Part I

Getting Started

Chapter 2

Overview of OpenBSD

OpenBSD is one of the most secure and well-designed operating systems available today. It has its roots in countless hours of research and development based on some of the best UNIX flavors of the past, and it boasts all the features of modern operating systems. The OS is widely considered one of the most secure general-purpose operating systems available today and it supports many key parts of the global Internet infrastructure.

2.1 A Brief History of OpenBSD

Although the roots of OpenBSD go all the way back to the 1970s and the early days of UNIX development, OpenBSD has since evolved into a leading UNIX system that has a strong following and is deployed in many places around the world. UNIX first forked off into BSD in the late 1970s and then again into FreeBSD and NetBSD in the early 1990s. Theo de Raadt, still the current project leader, split NetBSD into OpenBSD in October 1995 and released the 2.0 version of OpenBSD in mid-1997. This new flavor was designed to be ultra-secure and elegant in design.

OpenBSD shares a great deal with FreeBSD and NetBSD, and the development team has added many well-known packages for use on systems such as OpenSSH and *pf*. OpenBSD is a different system from Linux, but most UNIX applications that work under Linux are or will be available for OpenBSD systems.

2.2 OpenBSD Security

OpenBSD's top priority is to be secure, and it was designed with security in mind throughout every stage of development. OpenBSD has remained immune to most of the exploits and attacks that have surfaced during its lifespan. The code for the core OS and the packages required for its installation have been carefully audited, and are generally considered to be secure. An additional constraint placed on the OpenBSD secure development model is that the base system must attempt to be cohesive with the original BSD model, preventing applications from breaking when possible.

This can all change, of course, when the next insecurity paradigm arises—for example, with the discovery of format string attacks in the summer of 2000. The OpenBSD group tries to stay ahead of the curve and implement preventive security measures; so far, it has proved successful. The code undergoes continual scrutiny and audits, and apparently innocuous bugs are fixed on the chance that they may be deemed exploitable at some later date. The OpenBSD team can typically provide a vendor response along the lines of "This was fixed at some earlier date." By remaining an open development model (everyone can view the CVS change logs), OpenBSD is constantly sharing its experiences with the larger community. Additionally, some developers within the project work on other software projects and incorporate OpenBSD security fixes into their own software; OpenBSD also benefits from these other projects by incorporating some of their code or ideas.

The OpenBSD team was one of the first large-scale development teams to adopt the now basic software engineering standard: Fix bugs, and classes of bugs, rather than just vulnerabilities. This means that system security can be enhanced by ensuring code correctness. Handling error conditions wisely is another part of correct development. Lastly, OpenBSD attempts to operate with as little privilege when possible, by having programs drop privileges as soon as feasible or by increasingly using the privilege separation mechanism. By applying these principles throughout the system, OpenBSD enhances overall system security.

2.2.1 The OpenBSD Security Model

One of the major goals of OpenBSD is to be a secure BSD-based UNIX system, and remaining BSD-centric is important to the project. For example, the developers have decided against incorporating features such as POSIX.1e capabilities, commonly referred to as "Orange Book" features. The system is to remain as generic a UNIX version as possible, but as hardened as possible within that constraint. Enough room exists within the traditional UNIX model to allow for a very secure and reliable system.

To accomplish this goal, facets of security have been incorporated into the system in a variety of places. Within the kernel, for example, subtle changes have been in place for years, such as randomized process IDs. This may seem unimportant, but it is a useful measure. Several applications use process IDs as security markers, and their predictability can be exploited to abuse the system. The kernel also has a strong random number generator, which it employs to hand out "entropy" to processes. Likewise, TCP initial sequence numbers are extremely random—far more unpredictable than on any other operating system available.[1]

Cryptography has been extensively developed and fully incorporated into the system as well, including the Blowfish and Advanced Encryption Standard (AES) algorithms (the latter is based on Rijndael). This feature is available to applications through the regular APIs. The advantages of this approach are numerous. An example is the ability to encrypt pages of swapped memory. If an attacker is able to obtain access to pages of swap that contain sensitive information, he or she would have to decrypt the data first before using it for nefarious purposes.

