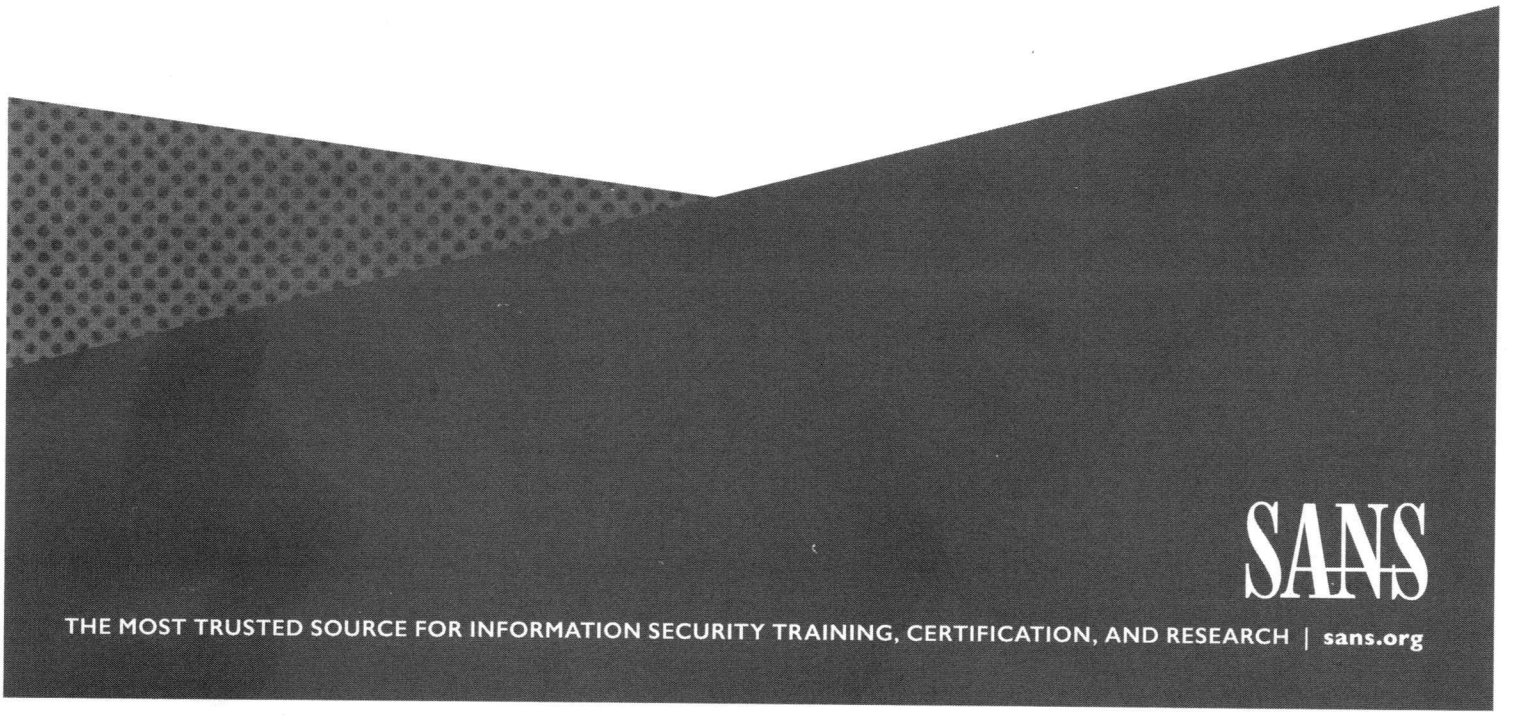
506.4-506.5
Application Security Sections 4-5



SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

SEC506.4 & 5 | Securing Linux/Unix

SANS

Linux Application Security

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Agenda

- Notes on `chroot()`
- The scponly Shell
- SELinux

- DNS and BIND
- Apache

The material in this course book is split over two days.

The first day is devoted to general techniques for securing applications on Linux/Unix. First, we'll briefly look at the `chroot()` mechanism, which can be used to improve the security of many different applications by locking the application up in a small subset of the file system. This may actually prevent certain exploits from working, and will certainly limit the options for the attacker. As an example of a chroot()ed application, we'll look at the highly useful scponly shell, which can be used as a more modern alternative to anonymous FTP. This afternoon we'll look at the SELinux type enforcement mechanism to further lock down what processes can do on our system.

Tomorrow's material looks specifically at the security settings in popular "Internet facing" applications that typically run on Linux and Unix systems: BIND and Apache. This course is not intended as a tutorial on how to administer these applications. Instead, we'll just be focusing on the "knobs" that you can use to improve the security of these applications.

Notes on `chroot()`

`chroot()` is a Unix system call that allows a process to give up access to all but a small portion of the file system. This enables programmers to create processes which run in a captive environment and therefore reduce many of the security risks to applications that have to face "hostile" networks—like the Internet.

What Does It Mean to `chroot()`?

- Process calls `chroot()`, specifying a directory argument
- Directory becomes root of private file system for process
- Process unable to access files outside of this directory
- Process is effectively "locked up" in a captive "jail"

When a process calls `chroot()`, it specifies a directory that the process will `chroot()` into. As far as the calling process is concerned, this directory becomes the root of the Unix file system for that process and the process is only able to access files from the `chroot()`ed directory and below. Full pathnames are interpreted as being relative to the `chroot()`ed directory. For example, suppose the process `chroot()`ed to the directory `/local/root` and then tried to access `/etc/passwd`, the process would actually be attempting to access `/local/root/etc/passwd`.

One of the primary uses for `chroot()` is to run networked services in captive environments so that security holes in these services can't be exploited against the entire machine. Typically, these are complex services like FTP, Web servers, and DNS servers or services which are insecure in other ways (like the TFTP service which permits file transfers with no authentication).

In the case of a buffer overflow attack or another remote exploit, even if the attacker gets access to the machine remotely they won't be able to "see" the entire file system to plant their back-doors. In fact, the exploit may fail because of the `chroot()`—if they're trying to `exec("/bin/sh")`, but the `chroot()` directory does not include the `/bin/sh` binary, then the attacker is out of luck.

How Do You `chroot()`?

Some daemons `chroot()` naturally:

- SSH (with PrivSep enabled)
- TFTP
- FTP (anonymous mode)
- BIND (optional)

`chroot` wrapper program can `chroot()` any command

Many daemons `chroot()` automatically or are at least equipped with the functionality to allow administrators to run them `chroot()`ed if they desire:

- Earlier in the curriculum, we looked at the "Privilege Separation" feature for SSH, where the SSH daemon calls `chroot()` for the potentially dangerous session startup and authentication phases.
- TFTP daemons will `chroot()` to `/tftpboot` (or a similar directory name) so that only files under this directory may be accessed by the TFTP server.
- Similarly, when an anonymous FTP session is started, the FTP daemon will `chroot()` itself to the top of the anonymous FTP directory structure created by the FTP server's administrator.
- BIND (the Internet standard DNS server implementation) allows administrators to run the name server in a `chroot()`ed environment and we will cover this configuration later in the curriculum when we talk about DNS security.

Since many services do not have built-in `chroot()` functionality, most Unix variants come with a `chroot` command which can be used to run any process in a `chroot()`ed directory structure. This is how you typically run applications like Apache in a `chroot()`ed environment.

Why Not `chroot()` Everything?

Requires complete "mini" Unix file systems for each process:

- Binaries and other helper applications
- Shared libraries
- Devices
- Configuration and log files

Programs with too many dependencies difficult to `chroot()`

The problem with running applications in a `chroot()` ed environment is that the environment itself is often difficult to set up initially. Remember that the process is completely trapped in the `chroot()` ed directory–all binaries, shared libraries, system devices, configuration files, etc. must be properly configured by the administrator in the `chroot()` ed hierarchy before the application can be run. Since the application gives few clues about which files and devices it needs to operate (other than failing to run if everything isn't set up exactly right), this process is fairly tedious and complicated.

It's not uncommon to find applications which are so deeply intertwined in the operating system that it's effectively impossible to run them in a `chroot()` ed environment. These sorts of applications should then be allowed to run by themselves on "sacrificial" machines where any security breaches can be more easily contained.

Only root Should chroot()!

- Attacker sets up `chroot()` area in their home directory
- Creates local password file with known root password
- Runs `chroot` on `su`, now `root` in `chroot()` ed area
- Attacker creates a set-UID shell here
- Attacker now has root shell in home dir!

The `chroot()` system call is only allowed to be called by processes running as root. If normal users are allowed to `chroot()`, then they can exploit this call to gain superuser privileges. For example, there's the attack that was documented in CERT Advisory CA-91.05 (see `http://www.cert.org/advisories/`) and affected DEC machines running Ultrix.

Apparently, the Ultrix developers felt that is was useful to allow any user to run the `chroot` program on their processes, so they made the program set-UID root. On the face of it, this sounds like a great idea—users can run suspicious programs in a `chroot()` ed jail and enhance the overall security of the system.

