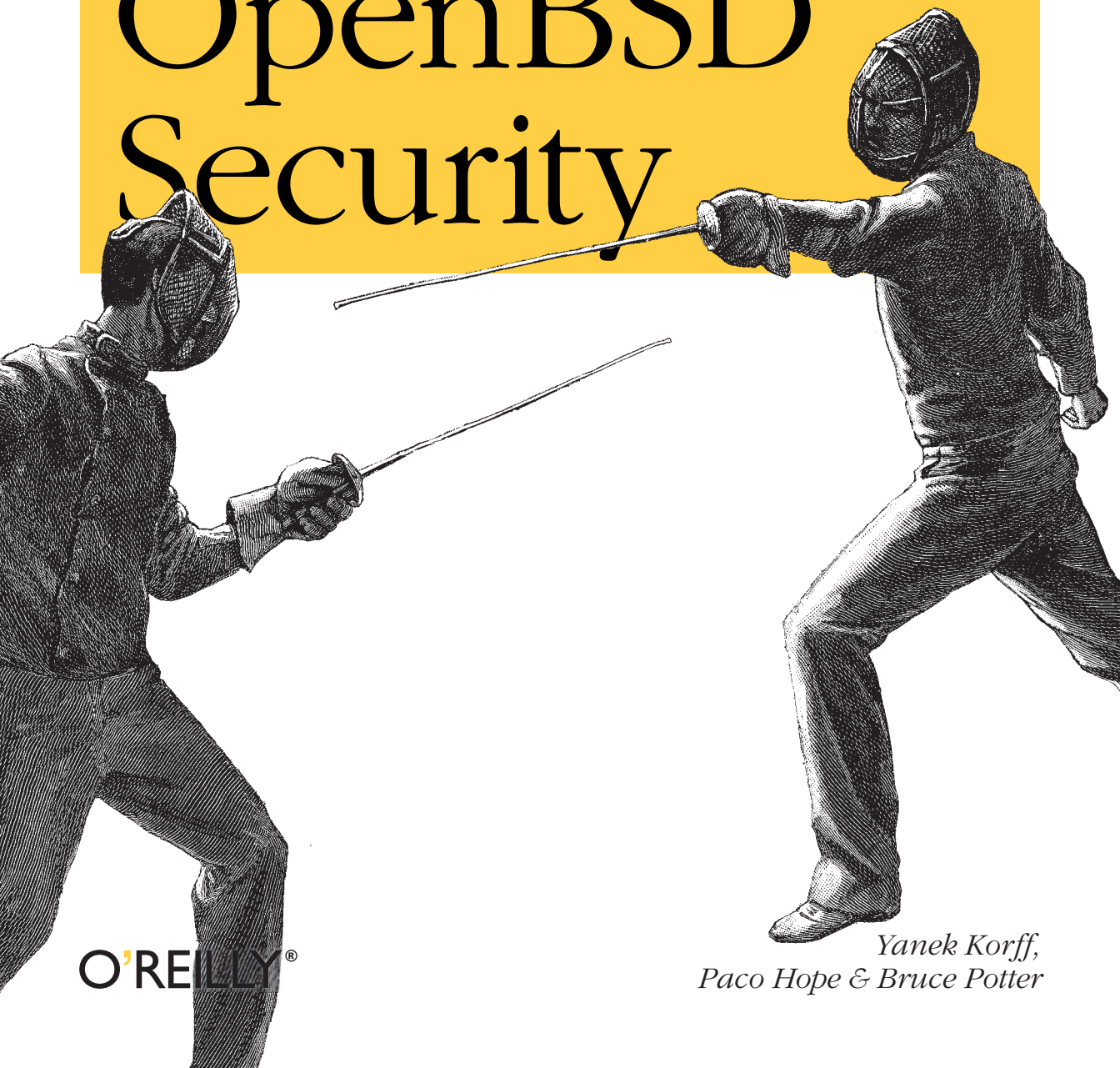


*Building, Securing, and Maintaining BSD Systems*

*Mastering*

# FreeBSD *and* OpenBSD Security



O'REILLY®

*Yanek Korff,  
Paco Hope & Bruce Potter*

# The Big Picture

*First we crack the shell, then we crack the nuts inside.*

—Rumble

*The Transformers: The Movie*

Security is hard. We have all heard this phrase as a rationale for insecure systems and poor administrative practices. What’s worse, administrators seem to have different ideas about what “security” entails. There are two common approaches to securing systems: some view security as a destination while others see it as a journey.

Those who see security as a destination tend to characterize system security in terms of black and white; either a system is secure or it is not. This implies that you can attain security. You can arrive at the end of a journey and you’ll somehow be secure; you win. One problem with this viewpoint is determining where “there” is. How do you know when you’ve arrived? Furthermore, how do you stay there? As your system changes, are you still at your secure goal? Did you move away from it, or were you not there to begin with? As you can probably tell, this is not our philosophy.

Instead of being a destination, we think security is best described as a journey—a product of ongoing risk management. Rather than trying to make your system impregnable, you continually evaluate your exposure to risks and keep the system as secure as you need it to be. An appropriate level of security is achieved when the risks facing a system balance against the level of effort spent mitigating those risks. No one buys a \$5,000 vault to safeguard a pair of fuzzy slippers. You judge the value of what you’re protecting against the kinds of threats it faces and the likelihood those threats will succeed, and then you apply appropriate safeguards. This is a much more practical way to view modern day information security.

When following a risk mitigation process, you will periodically pass up the opportunity to enable certain security mechanisms, even though you’re capable of doing so. The additional effort may not be warranted given the level of risk your organization faces. You will eventually reach a point of diminishing returns where you simply accept some risks because they are too costly to mitigate relative to the likelihood of

the threat or the actual damage that would occur. Sure, it may be fun to use encrypted filesystems, store all OS data on a CD-ROM, and deploy every other countermeasure you can think of, but do you really need to?

We define security in the context of risk. Risk is present as long as the system exists, and risks are constantly changing, so security cannot be a destination; it must be an ongoing process. “Doing security,” then, is an iterative process of identifying and responding to risks. This is the philosophy that we encourage you to take in securing your infrastructure.

As you’ll see in the rest of this book, FreeBSD and OpenBSD are robust operating systems that offer myriad ways to maintain secure systems. Throughout the book we provide security-minded walkthroughs of software installation, configuration, and maintenance. Along the way you’ll notice that we seem to point out more security-related configuration options than you care to implement. Just because we explore options doesn’t mean that you should implement them. Come at it from the perspective of managing risk and you’ll maximize the cost-benefit of “doing security.”

Before we get ahead of ourselves, however, we need to cover a few concepts and principles. In this chapter, we define system security, specifically for OpenBSD and FreeBSD systems, but also more generally. We look at a variety of attacks so that you, as an administrator, will have some perspective on what you’re trying to defend against. We’ll look at risk response and describe how exactly you can go about securing your FreeBSD and OpenBSD systems.

## What Is System Security?

Security professionals break the term security into three parts: confidentiality, integrity, and availability. This “CIA Triad” is a set of security requirements; if you’re not taking into account all three of these concerns, you’re not working towards providing security. We offer a lot of recommendations in this book that should help you work towards building secure systems, but we don’t tell you how these recommendations fit in with the CIA Triad. That’s not what this book is about, and it would detract from the real message. Nevertheless, as you’re looking at building encrypted tunnels for transferring files, jailing applications, and so on, think about what part of the Triad you’re focusing on. Make sure you’ve addressed all three parts before your project is done.

