A

MacOS Hardening Guide

The default configuration of MacOS remains quite permissive, but it's generally simple to enforce in MacOS a hardened level of security approaching (but still not quite as strong as) that of iOS.

Any security hardening may impact performance and usability across the system. It's highly recommended to try these recommendations out gradually on a test system before applying them fully in any production environment.

There are many approaches to hardening, and quite a few guides (such as CIS Apple OSX Security Benchmark), including automated tools (e.g. osx-config-check) exist. Most, however, go a little bit overboard in some recommendations (e.g. disabling Javascript in the browser which - while greatly improving security - propels the innocent user into the nostalgic WWW of the 1990s). The recommendations presented here try to greatly enhance the overall security posture, while minimizing user pain and suffering as much as possible. There are also some recommendations here which are overlooked by other common documents. I also tried to focus on "built-in" and "out of box" functions and settings, and stayed away from pointing out Anti-Virus, Anti-Malware, or in general third party products, save from pointing out they exist, where applicable.

Because approaches do differ, I originally did not think of adding such a guide to my book. Following a question by Sebastien Volpe, however, I realized it would be a good addition as somewhat of an informal "conclusion" to the book, and I am grateful to Sebas for it. I am likewise most thankful to Amit Serper (@0xAmit), who reviewed this document prior to its publication, and contributed some valuable insights!

This is the Appendix from "*OS Internals: Volume III - Security & Insecurity"

The other 400+ pages of the book are available through

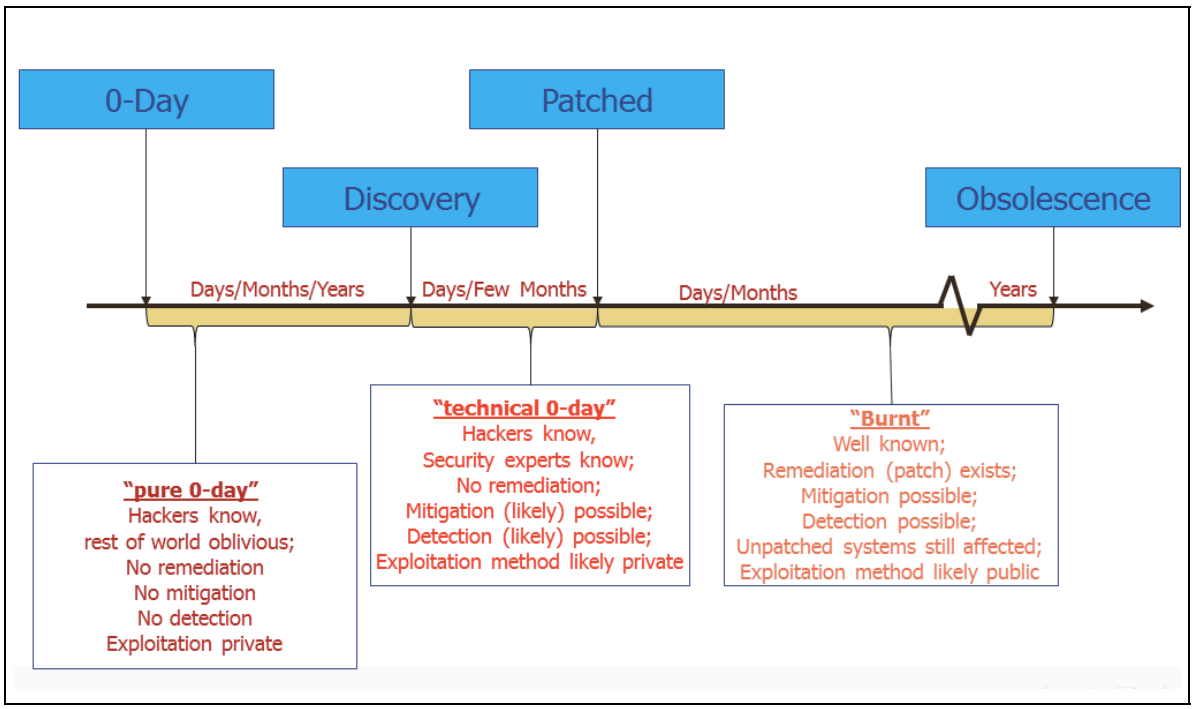
http://NewOSXBook.com/

Patching, Patching and Patching

If you haven't gone through the in-depth explanation of MacOS vulnerabilities in Chapter 12 of this volume, let me spoil the suspense and get you to its conclusion: Vulnerabilities in the core OS are inevitable, and you will automatically be affected by them. Though the security measures outlined throughout the rest of this appendix can certainly help, they will all fail in the face of a single kernel exploit.

