DAVID MALONE

security through obscurity

A REVIEW OF A FEW OF FREEBSD'S LESSER-KNOWN SECURITY CAPABILITIES

David is a system administrator at Trinity College, Dublin, a researcher in NUI Maynooth, and a committer on the FreeBSD project. He likes to express himself on technical matters, and so has a Ph.D. in mathematics and is the co-author of *IPv6 Network Administration* (O'Reilly, 2005).

dwmalone@maths.tcd.ie

IN THIS ARTICLE I'M GOING TO LOOK at some less well known security features of FreeBSD. Some of these features are common to all the BSDs. Others are FreeBSD-specific or have been extended in some way in FreeBSD. The features that I will be discussing are available in FreeBSD 5.4.

First, I'll mention some features that I don't plan to cover. FreeBSD jails are like a more powerful version of chroot. Like chroot they are restricted to a subtree of the file system, but users (including root) in a jail are also restricted in terms of networking and system calls. It should be safe to give root access to a jail to an untrusted user, making a jail like virtual system, not unlike some applications of UML [1] or Xen [2] but without running separate kernels for each system. A lot has already been written about jails [3], so I won't dwell on them further here.

Another feature that I don't plan to spend much time on is ACLs. ACLs are an extension of the traditional UNIX permissions system to allow you to specify permissions for users and groups other than the file's owner and group. Since ACLs are familiar to many people from Solaris, Linux, and Windows, to name just a few examples, I'll just refer to [4] for the details on FreeBSD.

File Flags

A lesser-known set of extended permissions is the "file flags" supported by BSD's UFS file system. Of interest to us here are the append-only, immutable, and undeletable flags. These names are reasonably self-explanatory: append-only files can only be appended to, immutable files cannot be changed in any way, and undeletable files cannot be deleted (or have their hard links removed—all names referring to the inode are protected).

Each of these flags comes in two flavors: system and user. System flags can only be set and cleared by root. User flags can be set and cleared by the file's owner and root. The `chflags` command can be used to set them and the `-o` flag to `ls` can be used to display them. With these commands the names used for the system version of the flags are `sappnd`, `schg`, and `sunlnk`, and the user versions are `uappnd`, `uchg`, and `uunlnk`.

For example, it often surprises newcomers to UNIX permissions that a file can be deleted by any user who can write to the directory the file is in. Below, user lmalone has set the undeletable flag, and now user dwmalone cannot remove it.

```
dwmalone@hostname% ls -ldo . normal undeleteable
drwxr-xr-x  19 dwmalone  wheel  -     3584 May 14 09:03 .
-rw-r—r—   1 lmalone   wheel  -        0 May 14 09:03 normal
-rw-r—r—   1 lmalone   wheel  uunlnk    0 May 14 09:03 undeleteable
dwmalone@hostname% rm normal undeleteable
override rw-r—r—  lmalone/wheel for normal? y
override rw-r—r—  lmalone/wheel uunlnk for undeleteable? y
rm: undeleteable: Operation not permitted
```

Even root cannot remove this file, until the flag has been cleared manually:

```
root@hostname# rm undeleteable
rm: undeleteable: Operation not permitted
root@hostname# chflags nouunlnk undeleteable
root@hostname# rm undeleteable
```

There are some obvious applications for these flags. Append-only files can be used as log files, protecting against accidental or malicious truncation. FreeBSD installs a number of important files as system immutable (libc, init) to keep them from being damaged accidentally.

The immutable flag can also be used to prevent people "stealing" a link to an SUID executable. Usually, in any directory for which a person has write permissions, he can make a hard link to any file in that filesystem for which he has read permissions. This means that if someone knows there is a vulnerability to be announced in some SUID executable, he can steal a hard link to it and still have access to the executable after the original file appears to the sysadmin to be deleted. If an immutable flag is set on a file, these sorts of games aren't possible [5].

Note that file flags can only be manipulated locally and cannot be set or cleared over NFS. This means that marking a file on an NFS server as immutable is a good way to keep anyone from changing it.