Whether we’re talking about physical security, information security, network security, or system security, the CIA Triad applies. The question is, exactly how does it apply to system security?

### Confidentiality

Confidentiality is all about determining the appropriate level of access to information. Confidentiality is often implemented at the most basic level on FreeBSD and

OpenBSD systems by traditional Unix permissions. There are a variety of files scattered across the filesystem that are readable only by the root user. Most notable, perhaps, is */etc/master.passwd*, which contains hashes for users' passwords. The vast majority of files are readable by everyone, however. Even system configuration files like */etc/resolv.conf*, */etc/hosts*, and so on are world readable. Is this wrong? Not necessarily. Again, confidentiality isn't about having to protect data from prying eyes; it's about classifying data and making sure that information deemed sensitive in some way is protected appropriately.

Filesystem level protections are of course only one facet of confidentiality. Data may be exposed through some service designed to serve information like DNS, or a web server. In these cases, the method you employ to protect data won't necessarily be filesystem permissions; perhaps you'll control what systems are allowed to query your DNS server, or which web-authenticated users are permitted to view a certain document tree. When you need to protect data from eavesdropping as it moves across a network, you'll probably use encryption. When implemented appropriately, it helps ensure that only the intended recipient can read the transmitted data.

## Integrity

Data integrity relates to trust. If you cannot guarantee the integrity of some information on your system, you can't trust it. Consequently, resources for which integrity is an important issue need to be identified and appropriately protected against modification.

Confidentiality may not have been an issue for your */etc/resolv.conf* file. Allowing users to see what resolvers your system depends on is okay. Allowing users to modify the list of resolvers is not! Your system's resolvers are a data source. When you access a server providing anonymous CVS access to your OpenBSD sources, your system will ask one of the servers listed in */etc/resolv.conf* to find the IP address for the name you provided. If you can't guarantee the integrity of the data in this file, you can't trust the IP address you get from the resolver. As a consequence, you can't trust the sources you download either.

Like confidentiality, the filesystem permissions model helps enforce data integrity. Unfortunately file permissions aren't enough by themselves. If someone has broken through your filesystem protections somehow, you won't know that your data has been tampered with. That is, not without good auditing. Moreover, you won't be able to restore a known good configuration without data backups.

Data integrity is also an issue during network transfers. How can you be sure that the information has not been modified in transit? The BSD operating systems will provide "signatures," which uniquely identify file distributions. When you download a package or source tarball or install a port, you can check your local files against the remote signatures. If it's a match, your file has not been modified while in transit.

## Availability

Often overlooked by administrators, availability is the last key component of security. Protecting your systems from information disclosure and tampering is important but not sufficient. If a user on your system “accidentally” copies his 120GB MP3 music collection to your file server and you run out of disk space on /, your system will suddenly cease being useful. If you had separated your home directories into their own partition, and additionally configured filesystem quotas, this would not have been a problem.

Availability does not only pertain to services, it can also apply to data, though most examples you might immediately think of are really data integrity issues. What would happen if a virus infected your workstation and destroyed the only private keys that decrypt vital data? You probably have a backup of that encrypted file and have otherwise taken care of integrity issues, but suddenly that data may as well have been deleted. It’s no longer available for use.

System availability can be one of the most difficult areas of providing system security. The number of ways (both physical and electronic) an attacker can make your server unavailable is staggering.

## Summary

So, after all this, what is system security? For our purposes, providing “system security” is about responding to risks that threaten the confidentiality or integrity of data that resides on or passes through our systems, and working to guarantee the availability of the services and data. If you made it through this section, congratulations. This is pretty dry stuff, but it’s important. We won’t explicitly talk about the elements of the CIA Triad in this book, but we encourage you to keep these principles in mind when working on protecting your systems.

## Identifying Risks

We are going to consider a variety of risks that we might face and then talk about how we can respond to them. In this chapter, we lay the groundwork for many general risks. In each of the application-specific chapters, we will identify application-specific risks that make concrete examples from the concepts here. We are not going to give you advice on how to identify lightning strikes as a risk and use FreeBSD or OpenBSD features to help protect your data against them. Instead, we think mostly about hackers and other malicious adversaries and how you identify and assess the damage they can wreak on your infrastructure.

## Attacks

An attack against a system is an intentional attempt to bypass system security controls or organizational policies to affect the operation of the system (active attack) or gain access to information (passive attack). Attacks can be classified into insider attacks in which someone from within an organization who is authorized to access a system uses it in an unauthorized way, or outsider attacks, which originate outside of the organization's security perimeter, perhaps on the Internet at large.

Attacks motivate system administrators, supervisors, and organizational leaders into caring about security, but where's the damage in an attack? What kind of major assault is required to wreak the kind of havoc required to get all these people worked up? Despite our definition of attack, we still do not have enough information to determine for a given system, where the risks lie.

In order for active and passive attacks to succeed, something must be at fault. Attacks necessarily leverage fundamental behavioral problems in software, improper configuration and use of software, or both. In this chapter, we examine these classes of attacks including the special-case denial of service (DoS) attack.



Remember that while attacks are necessarily intentional, system disruptions due to software failure or misconfiguration probably are not. In either case, the end result is the same: system resources are abused. In the latter case, however, this does not come as a consequence of someone's malevolence. Fortunately, defending against attacks tends to help defend against unintentional disruptions, too.

At a slightly higher level, it is also important to distinguish between attacks that originate locally on the system and those that may be carried out over a network. Keep in mind that, although this distinction is important when thinking about mitigating risk, local attacks can quickly become network-exploitable if the system can be breached over the network in any way.

In order to provide system security for a single DNS server, a firewall, or a farm of web servers, it is important to understand what you are actually defending against. By understanding the classes of attacks you will come under, and the technology behind these attacks, you will be in a better position to maintain a secure configuration.

## Problems in Software

Attacks that exploit problems in software are the most common. Security forums such as Bugtraq and Full Disclosure focus on software problems. These are somewhat high volume lists and over the years have recorded thousands of vulnerabilities in software. The kinds of vulnerabilities recorded vary as widely as the associated impact.

In order to be able to understand a security advisory and react accordingly, it is imperative that you understand the common types of software-based vulnerabilities. This knowledge will prove useful when you build and configure your systems, resulting in a more well-thought-out deployment of your infrastructure.

## Buffer overflows

Probably the most widely publicized software security flaw is the buffer overflow. Buffer overflows are discussed on highly technical security lists, mass media, and as a result of this press coverage, by company executives. Buffer overflows are the result of (sometimes trivial) tactical coding errors that often could have been prevented. Exploits can be devastating. Buffer overflows are made even more dangerous when the software is reachable over the Internet. For instance, the Morris Worm (1988), used a buffer overflow in the `fingerd` utility as one of its exploit mechanisms. The Code Red worm, which infected hundreds of thousands of computers in 2001, exploited a buffer overflow in Microsoft's IIS web server to do its damage.

So what is a buffer, and how does it overflow? A buffer is a portion of memory set aside by a program to store data. In languages like C, the buffer generally has some predefined size. The C program will allocate the exact amount of memory required to accommodate this buffer. The developer must ensure that the data put into the buffer never exceeds the capacity of the buffer. If, through programming error and some unexpected input, too much data is inserted into the buffer, it will “overflow” overwriting adjacent memory locations that store other information.

The developer may feel that there is no pressing need to ensure the data being placed into a buffer will fit. After all, if it does not fit, the program will probably crash. The user did something stupid, and the user will pay for it. In Example 1-1, if the user inputs more than 20 characters in response to the `gets(3)` function, the buffer will not be able to store all the data.