2.2.2 The Audit

Most people know about the full code audit of OpenBSD that began in the summer of 1996 and found thousands of bugs, including new types of software bugs leading other OSs to perform further security audits. This effort has produced a system well known for its reliability and correctness, plus an overall reputation of security throughout.

What few people realize is that the audit is continual. Almost every line of code is thoroughly audited, bugs are fixed, and many security vulnerabilities are eliminated.

One of the key facets of the OpenBSD approach to code auditing is the idea that "Bugs are what lead to security vulnerabilities, so bugs should be fixed." This is in contrast to many development efforts, which focus on vulnerabilities ahead of bugs. In taking its approach, the OpenBSD team has found that classes of programming errors produce a common bug pattern. This immediately leads to the notion that it is important to scour the source code for similar bugs, which the team periodically does when new classes of bugs are found.

This isn't to say that every bug that could be fixed is. In several cases OpenBSD has fallen victim to security problems, and in a few cases this OS has been the only one with a security bug. The development team is a group of humans and makes mistakes. Bugs get fixed and everyone moves on.

2.2.3 Cryptography

Along with strong security comes the use of cryptography wherever possible. A very secure operating system is useless unless you can do work securely with it. This is where cryptographic tools come into play. OpenSSH is used to allow secure connections and

[1]As was shown in a summer 2001 report by Michal Zalewski of the Bindview Group.

file transfers over the Internet. A strong random number generator is used throughout the system for everything from process numbers to salts for password generation. String hash transformations are used for the generation of MD5 hashes, S/Key passwords, and other items. The default cryptographic scheme of DES has been replaced with Blowfish to create much stronger user and system passwords. OpenBSD even supports a wide range of encryption hardware.

2.2.4 Proactive Security

OpenBSD is unique among operating systems because of its pervasive security goal; it has proactive security. OpenBSD developers are constantly working to find new security problems before they appear in the wild, and to apply these findings to current and future sources in the form of audits. In addition to making extensive use of tools and techniques such as chroot and privilege separation, everything is designed in such a way that problems are prevented and circumvented even before they came into being.

2.3 Which Applications Are and Are Not *Secure*?

Although the base installation is considered to be *secure*[2] the applications in the ports tree aren't generally considered to meet this standard. No thorough code audits of the applications in the ports tree are usually done by the OpenBSD team. Ports are, however, installed in such a way to help avoid most potential problems, and security problems sometimes do get detected and fixed in the porting process. Due to the large number of total ports, this cannot be said for the whole ports tree.

2.4 Licensing

Another goal of OpenBSD is to keep all of the source code free. To achieve this goal, as much software as possible that is included with OpenBSD is covered by the Berkeley Copyright.[3] This also means that almost all software that is included with OpenBSD must be at least as free as the Berkeley copyright. Recently, the entire OpenBSD source code and

[2]Saying that something is or isn't secure is hard to do. Even if an application has been audited, new techniques may arise that were never taken into consideration when the audit was done. Thus, while it is easy to say that a system is insecure, it is difficult to say with authority that an application is secure. The OpenBSD team works diligently to find most of the potential problems and to be as safe at they can be. OpenBSD does not attempt to employ any provable security mechanisms, however.

[3]**http://www.openbsd.org/policy.html.**

ports tree went through a massive license audit. This led to the change of many licenses by the respective code's authors and, in some drastic cases, the removal of code from OpenBSD source and ports.[4] In most situations it suffices for the authors of the software to be contacted to clarify their licenses. It is uncommon for software to be removed or replaced due to licensing issues.

As a consequence, OpenBSD can be used in commercial settings safely and with little fear of licensing issues. No fees—only the respect of the copyright and its statement—is required to use OpenBSD in a commercial product. In fact, there are numerous commercial products based on OpenBSD underpinnings.

2.5 The *Feel* of OpenBSD

OpenBSD feels different than many other UNIX systems. Its filesystem layout is more controlled and is designed primarily for security and functionalty, rather than to satisfy the needs of the marketing department.