The problem is that a malicious user can exploit `chroot` to get a set-UID root shell. All the user has to do is set up a `chroot()` ed environment which contains a password file that they've generated with a known root password, plus a copy of one of the Unix shell executables. The user then runs `su` (or some other similar program) in the `chroot()` ed jail. The `su` program looks up root's password in the dummy password file and the attacker is able to log into the `chroot()` ed area as root. At this point, the attacker sets the set-UID bit on the shell executable in the `chroot()` ed area and then logs back out. The attacker now has a set-UID shell that they can use to compromise the entire system.

The "fix" is to simply never set the set-UID bit on the `chroot` program and to never allow normal users to call the `chroot()` system call.

Also Look Out For ...

- Attacker exploits poor config, and manages to upload a set-UID program into `chroot()` zone
- Attacker can then trigger set-UID program in system environment not protected by `chroot()`:
  - Log in as normal user and execute
  - Buffer overflow in unprivileged program
- Could use this privilege escalation to escape `chroot()` (see upcoming slides...)

You also need to be careful about attackers being able to upload files, particularly set-UID executables, into your `chroot()` area. A set-UID executable in the `chroot()` area could be combined with an unprivileged exploit to gain remote root privileges.

For example, if your anonymous FTP server allows file uploads and doesn't reset permissions on uploaded files, you might be allowing attackers to upload set-UID programs into your upload area. Now suppose you were also running a Web server on the same machine (generally a bad idea for exactly the reason we describe here). The attacker might have a buffer overflow exploit against the Web server. Normally this would give the attacker access as the UID the Web server was running as, but if the attacker can have their exploit trigger the set-UID binary they uploaded into your FTP area then the attacker can possibly get root privileges on your machine!

Similarly, if you allow user logins on your anonymous FTP server, then one of your users might upload a set-UID program into your FTP area and then log in under their own unprivileged UID to trigger the exploit. Later in the class, we will show you good ways to configure your anonymous FTP server to prevent this sort of thing from happening.

Also, it turns out that an attacker who manages to get root privileges while in the `chroot()` area may be able to "escape" from the `chroot()` restrictions. Let's look at this issue in more detail in the next few slides...

Give Up `root` Privs!

Process must run as superuser in order to `chroot()`

Process *must* give up root privs ASAP, or escape possible:

- Attacker creates devices in `chroot()` area
- Attacker uses `ptrace()` on other apps
- Attacker loads kernel modules

As we've already described, `chroot()` is a privileged system call and can only be run by the superuser. However, after the process has called `chroot()`, it's important that the process give up root privilege as soon as possible and operate as a normal unprivileged user. If the process is running with root privilege and the attacker was able to execute arbitrary code (via a buffer overflow or some other attack), then even a `chroot()`ed process running as root can do a lot of damage.

For example, the attacker could create copies of the system's disk devices in the `chroot()`ed area and have access to the file system (or a copy of `/dev/kmem` to access the kernel). The `ptrace()` system call would allow the attacker to control other processes on the system—i.e., ones which are not locked up by `chroot()`, but instead, have access to the entire system. Or the attacker could exploit loadable kernel modules and put code directly into the kernel (assuming they could get the malicious kernel module into the `chroot()` area).

This is one reason why anonymous FTP servers run as the `ftp` user and why TFTP daemons run as `nobody`.

Note that it is extremely important that the process UID be changed using the `setuid()` system call, which changes both the "real" and "effective" UID of the process. DO NOT use the `seteuid()` call (which only changes the "effective" UID of the process) because many Unix systems will allow the process to switch back to the "real" UID (root in this case) at will.

Breaking Out of `chroot()`

- Process cwd not in `chroot()` area:
  - Process didn't `chdir()` before `chroot()`
  - Attacker creates subdir to `chroot()` into
- Attacker forces process to `chdir("..")` to root directory
- Calls `chroot(".")` to get access to entire file system
- Exploit requires root privs operate correctly

There is another well-known mechanism for escaping from `chroot()` restrictions if you can get root access within the `chroot()` area. A complete write-up (with exploit code) of this mechanism can be found at:

http://www.unixwiz.net/techtips/mirror/chroot-break.html

The basic problem is that `chroot()` doesn't necessarily change the current working directory of the `chroot()`ed process. This means the attacker can force the process to effectively "`cd ..`" all the way up to the root of the file system. Once at the root of the file system, the attacker then calls `chroot()` on the current working directory. At this point, the attacker can now see the entire file system.

Even if the application developer is careful to change the working directory of the process to a directory someplace under the `chroot()` directory, the attacker can still make this exploit work. First, the attacker creates a subdirectory under the `chroot()`ed hierarchy (or uses an existing subdirectory that's already there). The attacker forces the process to `chroot()` to this subdirectory, leaving the current working directory of the process alone, which means the process' current working directory ends up outside of the `chroot()` restriction. Now we're back to the previous case of the attacker being able to walk backward up to the root of the real file system and call `chroot()` there.

However, this exploit requires the attacker to call `chroot()` to be successful—so if the program that the attacker has compromised has given up root privileges, then this "escape hatch" doesn't work.

Beyond `chroot()`

- Kernel prevents "double `chroot()`"
- Application whitelisting with SELinux
- Virtual system instances:
  - Xen/VMware
  - Containers/Docker

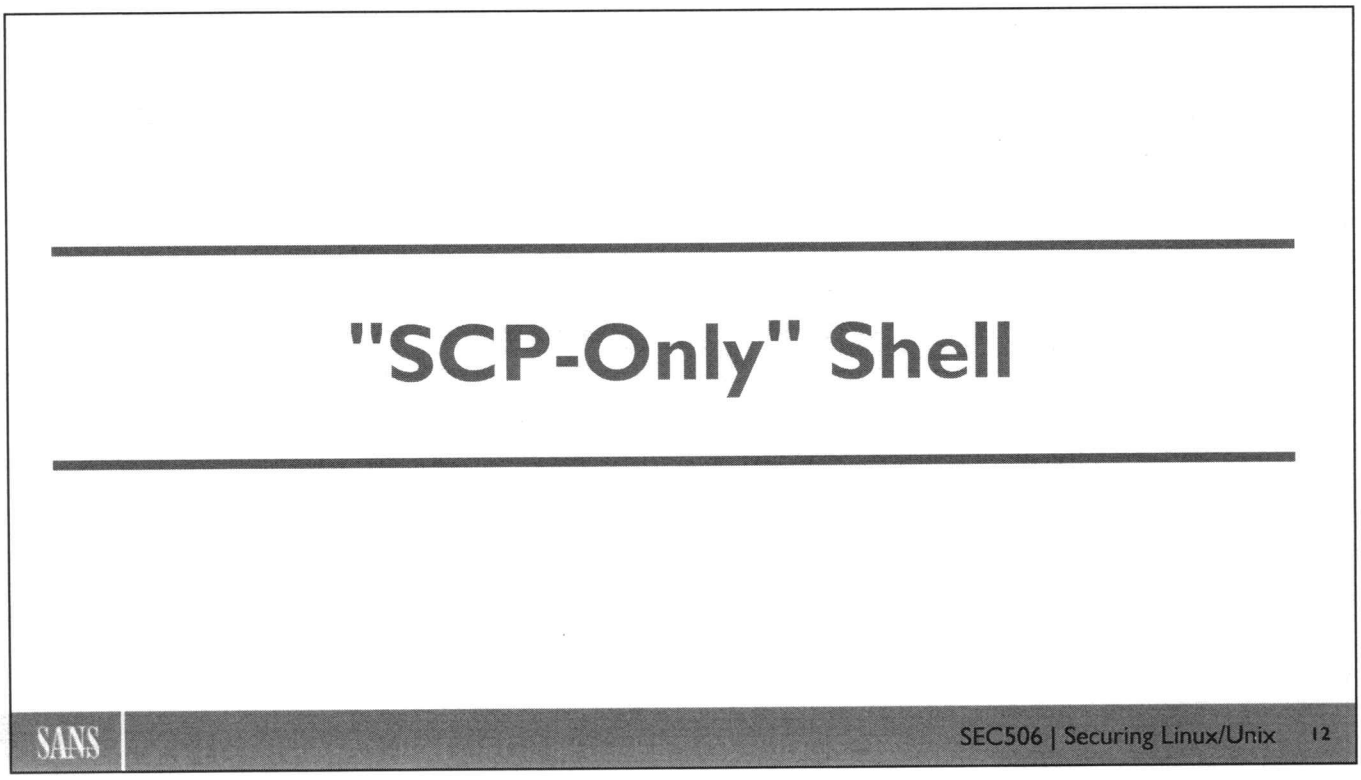
Since we've come to understand the limitations of `chroot()` pretty well at this point, various efforts have been made to either "shore up" know issues with `chroot()` or provide alternative security functionality to restrict access that compromised applications would have.

For example, the `chroot()` breakout exploit on the previous slide was possible because an application is allowed to call `chroot()` multiple times during its lifecycle. Perhaps an application should only be allowed to `chroot()` one time. This is exactly one of the restrictions that kernel hooks like the grsecurity kernel for Linux (*http://www.grsecurity.org/*) enforces along with other security restrictions.