*Example 1-1. A simple buffer that can overflow*

```
int main() {
    char buffer[20];
    gets (buffer);
    printf("%s\n", buffer);
}
```

The excess data will be written sequentially in memory past the end of the buffer into other parts of the processes' memory space. There are not necessarily security ramifications in this case—this example is purely academic.

Taken at face value, running past the end of a buffer seems like a pretty innocuous problem. The data ends up in other parts of the processes' memory space thereby corrupting the processes' memory and eventually crashing the program, right? Well,

attackers can use specially crafted data to intentionally overflow the buffer with instructions that ultimately will be executed by the operating system.

One popular attack strategy is to overwrite the return location of the function containing the exploitable buffer. Thus at the end of the function, instead of continuing to execute the program right after that function was called, the program's execution continues from some other point in memory. With a little knowledge and effort, an attacker can first inject malicious code (e.g., *shellcode*) which, if executed, would give the attacker access to the system. Then by setting the return location to point to their injected *shellcode*, the operating system will happily read the overwritten memory containing the new return location and execute the *shellcode* at the end of the function. This is only one example of how a buffer overflow can lead to some additional access on the system.

Fortunately the BSD operating systems have been extensively audited to protect against these, and similar, software flaws. OpenBSD includes built-in protections that prevent programs from writing beyond their allocated memory space thus preventing the execution of arbitrary code. This kind of protection can be very useful since not all of the third-party programs you install on your systems will have undergone the kind of scrutiny the operating system went through.

For non-programmers, this might seem complicated and difficult to digest. Computer science geeks will probably find this very familiar. Application security experts and others well versed in exploiting software vulnerabilities may already be wondering why we are not covering heap-based overflows in addition to the trivial example above. For the curious, additional resources are plentiful online. To succeed as a security-minded system administrator, understanding buffer overflows to this level should be sufficient.

## SQL injection

Although buffer overflow attacks get a great deal of attention, there are many other problems in software that give rise to vulnerabilities. With the increasing number of deployed web-based applications, attackers have become quite savvy at attacking web sites. Web-based attacks may target the web server software, a web-based application, or data accessed by the application. Some web-based attacks may even go right to the core operating system and attempt to exploit the host itself. The most common forms of web-based attacks, however, are those that lead to the manipulation of application data or escalation of application privileges.

Complex web applications often work directly with vast quantities of data. This is almost a guarantee when the web application is used by a financial institution or a data warehousing company but also common with smaller e-commerce sites, online forums, and so on. The only reasonable way to manage a large volume of data is by storing it in a database. When the application accesses the database, it must

communicate using some defined database query language, most often Structured Query Language (SQL).

SQL is a tremendously easy language to learn, and web developers often feel comfortable working with it after only a few days or weeks of exposure. Hence, there is an abundance of web applications that dynamically construct SQL statements as part of their response to user requests. In general, these statements are constructed based on what a site visitor has selected, checked, and/or typed into form fields on a web page.

Unfortunately, developers are often unaware of how their SQL query can be abused if the input from the user is not properly sanitized. An attacker can often inject her own SQL statements as part of the input to the web application in an effort to completely alter the dynamically generated SQL query sent to the database. These queries may result in data being changed in the database or may give the attacker the ability to view data she is not authorized to view.

There are, of course, ways to defend against SQL injection attacks from within web applications. One common approach is to parse every value provided by the user. Make sure it doesn't contain any undesirable characters like backticks, quotes, semicolons, and so on. Also ensure that the valid characters are appropriate for the value being returned. To get around the problem completely, developers may be able to use stored procedures and avoid dynamically creating SQL.

### **Other software problems**

We have only scratched the surface of application level vulnerabilities. What follows is a brief synopsis of additional software programming errors that have been known to lead to security problems.

#### *Format string error*

Format strings are used by some languages, notably C, to describe how data should be displayed (e.g., by *printf(3)* and derivatives). Unfortunately attackers can sometimes manipulate these format strings to write their own data to memory on the target host. This attack is like a buffer overflow in that it will allow an attacker to run his own code on the victim host. However, it is not actually overflowing a buffer so much as abusing a feature in the programming language.

#### *Race conditions*

Race conditions exist when there is contention for access to a particular resource. Race conditions are common software engineering problems, but they can also have security ramifications. For instance, assume an application wants to write password information to a file it creates. The application first checks to see if a file exists. Then the application creates the file and starts to write to it. An attacker can potentially create a file after the application checks for the file's existence but before the application creates the file. If the attacker does something interesting, like create a symbolic link from this protected file to a world

readable file in another directory, sensitive password information (in this example) would be disclosed. This specific kind of race condition is referred to as a time-of-check-to-time-of-use or *TOCTTOU* vulnerability.

### *Web-based attacks*

There are a staggering number of web-based attacks available in the hacker's arsenal. SQL injection, and other code injection attacks, comprise one class of web-based attack. Through poor web application design, weak authentication, and sloppy configuration, a variety of other attacks are possible. Cross-site scripting (XSS) attacks attempt to take advantage of dynamic sites by injecting malicious data. These attacks often target site visitors; web developers and site administrators may be left unaware. Handcrafted URLs can sometimes bypass authentication mechanisms. Malicious web spiders can crawl the Web, signing your friends and family up for every online mailing list available. These types of attacks are so common that there are many security companies that focus specifically on web-based security.

## **Protecting yourself**

As a system administrator or security engineer, you generally do not have control over the security within the software installed on your servers.



FreeBSD and OpenBSD are open source operating systems, as is most application software they run. Although you certainly have the option of modifying the source code of applications, you probably do not want to do that. System administrators rarely have time to properly develop and maintain software. As a result, managing custom code will usually lead to software maintenance nightmares.

You may be able to choose one server product over another: while building a mail relay, you might consider Sendmail, Postfix, and qmail. To help guide your decision, you might want to evaluate the particular software's security track record. Be wary, for much like mutual funds, past trends are not a reliable indicator of future performance. With custom, internally developed code you usually have no options at all; your company has developed custom software and your systems must run it.

At this point, you may be wondering if there is any way to mitigate the risks associated with vulnerabilities in software. As it turns out, there is. Being aware of vulnerabilities is a good first step. Subscribe to mailing lists that disclose software vulnerabilities in a timely fashion. Have a response plan in place and stay on top of patching. Watch your systems, especially your audit trails, and be aware when your systems are behaving unnaturally. Finally, be security-minded in your administration of systems. What this entails is described later in this chapter and embodied by the rest of this book.

## Denial of Service Attacks

DoS attacks are active—they seek to consume system resources and deny the availability of your systems to legitimate users. The root cause of a system or network being vulnerable to a DoS attack may be based on a software vulnerability, as a result of improper configuration and use, or both. DoS attacks can be devastating, and depending on how they are carried out, it can be very difficult to find the source. DoS attacks have a diverse list of possible targets.

### Target: physical

DoS attacks can occur at the physical layer. In an 802.11 wireless network, an attacker can flood the network by transmitting garbage in the same frequency band as the 802.11 radios. A simple 2.4 GHz cordless phone can render an 802.11 network unusable. With physical access to an organization's premises, cutting through an Ethernet cable can be equally devastating on the wired side.