There are no easy solutions to vulnerabilities, and no solutions at all to 0-day vulnerabilities. Once 0-days surface, however, they are (usually) promptly patched. But the patches are worthless unless they are applied. **ONLY** when a patch is applied can its vulnerability be dismissed as no longer a risk.

Figure 1: The vulnerability lifespan



As figure 1 demonstrates, however, dismissing a vulnerability as obsolete can only be achieved when every single system has either been patched or updated. While not an issue for the security-savvy household user, this can pose a significant challenge in corporate environments, which may number in the hundreds or even thousands of machines, and can be hard to keep up-to-date. As a result it is not uncommon to find old and antiquated OSes in these environemnts, which may harbor vulnerabilities readily exploitable with exploit-db.com style scripts.

Luckily, the software update functionality in MacOS can be performed at a non-interactive, command-line level, and can thus be automated. Using the `softwareupdate(8)` tool with the `-i (--install)` and `-r (--recommended)` will download and install available patches without any user interaction - and therefore no ability for the user to intervene with the process, either.

Note, that it is not uncommon for Apple to just "give up" and cease supporting older versions of MacOS, even when known vulnerabilities exist. A good example is MacOS 10.10, whose latest (and final) version - 10.10.5 is still very much vulnerable to the muymacho and tpwn exploits, described earlier in this book. Though there are sometimes point security updates, Apple generally assumes that its audience will update to MacOS 10.11 (and now 12), and therefore does not bother with supplying even a simple patch for vulnerabilities.

Logging and Auditing

Carefully monitoring the logging and auditing subsystems of MacOS will often provide early warning of a hack before it is successful: Few hack attempts "get it right" on the first try, but it is only through observing the trails of failed attempts or abnormal activitiy that they can be detected.

syslog/asl

Apple's logging infrastructure (until MacOS 10.12) follows that of the standard UN*X `syslog(1)`, with some Apple specific extensions (referred to as "ASL", for Apple System Log). The extensions enable a greater degree of verbosity and filtering, while at the same time maintaining compatibility with the traditional mechanisms and third party servers.

One of the most powerful features of `syslog` is the ability to log to a remote host. This is carried over UDP 514, and can be configured in the /etc/syslogd.conf using a "@" as the marker (along with IP or hostname), as well as designating a `loghost` entry in DNS or /etc/hosts. The remote host must be running `syslogd` with networking explicitly enabled, since the default logging in Mac OS X is over a local (UNIX domain) and not network socket.

Remote logging offers two distinct advantages:

- **Centralized logging:** to a single server greatly simplifies the task of log monitoring, which can be automated using third party tools or with the standard UN*X utilities of `grep(1)`, `awk(1)`, `perl(1)` and other filters.
- **Write-only access:** If the loghost is not otherwise accessible over the network (e.g. no SSH, remote login, or other facilities), records can be added to the log, but not read or removed. This greatly increases security, because an attacker cannot harvest the logs to glean any configuration or sensitive information. Furthermore, this makes the log far more reliable, as an attacker cannot erase or modify any previously logged entries. Note, that an attacker can still flood the log with bogus records, but cannot undo any previous records.

log (MacOS 12+)

MacOS 12 deprecates syslog/asl in favor of the new `os_log` subsystem. This is a more powerful infrastructure which abandons the traditional text-files based logging in favor of mostly in-memory logging, and a datastore. It is expected that, over time, ASL will be entirely deprecated, possibly transparently if Apple decides to implement `syslog(3)` and `asl(3)` APIs over `os_log`

The `os_log` subsystem does not (at the time of writing) support network logging. It is a fairly simple matter, however, to run the `log(1)` client command and pipe its output to `nc(1)` to another, remote host:

```
log --stream | nc remote.log.host ###
```

Consider, however, that this is only the basic method for redirecting output, and a more resilient solution - which considers network failure events and scalability - should be adopted.

