Improving Memory Permissions in FreeBSD

by Brooks Davis

The virtual address space of a process contains a number of physical pages mapped into memory. These might be pages from a program, a library, an ordinary file, or *anonymous* pages that begin life as a zeroed page. These mappings are maintained in a translation lookaside buffer (TLB). On modern architectures, the TLB allows pages to be mapped with a combination of read, write, and execute permissions. This enables things like read-only sharing of code and data between processes for physical memory utilization.

Older architectures (e.g., MIPS, early i386) only supported read and write permission, but modern CPUs generally support an execute permission as well. Used correctly, the execute permission can mitigate a number of common security vulnerabilities. For example, it used to be common to exploit a program by writing code (commonly known as *shell code*) to an improperly bounds checked string on the stack and changing the saved return address of the function to point to the string. By removing the execute permission from the stack, we can prevent this attack. Most FreeBSD architectures do this.

As expected, breaking simple, stack-based attacks leads attackers to look for other vulnerabilities. One of the simplest next steps was to find a way to write code to a page that was mapped executable followed by smashing the stack to point the return address to it. A popular mitigation for this is the write-XOR-execute policy (W^X). This policy prevents mapping pages with both the write and execute permission. For most programs, this works without program changes outside the runtime linker, but some programs such as Java virtual machines and web browsers use just-in-time (JIT) compilers to generate code and run it. These JITs are critical to achieving reasonable performance, but, implemented naively, they don't work with W^X. Fortunately, it is usually a simple matter to map pages writable, write generated code to them, and then make them executable. Not all programs are trivially converted though, so W^X implementations generally provide a way to disable W^X for certain programs.

FreeBSD does not currently support W^X, but work is in progress. The main difficulty has been implementing an appropriate framework for tagging binaries that must opt out and providing mechanisms to test opting in or out. We have now added a general mechanism (and ELF note) for setting opt-in and opt-out bits in binaries as well as flags in `procctl` which allow features to be enabled or disabled in a given execution of a program. We expect to have W^X available in FreeBSD 13 and hope to have it enabled by default (at least for new programs). The latter part will depend on our confidence in testing existing software.

Setting Maximum Page Permissions

In the kernel, each page has both a set of permissions and a set of maximum permissions. For example, a page backed by a file that the process can read, but not write, will not include write permission in either set. Anonymous pages have read, write, and execute in all cases in FreeBSD. File-backed pages depend on the process's access to the file and any restrictions placed on the file descriptor when it was opened.

While the kernel knows about these, the standard APIs for manipulating page permissions—`mmap()` and `mprotect()`—don't. This leads to excessive default maximum permissions. For example, pages allocated for `malloc()` using `mmap()` default to read and write permission, but can be made executable using `mprotect()`. In almost no circumstance is that desirable, yet today there is no way to prevent it.

Diversion to CHERI

Before discussing solutions to this problem, we will consider another problem. The CHERI architecture adds a new hardware type, the capability. These capabilities (not to be confused with Capsicum capabilities) grant access to regions of virtual memory. They specify a range of accessible addresses along with a set of permissions—load, stores, and execute—comparable to the permission on pages, but at byte rather than page granularity (typically 4KiB). CHERI capabilities are designed to be used at C pointer. In our work on CheriABI, we created a process runtime and compilation environment based on FreeBSD (our fork is called CheriBSD) where all pointers in a program are CHERI capabilities. This includes pointers returned by `mmap()`.

When setting bounds on pointers in CheriABI, we attempt to follow the principle of least privilege, which states that no more permission should be granted than necessary. With pointers returned by `mmap()`, our initial inclination was to return pointers with permissions mapped from the requested page protections (e.g., read and write becomes load and store). This mostly works but does not work with all usage patterns. First, the runtime linker makes the initial mapping for each library with no permissions, simply reserving space. It then maps portions of the file with appropriate permissions. If we returned a pointer with no permissions, we would be unable to access the file (or make new mappings with more permissions). Second, JITs need to map writable to start and then convert those mappings to executable. In both cases, we need a way to specify the maximum expected permissions of the mapping in order to create a pointer that has those permissions.

In addressing the problems with `mmap()` in CheriABI, we wanted our changes to be minimal. In particular, we wanted the source to continue to work on non-CHERI systems and ideally for any `mmap()` extensions to be small enough to make it easy to apply them to a cross-platform code base. In the end, we decided to steal some bits from the `prot` argument and add a second set of bits for maximum permissions. We created a macro `PROT_MAX()` to be ORed with permissions to specify the maximum permissions. For example, a library mapping previously mapped with `PROT_NONE` would be mapped `PROT_NONE | PROT_MAX(PROT_READ | PROT_WRITE | PROT_EXEC)`. For this code to work on systems that don't support `PROT_MAX()`, it can easily be defined to `0`. To avoid widespread code changes, we chose to treat the supplied permissions as the maximum permissions unless maximum permissions are explicitly specified. This required a change to the runtime linker, but, otherwise, code in the FreeBSD base system just works with this change in behavior.

`PROT_MAX()` in FreeBSD

When we started exploring W^X, one of the issues that stood out was that virtually all mappings have execute permissions in their maximum permission set. We realized that `PROT_MAX()` could address this issue. In June 2019, we committed a change adding `PROT_MAX()` support to the FreeBSD kernel. It differs from the CheriBSD version in that we keep legacy maximum permission logic unless the program requests `PROT_NONE`, a `sysctl` is set to imply the maximum permission, or procctl is used to explicitly enable implying maximum permissions for this process.

Much like W^X, we believe that implying maximum permissions with `PROT_MAX()` annotations where required provides the best implementation of the principle of least privilege and is the right approach. Work is in progress to test the extent to which maximum permission can safely be implied and we hope to eventually turn on implying them by default in a future FreeBSD release.

Compatibility and Limitations

Extensions to `mmap()` are rarely compatible across platforms. NetBSD has a similar extension `PROT_MPROTECT()` which adds permissions to the maximum permission set relative to the permission set. This has the disadvantage that is doesn't easily allow maximum permissions to be downgraded later via `mprotect()`.

We found no similar extensions in other operating systems.

The `PROT_MAX()` model does have some limitations. You cannot currently use it to set a maximum permission of `PROT_NONE`. You also can't downgrade maximum permission on a range of pages with mixed permissions without touching each sub-range separately or setting all of their permissions identically.

A major limitation of page-granularity memory permissions is that most programming language objects are much smaller than a page. Linkers group together objects that should have similar permissions to allow permissions to be limited, but it is harder to constrain memory allocated by `malloc()`.

Conclusion

Page-granularity memory permissions are a useful defense against a number of attacks. In particular, they allow the application of the principle-of-least-privilege to system memory. More advanced systems such as CHERI's capability-based pointers allow further application of fine-grained permissions.

This article has covered the basics of machine-independent memory permissions that fit the current `mmap()` permission model. The architecture-specific implementations vary and are beyond the scope of this overview as are other related mitigations such as Intel's Supervisor Mode Access/Execution Prevention (SMAP/SMEP) and ARM's Privilege-Access-Never (PAN), which are related but protect the kernel from userspace rather than protecting userspace from itself. ●

BROOKS DAVIS has been a FreeBSD developer for nearly two decades and is a member of the FreeBSD core team. He has worked on network stacks, high performance computing, build systems, and most recently on enhancing FreeBSD for the CHERI architecture. He currently works for SRI International.