Newer operating systems are starting to implement even more granular process rights restrictions in the kernel, for example, the SELinux hooks in newer Linux kernels (or similar functionality in the grsecurity patches or AppArmor) and the Privileges functionality in Solaris 10 and later. With this sort of functionality, you can define down to the level of individual files what objects in the operating system a given process may read, write, or execute. Fine-grained control over other system calls is also included. Perhaps if you develop a restrictive enough "white list" privilege model for a given process, then the `chroot()` call wouldn't be required at all.

The difficulty with these fine-grained kernel controls is that they can be extremely complicated to configure. While these tools generally support a "learning" mode that helps administrators build policies based on watching the normal behavior of the application, the resulting policy files can be complex and difficult to audit.

A final alternative would be to make use of the various different "virtual machine" technologies that have become available. This would be just like deploying your application on a stand-alone server, except that the "server" in this case is actually a software virtual machine running (possibly in parallel with other virtual machines) inside of a larger host operating system. Aside from the savings in hardware, running applications in virtual machines allows you to quickly recover from a compromise by shutting down the compromised image and simply "rebooting" the original "gold standard" image that was originally deployed (assuming you kept an off-line copy).



I mentioned in the last section that one of the reasons that we don't just `chroot()` every single application is that there's some fairly significant administrative overhead in setting up the directory structure that the `chroot()` will happen in. To give you an example of this process, I'm going to show you a useful little application called the SCP-only shell that can provide a more secure file transfer platform than anonymous FTP. The SCP-only shell normally is configured to run `chroot()`ed, so we'll need to go through the steps of setting up the appropriate directory structure for the app.

What is SCP-Only Shell?

- A more secure option to anonymous FTP
- Creates captive user accounts:
  - Can only be accessed with SFTP or SCP
  - Each user is `chroot()`ed to their own dir
- Implemented as a special shell in `/etc/passwd` entries

There are still a number of sites that are using FTP to exchange files with customers, business partners, and the like. It's also still used a lot in web-hosting environments as a mechanism to allow users to publish updates to their contents. The obvious problem, however, is that FTP is a clear-text protocol, so all those user credentials are susceptible to being sniffed off the wire. Plus, there's the session-hijacking threat to contend with.

The SCP-only shell is an SSH-based mechanism that gives you a lot of the features that people want out of anonymous or so-called "guest user" FTP servers, but in a more secure fashion because it's fully encrypted. The idea is that the SCP-only shell allows you to create "captive" accounts for users that only allow them to SCP or SFTP to a particular directory and which will never give them full interactive access on the server. In the typical implementation, each user is given their own password and a separate home directory for their own private use (though you could create shared accounts if you wish)—in this sense it's much more like "guest user" access on an FTP server rather than true wide-open anonymous FTP access.

It's called the SCP-only "shell" because you literally enable this functionality by setting the `scponlyc` binary as the user's default login shell in their `/etc/passwd` entry. Of course, there's some other setup you have to do as well …

Getting It Working

- Build package from source *[Easy]*
- Create `chroot()` directory *[Example]*
- Configure user accounts *[There's a trick]*

I'll also show you some automounter tricks that you might find helpful ...

The SCP-only shell is an Open Source package that's not normally bundled with your OS. So, you'll need to go to the SourceForge site for the project (*http://sourceforge.net/projects/scponly*) and download the code. The package comes with a typical `configure` script: I recommend you enable both the `--enable-chrooted-binary` option to turn on the chroot()ing option as well as the option `--enable-winscp-compat` which turns on support for SCP in addition to SFTP as well as adding backwards compatibility hacks for older versions of WinSCP (not strictly required anymore, but "be liberal in what you accept" applies here).

Once you've got the package compiled and installed, the hardest part of using the tool is setting up the directory structure that the `chroot()` is going to happen in. But I'm going to show you exactly how to do this so that you can get a sense of the process you go through to `chroot()` a typical application.

Once you've got the basic `chroot()` directory structure set up, you'll want to create a user account and test things out. However, the basic `chroot()` setup doesn't really scale well to the case where you have thousands of these accounts (if you're a web-hosting provider for example), so I'm going to show you some additional tricks to make this easier. More on this after we get the basic `chroot()` directory set up.

chroot() Dir Creation

1. Create basic directory structure
2. Copy binaries
3. Copy shared libraries
4. Make devices
5. Copy configuration files

No matter what application you're setting up a `chroot()` directory for, you still follow the same basic five steps:

1. Create the basic directory structure that the `chroot()` ed app is going to run in. Remember that from the perspective of the application this directory is the entire operating system in miniature. So, you end up creating copies of directories like `/dev`, `/etc`, `/usr/bin`, and so on. The files that you have to populate into those directories depend heavily on what the application needs in order to function.
2. So, the next step is to copy whatever binaries the `chroot()` ed application needs to function into your new directory structure. Sometimes it's obvious what applications are needed, sometimes you have to consult the documentation or the application source code, and sometimes you need to be more resourceful. More on that as we go.
3. Once you've copied the necessary binaries, you will typically need a number of shared libraries to make those binaries operable. Unix systems have a tool called `ldd` that can tell you the shared library dependencies for a given binary, but you get to manually copy those libraries into your `chroot()` directory.
4. The application may also need copies of certain device files in order to function. For example, network-oriented applications may need a copy of `/dev/tcp` in order to emit packets. So, you get to learn how to create device files too! Joy!
5. Finally, the app will also probably need copies of some configuration files—be they generic system configuration files from `/etc` or application-specific configuration files. Again, figuring out exactly which files are necessary can be tricky, but you can use the same techniques you would use to figure out which binaries or device files are created.

Let's work through a step-by-step example on the next several slides. The SCP-only shell is a fairly simple application to `chroot()` because it was designed with this functionality in mind. Other apps may have much more complicated dependencies, but at least this example will give you a taste of this process. By the way, the SCP-only shell comes with a script to set up the `chroot()` area for the application, but it doesn't do a very good job in my opinion of creating the most minimal possible set of dependencies.

Step #1: Create Directories

```
# mkdir -p /scponly/chroot
# cd /scponly/chroot
# mkdir -p dev etc lib usr/lib usr/bin \
      usr/libexec/openssh
# chown -R root:root /scponly/chroot
# chmod -R 111 /scponly/chroot
```


So first you need to pick a directory where the `chroot()` will occur and flesh-out the directory structure underneath that. The exact location of the `chroot()` directory generally doesn't matter—do whatever fits best with your site-specific disk layout policies. I often like to put `chroot()` directories in their own file system so that problems in the `chroot()` directory are less likely to impact the rest of the system (and vice versa).

Here we're going to `chroot()` into `/scponly/chroot`. Under that directory, we're creating a number of sub-directories to emulate a typical OS layout. Under normal circumstances, you'd probably end up creating these directories one at a time as you worked through the various application dependencies during the `chroot()`ing process, but having done this before I know what directories are required and I'm just creating them all at once. I'll be populating these directories as we move along through the later steps.

Notice that you can and should use very restrictive file directory ownerships and permissions—much tighter than the default OS permissions on these directories. Remember that the typical `chroot()`ed application will give up root privileges as soon as it `chroot()`s. So, you want to make as much of the directory structure as possible owned by root so that it cannot be modified by the user operating the `chroot()`ed app. We're also setting very restrictive permissions on all directories. Specifically, we're making the mode `111`—execute-only for all classes of users—which means you can access files and sub-directories under these directories but not write files or even get directory listings.

Obviously, we're going to need a directory for our remote user to upload and download files into. We'll look at creating this directory later when we start adding user accounts.

Step #2: Binaries

```
# cd /scponly/chroot
# cp /usr/bin/scp usr/bin
# cp /usr/libexec/openssh/sftp-server \
      usr/libexec/openssh
# chown root:root usr/bin/scp \
      usr/libexec/openssh/sftp-server
# chmod 111 bin usr/bin/scp \
      usr/lib/exec/openssh/sftp-server
```


Now I happen to know which binaries are going to be needed for SCP and SFTP—though the exact location of the `scp` and `sftp-server` binaries may vary from system to system. If you didn't know which binaries were required, you could read the OpenSSH source code, or possibly use the technique we'll be using in Step #4—there we'll be using it to figure out which devices are required, but it also works for all sorts of other application dependencies, including hard-coded binary pathnames.

In any event, once you've figured out what binaries you need, simply copy them from their normal locations in the file system to corresponding directories under your `chroot()` hierarchy. Again, we're setting very restrictive ownerships and permissions on these files.

Step #3: Libraries

- Run `ldd` against binaries from Step #2
- Merge shared library lists, copy into appropriate dirs
- Also include `libnss_compat.so.*`
- Shared libraries should be mode 555

SANS | SEC506 | Securing Linux/Unix 18

However, the programs on most Unix-like operating systems are dynamically linked—meaning they depend upon one or more shared libraries to function properly. These shared libraries must also then be copied into our `chroot()` ed directory structure.