Enabling Auditing

MacOS's strongest security feature is indubitably auditing. While not proactive, it is still nonetheless capable of tracking security-sensitive operations and events in real time. Unlike the aforementioned logging subsystems, which require voluntary record generation by applications, audit records are generated by the operating system itself.

Despite logging directly from kernelspace to the audit logs, a major drawback of auditing is nonetheless its local nature. If a system is compromised its audit logs cannot be considered trustworthy. Fortunately, a little bit of creativity with UN*X shell scripting can redirect the audit log directly to a centralized server. The same `nc(1)` trick which can be applied on `log(1)` can also be applied on the `/dev/auditpipe`. In fact, logging on the audit pipe may be conducted with or without `praudit(1)`, enabling a binary stream (which is more compact in nature), rather than first translating it to human readable format. Here, too, a more resilient wrapper (in a shell script or other) is recommended.

The `supraudit` tool, available from the book's website (but requiring a license for corporate use) has built-in networking functionality. It also has the ability to set different filters on the `/dev/auditpipe` than the default policy, which allows for faster auditing with less effect on system performance as less audit records get flushed to the local disk (which increases I/O considerably).

The exact specification of the audit policy is outside the scope of this recommendation, as it may greatly depend on the organizational policy. As a rule of thumb, however, remember that auditing is inversely proportional to performance. At a minimum, logging the classes of `lo` (login/logout), `aa` (authentication/authorization), `ex` (execution) and `pc` (process lifetime) are recommended. For high-security installations, wherein auditing is critical, consider using the `ahlt` flag, which stops the system on audit failure.

User-level Security

Login banner

In addition to the usual /etc/motd, the graphic `loginWindow` can be set to display a notice. This won't determine any hackers, of course, but does serve as a warning on usage policies, and might be required legally in some locations.

```
defaults write /Library/Preferences/com.apple.loginwindow LoginwindowText “lorem ipsum..."
```

Password Hints

It's possible to fine-tune the number of failed password attempts before any password hints are displayed. This can be used to disable password hints altogether.

```
defaults write /Library/Preferences/com.apple.loginwindow RetriesUntilHint -integer ###
```

Login/Logout hooks

A relatively little known, but highly useful mechanism in MacOS is that of Login/Logout hooks. These are paths to binaries (or, more often, scripts), that run as part of the login and logout processes.

```
defaults write com.apple.loginwindow LoginHook /path/to/execute
defaults write com.apple.loginwindow LogoutHook /path/to/execute
```

Using login hooks it is possible to run a program that will, for example, monitor the user's login and record or alert the administrator in real time. Likewise, a logout hook can be used to ensure removal of temporary files (for example, cleaning out any files in the Trash using `srm`).

Note, that Login/Logout hooks are also potential hiding places for malware seeking persistence, and should be checked periodically (prefereably, in every user session) for unauthorized modifications.

Password Policies

MacOS systems in the enterprise will automatically synchronize their password policies in most cases with that of their controllers, by virtue of recognizing and allowing a centralized authentication server. The Mac OS X Server App (or, for earlier systems, Workgroup Manager could be used to configure such systems as well.

From the command line, the `pwpolicy(8)` tool is available to set all aspects of the password policy. The tool (mentioned in Chapter 1) is properly documented in its manual page. The actual recommended policy will vary.

Screen Saver locking

Most users step away from their computer without bothering to lock the screen, and an unattended session poses significant security risks as passerbys may potentially use even a short window to steal information or run commands. It's therefore recommended to set the screen saver options - either through the System Preferences.app or through the `defaults(1)` command:

```
defaults write com.apple.screensaver askForPassword -int 1
defaults write com.apple.screensaver askForPasswordDelay -int 0
```

disable su

The venerable `su(1)` utility is not as security sensitive or featured as the more modern `sudo(8)`, and should therefore be disabled. Disablement is as simple as a `chmod u-s` operation on it, but it is recommended to further add a line containing `pam_deny.so` (as shown in the Experiment "Tinkering with PAM configuration files" from Chapter I).