### Target: network

At the data link and network layers, traffic saturation can interfere with legitimate communications. Flooding a network with illegitimate and constantly changing arp requests can place an extreme burden on networking devices and confuse hosts. Attempting to push a gigabit of data per second through a 100 Mbps pipe will effectively overrun any legitimate network traffic. Too much traffic is perhaps the quintessential example of a DoS attack.

Network-level DoS attacks can stop nearly all legitimate incoming and/or outgoing traffic thereby shutting down all services offered on that network. If a network-level DoS attack is conducted using spoofed source IP addresses, victims must often work with their ISPs to track down the source of the flood of data. Even still, this process is extremely time consuming and may not even be possible depending on the capabilities of the ISP. Worse yet, distributed denial of service (DDoS) attacks use different attacking hosts on different networks, making it nearly impossible to block all the sites participating in the attacks. DDoS attacks are difficult to defend against, especially at the host level.

### Target: application

Even at the application level, a DoS attack can be devastating. These DoS attacks generally use up some finite resource on a host such as CPU, memory, or disk I/O. An attacker may send several application requests to a single host in order to cause the application to consume an excessive amount of system resources. She may simply exploit a bug in code once that causes the application to spiral out of control or simply crash. Some services that fork daemons at every new connection may be subject to a DoS if tens or hundreds of thousands of connections are made within a short period of time.

These DoS situations may not always come as a result of an attack. Sometimes, an unexpected and sudden increase in the number of legitimate requests can render a service unusable.

### **Protecting yourself**

Physical and network-based DoS attacks are difficult to defend against and are out of the scope of host-based security. At the operating system level, you can do little to mitigate the risks associated with these kinds of attacks. Generally, some form of physical access controls help with physical attacks and specialized network devices like load balancers assist with network-based attacks.

IDS hosts may be used to help detect these kinds of attacks and automatically update firewall or router configurations to drop the traffic. Although this may protect one service, if the sheer volume of data is too much, blocking it at your firewall will not be useful. You will need to coordinate with your ISP.

Application-level attacks are something the security-minded system administrator can do something about. If you've been reading about the other forms of attacks, you might already have an idea of the kinds of mitigation techniques you can use against DoS attacks: secure architecture and build, controlled maintenance, and monitoring logs. Any mitigation techniques you have in place to protect from software vulnerabilities and avoid improper configuration and use will help make DoS attacks more difficult. Additional anomaly detection specific to identifying DoS attacks will go even farther.

## **Improper Configuration and Use**

Even if the software you are using is bulletproof, that does not mean that you are home free. Even good software can go bad when configured or used in an insecure fashion. Security options can often be disabled, user roles can be jumbled together allowing excessive access, and passwords can be sent across the network in the clear. High-quality software configured poorly, can be as vulnerable as poor-quality software doing its best.

### **Sloppy application configuration**

One of the most dangerous forms of improper use and configuration comes at the hands of the administrator. The security of a host is directly affected by the security of the applications running on that host. Careless configuration of installed services will almost certainly lead to trouble.

Let's say you need to build a mail server that supports user authentication so that your users can send mail from foreign networks. If you fail to provide an encrypted session over which the authentication information can travel, you will be exposing your organization's usernames and passwords.

You might also need to deploy a set of DNS servers. Without careful configuration restricting who is allowed to query the servers and how, you may be opening yourself up to denial-of-service attacks or worse. Again, careful configuration will help mitigate these risks.

Chapters 5–8 focus on providing common services from FreeBSD and OpenBSD systems. By understanding the application and the risks associated with providing service, you will be able to carefully configure and deploy these services safely.

## Protecting yourself

Insecure configuration and use comes in many shapes and sizes. There are nearly an infinite number of ways to misconfigure a host or an application and compromise its security.

Again, as with application security, auditing is vital. But in this case, it is not simply to catch attackers. In order to maintain a controlled configuration on production hosts, you must have a configuration management process. At a technical level this means structured change control, and possibly even revision control procedures. In even more formal environments, a daily meeting where production changes are discussed and approved can go a long way toward keeping configuration changes sane. Auditing hosts that are under configuration management allows you to detect when changes have been made. Any unauthorized changes will be discovered and can be backed out before they cause security problems.

Change control procedures are more extensively discussed in Chapter 4.

## Accounts and permissions

At the heart of the Unix security model are users and groups. The concept is pretty straightforward. Files and directories on a Unix filesystem are protected by user, group, and other (often referred to as world) permissions. The user represents the finest resolution of access control. Users can also be members of a group that has specific access rights over filesystem data. To make things more flexible, a user can be a member of multiple groups. Finally, the world permissions apply to all users on the system, regardless of group membership. Permissions are further broken down into access modes specific to the owner, the group, and world. Each class may have read (r), write (w), and/or execute (x) rights to the file. These filesystem permissions are likely familiar to anyone with background in Unix operating systems. However, despite being well understood, it is imperative that filesystem permissions are closely monitored.

One particularly dangerous set of permissions can make user and group ownership sticky. The *set-user-identifier* permission (*setuid* or *suid*) causes a program to assume the user ID of the owner of the file, not the person who executed the file. The *setuid*

permission is represented by an *s* in place of the user execute bit. For instance, the `traceroute(8)` program that is included with FreeBSD 5.x is *setuid* by default.

```
-r-sr-xr-x  1 root  wheel   23392 Jun  4 21:57 traceroute
```

When a normal, non-root user runs `traceroute`, the process nevertheless runs as root. This is done because `traceroute` needs access to low level network capability that a normal user does not have. Similarly, the *set-group-identifier* (*setgid* or *sgid*) permission can be set to change the group owner of a file or directory.

While all this can be useful for making programs work in certain ways, it can lead to mistakes. The *setuid* bit should only be applied to programs that absolutely need it. *setuid* root files are often the target of attackers who try to make the program do something it should not. If a *setuid* program can be made to execute arbitrary code, or clobber files, it will do so as root. This could easily lead to escalated privileges on the system. Programs should never have *setuid* bits applied to them after the fact—the fact that they will be running *setuid* should be part of the application design process.

To find *setuid* and *setgid* files on your BSD system, run the following command:

```
% find / -type f \( -perm -2000 -o -perm -4000 \) -print
```

If non-root users on your system don't need to run these *setuid* executables, the *setuid* and/or *setgid* bit can often be safely removed. OpenBSD administrators will be happy to know that the OpenBSD team has spent considerable time reducing the number of *setuid* binaries on the system, and using privilege separation to mitigate the risks of the remaining ones.

The BSD-based operating systems also have a special group: `wheel`. In order for users to use the `su(1)` command to launch a root shell, they must be in the `wheel` group. Other than controlling access to the root account, several files and devices on the operating system are group-owned by `wheel`. Be very careful about who you add to this group.



According to Eric Raymond's jargon file (<http://www.catb.org/~esr/jargon/>), the `wheel` group was derived from the term "big wheel," which means "a powerful person." When Unix hosts were less common and far more expensive, being in the `wheel` group was really a position of power. With the advent of free BSD-based operating systems that run on cheap x86-based hardware, being in the `wheel` group on your home PC does not seem like the honor it once was.

In general, permissions on files and directories should always be carefully controlled and audited. Administrators are often tempted while debugging problems to change the permissions on files or directories to `777` (everyone can read, write, and execute the file or change into the directory) in trying to determine whether the problem is permissions related. In some cases, especially in test environments, this may not be a

terrible thing to do—as long as permissions are quickly restored to normal. Unfortunately, in many cases the administrator will have made several changes at once; when the application starts working again he might be tempted to leave everything as it is. This can lead to a very dangerous situation, especially on production systems.