You can determine what shared libraries are required for a given program by running the `ldd` command on the binary:

```
# ldd /usr/libexec/openssh/sftp-server
        libresolv.so.2 => /lib/libresolv.so.2 (0x00f80000)
        libcrypto.so.6 => /lib/libcrypto.so.6 (0x00b10000)
        libutil.so.1 => /lib/libutil.so.1 (0x00f9d000)
        libz.so.1 => /usr/lib/libz.so.1 (0x00224000)
[... and so on ...]
```

So, the trick is to run `ldd` on all of your binaries and create a single merged list of all the shared library dependencies. Then copy the relevant libraries into the appropriate directories under your `chroot()` directory. Note that in order to function, shared libraries must be installed mode 555 (read and execute for all). However, the directories the libraries live in can be mode 111.

One interesting note for Red Hat systems like our test server is that there seems to be an additional dependency on `/lib/libnss_compat.so.*`, which doesn't show up in the output of `ldd`. I'm assuming that one of the libraries output by `ldd` has its own dependency. Unfortunately, there's no "recursive" option to `ldd` to easily identify this dependency. You usually end up finding this because your `chroot()` ed app fails to work, and you'll end up doing some debugging—more on this process on the next slide.

Here's some shell code you might find useful for populating your shared libraries in a more automatic fashion. I'm doing some clever tricks with Perl to pull the shard library names from the output of `ldd` and using that output in a loop:

```
cd /scponly/chroot
for i in `ldd /usr/bin/scp /usr/libexec/openssh/sftp-server | \
                perl -ne '($l) = m|\s((/usr)?/lib/\S+) \(0x|;
                          print "$l\n" if ($l);' | \
                sort -u`; do
      cp $i .$i
done
cp /lib/libnss_compat.so.* lib
chmod 555 lib/* usr/lib/*
```

It may be possible to eliminate all shared library dependencies by building statically linked copies of `scp` and `sftp-server` from the OpenSSH sources. But this is way outside the scope of this course. Also, some operating systems like Solaris intentionally make it difficult to compile statically linked executables. For more information on creating statically linked binaries under Solaris see

http://www.deer-run.com/~hal/sol-static.txt

Step #4: Devices

```
# ls -l /dev/null
crw-rw-rw- 1 root root   1, 3 Jan 14 16:17 /dev/null
# cd /scponly/chroot/dev
# mknod null c 1 3
# chown root:root null
# chmod 666 null
```

SANS | SEC506 | Securing Linux/Unix 20

Next, the administrator has to create any system devices that the program requires. The easiest way to figure out which devices are required is to run the program under `strace` (or `truss` under Solaris, or whatever system call tracing tool is appropriate for your OS) and see what device files are being opened. While the output of a program like `strace` is huge and difficult to read, what typically happens is that the daemon process attempts to open a non-existent device and aborts. Somewhere in the last couple of dozen lines of output, you'll see an `open()` call that fails with `ENOENT` (no such file or directory) and be able to read which device was being `open()`ed. Here's some sample output to show you what I'm talking about:

```
# ps -ef | grep sshd
root      2253     1  0 08:57 ?        00:00:00 /usr/sbin/sshd
# strace -f -p 2253
[... gobs of output not shown ...]
[pid 15766] open("/dev/null", O_RDWR|O_LARGEFILE) = -1 ENOENT (No
    such file or directory)
[pid 15766] write(2, "Couldn't open /dev/null: No such"..., 50) = 50
[pid 15766] exit_group(1)                 = ?
Process 15766 detached
[... lots more output ...]
```

The first step is to figure out the process ID of the master SSH daemon. Then call `strace` with the `-f` flag (follow forked child processes) on that process ID ("`-p 2253`" in this case). Picking the relevant `ENOENT` out of this morass can be non-trivial, especially when using "`-f`", so I recommend redirecting the `strace` output to a file and `grep`-ing for `ENOENT`. Generally, the last `ENOENT` is the one that caused the app to bomb out, so that's the one you want to look at (not all `ENOENT` errors are fatal).

Once you've figured out the device you need to create, it's time to fire up the `mknod` command. But how do you figure out the correct options to pass to the `mknod` program in order to create the appropriate device(s) in your

`chroot()` area? The easiest thing to do is just run `ls -l` on the actual system devices: the *device type* is the first letter of each line of output (either a "`c`" or a "`b`" for device files) and the other two numeric arguments, the *major* and *minor device numbers*, are displayed where the file size would normally be shown (device files have no data blocks associated with them, hence no file size need be reported). You can even copy the default ownerships and permissions from the output of `ls -l`.

Of course, even though we've created the `/dev/null` device that the app is dependent on, there may be other application dependencies that we haven't found yet. So, you go back and try your application again, and it will bomb out someplace else. Again, you look at the `strace` output to figure out what file or device you're missing and update your `chroot()` area appropriately. You keep doing this until the application "works." And if that sounds like a tedious, time-consuming process, it certainly can be. However, this same process will also help you figure out other missing application dependencies, including configuration files (Step #5, coming up next), binaries, and shared libraries. Of course, in the case of a missing binary, the `ENOENT` will occur on an `exec()` call rather than an `open()`.

Step #5: Config Files

```
# cd /scponly/chroot/etc
# cp /etc/group .
# grep ^root: /etc/passwd >passwd
# chown root:root *
# chmod 444 *
```

SANS | SEC506 | Securing Linux/Unix 22

Finally, you'll need to create some "stub" files under `/etc`. For example, `passwd` and `group` files are required so that the `chroot()`ed `scp` and `sftp-server` commands can display user and group names rather than user and group numbers. In general, you don't need the complete contents of either file—though in this case, we're just copying in the entire `/etc/group` file because it's easier, but stripping most of the `/etc/passwd` entries.

By the way, *never* create an `/etc/shadow` file or put real encrypted password strings in the `chroot()`ed hierarchy since these configuration files are not used for authentication (the authentication happens long before the SCP-only shell invokes `chroot()`). Some security administrators have created "honey pot" machines with bogus password entries and then raised alarms when somebody tried to log into the system with one of the bogus username/password combinations, but frankly, there aren't enough hours in my day to pay attention to stuff like that.

Note that the configuration files in the `chroot()`ed directory should be world-readable but not writable.

Single User Setup

```
# groupadd scponly
# useradd -c 'Test User' -g scponly \
        -M -d /scponly/chroot//data \
        -s /usr/local/sbin/scponlyc testuser
# cd /scponly/chroot
# mkdir data
# chown testuser:scponly data
# chmod 750 data
# grep ^testuser: /etc/passwd >>etc/passwd
```


The only remaining chore is to actually create a user account that logs into the `chroot()` directory we just got done setting up and to create a directory under the `chroot()` area where this user can put files and take files from.

The `useradd` command on the slide creates a user called "`testuser`" whose shell (`-s`) is set to `/usr/local/sbin/scponlyc` (the trailing "`c`" means the `chroot()`ing version of the `scponly` binary). Note the slightly funky way were specifying the home directory (`-d`) for the user—the "`//`" marker in the directory name indicates the place where the `chroot()` should occur, and the remainder of the path gives the default directory to put the user into when giving them access. So, in this case, `scponlyc` will `chroot()` to `/scponly/chroot` and the user will start in the `data` subdirectory.

Of course, we also need to create that `data` directory and set appropriate ownerships and permissions so it's writable by the "`testuser`" account. It turns out that you also need to put a copy of the "`testuser`" password entry in the `passwd` file under the `chroot()` directory structure (that's what the `grep` command in the last line on the slide does).

Wait Just a Minute There!

- What if you have *thousands* of users?
- Maintaining individual `chroot()` areas would be painful!
- Use automount magic, tweaking original directory layout ...

OK, how many of you realized the problem with our last example? What if you were a web-hosting provider with literally thousands of users with SCP-only access? Would you create a separate `chroot()` directory for each user? Of course, you wouldn't—that would be a nightmare to maintain and a huge waste of disk space to have all those duplicate shared libraries everywhere.

Wouldn't it be cool if we could have a single copy of the basic `chroot()` directory structure but still have separate `data` directories for each user? Turns out that there is a way to do this if your automounter kung fu is strong. Let me show you how …

The Automount Trick

In `/etc/auto.master`:

```
/scponly/mounted  /etc/auto.scponly
```

The `/etc/auto.scponly` map:

```
*       /           :/scponly/chroot   \
        /data       :/scponly/data/&
```

Yes, this craziness actually works...

SANS | SEC506 | Securing Linux/Unix 25

In automount parlance what we're doing in the `/etc/auto.master` file is defining an *indirect map* underneath the `/scponly/mounted` directory. We're going to change our user home directory entries in `/etc/passwd` so that they point to directories under this file system, rather than using the raw `/scponly/chroot` directory structure like we were using earlier. The definitions of what happens under `/scponly/mounted` are defined in the `/etc/auto.scponly` configuration file.

`/etc/auto.scponly` uses lots of cool automounter tricks, including *wildcarding* (that's the "`*`" at the front of the entry), *multiple mounts* (notice that we're driving two distinct mount points out of our single wildcard entry), and *bind mounts* (we use just a "`:`" prefix to indicate we're mounting local directories on a different mount point rather than file systems from some other server over the network). Yow!