Harden sudo

There is no argument that `sudo(8)` is better than the basic `su(1)`, but its default configuration can and should be hardened. For this, the following steps are suggested:

In a corporate environment, only selected commands should be enabled for sudo. These can include safer commands such as `shutdown(8)` and `reboot(8)`. Under no circumstances should any shell be enabled, because this effectively bypasses any `sudo` command restrictions as the user can simply `sudo bash` or similar.

`sudo` has a little known function in `tty_tickets`, which binds the superuser permissions to the terminal (tty) on which `sudo` was last authenticated successfully. Without this function, two user sessions on different terminals will automatically be able to obtain superuser privileges if one of them authenticated.

Other useful features are `log_input` and `log_output`. These can be set globally or on a per-command basis as well (using `[NO]LOG_[INPUT/OUTPUT]`). `sudo` can even be configured to mail alerts on success or failure. Extensive documentation on these and other options can be found in `sudoers(5)`.

Periodically check start up and login items

Malware seeks to persist, and it is therefore a good idea to periodically check the user's start up and login items. The exact period may vary, but can be weekly, or tied to a login event using a hook. Any newfound items should be considered audit-worthy. This includes checking login/logout hooks, and even the jobs themselves, as malware may seek persistence by scheduling itself to execute via `cron` or `at`.

Use MDM (or parental controls) to manage user sessions and capabilities

As described in Chapter 6, MacOS has quite a few software restrictions mechanisms, which are carried out through the `mcxalr` binary and its related kernel extensions. Software restrictions are quite powerful, and allow white-listing only selected apps, or even reducing the workstation to a "kiosk" mode.

Commercial MDM solutions integrate with the built-in mechanisms, and allow even more functionality. In absence of such, the parental controls offer a surprising amount of restrictions on the locally logged on user - from setting login times, through white listing applications, websites (in Safari), emails (in default Mail.app), messages (in iMessages app) contacts, peripheral access and more.

The exact limitations are left up to the administrative policy, but either mechanism is highly recommended.

Data Protection

Periodically obtain cryptographic snapshots of important files

Important system files, such as /etc/hosts (which bypasses DNS), /etc/passwd and others are often modified by malware or hackers for a variety of purposes. Merely relying on filesizes or timestamps is insufficient, as it is fairly easy to tweak file sizes or `touch(1)` their timestamps.

Cryptographic hash functions such as MD5 and SHA-1, however, cannot be easily collided. It is therefore a good idea to run a periodic check on important system configuration files, and **certainly** on files deemed immutable (for example, the various binaries in /bin, /usr/bin, etc). The exact list of important files will vary (and will need to be updated on OS patches or updates). Any change detected in critical files should flag an immediate alarm.

Periodically backup user data

User data can easily be lost - either by accidential deletion, targeted sabotage, or ransomware. Backing up data periodically can mitigate the potential damage. Backup scripts can be configured manually, or using a third-party management tool. In the case of manual configuration, using `find / -mtime ... | xargs tar zcvf` can work well.

As of MacOS 10.12, Apple provides the new APFS filesystem, which has built-in support for filesystem snapshots. Though the feature is immature at the time of writing, it is expected to be usable and reliable by 10.12.2 or later(?) and should be used when available.

Backups over the network are best operated when a single, trusted backup server communicates with the machines on the network over password-less, public-key enabled SSH sessions

Cloud Saving

MacOS is becoming more and more intgrated with iCloud, which is generally a great convenience to normal users, but perceived as a potential data leak in some cases. Should iCloud saving need to be disabled, it can be done easily with the following command line:

```
defaults write NSGlobalDomain NSDocumentSaveNewDocumentsToCloud -bool false
```

Enable hibernation

The manual of `pmset(1)` describes the various options for `hibernatemode`, and in particular mode 25, which is settable only via the command line utility. hibernatemode = 25 is only settable via pmset. The system will store a copy of memory to persistent storage (the disk), and will remove power to memory. The system will restore from disk image. If you want "hiberna- tion" - slower sleeps, slower wakes, and better battery life, you should use this setting.

Secure Deletion

Files in an HFS (or APFS) volume) don't actually get deleted - their filesystem node is unlinked, but the data blocks are not purged or reclaimed until a low disk space condition. It is possible to force secure deletion - which overwrites the contents of the blocks, by using either `srm(1)` or `rm -P`. Note that this method is not suggested for use on Flash or Fusion Drives, since it greatly increases the number of P/E cycles and can shorten the lifespan of the storage.