BSD Secure Level

As I described above, the file system flags provide some useful flexibility, and even some protection against shooting oneself in the foot as root. However, they provide little protection against a malicious root user, who could just clear all the flags before going about their nefarious business.

The BSD operating systems do provide a simple form of protection against a malicious root user in the form of numbered "secure levels," in which the higher the number, the greater the restrictions on what can be done. The secure level can be raised using the sysctl command, but cannot be lowered while the system is running. On FreeBSD the secure level is set to -1 by default, but the secure level for multi-user operation can be set in /etc/rc.conf by adding settings such as:

```
kern_securelevel_enable="YES"
kern_securelevel="2"
```

The restrictions placed on system operation at each secure level are as follows:

If secure level > 0 you can't:

- access hardware from user processes via /dev/mem, /dev/pci, I/O instructions, and so on;
- load or unload kernel modules;
- change system-level file system flags (unlink, immutable, append);
- run a debugger on init;
- or cause /dev/random to perform a reseeding operation by writing to it.

If secure level > 1 you also can't:

- open disks in /dev for writing (including SCSI pass-through devices);

- change firewall rules;
- or run the clock faster than twice its normal speed or turn time backwards.

If secure level > 2 you also can't:

- change certain secondary firewall features such as ipf's NAT and ipfw's dummynet configuration;
- change certain sysctl values (msgbuf_clear, ipport_reserved{high, low}).

At secure level 1, root can't change immutable files. A malicious root might decide to unmount the file system and edit the raw disk, circumventing file permissions, flags, and ACLS. At secure level 2, this isn't possible, as disks cannot be opened for writing. Interestingly, each jail actually has its own secure level, so a jail can run at a higher secure level than the host system.

This means that a careful combination of a high secure level and UFS file flags can prevent an intruder from installing rogue kernels or kernel modules, the sort of trickery described in Rik Farrow's "Musings" column last April [6]. For this to work, *all* the files and directories involved in the boot process need to be immutable—this would include /boot, /sbin, /etc, /bin, /lib, /usr/bin, /usr/lib, etc.

In practice, this isn't often done, as it reduces the amount that can be achieved with online system administration. To update libraries the system must be rebooted and the library installed in single-user mode before the secure level is raised.

I did say that the secure level could not be lowered while the system is running. There is a way around this that I have used. If you have chosen to include kernel debugger support in your kernel, then someone with access to the kernel debugger can reduce the secure level. For example, I can use CTRL+ALT+ESC to get to the debugger on the console of my server:

```
root@hostname# sysctl kern.securelevel=2
kern.securelevel: -1 -> 2
root@hostname# KDB: enter: manual escape to debugger
[thread pid 13 tid 100001 ]
Stopped at      kdb_enter+0x2f: nop
db> write securelevel 0
securelevel             0x2   =   0
db> continue
root@hostname# sysctl kern.securelevel
kern.securelevel: 0
```

This technique allows an administrator to run the system at a high secure level when appropriate but to lower it when needed. It is important to remember that access to the kernel debugger requires physical access to the system (either to the console or via FireWire) and requires debugger support in the kernel.

MAC Framework

The MAC (Mandatory Access Control) framework is part of the excellent work done by the TrustedBSD project [7] to bring new security features to FreeBSD. The MAC framework allows people to develop kernel modules that provide additional checks on what the kernel permits processes to do.

The MAC framework includes a number of sample modules implementing well-known security systems, such as the Biba integrity model and Multi-Level Security (MLS), which I'll just mention here since to do them justice would require many pages. The TrustedBSD project also provides a port of the SELinux [8] policy system. All the MAC modules that are shipped with FreeBSD are documented both in manual pages (man 4 mac) and in the FreeBSD handbook [9].

Along with these well-known modules are included a number of quirkier offerings. I'll mention three of these here: seeotheruids, bsdextended, and portacl.