What this entry means is when you attempt to access any directory ("`*`") under `/scponly/mounted`, two mounts are done (and, yes, they happen in the order specified in the entry) and pieced together into a coherent file system:

1. First, the `/scponly/chroot` directory is mounted on top of the root of the requested file system. So, if the system tries to access `/scponly/mounted/testuser`, then `/scponly/chroot` is mounted as this directory.
2. But each user needs their own private `data` directory. We're going to create a new directory structure called `/scponly/data` which has individual user-specific data directories-- `/scponly/data/testuser` and so on. Our automount map uses the special "`&`" macro, which means substitute whatever directory name was matched by our initial "`*`" wildcard. So, if `/scponly/mounted/testuser` is requested then `/scponly/mounted/testuser/data` is actually mounted from the `/scponly/data/testuser` directory. Slick, huh?

By the way, my original thought had been to actually mount `/scponly/chroot` with the "`-ro`" ("read-only") option, to make it even harder for attackers. But unfortunately, with bind mounts like we're doing here, you're not allowed to set mount options that aren't set on the original file system. I suppose you could put the `/scponly/chroot` directory into its own file system and mount that read-only, but I'll leave this as an exercise to the reader.

Need to Tweak Dir Structure

- **`/scponly/chroot/data`** is now just a stub for automounting other dirs
- Physical directories for user accounts should be **`/scponly/data/`***<user>*
- **`/scponly/mounted`** will be created automatically

Of course, using the automount map from the previous slide means we've got to tweak up our original directory structure a bit. We need to create `/scponly/data` and relocate the `/scponly/chroot/data` directory we created for our "`testuser`" account to `/scponly/data/testuser`. It's important when we do this that we leave behind an empty `/scponly/chroot/data` directory so that the automounter has a mount point to place the user's `data` directory on top of.

We actually don't need to create the `/scponly/mounted` directory. The automounter will automatically create this directory for us as soon as we create the automount maps that I showed you on the previous slide and restart the automounter (`/etc/init.d/autofs restart` on your CentOS virtual machines).

New User Config Is a Snap

```
# useradd -c 'New User' -g scponly \
      -M -d /scponly/mounted/newuser//data \
      -s /usr/local/sbin/scponlyc newuser
# cd /scponly/data
# mkdir newuser
# chown newuser:scponly newuser
# chmod 750 newuser
# grep ^newuser: /etc/passwd \
     >>/scponly/chroot/etc/passwd
```


This new automount-based scheme makes adding new users very simple. Say we wanted to create an account called "`newuser.`" The home directory entry for this user needs to use the `/scponly/mounted/newuser//data` path (and we should go back and fix the home directory for our "`testuser`" account to use this scheme too). Then we need to create a data directory for this user under `/scponly/data`-- `/scponly/data/newuser` in this case—with appropriate permissions. Finally, we need to make sure the `passwd` entry for "`newuser`" gets populated into the shared `/scponly/chroot/etc/passwd` file. That's it! Three steps and you're done!

Summary

- SCP-only is a good substitute for anonymous FTP
- Hardest part is convincing users to use WinSCP
- The automount trick is useful when you have lots of directories to manage

You really need to be shutting off clear-text login and file transfer protocols in your enterprise, and the SCP-only shell is a good replacement for most places where you've historically been using "guest" or anonymous FTP servers. It's so easy to configure, that normally the biggest hurdle is getting your Windows users to start using WinSCP instead of their favorite FTP client. By the way, WinSCP supports regular FTP too, so once your users switch they can continue to have a single file transfer client—that client will just happen to be WinSCP.

If you are using OpenSSH 5.x or later, there is now a built-in mechanism for setting up SFTP access in a `chroot()` environment. This can be turned on specifically for a particular group of users while allowing regular users of the system to SFTP in as normal. The setup is considerably less complex than setting up the SCP-only shell. For more details, see *http://www.thegeekstuff.com/2012/03/chroot-sftp-setup/*

I took a lot of time in this section going into the details of my little automounter hack because that kind of configuration ends up being useful in lots of situations where you have multiple `chroot()` directories to manage that are essentially identical. This comes up more often than you'd believe.

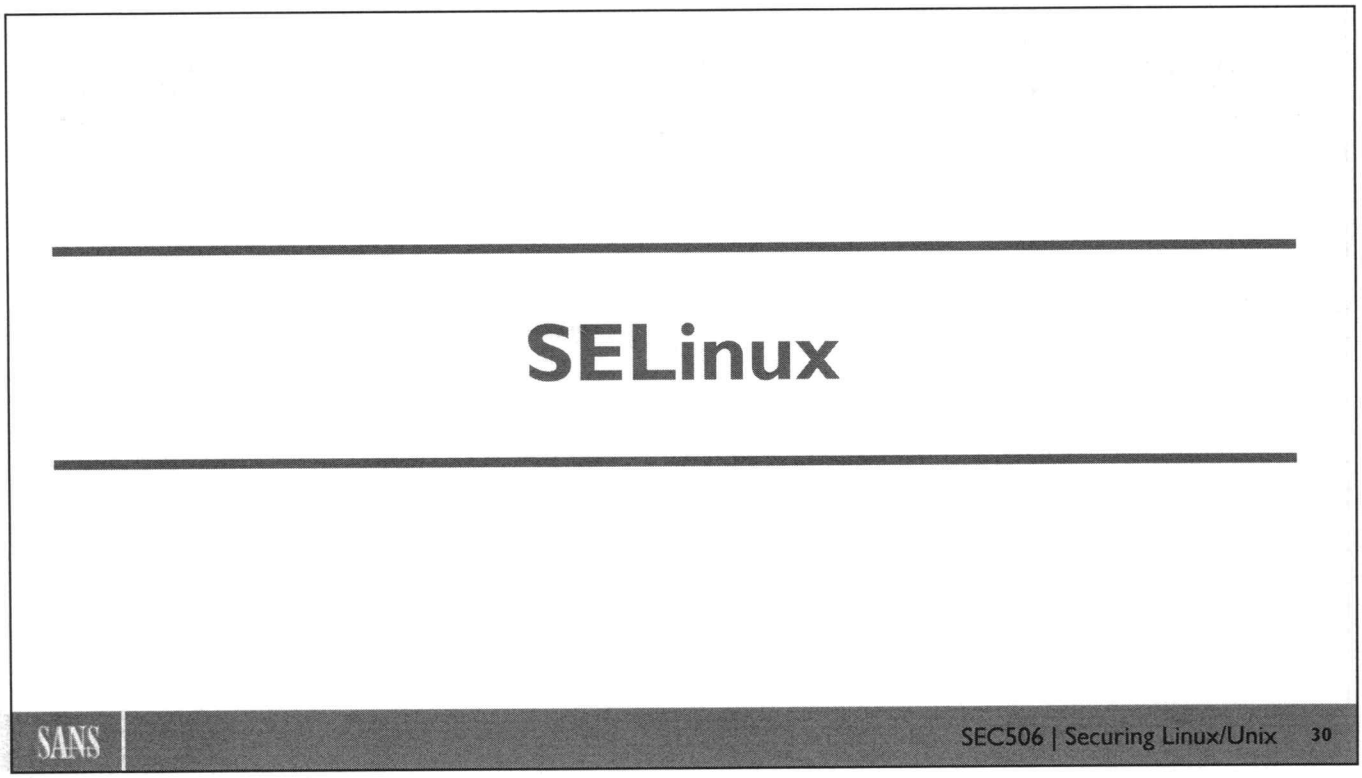
By the way, if you still think you need to set up an anonymous or "guest user" FTP server for some reason, there's an Appendix at the back of this book that goes into this in a great deal of detail. But really, don't do this to yourself—use SCP-only instead.

Lab Exercise

- Configuring the scponly shell
- **`chroot()`** and automounter games!

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 1, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 1
5. Follow the instructions in the exercise



For most people, their entire interaction with SELinux is turning it off during operating system installs. I think this is primarily due to a lack of understanding of the technology, which is not entirely surprising given the general lack of decent documentation out there. Hopefully, the information in this section will give you a clearer understanding of SELinux so you can at least make an informed choice before you turn it off. ☺