Physical Security

Firmware password

Setting a firmware password prevents any changes to the boot configuration, such as trying to boot from an alternate boot device. This greatly increases the security of your Mac. Apple documents the process of setting the password, which must be performed through the recovery filesystem, in knowledgebase article HT204455.

Find my Mac

Many people are familiar with setting the "find my i-Device" feature, but this also applies to Macs as well. Though usually less useful on the stationary Mac Pros and iMacs, this feature is a boon for Macbooks. Not only does it automatically set a firmware password, it also enables remotely locking or wiping the Mac if it is stolen or misplaced.

FileVault 2

FileVault 2 should be enabled. This important feature has been available as of Mac OS 10.7, and is reliable, transparent, and highly effective. While having virtually no noticeable effect on the system when running, it renders its data inaccessible should the device be compromised or rebooted by an unauthorized individual.

Remove key during standby

The FileVault 2 key used remains as plaintext in physical memory when a Mac goes into standby. This could allow certain types of hardware-based attacks to determine the key by capturing and dumping the RAM image. Setting `pmset destroyfvkeyonstandby 1` will remove the key from memory, but at the cost of forcing the user to re-login when the computer emerges from standby.

Note that this setting has been known to interfere with normal computer standby, and powernap, and so these two settings should also be disabled (using `pmset -a [standby/powernap] 0`.

Disabling USB, BlueTooth, and other peripherals

Floppy disks are long gone relics of a distant past, as are CD-ROMs. USB, however, remains widely used, and a potential entry vector for malware. Deployments of MacOS in high security environments may wish to disable USB mass storage devices. This can be done by removing the IOUSBMassStorageClass.kext from /System/Library/Extensions, remembering to `touch(1)` the directory so as to rebuild the kernelcache. It is also possible to use a similar method to disable USB together, though that is often impractical because of USB keyboards. A similar method on IOBlueToothFamily will remove BlueTooth functionality.

Do note, that these measures, though reversible (by replacing the removed kernel extensions and rebuilding the cache), might be a bit extreme. It's possible to apply them temporary by merely using `kextunload` (as root) on the extensions, but to leave them in-place. A better way still is to restrict functionality to specific devices. This can be done by installing a third party kernel extension which will be first in line to intercept device notifications - much in the same way that VMWare Fusion and other virtualization programs usurp control of the USB. Such a kext would define an `IOKitPersonalities` key similar to the following:

```
<key>IOKitPersonalities</key>
	<dict>
		<key>UsbDevice</key>
		<dict>
			<key>CFBundleIdentifier</key>
			<string>....  </string>
			<key>IOClass</key>
			<string>....  </string>
			<key>IOProviderClass</key>
			<string>IOUSBDevice</string>
			<key>idProduct</key>
			<string>*</string>
			<key>idVendor</key>
			<string>*</string>
			<key>bcdDevice</key>
			<string>*</string>
			<key>IOProbeScore</key>
			<integer>9005</integer>
			<key>IOUSBProbeScore</key>
			<integer>4000</integer>
		</dict>
		<key>UsbInterface</key>
		<dict>
			<key>CFBundleIdentifier</key>
			<string>....  </string>
			<key>IOClass</key>
			<string>....  </string>
			<key>IOProviderClass</key>
			<string>IOUSBInterface</string>
			<key>idProduct</key>
			<string>*</string>
			<key>idVendor</key>
			<string>*</string>
			<key>bcdDevice</key>
			<string>*</string>
			<key>bConfigurationValue</key>
			<string>*</string>
			<key>bInterfaceNumber</key>
			<string>*</string>
			<key>IOProbeScore</key>
			<integer>9005</integer>
			<key>IOUSBProbeScore</key>
			<integer>6000</integer>
		</dict>
	</dict>
```

The matching dictionaries shown above would match any USB device, but they can easily be made specific so as to blacklist or (preferably) whitelist known device classes. Creating a simple kext to handle hardware devices by simply ignoring them is discussed in Volume II.

