Kernel Self-Protection

Kernel self-protection is the design and implementation of systems and structures within the Linux kernel to protect against security flaws in the kernel itself. This covers a wide range of issues, including removing entire classes of bugs, blocking security flaw exploitation methods, and actively detecting attack attempts. Not all topics are explored in this document, but it should serve as a reasonable starting point and answer any frequently asked questions. (Patches welcome, of course!)

In the worst-case scenario, we assume an unprivileged local attacker has arbitrary read and write access to the kernel's memory. In many cases, bugs being exploited will not provide this level of access, but with systems in place that defend against the worst case we'll cover the more limited cases as well. A higher bar, and one that should still be kept in mind, is protecting the kernel against a _privileged_ local attacker, since the root user has access to a vastly increased attack surface. (Especially when they have the ability to load arbitrary kernel modules.)

The goals for successful self-protection systems would be that they are effective, on by default, require no opt-in by developers, have no performance impact, do not impede kernel debugging, and have tests. It is uncommon that all these goals can be met, but it is worth explicitly mentioning them, since these aspects need to be explored, dealt with, and/or accepted.

Attack Surface Reduction

The most fundamental defense against security exploits is to reduce the areas of the kernel that can be used to redirect execution. This ranges from limiting the exposed APIs available to userspace, making in-kernel APIs hard to use incorrectly, minimizing the areas of writable kernel memory, etc.

Strict kernel memory permissions

When all of kernel memory is writable, it becomes trivial for attacks to redirect execution flow. To reduce the availability of these targets the kernel needs to protect its memory with a tight set of permissions.

Executable code and read-only data must not be writable

Any areas of the kernel with executable memory must not be writable. While this obviously includes the kernel text itself, we must consider all additional places too: kernel modules, JIT memory, etc. (There are temporary exceptions to this rule to support things like instruction alternatives, breakpoints, kprobes, etc. If these must exist in a kernel, they are implemented in a way where the memory is temporarily made writable during the update, and then returned to the original permissions.)

In support of this are `CONFIG_STRICT_KERNEL_RWX` and `CONFIG_STRICT_MODULE_RWX`, which seek to make sure that code is not writable, data is not executable, and read-only data is neither writable nor executable.

Most architectures have these options on by default and not user selectable. For some architectures like arm that wish to have these be selectable, the architecture Kconfig can select ARCH_OPTIONAL_KERNEL_RWX to enable a Kconfig prompt. `CONFIG_ARCH_OPTIONAL_KERNEL_RWX_DEFAULT` determines the default setting when ARCH_OPTIONAL_KERNEL_RWX is enabled.

Function pointers and sensitive variables must not be writable

Vast areas of kernel memory contain function pointers that are looked up by the kernel and used to continue execution (e.g. descriptor/vector tables, file/network/etc operation structures, etc). The number of these variables must be reduced to an absolute minimum.

Many such variables can be made read-only by setting them "const" so that they live in the .rodata section instead of the .data section of the kernel, gaining the protection of the kernel's strict memory permissions as described above.

For variables that are initialized once at `__init` time, these can be marked with the `__ro_after_init` attribute.

What remains are variables that are updated rarely (e.g. GDT). These will need another infrastructure (similar to the temporary exceptions made to kernel code mentioned above) that allow them to spend the rest of their lifetime read-only. (For example, when being updated, only the CPU thread performing the update would be given uninterruptible write access to the memory.)

Segregation of kernel memory from userspace memory

The kernel must never execute userspace memory. The kernel must also never access userspace memory without explicit expectation to do so. These rules can be enforced either by support of hardware-based restrictions (x86's SMEP/SMAP, ARM's PXN/PAN) or via emulation (ARM's Memory Domains). By blocking userspace memory in this way, execution and data parsing cannot be passed to trivially-controlled userspace memory, forcing attacks to operate entirely in kernel memory.

Reduced access to syscalls

One trivial way to eliminate many syscalls for 64-bit systems is building without `CONFIG_COMPAT`. However, this is rarely a feasible scenario.

The "seccomp" system provides an opt-in feature made available to userspace, which provides a way to reduce the number of kernel entry points available to a running process. This limits the breadth of kernel code that can be reached, possibly reducing the availability of a given bug to an attack.

An area of improvement would be creating viable ways to keep access to things like compat, user namespaces, BPF creation, and perf limited only to trusted processes. This would keep the scope of kernel entry points restricted to the more regular set of normally available to unprivileged userspace.

Restricting access to kernel modules

The kernel should never allow an unprivileged user the ability to load specific kernel modules, since that would provide a facility to unexpectedly extend the available attack surface. (The on-demand loading of modules via their predefined subsystems, e.g. MODULE_ALIAS_*, is considered "expected" here, though additional consideration should be given even to these.) For example, loading a filesystem module via an unprivileged socket API is nonsense: only the root or physically local user should trigger filesystem module loading. (And even this can be up for debate in some scenarios.)

To protect against even privileged users, systems may need to either disable module loading entirely (e.g. monolithic kernel builds or modules_disabled sysctl), or provide signed modules (e.g. `CONFIG_MODULE_SIG_FORCE`, or dm-crypt with LoadPin), to keep from having root load arbitrary kernel code via the module loader interface.