Furthermore, OpenBSD attempts to adhere to its BSD 4.4 roots and do things "the BSD way" when possible. Many commercial and even some other free operating systems have adopted many System V features and characteristics.

2.5.1 Filesystem Layout

One of the first things that most new users will notice about OpenBSD is how the layout feels different than that of most other UNIX systems. One of the goals of OpenBSD is to make the system elegant. The result is that files are where you expect them to be. All system binaries are stored in **/sbin** folders, while the userland binaries are in **/bin**. As the software was developed mostly from the ground up, OpenBSD doesn't have any extra backward compatibility, like the support for **/opt** or **/usr/ccs/bin** seen on some other UNIX systems. This is possible because the number of people who are able to make changes to the system is kept to a minimum.

There is similar control for the ports tree—the main way new applications are installed. There is good control to prevent ports from installing files outside of **/usr/local**. All of these constraints help the operating system have a clean feel throughout, which many users see as a great selling point for the system.

[4] As an example, the license for the old firewall code used under OpenBSD kernels, *ipf*, had to be removed because the license was not compatible with the spirit of the Berkeley copyright.

Palmer Palmer˙Ch02 March 26, 2004 11:36

2.5.2 Security

OpenBSD is different from most other systems in another way: It is built to be secure by default. On a new OpenBSD installation, there are very few daemons running and few network services started. Many other systems don't behave in this manner. Most systems turn on almost every service possible to save the user from having to do the setup. In contrast, on an OpenBSD system, the user needs to go through the configuration steps to enable a service. This "secure by default" stance has prevented security problems on many occasions. Although a threat may occur on many other systems, since it wasn't configured or enabled on OpenBSD systems, the security of the system remains intact.

This stance also applies to all services that are enabled by default; the least necessary access is given and the most secure setup is configured. This sometimes does have the side effect of closing off some expected functionality. As an example, by default the *OpenSSH* daemon is configured with X forwarding disabled. Most users would prefer to have this service turned on, but it's an unnecessary feature and it might turn out to be a security threat.

This isn't to say that OpenBSD is a minimalist system. The basic installation contains an SSL-enabled Apache Web server, an FTP server, an NIS client and server, an SSH server (enabled by default), a routing server, and much, much more. Most of these services are controlled via central configuration files (discussed in later chapters). The system also supports a number of third-party packages, such as the Network Time Protocol (NTP), by enabling it in the start-up of the system when it is installed.

2.5.3 User Friendliness

As can be seen from the discussion of its security in Section 2.4.2, OpenBSD seems to be designed differently than a lot of other UNIX systems. Most systems are designed to be easy to use and friendly to the user, to the point of sacrificing security. The development of OpenBSD is driven solely by the ideals of its team of developers. Although many people are upset by some of the opinions of the leading group, this strategy has the side effect of keeping the system clean.

Another place where this attitude can be seen is in the support (and lack of support) for some hardware. NetBSD aims to support as many hardware architectures as possible and has support for almost any piece of hardware that can be bought. Solaris is designed to run well on the SPARC and i386 processor lines. The hardware that is supported by OpenBSD is directly related to the task most OpenBSD systems perform: networking services. As a result, a good number of strong and stable network cards are supported, but other, more error-prone cards (e.g., old NE2000 cards) are not as well supported. Any hardware that OpenBSD does support, however, typically works well.

2.6 Packages and Ports

OpenBSD supports the concept of packages and ports. As with other operating systems, packages are programs that can be installed without the need to compile them (in binary form). A vast number of packages are shipped with the OpenBSD CDs, and more are available on the OpenBSD FTP mirrors, to fill the needs for most users. Along with the collection of packages, OpenBSD comes with a ports tree (discussed in later chapters), which contains the recipes used for building the packages. The ports tree is designed to identify dependencies needed to build and install packages automatically as necessary. The resulting packages can be installed and distributed across systems. Packages and ports are discussed more in Chapters 12 and 13, respectively.