Application Level Security

Enabling SIP (MacOS 11+)

System Integrity Protection (SIP) is hands-down the most significant security mechanism introduced into MacOS since the sandbox. While not a panacea, it definitely hardens the attack surface of the operating system by introducing another trust boundary, in between root and the kernel.

SIP is on by default as of MacOS 10.11, and there is no real reasoning (aside from on development machines) to try to disable it. Using `csrutil` it is possibly to perform selective disablement of some of its protections, although those should be done on a case by case basis. The protections against unsigned kexts must remain in place, and kernel extension developers should be encouraged to test only signed extensions.

Enforcing code signing

As discussed in Chapter 5, Code Signing in MacOS can be made just as stringent as in *OS - but it requires the setting of `sysctl(8)` variables. The list of recommended `sysctl` variables can be seen in the following output:

```
vm.cs_force_kill: 1    # Kill process if invalidated
vm.cs_force_hard: 1    # Fail operation if invalidated
vm.cs_all_vnodes: 1    # Apply on all Vnodes
vm.cs_enforcement: 1   # Globally apply code signing enforcement
```

Block-Block/Flock-Flock/etc-etc

Many third party tools, both open source and commercial, attempt to provide real-time monitoring of applications (processes) on the local system. Some are passive, collecting information, while others go to the level of inspecting specific APIs, possibly blocking system calls and mach traps. The same holds for Anti-Virus/Anti-Malware (though most are reactive, rather than proactive). This recommendation steers clear of suggesting this or another tool, though the common ones are those mentioned.

Sandboxing

As discussed in Chapter 8 of this book, the Sandbox is a truly powerful containerization mechanism, built-in to all of Apple's OSes. Using the `sandbox-exec(1)` tool, you can force unknown or untrusted binaries to run in a containerized environment. You can further using the tracing functionality of the sandbox to get a clear report on every single operation (at the system call level) that the binaries perform. Note, that applying a restricting profile on binaries can and often does result in breaking functionality, as many are poorly coded without any restrictions in mind.

Virtualization

Computing power has grown tremendously over the past generation, and for most users, the capabilities of the CPU far exceeds the demands of computation. Virtualization can be harnessed in quite a few ways to enable security:

- Quarantining downloads and attachments: A virtual machine provides a containerized environment, wherein even if malware runs amuck, it can do little damage. Suspect programs and downloads can be opened in this environment, which can quickly be discarded or paused in case of infection.
- Snapshots and clones: Allow for quick setup and deployment of known, secure configurations. In case of any compromise, it is a simple matter to revert to a trusted configuration at the click of a button.

Network Security

Application Layer Firewall

Enabling the Firewall from the "Security & Privacy" launches /usr/libexec/ApplicationFirewall/socketfilterfw (via `launchd(8)`'s com.apple.alf.agent.plist `LaunchDaemon` propertyl list. `launchd` redirects the daemon's standard error and output to `/var/log/alf.log`, but by default the logging is performed to /var/log/appfirewall.log

Apple documents the Application Firewall basics in a knowledgebase article, naturally not disclosing anything as per its implementation. More detail on the operation of this powerful mechanism can be found in Volume II. In a nutshell, however, the configuration is stored in /Library/Preferences/com.apple.alf.plist (and may be accessed via the `defaults(1)` command). The important keys to note are:

- **`allowsignedenabled`**: Allows an automatic exemption to code-signed applications
- **`applications`**: An array of application identifiers, usually empty
- **`exceptions`**: An array of dictionary objects, one for each exception. Applications are identified by their `path`, and the exception also holds a `state` integer.
- **`explicitauths`**: An array of dictionary objects, each one holding an application (bundle identifier). These are used for interpreters or execution environments, like Python, Ruby, a2p, Java, php, nc, and ksh.
- **`firewall`**: A dict of keyed services, which each key containing the `proc`ess name and the integer `state`.
- **`firewallunload`**: An integer specifying if firewall is unloaded. Must be zero
- **`globalstate`**: An integer specifying state. Must be two
- **`loggingenabled`**: An integer specifying if alf.log is used - must be one.
- **`loggingoption`**: An integer specifying logging flags. May be zero.
- **`stealthenabled`**: An integer specifying if host will respond to ICMP (e.g. `ping(8)`). Should be one.

pf

In addition to ALF, MacOS also provides a packet level filter called `pf`. This is an in-kernel facility, but may be controlled from user mode through the /dev/pf[m] character devices, and the `pfctl(8)` command, as well as files in /etc, notably pf.conf (5). This is the main config file, which loads the ruleset on startup, and may redirect/include other files (usually, in /etc/pf.anchors) as well. `pf` is borrowed from BSD, and is similar in some respects to Linux's netfilter mechanism (the foundation of its `iptables` firewall).