## Passwords and other account problems

Beyond permissions, accounts themselves can be the source of security problems. First and foremost, weak passwords are a bane to any security-minded system administrator. Make an effort to enforce strong passwords on your systems and for your applications.

### Strong Passwords

Strong passwords are generally at least eight characters in length and should contain a variety of letters (both lowercase and uppercase), numbers, and special characters. Avoid passwords that closely resemble dictionary words, proper nouns, or other words and numbers with any personal significance. Here are a few quick examples.

Donut is clearly a lousy password. The introduction of some number substitution in `D0nut1` may make you feel better but will not really improve the strength of the password very much. The password `sremrofsnart` may look good at first glance, but it is merely transformers spelled backward. Likewise, substitution in `sremrofsn@rt` does not significantly improve the password. A mnemonic such as `Tsij6w!` (This song is just 6 words long!) is a better all-round password than the others. Arbitrary choices of letters, numbers, and special characters that are easy to type are often also good candidates. Finally, passwords should be something easy enough to remember that they do not have to be written down.

Your goal in enforcing strong passwords that satisfy the aforementioned requirements is to prevent passwords from being cracked or guessed. To better understand this concept, explore password crackers and develop an understanding of how they operate. These are the tools that will be used against your password database, given the opportunity.

As time passes, even good passwords can lose strength, in a way. There is a continually increasing chance that a given password has been stolen. This can be accomplished by sniffing traffic on the network, watching users type them in, or compromising a host and running a password-cracking program against `/etc/master.passwd`. After some period of time, your password should be changed. For some organizations this period is annual, for others it is monthly.

Improper use of accounts is another common problem on production machines. Administrators will sometimes create a single account for many people to use. This happens often in technical support groups where turnover is high. Rather than find-

ing an easier way to perform account maintenance, an administrator may make a single account for all tech support people and give everyone the password. It is also commonplace to use system accounts with a generic name such as *www* and let several web developers log in under this user ID.

The practice of assigning multiple human beings to one account on the system makes auditing impossible. There is now no way to know what human being performed what actions on a given host because, from the perspective of the operating system, there is only one user involved. Changing passwords on these kinds of accounts is also a hassle as it must be done whenever someone leaves, and everyone must be told the new password immediately.

We continue to discuss managing user permissions in Chapter 4.

## Network Versus Local Attacks

We know that attacks can target faults in software, faults in configuration, or both, and this helps us tell how attacks might succeed, but it's also important to consider where attacks come from. A buffer overflow that allows for code execution on a host can lead to a host being compromised. However, if the buffer overflow vulnerability exists in some *setuid* binary that is only accessible to logged-in users, the risk associated with the vulnerability is severely limited. In fact, any vulnerability that requires local access is difficult to exploit, unless would-be attackers can gain access to the host.

On the other hand, if a buffer overflow exists in software that is reachable from the network, *ftpd* for example, then the danger is much greater. Simply being connected to the Internet exposes this vulnerability to everyone on the Internet. The risk associated with remotely exploitable vulnerabilities is often so much greater than with local exploits that administrators tend to respond much more quickly to these kinds of problems.

Just because network-based attacks are more dangerous does not mean you should ignore, or even put off fixing, local vulnerabilities. System administrators, due to lack of time, carelessness or both, sometimes patch the most critical vulnerabilities early and leave the rest “for later.” Eventually, a few months or years down the road, this practice is likely to lead to a compromise. It may become possible to combine a vulnerability that had previously only been locally exploitable with another, remotely exploitable, vulnerability to compromise the system. An attacker may be able to break into the system through this service's vulnerability and gain a shell prompt. At this point, just one of those “minor,” locally exploitable vulnerabilities is exactly what the attacker needs to gain escalated privileges and compromise the host. From there, other presumed minor vulnerabilities (in services accessible only if you are already behind the firewall, for instance) become prime targets and a means to compromise the rest of your network.

The lesson here is to follow your instincts in patching the most critical vulnerabilities first. However, if you fail to also patch minor software issues in a timely manner, they will be exactly what the attacker needs to own your network.

## Physical Security

From boulders to armed guards, physical security can take many different forms. However in most organizations, physical security and information security are controlled by two separate parts of the organization. Firewall administrators usually do not take care of giving out physical keys for the office doors. Sometimes it's tough to remember that physical security is an integral part of computer security.

If physical security breaks down, nearly all computer security constructs are rendered useless. An attacker who has physical access to a host has completely bypassed any network protections in front of the host. No one has invented a firewall (yet) that will detect a physical intruder, remove itself from a rack, and beat the intruder senseless.

Once physical security has been breached, an attacker can remove hard drives, or force a machine to boot to alternate media in order to subvert the core operating system. This will obviously cause a service interruption for the host, but a truly motivated attacker with physical access may not care about completely destroying a host in order to steal data or simply wreak havoc.

While this may seem abundantly obvious, security professionals often lose sight of the importance of physical security. We constantly weigh risks, decide which firewall configuration is best, or determine how best to handle groups on a server. However, decisions like that may be pointless if the data center holding your hosts is in an unlocked or an unattended building. Remember to take physical security into account when weighing risk. It would be a shame to get `ipfw` or `pf` configured on your BSD firewall only to see some guy running down the street with it the next day.

## Summary

At this point, you should have a pretty good idea of the attacks your system will face after you attach it to a network. Throughout this book we point out how you can defend against these attacks: where you should go for software updates, how to keep track of and respond to security advisories, and how to be diligent and careful in your system administration practices. In the next chapter, we'll describe in detail some of the building blocks the BSD operating systems provide to further mitigate the risks of system compromise. Before we get there, though, we close the loop on the topic of risk.

# Responding to Risk

Risk mitigation starts at the top. If you're working in an environment where you identify systems that need protection, determine how much effort you will put forth to protect systems, and perform the subsequent remediation, then you're doing too much.

In order to be able to respond to risk, the first step is to identify the resources that are important. Your organizational leadership is responsible for providing this high level assessment of information system criticality. Once this is done, senior security professionals get involved so that management can understand the kinds of risks given resources will face and perform a cost/benefit analysis for risk remediation.

## The Security Policy

One important artifact that comes from the discussion about risk by management and security professionals is the organization's security policy. This document provides a very high level overview of security requirements for the organization.

If your organization does not yet have a security policy, now is a great time to push for its development. The security policy is a document driven by requirements at the highest level of the organization. With organizational leadership supporting a policy, the administrator has the motivation from management to provide a security infrastructure and the support to do so from management.

There are a variety of resources to assist in the development of a security policy. RFC 2196 provides a guide to setting up security policies. The SANS Security Policy Project at <http://sans.org/resources/policy/> will help you develop a security policy by providing templates and guidance.

So how is this analysis done? If you work for a fairly small organization, you may be very involved in this process. So, let's take a moment to look at how to decide how much security is the "right amount." This is useful both in the context of defining priorities for the organization and at a smaller scale. For instance, your security policy may state that your DMZ systems must be protected by "strong authentication." Security policies in general are not much more explicit than this. It's up to security (and possibly also system) administrators to figure out how to get this done.

## How Much Security?

There are two considerations that influence how much time you spend "doing" system security: risk/consequence considerations and usability. The greater the risk or the higher the consequence, the more effort you must spend securing your systems. With respect to usability, if your application of security principles makes

administration more difficult—worse yet, if it discourages administrators from maintaining systems because of the hassle—you may have gone too far.