2.7 Where Is OpenBSD Used?

Most people do not use OpenBSD as a desktop OS, though there are some people who swear by it. Some mainstream applications are not supported by OpenBSD, causing many newer-generation UNIX users to shy away from it. Many people do decide to use OpenBSD as a desktop due to its numerous advantages, and they either make sacrifices by not being able to use the missing software or actually make it work. However, OpenBSD is untouchable as a secure network services platform. Most OpenBSD systems are deployed as firewalls or other edge servers where high security is vital. The systems also shine as Web, e-mail, DNS, and intrusion detection servers. Almost anywhere that security is a high concern, OpenBSD is well designed to fit the role.

For users who wish to have modern office, multimedia, and productivity applications, OpenBSD may not be the best choice. The support for third-party applications that meet these requirements is growing, but relatively thin. Java support in OpenBSD, for example, is poor.

OpenBSD also forms the base of several commercial applications—specifically, networking and security applications. Several of the project members are employed as software architects at different companies. The license of the project allows for the reuse of the software in commercial, closed-source applications. Nevertheless, many of the benefits and bug fixes of commercial development eventually return to the project as reliability patches and enhancements.

Chapter 29
systrace

29.1 Introduction

The OpenBSD default system comes with a policy enforcement tool named *systrace*, which provides a way to monitor, intercept, and restrict system calls. The *systrace* facility acts as a wrapper to the executables, shepherding their traversal of the system call table. The *systrace* facility then intercepts the system calls and, using the **systrace** device, processes them through the kernel and handles the system calls.

Getting started with *systrace* is quite easy. You can run your programs under *systrace*, generate policies based on the observed behavior, and then enforce this policy on the program in subsequent runs. There are, however, two problems with this approach:

- This approach assumes that the executable behaves entirely correctly and within the expected bounds. Violations of this assumption can include the use of a modified executable that has been reconfigured by an attacker. Subsequent uses of *systrace* will allow the malicious behavior to continue. To remedy this problem, policies should be reviewed after their generation to ensure that the anticipated behavior is observed, and trusted executables should be used in policy generation. Generally, automated policy generation should be undertaken only with trusted applications. Unknown applications can be used with interactive policy generation, so that decisions can be made before any damage is done by a rogue application.
- This approach assumes that the initial runs of the *systrace* policy generator fully exercise the range of actions for which the policy is intended. In reality, the *ls* executable, for example, may not know that it is allowed to list the files in a publicly allowed directory that was skipped in the original run.

To remedy this situation, it is possible to bootstrap the policy that *systrace* knows about by using arbitrary external policies and the *-f* flag. In this scenario, a base policy can be built and extended. Furthermore, one can generate filters that use wildcards, which eliminate the need for fully itemized lists:

```
native-open: filename match "$HOME/*" and oflags sub "ro" then permit
```

In this example, one can read (but not write to) any files and directories under the current user's home directory. This filter is forward adaptable and condensed.

Executables run under the *systrace* facility can pass policies on to their children and inherit policies from their parents. This is useful for login shells, for example, where you may wish to restrict a user's behavior using *systrace*. Any children from this shell will have a policy that has been inherited from their parent. A simple *systrace* login shell would look like the following:[1]

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

int
main(int argc, char *argv)
{
        char           *args[4];

        args[0] = "-Ua";          /* system policies, auto enforce */
        args[1] = "/bin/ksh";     /* run ksh */
        args[2] = "-l";           /* login shell */
        args[3] = NULL;
        if (execv("/bin/systrace", args) < 0) {
                fprintf(stderr, "loging in failed.");
                exit(-1);
        }
        /* NOTREACHED */
        return (0);
}
```

[1]Newer, more flexible versions are maintained by one of the authors and are available at **http://monkey.org/jose/software/stsh/**.

The series of steps to enable this system for users would look like the following:

```
$  mkdir -p /usr/local/src/bin/stsh/
$  vi /usr/local/src/bin/stsh/stsh.c
 enter above code
$  gcc -o /usr/local/src/bin/stsh/stsh /usr/local/src/bin/stsh/stsh.c
$  sudo cp /usr/local/src/bin/stsh/stsh /bin/stsh
$  sudo vi /etc/login.conf
 edit the variable ‘‘shells’’ to be /bin/stsh
```

This can be applied in the *default* class for all users or just to users in a particular login class.

Before you begin, make sure you have a policy for **/bin/sh** in the directory **/etc/systrace** (saved as **bin_sh**). Now test this setup (leaving at least one user with a normal shell for login purposes). Also, this method disables the utility *chsh*, which users can use to change their shells. They cannot disable their use of *stsh* and use a non–systrace-wrapped */bin/ksh*, for example. Instead, their parent shell will always be a *systrace*-wrapped */bin/ksh*. Newer versions of *stsh* can spawn any shell the user chooses, wrapped in *systrace*.

The target uses of *systrace* are threefold. First, it is designed for untrusted data paths, such as executables from potentially untrusted sources or applications that handle untrustworthy data. These can include daemon processes, for example, which are open to the world. By using *systrace*, an administrator can restrict the arbitrary execution of commands. Second, this program is very useful for machines with untrusted users operating in their shells. By spawning the login shell under the control of *systrace* and then forcing children to inherit this policy, transparent sandboxing of the system can occur. Third, *systrace* can protect users from their own processes. Some applications are untrusted or otherwise potentially damaging to the system or accept untrusted data from the network. Cradeling their execution by using *systrace* can help mitigate any damage they may cause.

Global system policies live in the system directory **/etc/systrace**. Examples policies exist for two daemons, *lpd* and *named*, which provide robust sandboxing for the executables. These examples show what can be done to secure a system using *systrace*.

User-specific policies are found in **/home/*username*/.systrace**. If a user modifies a global policy, the modified version is saved in his or her home directory. This prevents one user from modifying the execution environment of other users' applications.

Note that *systrace* does require a modest level of understanding regarding system calls and their consequences. It is easy to write a policy that is impossible to use by ignoring fundamental actions, which is why it is advisable to start with automatically generated policies. Also, some large, complex applications may be difficult to run under *systrace* due to the large number of system calls they make. In these situations, it may be wise to attempt to allow nearly everything except a subset of commands. For example, your Web browser may be allowed to open arbitrary sockets above 1024 but not allowed to spawn a child shell.

29.1.1 Example Use

As described previously, it is possible to use *systrace* to automatically generate a policy for an executable. With the *-A* flag set, *systrace* will accept all actions as permitted and use them to build a policy. The following example shows the geneation of a simple policy allowing **/bin/ls** to read the user's home directory:

```
$  cd
$  systrace -A ls
```

By default, *systrace* will store the generated policies in the directory **/home/*username*/.systrace**. For our example run of *ls*, a policy named **bin_ls** will appear with the contents of the policy for that executable:

```
Policy: /bin/ls, Emulation: native
    native-__sysctl: permit
    native-mmap: permit
    native-mprotect: permit
    native-ioctl: permit
    native-getuid: permit
    native-fsread: filename eq "/etc/malloc.conf" then permit
    native-issetugid: permit
    native-break: permit
    native-fsread: filename eq "/home/jose" then permit
    native-fchdir: permit
    native-fstat: permit
    native-fcntl: permit
    native-fstatfs: permit
    native-getdirentries: permit
    native-lseek: permit
    native-close: permit
    native-write: permit
    native-munmap: permit
    native-exit: permit
```

This simple, minimalistic policy is nevertheless very restrictive. When we try to use this policy to enforce actions, we can see the effect. Using the command *systrace -a*, we can automatically enforce the policy we have installed:

```
$  systrace -a ls /etc/
ls: /etc/: Operation not permitted
```

Additionally, a message is logged to the central system logs via the *syslog* mechanism. By default, these messages will appear in the file **/var/log/messages**. Reading these messages can be useful for security monitoring or policy review and adjustment purposes:

```
May 30 07:12:11 superfly systrace: deny user: jose, prog:
/bin/ls, pid: 1664(0)[17057], policy: /bin/ls, filters: 129, syscall:
native-dup2(90), args: 8
```

The denial of system calls can be controlled by using the specific signal sent to the executable making the request. For example, it may be advisable to send *ls* a "permission denied" signal when it attempts to show the files in the **/etc** directory. The *ls* command gracefully reports the error to the user and exits.

More complicated policies can be generated interactively. When the X11 environment is available, the application *xsystrace* is used to provide responses to policy queries. In a text-only environment, the responses are handled on the command line. Responses are "permit" and "deny" with options that match those found in the policy file. For example, generating a policy for *tcpdump* would look like the following:

```
#  systrace tcpdump
/usr/sbin/tcpdump, pid: 8159(0)[0], policy: /usr/sbin/tcpdump,
filters: 0, syscall: native-issetugid(253), args: 0
Answer:  permit
```

Note that *systrace* uses the shell from which it was started to make these policy queries. If the shell has been closed, the application will hang while waiting for a policy decision.

The *systrace* system understands the following environmental variables and expands them as macros:

- **HOME** The user's home directory (e.g., **/home/jose**).
- **USER** The user's name (e.g., *jose*).
- **CWD** The current working directory (also known as **.**).

These variables can be substituted in the *systrace* policy and allowed to expand. An example setup using such macros would appear as follows:

```
native-fsread: filename eq "$HOME/.gaim" then permit
native-fswrite: filename eq "$HOME/.gaimrc" then permit
```

These examples were taken from a policy for the IM chat client *gaim*, generated automatically by *systrace -A*, and then smoothed over by manual editing.

29.2 Creating Policies

Creating a *systrace* policy for an application is relatively straightforward, but entails an interactive series of steps. Policies must be edited to support matching arbitrary versions of shared libraries of network sockets, for example. The steps outlined here illustrate how the policy for the chat client program *gaim* was developed on one of the authors' systems.

Creating an initial policy is done using the built-in *systrace -A* command:

```
$  systrace -A gaim
```

At this stage, an initial policy is created (in this case, in the directory **/home/jose/.systrace/usr_local_bin_gaim**). It contains specific network addresses, shared library versions, and filenames. Many of these can be edited to support arbitrary versions.

29.2.1 Editing Policies

Policy editing consists of two main questions. The first question is "which attributes can be generalized into expressions to match, rather than specific instances?" Here the policy is edited to move from "eq" tests to "match" tests with some form of globbing. The second question is "What kind of filesystem limitations should be added?" For applications that write to the filesystem, it may be worthwhile to control the directories to which they have access.

For network applications, one of the major issues in policy editing is the handling of name servers. The file **/etc/resolv.conf** allows for multiple DNS servers, yet the application typically uses only one:

```
    native-connect: sockaddr eq "inet-[192.168.4.3]:53" then permit
```

In editing this entry, it is important to examine both the resolver configuration and the network environment. A laptop that uses DHCP on several networks and unknown DNS servers should edit the line to look like the following:

```
native-connect: sockaddr match "inet-*:53" then permit
```

Here any IP address on port 53 can be accessed. If a predefined list of servers is sufficient, then they can be enumerated:

```
native-connect: sockaddr eq "inet-[192.168.4.1]:53" then permit
native-connect: sockaddr eq "inet-[192.168.4.3]:53" then permit
native-connect: sockaddr eq "inet-[192.168.4.4]:53" then permit
```

In the case of *gaim*, the client connects to a variety of servers for load-balancing efforts and possibly on different ports, depending on the protocol. Here blanket socket connections could probably be allowed.

In the case of filenames, specifically for shared libraries, one can convert the policy to use the "match" operator and globbing rules. For example, a policy rule such as

```
native-fsread: filename eq "/usr/lib/libc.so.29.0" then permit
```

is easily rewritten as

```
native-fsread: filename match "/usr/lib/libc.so.*" then permit
```

and made more portable when the system is upgraded. If you wish to allow arbitrary library access and not enumerate libraries by name, the entries

```
native-fsread: filename eq "/usr/lib/libz.so.2.0" then permit
native-fsread: filename eq "/usr/lib/libperl.so.8.0" then permit
native-fsread: filename eq "/usr/lib/libm.so.1.0" then permit
native-fsread: filename eq "/usr/lib/libutil.so.8.0" then permit
native-fsread: filename eq "/usr/local/lib/libintl.so.1.1" \
    then permit
native-fsread: filename eq "/usr/local/lib/libiconv.so.3.0" \
    then permit
native-fsread: filename eq "/usr/lib/libc.so.29.0" then permit
```

can be rewritten as follows:

```
    native-fsread: filename match "/usr/lib/*" then permit
    native-fsread: filename match "/usr/local/lib/*" then permit
```

The application can now read any file in the directory **/usr/lib** or **/user/local/lib**. Note that it will not be allowed to write to any file in that directory unless an action is stated to permit that operation.

After editing the policy, the application should be rerun with the policy used:

```
$  systrace gaim
```

Errors will be handled in two ways. First, with no auto-enforcement (*systrace -a*) in use, the system will ask you how you want to handle policy violations. Second, denied actions will be logged to the system's logs:

```
/var/log/messages.1.gz:Jan 25 23:58:29 tank systrace: deny user: jose,
prog: /usr/local/bin/gaim, pid: 27552(0)[0], policy: /usr/local/bin/gaim,
filters: 104, syscall: native-connect(98), sockaddr: inet-[192.168.4.2]:53
```

After examining both of these feedback mechanisms, the policy can be edited to remove them and gracefully handle errors. As stated earlier, this strategy involves an interactive process of policy editing and testing.

29.2.2 The Benefit of a Local Caching Name Server

One of the complexities of a *systrace* policy with respect to a dynamic system (such as a laptop) is the variety of networking changes it generates as it travels around. For example, the DNS servers used by the system will change for each network used, as reflected in the varied entries in **/etc/resolv.conf**. As network client applications are used, they will contact these new DNS servers. If the *systrace* policy is restrictive in terms of the IP address of the socket used for DNS, then the application will fail as it enforces this policy.

One option is to build a generic *systrace* policy that can connect to any address on port 53 (for DNS):

```
    native-connect: sockaddr match "*:53" then permit
```

Any IP address will then be matched and DNS will be allowed.

Another option is to use a local, caching-only DNS server as your primary DNS system, along with a configuration that keeps this option static. For example, a name server entry in the file **resolv.conf** that specifies *nameserver 127.0.0.1* will cause client applications to use the local DNS server. For DHCP users, not requesting the *domain-name-servers* option will also be useful. In the file **/etc/dhclient.conf**, such a configuration will request other information, but not DNS server information, and actively reject the DNS server information offered:

```
interface "fxp0" {
    request subnet-mask, broadcast-address, time-offset, routers,
        domain-name, host-name;
    supersede domain-name-servers 127.0.0.1;
}
```

Now the local DNS server will be used. On the one hand, this scheme requires the additional complexity of running a local, caching-only DNS server, which is not desirable for some systems. On the other hand, it gives a static *systrace* option for all network client applications.

29.3 Privilege Elevation with *systrace*

The *systrace* system can also be used to remove *setuid* and *setgid* binaries. Normally, only a single step or two needs to be performed as an elevated privilege user. This can include binding to a low-numbered socket or reading a protected file.

In *systrace*, the *permit as* action can be used to allow a system call to proceed as a specified user. For example, an application that captures packets from the network will have to read these packets using the BPF devices as root. This privilege can be allowed for non-root users using the *permit as root* option:

```
    native-fswrite: filename eq "/dev/bpf0" then permit as root
    native-fswrite: filename eq "/dev/bpf1" then permit as root
```

Note that the parent *systrace* command must be run as root for this technique to work, as arbitrary users cannot run various system calls as elevated privilege users. Failure to do so will result in an error in the execution of the program:

```
$  systrace dnstop wi0
Privilege elevation not allowed.
```

The program will attempt to operate normally under these circumstances, but will typically fail. Instead, use the *-c* option to *systrace* to set the user ID (and optionally the group ID) for the child process. Here, root runs the *dnstop* program as user 1000, but is still allowed to open the BPF devices (normally accessible only by root):