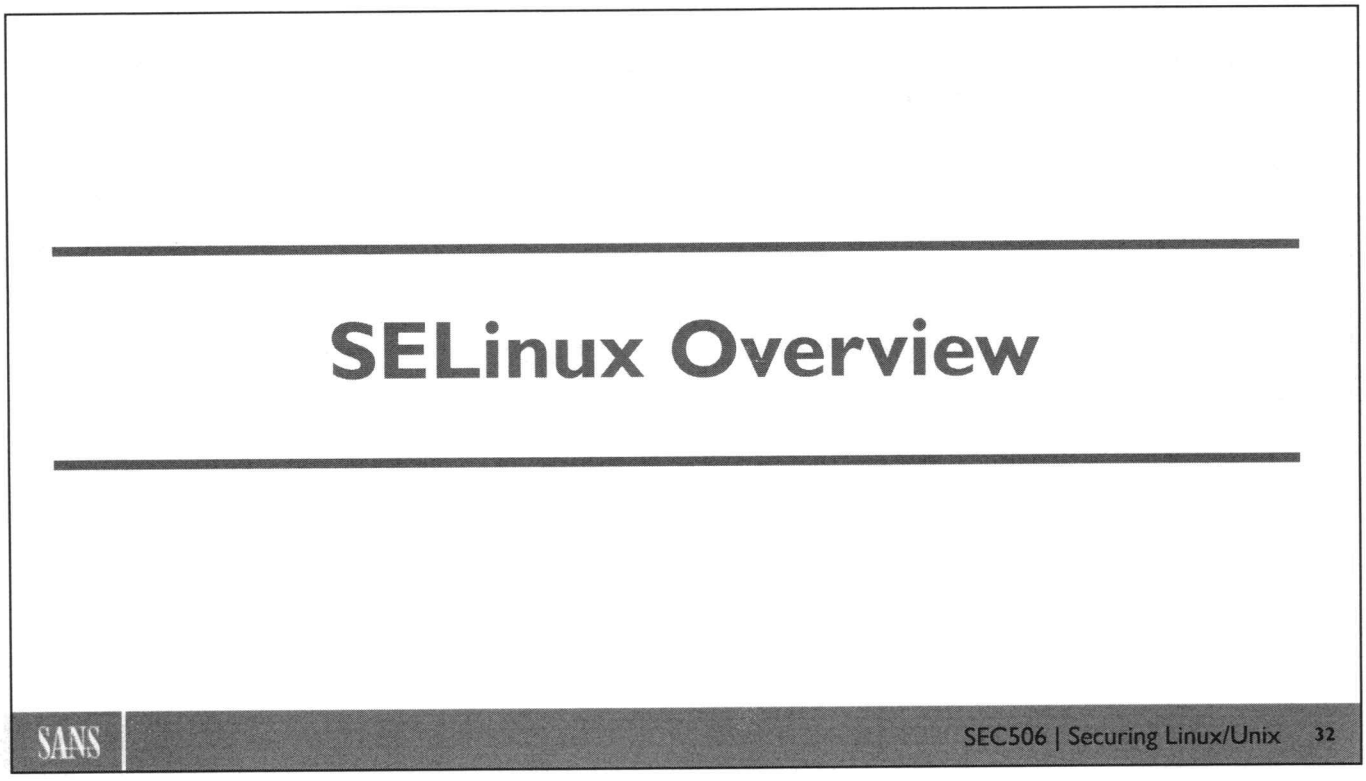
SELinux Topics

- Overview
- Common Troubleshooting Tasks
- Writing Policies from Scratch

The *Overview* section is a quick, high-level introduction to SELinux and some of the outermost administrative interfaces to the SELinux environment.

Next, we'll look at some *Common Troubleshooting Tasks* that tend to come up frequently in environments running SELinux, using some web server configuration examples.

Finally, in the *Writing Policies from Scratch* section, we'll actually work through a complete example of creating a brand new SELinux policy for a real network service.



First a general overview and high-level introduction to SELinux…

What Is SELinux?

Kernel-based access controls:

- MLS/MCS: think Secret, Top Secret, ...
- RBAC: fine-grained user privilege controls
- Type Enforcement: process whitelisting, app isolation

Type Enforcement is the most commonly used piece

SELinux is a very fine-grained set of access controls implemented in a kernel module. It actually covers several different types of functionality:

Multi-Level Security (MLS)/Multi-Category Security (MCS)—These sorts of security controls are typically required at high-security sites that need to rigidly partition different categories of data (unclassified, classified, secret, top secret, etc.) on their networks. Outside of these kinds of environments, however, this level of access control is not widely used or needed. Luckily, you can simply ignore this functionality in a typical SELinux configuration by simply not using it—all objects will be placed in the same minimum-security level by default and can freely interact with each other.

Role-Based Access Control (RBAC)—One historic problem with the Unix security model has been "all or nothing" administrative access via the root account. RBAC is an attempt to address this deficiency by allowing sites to assign specific aspects of administrative privilege to a number of different roles. For example, you might have a "network administrators" role that allows somebody to configure the network interfaces on the device, while the "printer admin" role allows somebody control the print queues. The end game would be to completely disable the root account and just assign people to specific roles based on their job function.

However, there are a couple of issues:

- Nobody has actually published a meaningful RBAC policy for a typical enterprise Linux environment—even a decent subset of one—and creating one from scratch is an enormous effort. Even if somebody did create and publish a straw-man policy, it might end up being too site-specific to be useful to other organizations.
- While different flavors of Unix support RBAC in their kernels, every vendor's implementation is completely different from a configuration syntax and administrative implementation perspective. So, if you spend a lot of time crafting an SELinux-based RBAC policy, you'd have to spend a bunch more time "porting" that policy to your Solaris, AIX, etc. systems.

This is why `sudo` still tends to be quite popular. It gives you 80% of the fine-grained access controls you'd get from RBAC, and it's both easier to configure and portable across multiple Unix variants.

Type Enforcement (TE)—Type Enforcement is essentially an application "whitelisting" facility. Type Enforcement policies attempt to specify exactly which components of the operating system (files and directories, devices, sockets, etc.) a given application needs to interact with and what level of access the application needs (read access, write access, etc.). If an application attempts to stray outside the bounds of the defined policy, the kernel refuses to grant access to the requested resource. More often than not, this will cause the application to fail.

The goal is to prevent an application that has been compromised by an attacker from misbehaving in ways that would allow the attacker to compromise the larger systems. So, Type Enforcement is really an "application isolation" strategy, similar to `chroot()`. However, where `chroot()` can really only control the application by limiting to a particular subset of the file system, Type Enforcement can control many other aspects of the application's behavior (access to network sockets, for example, or finer-grained access controls to specific files) and still allow the application to run in the default OS directory structure (meaning no need to set up a `chroot()` directory with copies of binaries, libraries, etc.). You could even run an application with a combination of `chroot()` and Type Enforcement restrictions, though most organizations see Type Enforcement functionality as a superset of the security controls they get in a typical `chroot()` environment.

These days, most sites are at most adopting Type Enforcement only and largely ignoring both MLS/MCS and RBAC. The most popular Type Enforcement strategy is the so-called "targeted policy" implemented by default in Red Hat Enterprise Linux (and also used by Gentoo Linux). As of RHEL5, the targeted policy covers nearly all standard OS daemons that can be accessed from outside the box and a number of other programs as well, while leaving normal user and internal administrative processes alone (thus reducing the pain of implementing SELinux in most environments).

The original [SELinux] policy published at the NSA was the strict policy. Its goal was to lock down the entire operating system, controlling not only the daemons that live in system space but controlling the user space as well. Strict policy ... adds the largest burden on users ...

During the development of Fedora Core 2, we attempted to use strict policy as the default policy. We had multiple problems with this because the strict policy was governing the way that users were running their systems. We had to cover all possible ways that a user would be able to setup their system. As you can imagine we had a ton of problems and bug reports. Most people when confronted with SELinux at that time, just wanted it turned off...

After our experiences with the strict policy, we went back and reflected on what our goals were. We wanted a system where the user was protected from System applications that were listening on the network.

These applications were the doors and windows where the hackers would enter the system. So we decided to target certain domains and lock them down while continuing to leave userspace to run in an unconfined nature. Targeted policy was born ...

From https://fedoraproject.org/wiki/SELinux/Policies

Alternate SELinux Universe

- *Objects* are anything in OS: files, devices, sockets, processes
- Objects have SELinux *contexts*—just labels chosen by us
- SELinux *policy* defines how an object in one context interacts with other objects

All of this is completely independent of normal Unix ownerships & privileges ...

When SELinux talks about *objects*, it just means something in the operating system—whether that's a file, a directory, a device, a network socket, a process, or what have you. Each object is assigned a label that describes the *context* associated with that object. These context labels are just a set of conventions established by the folks who developed the SELinux policy on your system. When you're developing your own policy for new applications, you also end up creating new labels to identify the pieces of the operating system that are specific to your application.

SELinux policy rules control how an application running in a particular context—for example, an Apache web server running in the "`httpd_t`" context—can interact with other objects in the operating system—files in the web docroot, for example, which have context type "`httpd_sys_content_t`" in the default targeted policy. A policy statement might say that processes in the `httpd_t` context may read files that have `httpd_sys_content_t` context. Anything not specifically permitted by the SELinux policy is denied, so if the policy only grants the web server read access to files in the docroot, then if the process tried to overwrite one of those files (web defacement) the kernel would block the write attempt, returning an error code to the process. Depending on how the application is coded, the process may catch the error and keep running, or terminate and give up.

It's important to understand that SELinux essentially exists as an "alternate universe" from the normal Unix ownership and permissions rules that you're used to. That being said, it's important to maintain good discipline around normal Unix ownerships and permissions, because again most sites only implement SELinux Type Enforcement on certain critical processes and rely on normal Unix permissions to control access for interactive user sessions.