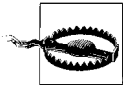


Be careful to differentiate between inappropriate security requirements and lazy administration. Not every administrator who shirks at jumping through a few hoops is pointing out your paranoia.

## Risk and consequence

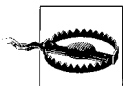
The role of your system combined with its exposure to risk helps you determine to what lengths you should go in locking down your system. If you are building a server that provides DNS functionality to your network, a failure or compromise of this system would easily lead to widespread problems. An incident involving a server that provides only NTP however may merely disrupt clock synchronization until the service can be restored, but may not *immediately* affect the rest of your network.

The location of your system both physically and logically on your network is also an important consideration. Systems located on a perimeter network are exposed to external attacks frequently. A computer providing Internet access at a library may have hundreds of users a day—and not all of them trustworthy. Servers on an office network are only directly exposed to other systems on that network.



Be careful here! Many administrators blithely assume an internal system is not prone to attack and spend almost *no* time or effort securing the system. This is why the few attacks that originate within the organization so often succeed.

So, what if there were a compromise of a system? What would the impact be to you and your organization? Your organization may suffer from bad press and/or loss of revenue. You may be deemed incompetent and then fired. The effects of a security compromise are not always obvious. The higher the cost of a security breach of the system currently being built, the more time you need to spend securing the system.



Some argue that, because the system they are building is not that important, the security of that system is not worth any great effort. However, compromised systems often are not an end in themselves; instead they provide attackers with a staging ground from which to initiate further attacks.

From a risk perspective, the amount of effort you must put forth “doing” system security relates directly to the amount of risk involved and the expected consequences of risk realization. Table 1-1 summarizes the effect that probability and impact have on risk.

Table 1-1. Relationship between risk and probability/impact

Impact	Probability		
	Very probable	Possible	Very unlikely
Disasterous	HIGH RISK	HIGH RISK	Medium Risk
Significant	HIGH RISK	Medium Risk	Medium Risk
Moderate	Medium Risk	Medium Risk	Low Risk
Minimal	Medium Risk	Low Risk	Low Risk

This makes sense. Buying a house on an eroding precipice is a high risk proposition. The chances of your house sliding into the abyss are pretty likely. The impact to your house and your belongings certainly qualify as “disasterous.” Maybe you’d rather build a house on a mountain. The chances of being buried under a mudslide or avalanche are probably on the low side, depending of course on the mountain’s history. The impact would likely be fairly significant though. We’re looking at medium risk proposition here. You get the idea. The higher the impact and/or probability, the higher the risk.

### Security versus functionality

You might wonder why you don’t just use every security tool you have on and around every system. Security costs something. At the very least, the time you spend performing security-related tasks is one kind of cost. The associated loss of functionality or convenience is another form of cost.

Envision building a webmail gateway so that users are able to access internal email even when they are offsite. This has the potential to expose private email to users on the Internet, so you may decide that a few extra notches of security are necessary. So you lock down the webmail system. You remove all webmail plugins except those that are absolutely necessary. You enable HTTP-digest based authentication in your web server. You require that users also come through a proxy server that requires a different layer of authentication. Finally, you mandate that users carry around signed certificates so that their identity can be validated to the server.

This example may be contrived, but security overkill is not always so obvious. In our example, users will become so frustrated with the passwords they must remember, the USB drives they must carry, and the overall hassle that they will eschew webmail altogether. Instead, they may very well start forwarding all their corporate mail to their personal account, which is far more accessible. As a security-minded system administrator, you have just lost: security came at too great a loss of convenience. Similar problems arise if the balance between security and maintainability is lost.

In less structured environments, administrators often have a lot of leeway. If you choose to boot your OpenBSD system from CD-ROM and store all binaries and libraries on this read-only media, you may have gone a long way to keeping your

system inviolate. However, what happens when a critical vulnerability is announced and you have to immediately patch vulnerable servers? It will take a great deal of effort to build a new standard configuration onto a CD, test it in your QA environment, and finally deploy it in production. This effort may make you want to wait to perform the upgrades later, delaying your response to a critical event. The maintenance hassle of making your security posture “go to 11” just increased the window of opportunity for attackers.

The danger here is obvious. If you put off patching known vulnerabilities because of the effort required to do so, then you can be worse off than if you had spent a little less effort on your secure configuration. In some very strict environments, you may not have a choice and be required to build, test, and deploy new CD boot images that night.

## Choosing the Right Response

After the important resources have been identified and some cost/benefit analysis has been performed to figure out what it’s going to take to secure the resource, it’s time to decide whether to do it or not. It’s pretty obvious what you do when the cost is low and the benefit is high: you act! But what about when the cost is high? If the benefit is high, do you do it anyway? The following outlines what you can typically do with risks after you’ve identified them.

### Mitigate risk

If the fallout from a realized risk is significant and the costs of mitigation can be tolerated, you should take action to mitigate the risk. Some, but perhaps not all, tasks will fall to the system administrator. She is responsible for things like patching vulnerabilities to keep attackers at bay and system backup to prevent accidental or intentional data destruction.

### Accept risk

These risks are identified, evaluated, but not mitigated. The realization of these risks fails to significantly affect the organization, the likelihood of risk realization is too low, or no mitigation can be enacted for some reason. For example, a university might categorize the potential for an outside observer using intelligence-grade surveillance equipment to remotely observe the registrar’s grade entry as a risk. The likelihood of this occurrence is as low as the cost of mitigation is high. Should the risk be realized, the consequences are probably not dire. Therefore, the risk remains and the university will accept the ramifications if this risk is realized.

## Transfer risk

Risk transference is a form of mitigation in which the risk is transferred to another party. This might involve outsourcing a system or process, or keeping it in-house while taking out insurance to cover potential loss. If your system is processing credit card information, there is a risk that the credit card numbers stored on your systems could be stolen by an attacker or insider. Standard mitigation techniques (firewalls, strict access control, and encrypted data) may keep out external eyes, but a rogue employee with access rights may be able to steal and sell this information. The costs associated with the realization of this risk are too immense for the organization to handle, but the likelihood of it happening is low to moderate. A risk like this is a good candidate for being transferred to another entity, like an insurance company.

## Security Process and Principles

We now have an idea of the kinds of attacks your host will face. Management has made it clear what resources are important, what must be protected, and what is too expensive to protect. What we're left with is a bunch of risks that need mitigation. We know that to mitigate the risks, we have to balance security needs against required functionality and convenience. We also know that the more critical the resource, the more effort we should go through in securing systems. So let's bring this all into focus in terms of FreeBSD and OpenBSD system administration.

This book is about “doing security” for your BSD systems. This book presents one major framework for building security into your BSD system deployments: security through the system lifecycle. What does this mean? Well, this is a daunting undertaking, to be sure. To make these kinds of tasks easier, we break them into discrete parts; building a secure initial configuration, performing ongoing maintenance, and auditing and incident response.

## Initial Configuration

Secure initial configuration is an obvious topic when discussing system-level security. And why not? A host with a secure initial configuration is more likely to stay that way. Careful installation and good decisions early on will leave you with a well-configured, fairly secure system. There is a strong motivation to be diligent in ongoing maintenance because it is easier to maintain a clean slate than it is to solve security issues while not breaking functionality. Furthermore, well-maintained systems who were built from secure initial configurations assist in containment. An attacker that manages to break into a well-maintained box somehow will have a hard time continuing his assault because unneeded services are disabled, file permissions are carefully controlled, and applications are tightly secured.