```
#  systrace -c 1000:1000 -a dnstop wi0
```

This approach can be used to greatly limit the scope of programs that would otherwise require root privileges.

29.4 Where to Use *systrace*

One ideal place to run *systrace* with complete and restrictive policies is on network servers. Protecting the execution environment of exposed services with *systrace* can considerably minimize the ability of an attacker to cause a dameon program to begin executing arbitrary actions. This can include remote SSH servers, name servers, and Apache Web servers. Generating these policies can be a bit time-consuming. Nevertheless, with *systrace -A*, a thorough exercise of the program, and a review of the policies, security can be enhanced.

For publicly accessible shell servers—for example, on common lab systems—one target for *systrace* policies is the control of *setuid* root executables. Wrapping the execution of these programs in a controlled policy minimizes the potential damage that can be generated by a malicious user. The *setuid* root bit can be removed, and operations that require privileges can be replaced by *permit as* statements in the *systrace* policy. Wrapping the executable in a small program that ensures it is run under *systrace* can complete this security enhancement.

On network clients, Internet-exposed client applications can be wrapped in *systrace* policies. The earlier example, which showed the generation of a policy for the *gaim* client, can be extended to a variety of network clients, including *irc* clients and *ssh* usage. The *systrace* system can protect the local system from malicious servers or P2P clients that might attempt to execute arbitrary actions on the client system.

29.5 System Coverage with *systrace*

Achieving total system coverage with *systrace*, where no avenue remains in which to execute arbitrary commands or handle user-supplied data, is the ultimate goal for a system protected by *systrace*. It is best accomplished by performing three actions. The first action is to ensure that complete, up-to-date policies have been generated for the applications. It is perhaps best to run the system using *systrace -A* for a short while to fully exercise applications. The second action is to start any network daemons that are launched from processes such as **/etc/rc** using *systrace*, which requires minor amounts of script editing. The third action is to give users shells wrapped in *systrace*. Any executable that the users will run will require a policy, as *systrace* also wraps child processes.

This difficult-to-achieve process requires an in-depth understanding of the system as well as the implications of system calls. For most users, running *systrace* on their network daemons will suffice.

29.6 Additional Uses for *systrace*

Beyond application sandbox enforcement, the *systrace* facility has other uses. These are just now starting to be explored.

29.6.1 Software Testing

One interesting use of *systrace* is to test the error-handling abilities of various applications. The *systrace* system can be used to reliably and predictably force failures with various error conditions on a per-system call level. For example, to examine how a process reacts if it is unable to read the configuration file for *malloc*, a line such as the following would be integrated into a *systrace* policy for the process:

```
native-fsread: filename eq "/etc/malloc.conf" then deny[enoent]
```

This would return a “file not found” error for this file. The application’s handling of this error condition could then be tested to look for graceful handling of the error. Note that denying an application the right to peform a *native-exit* will force it to abort, which will produce a core dump.

29.6.2 IDS Logging

Another use of *systrace* is as an intrusion detection logging system. This is best done with the logging of *native-exec* entries. For example, to enable logging of all file openings by a network daemon process, the *systrace* policy for the daemon would include a line like this:

```
native-fsread: filename eq "*" then permit log
```

Now every file opening carried out by the process will be logged by the application. An otherwise complete policy will have to be created for the process as well.

Palmer Palmer˙Ch29 March 26, 2004 11:59

29.7 Limitations of *systrace*

Despite its many features, *systrace* has a number of limitations that bear mentioning. First, it lacks a facility to specify that you can "permit once" for a system call, such as binding to a socket. This can allow an attacker to recycle a system call, potentially at elevated privilege.

Second, system calls have no exclusive or. For example, an application might be permitted to open a file or a device, but not both. This weakness could ultimately be leveraged by an attacker who seeks to do more than a program was intended to do.

Lastly, the parent process has no control over spawned processes. For example, if you allow */bin/sh* to be executed, you cannot control it beyond its own *systrace* policy. One way to get around this limitation is to specify a policy for the child process to inherit if it is to be less liberal than the normal system policy. This would be done via *systrace -i*.

29.8 Resources

Niels Provos, the primary author of *systrace*, presented a paper at Usenix 2003 on the architecture of *systrace*. Users who are interested in the internal workings and design of the system may want to read his paper.

Niels also maintains the Web site **http://www.systrace.org**, which houses information on *systrace*. Various example policies are available there.