Toasterkit - A NetBSD Rootkit

Anthony Martinez
Thomas Bowen
http://mrtheplague.net/toasterkit/

Toasterkit - A NetBSD Rootkit

1. Who we are
2. What is NetBSD? Why NetBSD?
3. Rootkits on NetBSD
4. Architectural Overview
5. Our contributions
6. Demo
7. Protection
8. Prevention

Who we are - Anthony Martinez

Anthony is a system administrator for the New Mexico Institute of Mining and Technology's Computer Center, as well as an undergraduate Computer Science student at the university. He originally proposed the project that evolved into Toasterkit.

Who we are - Thomas Bowen

Thomas is a system administrator for the New Mexico Institute of Mining and Technology's Computer Center. He is also enrolled in the Computer Science program with emphasis in Information Assurance.

Why NetBSD?

NetBSD is a popular operating system for embedded systems. It is also extremely source-portable, meaning that when written properly, anything targeting the kernel is equally so. This way, the rootkit can work on any NetBSD port!

Additionally, NetBSD is something of a research tool — new ideas such as Veriexec and the `kauth` framework[a] are being worked on in NetBSD, and nobody (else) is targeting them.

[a] Which originated in Mac OS X.

History of Rootkits on NetBSD

- Chkrootkit hasn’t been updated since NetBSD 1.6. We’re at 4.0, with 5 soon to be released.
- If there are any rootkits targeting recent versions of NetBSD, none of them appear to be public.

Overview of NetBSD Architecture

- Portable across hardware
  The slogan: “Of course it runs NetBSD”. Excellent support for cross-building.
- Architecture of code
  All of the architecture-dependent pieces are abstracted behind common functions; we don’t have to worry about byte order, memory-manager specifics, etc.
- Loadable Kernel Modules (LKM)
  Not commonly used, but are enabled by default. It also allows code to infiltrate the kernel *ex post facto*; security-conscious administrators might disable this. Modules can add syscalls, `sysctl` nodes, executable formats, filesystem drivers, etc.

Overview of NetBSD Architecture

- Process security (`kauth`)
  The `kauth` framework acts as a gatekeeper between the kernel's own routines, and is designed to be more fine-grained than the previous UNIX superuser approach of "all-or-nothing". The `kauth` system is used in the construction of other security models.
- Security models
  NetBSD supports custom security models. The default model, `bsd44`, is the standard BSD securelevel and superuser scheme. Also documented in the manual page for `secmodel` is a sample module allowing users with a uid below 1000[a] to bind to the normally-reserved port range below 1024.

[a] This range is generally used for system daemon accounts

What we've done and how we've done it

- Elevated privileges within the kauth framework.
- Made processes hidden via direct kernel object manipulation
- Portably removed write-protection from kernel memory areas; required for modifying some kernel tables.
- Hooked `sysctl` and `ioctl` functions in order to hide sockets and modules.

Code Skeleton

- NetBSD includes example code for kernel modules in `/usr/src/sys/lkm/{misc,syscall}/example`, and several fully-featured modules in `/usr/src/sys/lkm`. The sample modules do very little, but provide a skeleton to build other modules on.
- Specifically, the `misc` example, originally intended to show how a system call is inserted “by hand”, can be modified to hook a system call.
- A sample Makefile is also included, which is simply a call into the already-existing NetBSD build process.

System Calls, hooking

- System calls are exposed, among other ways, via a global `sysent` array, though accessing this array is not the standard way of placing a system call.
  Each element of `sysent[]` is of type `struct sysent`, containing information for the userspace — number of arguments, size of arguments, flags[a], and the function to be called.
- We can modify existing behavior by changing the function pointer (`sysent[n].sy_call`) to one of our own design, if done carefully enough.
- All system calls have a uniform prototype for use in the kernel, and access any userspace arguments indirectly.

[a] As of NetBSD 4.0, the only flag is whether or not the syscall is multiprocessor safe.

Building, loading, using an LKM

- We use two types of loadable modules:
  1. `misc` modules, which provide no automatic initialization
  2. `syscall` modules, which automatically find the next unused system call number and insert themselves there.
- The NetBSD build system provides Make targets for loading, unloading, and building the module, no matter what its type.
- The module system is controlled by `ioctl` commands on `/dev/lkm`. This comes into play later, when we are hiding modules.

Privilege Escalation with Kauth

- The first module is relatively simple. It adds a system call that gives the user escalated privileges. Since NetBSD uses `kauth`, however, we can't just set the process's User ID to 0 (root) and call it done.
- Instead, we need to operate within `kauth`'s bounds. The interface is documented in the manual pages, and we use the `kauth_cred_dup` function on the credentials of process #1 — `init`.
- Since `init` shouldn't be running under any restrictions, considering its responsibilities, we considered it a fair process from whom to "steal" credentials.