Properly configuring a host requires a solid understanding of the technology at hand. Unless an administrator knows the ins and outs of the core operating system and the

applications running on it, she will not be able to know what actions to take to lock down the host. While there are plenty of host-lockdown templates on the Web for various operating systems, none of them is a one-size-fits-all solution. Administrators often blindly follow security templates only to make their host so secure that it becomes unusable. Or, they follow a template that doesn't apply well to their actual situation. It gives them a feeling of added security because they took proactive security initiatives. But its poor applicability means they have left important weaknesses unaddressed. We urge you to understand the service you are trying to provide and the risks associated with it. From there you can figure out how to appropriately lock-down your system.

Initial configuration is a common area of focus for many books. Configuration is very tangible and the myriad options can be very confusing. Every application has its own unique options and architecture considerations. On any modern OS, there are many applications and aspects to the core system that require specific attention. Authors and readers find it easy to focus on these issues because there is so much ground to cover.

However, securing the initial configuration is not the only aspect of system security. System security is a complicated subject, users rarely see the big picture. Even systems that are thought to have a secure initial configuration can be administered poorly, eventually causing gaps in the security stance of the machine. It is important to look beyond secure configuration options and think about the broader security picture.

Building a secure initial configuration, as far as the operating system is concerned, is the primary focus of Chapter 3. Every application-specific chapter in this book will also focus on the secure initial configuration of the relevant application.

## Ongoing Maintenance

Regardless of the role a host plays, once it is deployed, it begins to change. Managing this change is key in maintaining the integrity and security of the host. Assuming you have paid attention to the details of configuring the host, it is in a known good state when it is deployed. Every service request, attempted attack, applied patch, and administrative login has the potential to change the security stance of the machine.

Staying ahead of this change requires a disciplined and coordinated effort. For instance, patch management procedures need to be in place long before you start applying patches. Vendor patches are often released in response to a vulnerability discovered and announced in a public forum. From the point a vulnerability is discovered to the point your systems are patched, the security of your host is a matter of chance. A worm may be written to automatically crawl through the Internet exploiting this vulnerability. An attacker may use the vulnerability to target your organization directly. Thus response to security advisories must be quick and effective.

While this may sound simple as a concept, in practice, patching can be very disruptive. Patches may interrupt service when they are installed. They may even have adverse effects that force you to roll the patch out of your systems until the effects can be mitigated. Successful patch management is not simply composed of technical aspects like how to download and install the patch. Patch management includes regression testing the patch in a lab, getting buy-in from stakeholders and assuring them that the patch will not interfere with organizational operations, and being prepared to roll back when necessary. By understanding the subtleties in patch management, you are helping ensure the security of a host over the long haul.

Other issues, such as using secure transport mechanisms when accessing a host for administrative purposes and proper user management, are also vital for the long-term security of a host. Understanding the ins and outs of secure management is critical on any platform. Implementing these secure management processes can vary dramatically depending on the operating system and applications being used. Luckily, FreeBSD and OpenBSD have a long history of being very maintainable systems. These operating systems are implemented in a manner that makes keeping them secure straightforward and relatively easy to upgrade when security vulnerabilities are discovered.

System maintenance with a goal of maintaining system security is the primary topic of Chapter 4.

## **Auditing and Incident Response**

Sometimes, bad things will happen to good hosts. Even with a secure initial configuration and proper administration techniques, an attacker will periodically successfully bypass the security perimeter of a host. At the very least, auditing will help you determine when you're dealing with an incident. You then need to be prepared to ensure service is restored as rapidly as possible and the damage contained.

Proper incident response relies on both technical and business knowledge. On the technical side, individuals responding to a security incident must have a playbook already created that describes how to get machines redeployed in a secure fashion. Every incident will differ to some extent causing changes in the original plan to restore service. By understanding the core operating system, the applications running, and the manner of attack being executed, an administrator responding to an incident can modify the path to recovery to match the attack that was used.

From a business perspective, the administrator needs to understand the impact the attack has on the business and react accordingly. For instance, if a new product has recently been deployed on the web servers, restoring service immediately may take precedence over preserving the attacker's footprints. Or if an attacker has a history with your organization, you may want to verify that you have a complete audit trail of her actions before restoring servers to known good states. Security is a means to

an end. The organization's goals are the ultimate end in most cases, so your actions when responding to an incident should reflect that.

After an incident has been contained through proper response, there may be forensic work required. Some organizations choose to analyze the technical aspect of every incident in an effort to learn what actions the attacker took. This allows them to determine the real loss caused by the attack as well as whether or not civil and criminal charges should be pursued. Other organizations generally do not do forensic investigation unless the attack is obviously damaging enough to pursue legal action. These types of organizations have determined that the reward for the investigation does not normally warrant the effort required to determine what the attacker has done.

Regardless of your organization's stance on incident response, you may be called upon to perform forensic analysis of a compromised host. The level of diligence required when performing analysis will vary depending on if the investigation is internal or if the data will be used in a court of law. However, from a technical perspective, performing a forensic analysis requires deep technical knowledge of the structure and operation of the operating systems and applications in question. It also requires understanding of the tricks attackers may use to hide or store data and processes.

Forensic analysis is really detective work. It involves looking for clues and understanding the motivation of the attacker. It also involves knowledge of how things work. The lead character in any TV detective show is not just a good interrogator. These fictitious detectives generally have years of broad fictitious experience they can leverage to solve heinous fictitious crimes. Examining a compromised host is very similar; but your experience had better be real and the more you know when you start the analysis, the better analysis you will be able to perform.

Auditing and incident response are the major focuses in the last section of this book.

## **System Security Principles**

Security through the system lifecycle is a useful framework for understanding how security can be woven throughout your administrative duties. This lifecycle is “what you do” for system security. It's time to turn our attention to “how you do it.”

We've talked about building a secure system, maintaining it in a clear and controlled way, and responding to threats. Following this system administration lifecycle helps us maintain a secure environment and maintain our organizational operations. However, the details of how to actually conduct these lifecycle activities are not nearly as straightforward.

Understanding how to build, deploy, and maintain a secure system involves technology specific information. In this book, we will give you a great deal of information on

FreeBSD and OpenBSD, arming you with the “domain expertise” required to use these operating systems securely. But beyond that, you must have the right mindset. To be a security-minded system administrator, you need to have the right set of guiding principles. The principles outlined in this chapter should be applied at every stage of a system’s lifecycle. Whether you’re designing your system or dealing with an ongoing incident, these principles should be valuable in making the right decisions along the way.

As we walk through the system lifecycle in this book, you’ll notice that in making decisions or justifying claims, we refer to a variety of security principles along the way. In the book *Building Secure Software* (Addison Wesley) by Gary McGraw and John Viega, the authors present 10 guiding principles for software security. Our focus in this book is not specifically on writing secure code (software security) but in building secure FreeBSD and OpenBSD systems. Nevertheless, many of these principles map directly to secure administration practices and we present them explicitly here.

## Apply Security Evenly

The application of security must be consistent across everything you do. The “weakest link” principle means that the strength of your overall security posture will be no stronger than the weakest link. This makes sense. We’ve already looked an example of this. If your information security stance is strong—your host has been defended against all network-based and local shell-based attacks—you’re in good shape. But if your server is located in an unattended or unlocked co-location facility, all someone really has to do is walk up and take your machine away.