Note, while some of these modules can be loaded at any time, they all require that the MAC framework be compiled into your kernel by adding options MAC to your kernel config file. The more complex modules must either be loaded at boot time or compiled into your kernel.

SEEOTHERUIDS MAC MODULE

The seeotheruids module prevents users from seeing processes owned by other users. Usually ps, netstat, /proc and top will display the processes and sockets on the system belonging to all users. When the seeotheruids module is enabled, normal users can only see their own processes. Root is not a normal user, so it can see everyone's processes. It is also possible to allow a particular group of users to see the processes of others; for example, we can arrange for users in group wheel (with GID 0) to be able to see everyone's processes:

```
root@hostname# kldload mac_seeotheruids
root@hostname# sysctl security.mac.seeotheruids.enabled=1
security.mac.seeotheruids.enabled: 0 -> 1
root@hostname# sysctl security.mac.seeotheruids.specificgid=0
security.mac.seeotheruids.specificgid: 0 -> 0
root@hostname# sysctl security.mac.seeotheruids.specificgid_enabled=1
security.mac.seeotheruids.specificgid_enabled: 0 -> 1
```

BSDEXTENDED MAC MODULE

The bsdextended MAC module allows you to define more complex relationships between users and what files they are permitted to access. Unlike file permissions, ACLs, and flags, these rules are not attached to particular files but are systemwide.

This is perhaps best explained using another example. Suppose we have a sandbox user "pproxy" whose sole purpose is to run a POP proxy daemon. This user probably only needs read access to a few files on the system but, in fact, has access to all the files that are readable by "others."

```
pproxy@hostname% ./ls -l
total 8068
-r-xr-xr-x  1 pproxy  wheel  4096352 Apr 30 12:07 cat
-r-xr-xr-x  1 pproxy  wheel  4096352 Apr 30 12:07 ls
-rw-r—r—  1 pproxy  wheel       6 Apr 30 12:27 myfile
-rw-r—r—  1 root    wheel       4 Apr 30 12:27 otherfile
pproxy@hostname% ./cat myfile
hello
pproxy@hostname% ./cat otherfile
bye
```

By loading the bsdextended module we can use the ugidfw command to define rules stating which files the pproxy user can access. The following commands set up rules that allow the pproxy user read and execute access to a file and its attributes if it belongs to user pproxy and no access to any other files:

```
root@hostname# kldload mac_bsdextended
root@hostname# ugidfw add subject uid pproxy object uid pproxy mode srx
root@hostname# ugidfw add subject uid pproxy object not uid pproxy mode n
```

The "subject" part of a ugidfw command describes the user or group that the rule applies to. The "object" part describes the files the rule applies to, by specifying their owner (either by UID or GID). The mode describes the permitted operations.

- r normal read access to the file
- w normal write access to the file
- x execute/search access
- s read access to file attributes, such as permissions, owner, etc.

a administrative operations such as chmod, etc.
n no access

The first matching rule is used to decide what access is permitted by mac_bsdextended. Remember that for an action to be permitted it must also be allowed by traditional permissions and any other MAC modules, ACLs, or file flags in force.

After creating these rules with ugidfw, the pproxy user has greatly reduced access to the system. They can no longer read world-readable files (even /etc/passwd and /etc/group are inaccessible), and they cannot write to any files:

```
pproxy@hostname% ./ls -l
ls: otherfile: Permission denied
total 8066
-r-xr-xr-x  1 3007  0  4096352 Apr 30 12:07 cat
-r-xr-xr-x  1 3007  0  4096352 Apr 30 12:07 ls
-rw-r—r—  1 3007  0        6 Apr 30 12:27 myfile
pproxy@hostname% ./cat myfile
hello
pproxy@hostname% ./cat otherfile
cat: otherfile: Permission denied
pproxy@hostname% echo > myfile
myfile: Permission denied
```

Those who read the preceding two examples carefully will have noticed that I used copies of ls and cat belonging to the pproxy user. In fact, I also had to statically link these commands, as the pproxy user cannot read the normal copies of cat, ls, or libc because of the ugidfw rules! This is much the same situation as setting up a chrooted environment, where the correct executables and libraries need to be available to the user.

