a stray write flaw. Bounds checking similarly restricts the cases where a stray write flaw may exist to begin with. If every index into an array is verified to be within the size of the given array, no amount of an attacker's wishing will escape the checks.

By far the most fundamental protection in the kernel is correct memory permissions. This is collected under the poorly named `CONFIG_DEBUG_RODATA` [3]. While it was at one time used for debugging, kernel memory permissions are used to enforce memory integrity. And while it once only controlled making read-only data actually read-only, it also now makes sure that the various safe combinations of memory permissions are in place: kernel code is executable and read-only, unchanging data is read-only and not executable, and writable data is (obviously) writable but additionally not executable. Fundamentally, nothing should ever be both executable and writable: such memory areas are trivial places attackers could use to gain control.

In the face of proper kernel memory protection, attackers tend to use user space memory for constructing portions of their attacks. As a result, the next most fundamental protection is making sure the kernel doesn't execute or (unexpectedly) read/write user space memory. The idea isn't new that kernel memory isn't available to user space (this is the whole point of system calls), but this protection is the inverse: user space memory isn't available to the kernel. If an attack confuses the kernel into trying to read or execute memory that lives in user space, it gets rejected. For example, without this protection it's trivial for an attacker to just write the executable portion of their attack in user space memory, entirely bypassing the permissions that make sure nothing is writable and executable in kernel memory.

Some models of CPUs have started providing this protection in hardware (e.g., SMEP and SMAP on x86 since Skylake, and PXN and PAN on ARM since ARMv8.1), but they are still rare, especially on server-class systems. Emulating these protections in software is the next best thing. 32-bit ARM systems can do this with `CONFIG_CPU_SW_DOMAIN_PAN` since Linux v4.3, and 64-bit ARM systems can do this with `CONFIG_ARM64_SW_TTBR0_PAN` since Linux v4.10. Unfortunately, as of v4.10, emulation for SMEP and SMAP was still not available for x86 in the upstream kernel [4].

The places where the kernel explicitly reads and writes user-space memory is through its internal calls to, respectively, `copy_from_user()` and `copy_to_user()`. Since these calls temporarily disable the restriction on the kernel's access of user-space memory, they need to be especially well bounds checked. Bugs here lead to writing past the end of kernel memory buffers, or exposing kernel memory contents to user space. While some of the bounds checking already happens at kernel compile time (especially since v4.8), many checks need to happen at runtime. The addition of `CONFIG_HARDENED_USERCOPY` in v4.8 added many types of object-size bounds checking. For example, copies performed against kernel heap memory are checked against the actual size of the object that was allocated, and objects on the stack are checked that they haven't spanned stack frames.

The kernel stack itself gained protections on x86 in v4.9 and arm64 in v4.10. Prior to `CONFIG_VMAP_STACK`, the kernel stack was allocated without any guard pages. This meant that when an attacker was able to write beyond the end of the current kernel stack, the write would continue on to the next kernel stack, allowing for the (likely malicious) manipulation of another process's stack. With guard pages, these large writes will fail as soon as they run off the end of the current stack. Introduced at the same time, the addition of `CONFIG_THREAD_INFO_IN_TASK` moves the especially sensitive `thread_info` structure off the kernel stack, making an entire class of stack-based attacks impossible.

Future Work

While not yet in the kernel as of v4.10, another interesting probabilistic protection that will hopefully arrive soon is struct randomization [5]. This will randomly reorganize the layout of commonly attacked memory structures in the kernel. This protection is less useful on distribution kernels (since the resulting layout is public), but still makes exploitation more challenging since an attacker now has to track this layout on a per-distribution and per-kernel-build basis. For organizations that build their own kernels, this makes attacks much more difficult to mount because an attacker doesn't know the layout of the more sensitive areas of the kernel without also being able to first gather very specific details through information exposures.

Building on the deterministic memory protection provided by `CONFIG_DEBUG_RODATA`, there has been some upstream work to further reduce the attack surface of the kernel by making more sensitive data structures read-only [6]. While many structures can already be easily marked read-only, others need to be written either once at initialization time or at various rare moments later on. By providing a way to make these structures read-only during the rest of their lifetime, their exposure to an attacker will be vastly reduced.

Another area under current development, as of v4.10, is protecting the kernel from reference-counting bugs. When there is a flaw in reference counting, the kernel may free memory that is still in use, allowing it to get reallocated and overwritten leading to use-after-free exploits. By detecting that a reference count is about to overflow [7], an entire class of use-after-free bugs can be eliminated. The work underway is to create a specific data type that is protected and only used for reference counting, and then replace all the existing unsafe instances.

Staying Updated

By far the best way to protect Linux systems (or any systems) is to keep them up-to-date. This isn't new advice, but it usually only takes the form of recommending that all security updates be installed. While that is absolutely a best practice to adhere to, it only addresses known flaws. The idea must be taken a step further: to get the latest kernel self-protection technologies, systems need to be running the latest Linux kernel.

If products are built using the Linux kernel, they need to be able to receive the latest kernels as part of their regular update cycle. This can end up being a fair amount of up-front cost, since drivers need to be upstreamed and proper automated testing procedures need to be implemented. The long-term results will quickly pay dividends since the burden of code maintenance is shared with upstream and the test environment will catch bugs as soon as they are introduced instead of months or years later.

If systems are built around a Linux distribution, they need to be kept upgraded to the latest distribution release. Many distributions have a "long term support" release that requires waiting a couple of years or more between upgrades. If, instead, a system is upgraded to the regular releases that usually come out on a six-month cycle, they will be much closer to the latest kernel. While distribution kernels will still lag slightly behind the latest kernel release, it's a reasonable tradeoff to make: the system has a more current kernel, but it is still supported by the distribution (unlike rolling your own kernel on top of a distribution).

The work to stay updated tends to be spread thinly across a longer time frame, rather than stacking up only to be addressed in bulk every few years. This generally means fewer emergencies and a smoother planning cycle. Beyond the other benefits of having more modern software, it'll also come with an ever increasing series of defenses designed to stop attacks before they begin.

References

[1] http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project.

[2] https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/.

[3] Along with CONFIG_DEBUG_SET_MODULE_RONX.

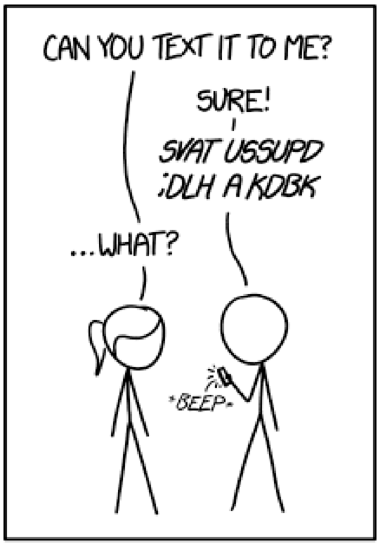
[4] Available in grsecurity via CONFIG_PAX_MEMORY_UDEREF.

[5] Available in grsecurity via the RANDSTRUCT GCC plugin.

[6] Available in grsecurity via CONFIG_PAX_KERNEXEC and the CONSTIFY GCC plugin.

[7] Available in grsecurity via CONFIG_PAX_REFCOUNT.

XKCD



SETTING MY PHONE'S SPEECH RECOGNITION TO DVORAK WAS A PAIN AT FIRST, BUT IT'S MORE EFFICIENT IN THE LONG RUN.

xkcd.com