## Practice Defense in Depth

One of the most important and frequently touted principles is *defense in depth*, also referred to as the “layered approach.” Defense in depth suggests that multiple levels of security are better than one single layer of protection. In our example of the physically unprotected server, to apply defense in depth we would move the system to a co-location facility that is locked. Not only should the building be locked, the system should be in a locked cage. Cameras should monitor arrivals and departures. Guards should also be posted to discourage would-be attackers.

Will all of this make our server impenetrable? No, but every additional layer of security makes compromise less likely.

## Fail Safe

In case of failure, fail in a safe way. Err on the side of caution. If our server that’s now in a guarded vault performs virus checking for your organization’s incoming mail,

what should happen when the server fails? Should your mail servers simply say “Hmm... my virus-checking host is unreachable. Well, the mail must go through!” Ideally, no. If your organization can tolerate a slight delay in mail delivery, your mail server should probably allow mail to spool until the virus-checking host is available again.

## Enforce Least Privilege

Least privilege is the concept that an entity, be it a person, process, or otherwise, is given the bare minimum privilege required to carry out a particular task. The idea is relatively straightforward.

Let’s say you send one of your interns to go fix the broken server, but the server is in the same cage as a bunch of your organization’s financial systems. The principle of least privilege would indicate that you make sure that the intern you’re sending can access the broken server and nothing else. Perhaps you could ask the co-location facility staff to disconnect the host and leave it in a room with a crash cart and a network drop so your intern can work on it without having access to anything else.

One useful way to build least privilege into your infrastructure is to approach deployment and configuration with a default deny mindset. In a firewall context, this means your first rule is to block all traffic. On a system, you only add a user account if and when a specific user needs access. By default, the user is placed into a group that has no access to anything on the system. Should the need arise, the user can be added to additional groups. Even the now well-guarded co-location facility follows this strategy. By default, they will not let anyone access the systems they host for you. You need to specifically authorize users.

## Segregate Services

When it comes to system security, don’t put all your eggs in one basket. When given the opportunity, separate the services running on systems as much as possible. In some cases, you may want to give one single role to multiple systems so that if one system fails, the service can still be available.

We might have been foolish enough to put our financial data on the same system that performs virus checking. Perhaps, given the volume of mail we receive, virus checking is barely utilizing the resources of the system. Coupling virus checking with data storage at least allows us to use some of that barren 200GB of mirrored disk space that came with the host. It’s a good thing we didn’t do that. When the system failed, not only would our mail flow have been temporarily interrupted, our financial group would have been unable to issue invoices, reconcile their registers, or (heaven forbid) perform payroll processing!

## Simplify

Complexity is a bane when it comes to maintenance. It is easy to maintain a system that is configured in some sane way. When you create too many interdependencies and complex configurations, however, maintenance quickly becomes a nightmare. You're more likely to break something when you touch the machine than fix something.

When your intern finally gets to the system to repair it, he might need to find out what went wrong. If it's a standard FreeBSD install on a pair of hardware-mirrored (RAID 1) drives with packages where one would expect to find them, he stands a good chance of figuring out the problem. What if, on the other hand, you decided that installing using ports or packages wasn't good enough? You compiled your virus scanning software from source and linked all the binaries against custom libraries. If your intern hasn't been fully briefed as to how this works, he'll never discover that the last system upgrade wiped out your custom binaries and that's why the virus scanning software broke.

Keeping things simple ensures that anyone with the right skill set will be able to fix the problem. When complex configurations are unavoidable, they can be simplified through comprehensive documentation.

## Use Security Through Obscurity Wisely

People say that security through obscurity is no security at all. Sure enough, if obscurity is your only means of providing security, you are not providing security after all. On the other hand, there are a variety of tactics you can use to be a little more secure in conjunction with other secure configuration and administration techniques. These are generally less effective than "real" security, but a little additional obscurity (defense in depth) doesn't hurt.

Let us suppose that after your intern finishes fixing your virus scanning server, you tell him that he needs to reconfigure the web server on the same system that acts as a frontend management tool for the virus software. Instead of running this server on port 80, he should make it listen for connections on 4000. Like most security-through-obscurity techniques, this alone has limited value. While some vulnerability scanners might just wander across systems finding systems with a listening port 80, most probe thousands of ports per system. Moving your web server is of minimal use because network probes will eventually find it anyway. If he were to also configure the web server so that it doesn't announce what kind of service it is, network scanners will have a little more difficulty telling the person running the scan what kind of service is listening on port 4000.

The key here is that you shouldn't waste time on obscurity when you can spend your time constructively on security. Secure your systems and your services. Document

your configuration. If you happen to have a few extra seconds to obscure some information an attacker could otherwise get for free, then by all means do so.

## Doubt by Default

If you can help it, don't trust anything. Seems a little paranoid, and taken to the extreme, paranoia will certainly be more of a hindrance than an aid. Still, a little doubt by default will go a long way.

Examples of this are everywhere. If you were to receive a phone call from a director you've never talked to before, and she's asking for the dial-up password—would you tell her? No. At least, not until she's verified that she is who she claims to be. When you visit an SSL-enabled site, does your browser automatically install the certificate and consider it trusted? Of course not. The certificate needs to be signed by a trusted certification authority, it must be valid, and the name on the certificate must correspond to the name of the site you are visiting. Finally, will the co-location staff let your intern in simply because he says he works with Doughnuts, Inc? No way. They'll check his ID and cross-reference his name with their access list.

Every system you build will have services that interact with other systems and users. Think about what you can do to help your running services doubt by default.

## Stay Up to Date

Being up to date applies both to your systems and to administrators. Both OpenBSD and FreeBSD are easy to maintain through well-documented upgrade processes. Upgrading application packages can also be a straightforward procedure with tools like *portupgrade* in FreeBSD. Keeping your systems up to date will help ensure vulnerabilities get patched.

The resources available to the system administrator are vast. As it turns out, Google (or your other favorite search engine) is one of the best tools you have at your disposal for information gathering. Keep up to date on what's happening with FreeBSD and OpenBSD. Stay abreast of trends, subscribe to mailing lists, and receive security advisories. Research products thoroughly before you decide to install something. Last, but not least, talk to your peers and share knowledge and ideas. You'll learn something. They will, too.

## Conclusion

System security is a complex problem. It requires knowledge of the underlying operating systems, the applications running on the host, the network infrastructure, the business goals, and your potential attackers. System security is also divided into phases revolving around the lifecycle of a host. All these aspects must be pulled together in an attempt to keep a host secure and the system usable. As complicated

as this may sound, there is hope. Many resources are available to assist in understanding the scope of system security. This book will help you install and configure a secure FreeBSD or OpenBSD system to provide critical services for your organization and hopefully provide insight that will be useful to you as you attempt to tackle the problems that you face.

## Resources

The following is a list of resources pertaining to the topics covered in this chapter.

### General Security Resources

- *Building Secure Software*, John Viega and Gary McGraw (Addison-Wesley), 2001
- Full Disclosure: <http://lists.netsys.com/mailman/listinfo/full-disclosure/>
- Security Focus (BugTraq et al.): <http://www.securityfocus.com/>
- “Smashing the Stack for Fun and Profit,” Elias Levy, Phrack 49: article 14 (<http://www.phrack.org>)

### General Security-Related Request for Comments (RFCs)

- RFC 2196: Site Security Handbook
- RFC 2504: Users’ Security Handbook
- RFC 2828: Internet Security Glossary
- RFC 3013: Recommended Internet Service Provider Security Services and Procedures
- RFC 3365: Strong Security Requirements for Internet Engineering Task Force Standard Protocols
- RFC 3631: Security Mechanisms for the Internet