"-Z" Shows Contexts

```
# ps -eZ | grep httpd
system_u:system_r:httpd_t:…  2773 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2775 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2776 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2777 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2778 ?  00:00:00 httpd
…
# ls -dZ /var/www/html
d… system_u:object_r:httpd_sys_content_t /var/www/html
# id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```


On a system that includes SELinux functionality, many of the standard OS programs have an added "`-Z`" flag which displays the SELinux context information.

The context label is actually a triple of "user", "role", and "type." But if you're not using RBAC, "user" and "role" generally don't matter. The "user" field tends to be either "`system_u`" (for system processes and other objects in the file system) or "`user_u`" (for normal users and their processes). Processes and users tend to get shoved into the role "`system_r`" by default, while static objects like files in the file system are labeled as "`object_r`" in general. It's the "type" field that's really used to discriminate between objects in the typical Type Enforcement targeted policy. Note that the `*_u`, `*_r`, and `*_t` suffixes are purely conventional—they're just put there to make it easier to interpret the labels.

You'll also often see a string like "`s0-s0:c0.c1023`" appended to each context. This is the MLS/MCS label for each object. The first part is a range of "sensitivity levels" (think *confidential*, *secret*, and so on) that can see the object—by default all objects in the OS are shipped at sensitivity level zero (the lowest level) so that they can be used by all users on the system. The "`c0.c1023`" is a range of "categories"—think "proprietary" and so on-- which when combined with the sensitivity label lets you say things like "*confidential and proprietary*" in you MLS/MCS policy. Again, the default here is "`c0.c1023`", covering the entire range of possible values. So once again any object on the system can interact with any other object without interference from the MLS/MCS policy.

Labels are inherited. If the `httpd` process runs a CGI script, that script will run in the `httpd_t` context unless the SELinux policy explicitly says otherwise (this is called a *transition* and does happen, as for example when `init` starts up a new daemon that wants to run in its own private context). Similarly, if you create a new file in a directory, it will tend to inherit the context of the parent directory by default.

You'll notice that our user session as reported by "`id -Z`" is labeled with "`unconfined_t`". The targeted policy has a generic rule that doesn't put any restrictions on objects in the `unconfined_t`. Effectively these objects, which include all user sessions and the processes they spawn, will not be constrained by SELinux at all, though they are still subject to normal Unix ownership and permission restrictions.

Is It Turned On?

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
# setenforce 1
# getenforce
Enforcing
```


For most people, the only question they have about SELinux is, "Is that (expletive deleted) SELinux running?" Actually, SELinux cannot only be enabled or entirely disabled, but also if enabled you have the choice between *permissive* and *enforcing* mode. *Permissive* means that the kernel will log violations of the SELinux policy for the machine but still allow the application to continue. Typically, this mode is used for testing and development of new policy. If the system is in *enforcing* mode, then the kernel will terminate applications that violate policy.

`sestatus` will tell you whether or not SELinux is enabled and what mode it's running in. You can switch between permissive and enforcing mode with the `setenforce` command: use "`setenforce 0`" (or "`setenforce Permissive`") to set permissive mode, "`setenforce 1`" (aka "`setenforce Enforcing`") turns on enforcing mode. `getenforce` will tell you which mode you're in.

Note that changes you make with `setenforce` will not persist across reboots. To make persistent changes, edit `/etc/sysconfig/selinux`, where you can choose to enable or disable SELinux and what mode to run in by default. If you're planning on changing between enabled and disabled states, you have to do it via a reboot.

SELinux Booleans

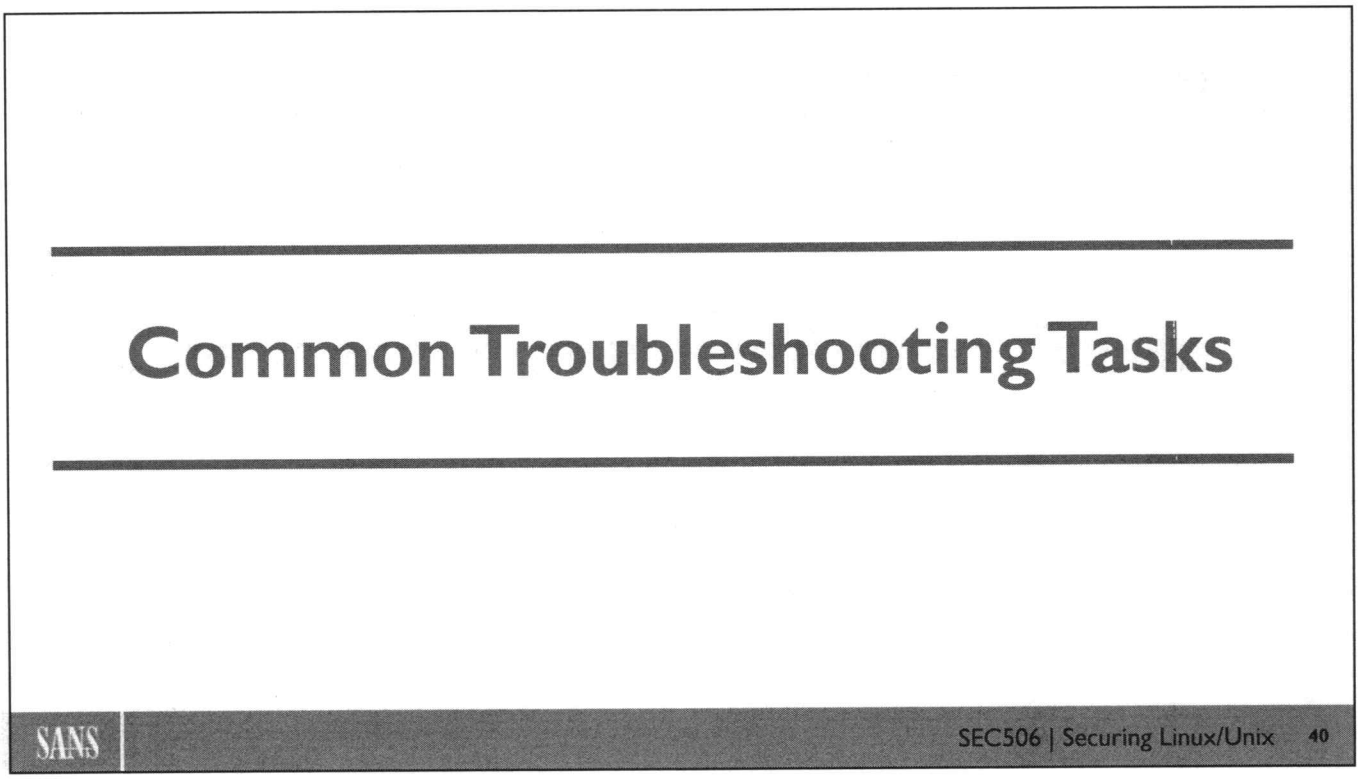
```
# getsebool -a | grep http
...
httpd_can_network_relay --> off
httpd_can_sendmail --> off
httpd_dbus_avahi --> on
httpd_enable_cgi --> on
httpd_enable_ftp_server --> off
httpd_enable_homedirs --> off
httpd_execmem --> off
...
```


When enabled, the standard targeted SELinux policy is enabled by default. However, the policy modules for many applications include optional functionality that can easily be enabled or disabled via SELinux *booleans*—basically simply on/off switches for the various bits of functionality.

You can get a list of all possible booleans and their current settings with "`getsebool -a`", or you can specify a single boolean to query as shown in the final example above. As you might expect, `setsebool` allows you to turn a given boolean on or off (1 for on, 0 for off). Note, however, that the changes you make with `setsebool` will not persist across reboots: use "`setsebool -P ...`" to make persistent changes.

In most cases, the name of the boolean pretty well describes what functionality it enables, particularly if you're familiar with the normal functioning of the application in question. However, if you're curious about exactly what each boolean enables, your only recourse is to get the source RPM for the SELinux policy for your system and look at the source code (the actual running policy files are stored in a binary format under `/etc/selinux` and are not human readable or easily reverse-engineered). Obviously, this is not an ideal method of documentation. More information on obtaining the SELinux policy source RPMs will be provided a little bit later in this module.

In the targeted policy that was implemented in RHEL5, each application had a `*_disable_trans` boolean associated with it. The idea was that when you turn these booleans on, it was supposed to allow the application to violate the SELinux policy even when the system as a whole was in enforcing mode. Thus, they were supposed to be a temporary escape mechanism to test new functionality or quickly work-around SELinux-related problems in production. Unfortunately, in practice, the `*_disable_trans` booleans didn't really end up working. In fact, these booleans have been completely removed from recent versions of the targeted policy and you won't find them at all in RHEL6 and later.



Now that we've oriented ourselves a bit in this new SELinux universe, let's look at some of the kinds of problems you will run into when you start using SELinux. This will also allow us to introduce some new commands for controlling objects in an SELinux environment.

Some `httpd` Examples

Move docroot to new partition:

- *Setting proper SELinux context on new dir*

Using an alternate port number:

- *Granting access to network ports*

SANS | SEC506 | Securing Linux/Unix 41

While the default SELinux policy works OK with the default configuration as provided by your OS vendor, the minute you start changing things you start creating SELinux policy violations. On enforcing systems, this means your application gets terminated by the kernel. At this point, most people give up and disable SELinux because they don't know how to troubleshoot and fix the problem. So, let's work through some common kinds of failures.