The operation of `pf` is explained in more detail in Volume II; Note, that this level of firewalling filters on a per-packet basis, rather than ALF's, which "sees" much higher into the OSI stack at the application level. This brings advantages (e.g. packet-level dropping, NAT, masquerading and more), and disadvantages (inability to reassemble packets).

Eliminate all unnecessary services

All unnecessary services should be disabled. In particular, Remote Apple Events and Internet Sharing must be disabled, as they are insecure. This especially holds true for Apple Events (as convenient as the service is) as it may lead to serious compromise.

Secure all necessary services

Where services are deemed required, for backup or network access, consider replacing them with secure variants. Mac OS no longer allows plain telnet and uses ssh, but even ssh's security can be further bolstered by using PKI in addition to or instead of passwords (the secure recommendation suggests using both). Likewise, FTP can be replaced by SFTP, and virtually any insecure legacy protocol - POP, IMAP - can either be instructed to use SSL, or be tunneled over SSH or SSL

Consider decoys for unused services

Network attackers, particularly automated worms, scan the network trying to fingerprint remote hosts or automatically take advantage of known vulnerabilities in order to propagate. Setting up a decoy - in the form of an open `nc -l` - can trap these attacks, immediately giving early warning of an impending attack.

Little Snitch/Big Brother/lsock

Third party tools, such as "little snitch" and "big brother" run in the background and constantly monitor network activity, including a powerful GUI. The `lsock` tool (available as open source from the book's companion website) and Apple's own `nettop(1)` achieve similar (but to some extents, lesser) functionality by running in a command line. `nettop(1)` and `lsock` use curses for full terminal screen capabilities, but the latter can be instructed to work in filter-friendly form with the `nc` (no-curses) command line.

Running network monitoring tools in the background is an efficient way of detecting both outbound and inbound connections, which may indicate open or covert channels by means of which malware may be communicating with remote servers. Note, that such tools are best when their output is supplemented by logs from the segment's firewall or router.

For the paranoid

Recompiling the kernel

The XNU kernel (for MacOS) remains in the opensource domain, and it is possible to compile it - from the sources at OpenSource.Apple.com - and create a kernel that is identical to the distribution kernel. It is also possible to change compilation settings, notably adding much debug functionality, and also securing the kernel.

A secure kernel is created by simply setting the `SECURE_KERNEL` `#define` to 1 (the way it is on iOS). This affects the kernel in several ways:

- **Core dumps will no longer be created:** bsd/kern/kern_exec.c's `do_coredump` is set to 0, and disables cores system-wide, rather than the non-secure default setting, which is only for s[u/g]id binaries. Additionally, the corresponding `sysctl(8)`s (from bsd/kern/kern_sysctl.c) are disabled.
- **Code signing is significantly hardened:** The `cs_enforcement_enable` and `cs_library_val_enable` variables (in bsd/kern/kern_cs.c) are toggled to 1, and defined as `const`, so they cannot be changed by `sysctl` calls. Additionally, the `cs_enforcement_disable` boot-argument is no longer honored. Similarly, in kernels before 37xx the `vm.cs_validation` `sysctl(8)` (from bsd/kern/ubc_subr.c is removed.
- **User-mode ASLR is mandatory:** As the `POSIX_SPAWN_DISABLE_ASLR` option is no longer honored in bsd/kern/kern_exec.c.
- **The NX sysctl(8) is disabled:** meaning that by default, data segments are marked not-executable and this cannot be changed.
- **The `vm.allow_[data/stack]_exec` are both disabled** (in bsd/vm/vm_unix.c, but removed in MacOS 12 anyway)
- **The `kern.secure_kernel` and `kern.securelevel` `sysctl(8)`s are set:** The former is set to 1 (true) and the latter to the "security level". This is well defined in bsd/sys/system.h as follows:

```
/*
 * The `securelevel' variable controls the security level of the system.
 * It can only be decreased by process 1 (/sbin/init).
 *
 * Security levels are as follows:
 *   -1 permannently insecure mode - always run system in level 0 mode.
 *    0 insecure mode - immutable and append-only flags make be turned off.
 *      All devices may be read or written subject to permission modes.
 *    1 secure mode - immutable and append-only flags may not be changed;
 *      raw disks of mounted filesystems, /dev/mem, and /dev/kmem are
 *      read-only.
 *    2 highly secure mode - same as (1) plus raw disks are always
 *      read-only whether mounted or not. This level precludes tampering
 *      with filesystems by unmounting them, but also inhibits running
 *      newfs while the system is secured.
 *
 * In normal operation, the system runs in level 0 mode while single user
 * and in level 1 mode while multiuser. If level 2 mode is desired while
 * running multiuser, it can be set in the multiuser startup script
 * (/etc/rc.local) using sysctl(1). If it is desired to run the system
 * in level 0 mode while multiuser, initialize the variable securelevel
 * in /sys/kern/kern_sysctl.c to -1. Note that it is NOT initialized to
 * zero as that would allow the vmunix binary to be patched to -1.
 * Without initialization, securelevel loads in the BSS area which only
 * comes into existence when the kernel is loaded and hence cannot be
 * patched by a stalking hacker.
 */
```

With a bit of daring, XNU's functionality can be dramatically altered by editing the source code. But one example is that MacOS can be made to adopt the *OS-like handling of kexts, by disabling the loading of kexts from user-mode by kext_request (host priv #425). Known safe kernel extensions can be a priori linked into the kernel cache, and all run-time linking functionality disabled. Bear in mind, however, that modifying the kernel sources effectively creates a branch from Apple's mainline, which may be challenging to keep up with, given newer OS releases.

XNU's open sources can compile quite cleanly when following the guidelines outlined in the README file. One glaring omission from the guidelines is the list of dependencies. As discussed in Volume II of this work, this boils down to obtaining the following packages:

- Cxxfilt: The real name of this package is C++filt, but + is an illegal character in DOS filenames.
- Dtrace: Current version: x.x. Required for CTFMerge.
- Kext-tools: Current version:
- bootstrap_cmds: Current version: xxx. Required for `relpath` and other commands

This can be checked in the user's ~/Library/Preferences/com.apple.Terminal.plist under the `SecureKeyboardEntry` setting as a boolean `true`.

Demote setuid binaries

The classic model of UN*X `setuid(2)` is an anathema to security. As shown in Output xx-suid, MacOS has gradually reduced the number of binaries to about a dozen presently - but even those aren't necessarily required in every day use. The binaries can be maintained, but should be considered for demotion (by `chmod u-s`). As shown in Chapter 12, vulnerabilities in `dyld` yielded instant root because of executing under setuid.

Demoting setuid binaries would close an important vector for local privilege escalation, but may or may not impact usability. For example, if the `at(1)` facility is not in use, both /usr/bin/at and /usr/bin/atq can be removed with no ill-effects. Some binaries, however, notably `sudo(1)` and `security_authtrampoline(8)` cannot be demoted without entirely breaking their functionality. In the case of the former, this isn't an issue if the facility is not in use. In the case of the latter, however, the binary is used internally in the system, and cannot be demoted.

Remove unnecessary binaries

The average users don't go so far as to use the terminal and shell environments. Even when they do, their command set is fairly limited. At the administrator's descretion, certain binaries - even entire apps - can be removed. This should be exercised with caution. Removing/disallowing the `Terminal.app` can go a long way.

Consider patching/recompiling specific binaries

There are certain binaries in the system that, while innocuous in and of themselves, may be useful in the process of a hack.

For example, consider an attacker who has somehow managed to obtain shell access.

> A versatile hacker can probably find a way around these restrictions (such as in the case of `chmod(1)` shown here, because inevitably it is a system call. This suggestion will nonetheless prove useful in the case of automated hacks - especially of the kind where a shell script is injected and executed, leading to the download of a binary - and relying on `chmod(1)` to execute it