PORTACL MAC MODULE

The portacl module provides more flexible control of who can use which network ports. The traditional UNIX-style rules controlling who can listen for data on a port are:

- Root can listen anywhere.
- Everyone else can listen on ports > 1023.

Thus certain daemons, such as Web servers and news servers, need to at least begin life running as root in order to get access to the required ports (port 80 and port 119, respectively).

The portacl module allows rules like "user www can bind to port 80," which means that the Web server never needs to run as root. Let's take that as an example:

```
root@www# kldload mac_portacl
root@www# sysctl security.mac.portacl.rules=uid:80:tcp:80,uid:80:tcp:443
security.mac.portacl.rules:  -> uid:80:tcp:80,uid:80:tcp:443
```

Here we're saying that UID 80 (i.e., user www) should be permitted to bind to tcp port 80 and tcp port 443. We do not need to make any other change. Since the constraints enforced by the MAC framework are in addition to the normal constraints enforced by the kernel, we need to tell the kernel to relax its usual restrictions and let portacl do the work.

```
root@www# sysctl net.inet.ip.portrange.reservedlow=0
net.inet.ip.portrange.reservedlow: 0 -> 0
root@www# sysctl net.inet.ip.portrange.reservedhigh=0
net.inet.ip.portrange.reservedhigh: 1023 -> 0
```

Portacl actually has an implicit rule that limits ports 1–1023 to root, unless otherwise permitted by your setting of security.mac.portacl.rules, so we don't need any further rules to have the other ports behave as usual.

We can then start Apache as user www—provided user www can write to the necessary log files. In order to do this you could move the log files to /var/log/www and have that directory owned by user www. The necessary changes to the default Apache conf file look like this:

```
Listen 0.0.0.0:80
LockFile /var/log/www/accept.lock
PidFile /var/log/www/httpd.pid
ErrorLog /var/log/www/httpd-error.log
CustomLog /var/log/www/httpd-access.log combined
```

In some cases, there will be no downside to starting a daemon as a non-root user. However, there are some minor downsides to starting Apache as user www. As the logs are owned by user www, a vulnerability in a CGI or PHP script may give write/truncate access to log files. Of course, file flags could be used to mitigate this.

Unfortunately, there is no equivalent of the net.inet.ip.portrange.reserved* sysctls for IPv6, which means that the mac_portacl module is of less use in combination with IPv6. This is why the example uses the wildcard IPv4 address explicitly: to stop Apache using the IPv6 wildcard address. This omission should be fixed in a future release of FreeBSD.

GEOM and Disk Encryption

The last feature of FreeBSD that I'll mention is GEOM. GEOM is a framework for dealing with disk-like objects in the FreeBSD kernel. For example, the physical disks on the system are registered with GEOM. GEOM has classes that understand PC partition tables and FreeBSD disk labels. These classes can examine the disk and make the partitions and subpartitions of the disk available in /dev.

GEOM isn't restricted to just recognizing subpartitions; it can also perform more complex transformations such as striping, network-based disks, and encrypted disks.

A sample module for doing disk-based encryption, called BDE, is included with GEOM. Using BDE is actually quite straightforward. Naturally, you need a spare partition to house the encrypted disk—in this case, we use /dev/ad0s1g. First, we initialize the disk and choose a passphrase—as usual, choosing a good passphrase is essential:

```
root@hostname# gbde init /dev/ad0s1g -L /etc/ad0s1g.lock
Enter new passphrase:
Reenter new passphrase:
```

The lock file specified in the command will have some information about the encrypted disk's "lock sector" written into it. This file should be treated with some care—it needs to be kept backed up, and knowing its contents will make an attacker's life easier. Next, we can attach the disk, create the file system, and check that everything works OK:

```
root@hostname# gbde attach ad0s1g -l /etc/ad0s1g.lock
Enter passphrase:
root@hostname# newfs /dev/ad0s1g.bde
root@hostname# mount /dev/ad0s1g.bde /stuff
root@hostname# df /stuff
Filesystem     1K-blocks Used    Avail Capacity  Mounted on
/dev/ad0s1g.bde  60667770    4 55814346     0%    /stuff
```

Now the file system can be used as usual, but BDE will encrypt each block before it is written to the disk.