Memory protection woes

- NetBSD requires that a CPU support memory management.
- Some parts of kernel space are protected against memory writes. This frustrated our immediate efforts to hook functions that weren't designed to be hooked.
- In general, memory pages can be marked as any combination of readable, writable, and executable.
- Low-level details are machine-dependent
- Thankfully, NetBSD provides us with `uvm`, a virtual-memory system designed at WUSTL, which abstracts memory management.
- Documented in the manual page for `uvm` is a function called `uvm_map_protect`, but calling it has no effect on kernel pages.

Memory Unprotection

- Removal of write-protection is required to modify certain parts of kernel memory:
  - Character device tables, specifically `lkm_cdevsw`, are protected against writes.
  - The sysctl tree is similarly marked read-only.
- Removing this type of write-protection should be done generically so as to maintain cross-platform compatibility: we can't go mucking around in the page tables ourselves.
- It also turns out that this work has already been done for us: Hidden underneath a `#ifdef KGDB` in `uvm_glue.c`, there is a function called `uvm_chgkprot`. It does what we need, so we copied it.

Hiding modules

- Modules are easily visible by means of `modstat` — without some way to hide this list, a rootkit is very obvious.
- `modstat` and friends operate by way of `/dev/lkm`, making `ioctl` calls to load, unload, and request the status of modules. Some way is needed to hook only these `ioctl` operations, since hooking the actual system call would be far too broad.
- The functions for device nodes are stored in `struct cdevsw` variables corresponding to each device: the one for `/dev/lkm` is named `lkm_cdevsw`. One of the slots in the structure is the responsible function for `ioctl`. Inserting a hook function, which returns “does not exist” whenever the module’s name begins with “rootkit”, only requires unprotecting the memory.

Sysctl

- `sysctl` is a tree structure originally from BSD, which was designed to allow an administrator to modify system parameters on-the-fly (without rebuilding the kernel), and is still used for that, but is now also used to report system information.
- Each node in the tree can either contain a value for the corresponding “key” or a pointer to a helper function that is to handle processing of that particular branch of the `sysctl` tree.
- Node entries are write protected and modification of helper functions is difficult using the documented (`sysctl(9)`) API.

Sysctl - Hiding network sockets

Our solution is to:

1. Scan the sysctl tree, getting to the level above where the function is to be found (using `sysctl_locate`).
2. Once the helper function node we want to modify is found, unprotect the memory
3. Insert our own hook function, based on the original

The user utility `netstat` accesses open port data via a `sysctl` helper function. Overriding this function allows us to hide open network ports.

Process hiding

- Hiding processes is accomplished by way of module implementing a system call that takes the name of a process to hide, and directly removes it from the `allproc` global kernel list, as well as a few other lists.
- This doesn’t prevent it from getting scheduled and running, since the NetBSD scheduler doesn’t operate on processes, instead working at thread granularity.
- This type of attack is referred to as “direct kernel object manipulation”.

Demo

Protection - Detecting hooks

Using a friendly loadable module, compare function pointer in tables with address of actual function.

- System call hooks
  Check sysent table against the addresses of the expected functions; see `/usr/src/sys/kern/init_sysent.c`.
- `sysctl` and `ioctl` hooks
  Check specific helper function nodes. A full sweep is a bit more difficult because there is no single source for the “correct” functions.

Detecting other stuff

- Detecting `kauth`
  We don't think there's a viable solution to this. There are many occasions where `kauth_cred_dup` is appropriate and correct, so emitting a warning each time it's used would just be noise.
- Detecting unprotection
  Again, no easy detection. There isn't a standard utility to display the kernel's memory mapping, but perhaps `pmap(1)` can be extended to do so.

Prevention

- The easiest way to prevent against attacks via loadable modules is to rebuild your kernel without `options LKM` enabled. Loadable modules are not frequently used in NetBSD, but if your system does require one, this might not be a usable solution.
- Another, though more intrusive, suggestion is to use security levels. Once the system has gone multi-user, kernels compiled without `options INSECURE` apply a variety of restrictions, including that kernel modules cannot be loaded.
- Unfortunately, common architectures such as i386/amd64, and mac68k/macppc, default to `INSECURE`. This also doesn't prevent someone sufficiently clever from loading modules before the securelevel gets raised.

Questions?

References

Our primary reference text for this project was *Designing BSD Rootkits*, by Joseph Kong. Other than that, the NetBSD source code was an excellent asset, as well as being well-documented and clear.