Memory integrity

There are many memory structures in the kernel that are regularly abused to gain execution control during an attack, By far the most commonly understood is that of the stack buffer overflow in which the return address stored on the stack is overwritten. Many other examples of this kind of attack exist, and protections exist to defend against them.

Stack buffer overflow

The classic stack buffer overflow involves writing past the expected end of a variable stored on the stack, ultimately writing a controlled value to the stack frame's stored return address. The most widely used defense is the presence of a stack canary between the stack variables and the return address (`CONFIG_STACKPROTECTOR`), which is verified just before the function returns. Other defenses include things like shadow stacks.

Stack depth overflow

A less well understood attack is using a bug that triggers the kernel to consume stack memory with deep function calls or large stack allocations. With this attack it is possible to write beyond the end of the kernel's preallocated stack space and into sensitive structures. Two important changes need to be made for better protections: moving the sensitive thread_info structure elsewhere, and adding a faulting memory hole at the bottom of the stack to catch these overflows.

Heap memory integrity

The structures used to track heap free lists can be sanity-checked during allocation and freeing to make sure they aren't being used to manipulate other memory areas.

Counter integrity

Many places in the kernel use atomic counters to track object references or perform similar lifetime management. When these counters can be made to wrap (over or under) this traditionally exposes a use-after-free flaw. By trapping atomic wrapping, this class of bug vanishes.

Size calculation overflow detection

Similar to counter overflow, integer overflows (usually size calculations) need to be detected at runtime to kill this class of bug, which traditionally leads to being able to write past the end of kernel buffers.

Probabilistic defenses

While many protections can be considered deterministic (e.g. read-only memory cannot be written to), some protections provide only statistical defense, in that an attack must gather enough information about a running system to overcome the defense. While not perfect, these do provide meaningful defenses.

Canaries, blinding, and other secrets

It should be noted that things like the stack canary discussed earlier are technically statistical defenses, since they rely on a secret value, and such values may become discoverable through an information exposure flaw.

Blinding literal values for things like JITs, where the executable contents may be partially under the control of userspace, need a similar secret value.

It is critical that the secret values used must be separate (e.g. different canary per stack) and high entropy (e.g. is the RNG actually working?) in order to maximize their success.

Kernel Address Space Layout Randomization (KASLR)

Since the location of kernel memory is almost always instrumental in mounting a successful attack, making the location non-deterministic raises the difficulty of an exploit. (Note that this in turn makes the value of information exposures higher, since they may be used to discover desired memory locations.)

Text and module base

By relocating the physical and virtual base address of the kernel at boot-time (`CONFIG_RANDOMIZE_BASE`), attacks needing kernel code will be frustrated. Additionally, offsetting the module loading base address means that even systems that load the same set of modules in the same order every boot will not share a common base address with the rest of the kernel text.

Stack base

If the base address of the kernel stack is not the same between processes, or even not the same between syscalls, targets on or beyond the stack become more difficult to locate.

Dynamic memory base

Much of the kernel's dynamic memory (e.g. kmalloc, vmalloc, etc) ends up being relatively deterministic in layout due to the order of early-boot initializations. If the base address of these areas is not the same between boots, targeting them is frustrated, requiring an information exposure specific to the region.

Structure layout

By performing a per-build randomization of the layout of sensitive structures, attacks must either be tuned to known kernel builds or expose enough kernel memory to determine structure layouts before manipulating them.

Preventing Information Exposures

Since the locations of sensitive structures are the primary target for attacks, it is important to defend against exposure of both kernel memory addresses and kernel memory contents (since they may contain kernel addresses or other sensitive things like canary values).

Kernel addresses

Printing kernel addresses to userspace leaks sensitive information about the kernel memory layout. Care should be exercised when using any printk specifier that prints the raw address, currently %px, %p[ad], (and %p[sSb] in certain circumstances [*]). Any file written to using one of these specifiers should be readable only by privileged processes.

Kernels 4.14 and older printed the raw address using %p. As of 4.15-rc1 addresses printed with the specifier %p are hashed before printing.

[*] If KALLSYMS is enabled and symbol lookup fails, the raw address is printed. If KALLSYMS is not enabled the raw address is printed.

Unique identifiers

Kernel memory addresses must never be used as identifiers exposed to userspace. Instead, use an atomic counter, an idr, or similar unique identifier.

Memory initialization

Memory copied to userspace must always be fully initialized. If not explicitly memset(), this will require changes to the compiler to make sure structure holes are cleared.

Memory poisoning

When releasing memory, it is best to poison the contents, to avoid reuse attacks that rely on the old contents of memory. E.g., clear stack on a syscall return (`CONFIG_GCC_PLUGIN_STACKLEAK`), wipe heap memory on a free. This frustrates many uninitialized variable attacks, stack content exposures, heap content exposures, and use-after-free attacks.

Destination tracking

To help kill classes of bugs that result in kernel addresses being written to userspace, the destination of writes needs to be tracked. If the buffer is destined for userspace (e.g. seq_file backed `/proc` files), it should automatically censor sensitive values.