The problem with encrypted disks is getting the passphrase to the system when it wants to use the disk. Naturally, you can't store the passphrase on an unencrypted disk; the encryption would be pointless! One option is to manually issue the gbde attach command whenever the administrator wants access to use the encrypted disk.

Another option is to require the administrator to enter the passphrase at boot time, when filesystems are mounted. This can be done by creating an appropriate entry in /etc/fstab and /etc/rc.conf:

```
root@hostname% fgrep /stuff /etc/fstab
/dev/ad0s1g.bde      /stuff      ufs   rw          2     2
root@hostname% fgrep gbde /etc/rc.conf
gbde_devices="AUTO"
```

With this configuration, the administrator will be prompted for the passphrase for /dev/ad0s1g at boot time. The boot scripts also support encryption of the swap partition. In this case, the key can be chosen randomly, as the contents of a swap partition aren't required after a reboot:

```
root@hostname% fgrep swap /etc/fstab
/dev/ad0s1b.bde      none        swap  sw          0     0
root@hostname% fgrep gbde /etc/rc.conf
gbde_swap_enable="YES"
```

More details about how to operate the GEOM BDE system, including how to detach and destroy encrypted disks, can be read in the gbde manual page. For a description of BDE internals, see [10]; a discussion of the strengths and weaknesses of its designs can be found at [11]. The GEOM system should also make it easier for FreeBSD to support disk encryption schemes used by other systems, such as NetBSD's CGD [12].

Summary

In this article we've covered some older features (file flags and secure levels) and some newer ones (the MAC and GEOM frameworks). These features basically provide a richer set of choices in the design of a secure system, providing options that aren't available with the plain UNIX security model. As usual, the tricky bit is the care required to use these features correctly.

More features are in the pipeline. In particular, there should be support for event auditing and OpenBSM available in the FreeBSD 6 family of releases. It will also be interesting to see what interesting applications of the MAC and GEOM frameworks people can come up with. Companies and individuals are already developing third-party modules for use in their own environments or in FreeBSD-based products.

REFERENCES

[1] User Mode Linux: http://user-mode-linux.sourceforge.net/.

[2] Xen virtual machines: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

[3] The original jail paper is available at http://docs.freebsd.org/44doc/. Many tutorials are also available; see, e.g., http://www.freebsddiary.org/jail.php.

[4] FreeBSD Handbook, "Security," File System Access Control Lists section, http://www.freebsd.org/handbook/; see a tutorial article at http://ezine.daemonnews.org/200310/acl.html.

[5] FreeBSD also provides the sysctls security.bsd.hardlink_check_uid and security.bsd.hardlink_check_gid, which prevent users from making hard links unless their UID/GID matches that of the file. However, these sysctls are only enforced against locally running processes, and so hard links can still be made over NFS.

[6] "Musings," *;login:*, April 2005.

[7] TrustedBSD project: http://www.trustedbsd.org/.

[8] SELinux: http://www.nsa.gov/selinux/.
SEBSD port: http://www.trustedbsd.org/sebsd.html.

[9] FreeBSD Handbook, "Mandatory Access Control," http://www.freebsd.org/handbook/.

[10] gbde—GEOM-Based Disk Encryption: http://phk.freebsd.dk/pubs/bsdcon-03.gbde.paper.pdf.

[11] GBDE discussion on the Cryptography Mailing List threads:
http://www.mail-archive.com/cryptography@metzdowd.com/msg03636.html;
http://www.mail-archive.com/cryptography@metzdowd.com/msg03671.html.

[12] The CryptoGraphic Disk Driver: http://www.imrryr.org/~elric/cgd/.