The first kind of failure happens when you try to use a different directory configuration than the default—like when you try to get your web server to use an alternate document root. The normal problem here is that the alternate directories you're using don't have the proper SELinux context labels on them, so access is not permitted by the standard policy. The fix is to put the right labels on the new directories, and I'll show you how to do that.

Another common failure mode happens when you try to use an alternate port number for a service—for example trying to get your web server to use an alternate port like 8888/tcp. Just like files and directories, sockets have contexts as well and you have to make sure the port you're planning on using is associated with the proper context. So, we'll also get to see how to manage contexts on port objects.

However, there are some problems that are not easily solved simply by relabeling. In these cases, we actually have to extend the default policy with our own rules. We'll look at these tools in the final section when we look at creating our own policies from scratch.

Docroot of Doom!

The scenario:

- You decided to create `/docroot` dir
- Updated `httpd.conf`
- Added some content

Mysterious errors when you restart Apache

Must be an SELinux problem, right?

SANS | SEC506 | Securing Linux/Unix 42

So, suppose you decided to create a new `/docroot` directory to hold your web content. After creating the directory, you update your `httpd.conf` file appropriately and restart the server. In RHEL5, the server would bomb with a mysterious error message. Looking in your Apache error logs, you see:

```
[notice] SIGHUP received.  Attempting to restart
Syntax error on line 281 of /etc/httpd/conf/httpd.conf:
DocumentRoot must be a directory
```

In RHEL6 and later, the web server starts up, but when you go to access any document under `/docroot` you get a `"Permission denied"` error:

```
[Wed Jun 10 16:14:54.732335 2015] [core:error] [pid 15647]
(13)Permission denied: [client ::1:35764] AH00035: access to
/index.txt denied …
```

Now you're a little crazed because you can do an "`ls -ld /docroot`" and see pretty clearly that it's a directory and the permissions are set correctly. And all the files under `/docroot` are world-readable.

You're pretty sure that this problem must have something to do with that wacky SELinux thing, but how can you be sure?

Is It Really SELinux?

```
# ausearch -f /docroot
----
time->Wed Jun 10 16:14:54 2015
...
type=AVC msg=audit(1433967294.731:635): avc:  denied
{ getattr } for  pid=15647 comm="httpd"
path="/docroot/index.txt" dev="dm-0" ino=264552
scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0
tclass=file
```

Seems like it could be an SELinux issue ...

SELinux uses `auditd` for logging events. That means we can use our old friend `ausearch` to find out if we have any events related to the `/docroot` directory:

```
# ausearch -f /docroot
----
time->Wed Jun 10 16:14:54 2015
...
type=AVC msg=audit(1433967294.731:635): avc:  denied  { getattr }
for  pid=15647 comm="httpd" path="/docroot/index.txt" dev="dm-0"
ino=264552 scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=file
```

Obviously, these audit logs are not designed with readability as the foremost goal (actually if you're on the console using the GUI, these SELinux alerts will also pop up from time to time and you can use the GUI-based tools to get more detailed human-readable output). But you can puzzle out what's going on here once you've been exposed to the madness for long enough:

1. First, we see that access is being "`denied`" to "`getattr`". "`getattr`" is short for "get attributes", and corresponds to trying to `stat()` a file or directory to figure out whether it exists and/or what kind of object it is.
2. The process triggering the denial is "`httpd`" (pid `15647`).
3. It's trying to access "`/docroot/index.txt`" (inode `264552` on `/dev/dm-0`) which is a file ("`tclass=file`")
4. The context of the `httpd` process (the source context, "`scontext`") is `...:httpd_t` and the context of `/docroot` (the target context, "`tcontext`") is `...:default_t`.

Other `ausearch` options that might be useful in these cases include:

```
ausearch -c httpd      # search for entries where the "command" matches httpd
ausearch -se httpd     # search for entries where the SELinux context matches httpd
ausearch -m AVC        # search for SELinux alerts with "type=AVC"
```

`aureport -a` can also be useful to see a summary of `type=AVC` alerts. You can then use `ausearch -a` to get more details about specific events:

```
# aureport -a

AVC Report
============================================================
# date time comm subj syscall class permission obj event
============================================================
1. 05/14/2015 10:06:54 ? system_u:system_r:init_t:s0 0 (null) (null) (null)
unset 453
...
8. 06/10/2015 16:14:54 httpd system_u:system_r:httpd_t:s0 4 file getattr
unconfined_u:object_r:default_t:s0 denied 634
9. 06/10/2015 16:14:54 httpd system_u:system_r:httpd_t:s0 6 file getattr
unconfined_u:object_r:default_t:s0 denied 635
# ausearch -a 635
----
time->Wed Jun 10 16:14:54 2015
type=SYSCALL msg=audit(1433967294.731:635): arch=c000003e syscall=6
success=no exit=-13 a0=7fba85170f50 a1=7fff0efc0e40 a2=7fff0efc0e40 a3=0
items=0 ppid=15645 pid=15647 auid=4294967295 uid=48 gid=48 euid=48 suid=48
fsuid=48 egid=48 sgid=48 fsgid=48 tty=(none) ses=4294967295 comm="httpd"
exe="/usr/sbin/httpd" subj=system_u:system_r:httpd_t:s0 key=(null)
type=AVC msg=audit(1433967294.731:635): avc:  denied  { getattr } for
pid=15647 comm="httpd" path="/docroot/index.txt" dev="dm-0" ino=264552
scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=file
```

So How Do I Fix This?

- Server works fine with the default document root (`/var/www/html`)
- Need to set the same SELinux *context* on our new document root directory
- Time for some more new commands...

We know that the SELinux policy allows Apache to operate fine with the default document root—`/var/www/html` on Red Hat systems. That implies that if we could just copy the context that's set by default on `/var/www/html` and apply it to our new document root then things would probably start working.

We've already seen "`ls -Z`" to display the current context of a file or directory. But now we need to learn how to change context labels on file system objects…

Lab Exercise

- Troubleshooting SELinux Part I
- Try it, you might like it ...

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 2, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 2
5. Follow the instructions in the exercise

Setting File/Directory Contexts

Two-step process:

- First use "`semanage fcontext ...`" to update policy
- Then "`restorecon -FR ...`" to fix contexts

Files created later automatically inherit context of parent dir

Let's start by using "`ls -Z`" to list the contexts on `/docroot` and `/var/www/html`. Actually, we already know the context on `/docroot` because it was the "target context" in the `audit.log` output, but let's see it again side-by-side:

```
# ls -Zd /var/www/html /docroot
d... root root unconfined_u:object_r:default_t:s0 /docroot
d... root root system_u:object_r:httpd_sys_content_t:s0 /var/www/html
```

OK, we need to set the context "`system_u:object_r:httpd_sys_content_t`" on `/docroot` and all the files underneath that directory. Just like `chown` and `chmod`, there's a `chcon` command for changing SELinux contexts on files and directories. However, the changes you make with `chcon` are not persistent. So, in general, `chcon` is not the way to go here (though it's occasionally useful for making quick changes for testing purposes).

In order to make useful changes, you have to create a new entry in SELinux's internal file context policy table. We do this with "`semanage fcontext ...`":

```
# semanage fcontext -a -t httpd_sys_content_t '/docroot(/.*)?'
# semanage fcontext -l | grep docroot
/docroot(/.*)?  all files  system_u:object_r:httpd_sys_content_t:s0
```

The first command above adds ("`-a`") a file context entry for '`/docroot(/.*?)`'—a regular expression that matches the `/docroot` directory itself and everything underneath it. In this case, we're specifying the default type attribute ("`-t`") "`httpd_sys_content_t`" for this directory and everything underneath. Since we didn't

specify the user attribute or the role portions of the label, these properties were set to sensible default values. You can see this when we use "`semanage fcontext -l`" to list the file context table entries. Note that if you make a mistake when you add a file context entry, you can delete it by using "`semanage fcontext -d ...`"—you use exactly the same syntax and all the same arguments you used when you did the "`-a`" version to add the rule initially.

While "`semanage fcontext ...`" establishes a default policy for a given directory, it doesn't actually change the existing context labels. The preferred method for changing the labels is the `restorecon` command, which looks at the defaults in the file context table and applies the appropriate labels to the directory you specify:

```
# restorecon -FRvv /docroot
restorecon reset /docroot context user_u:object_r:default_t:s0-
>system_u:object_r:httpd_sys_content_t:s0
restorecon reset /docroot/index.txt context user_u:object_r:default_t:s0-
>system_u:object_r:httpd_sys_content_t:s0
```

Here we're using the recursive ("`-R`") option to make sure we adjust the context labels for `/docroot` and all our content beneath it, and the verbose ("`-vv`") option so we can see the changes as they're being made. "`-F`" forces `restorecon` to change the user and role portions of the context along with the type. We can confirm that our changes fixed the problem by attempting to access documents from our web server. You'll find that our changes fix the issues we were having.

One more useful tidbit: suppose you've been making massive file system changes and you want to make sure that the file contexts on all directories reflect the default settings in the file context table. This often comes up after you've done a full restore of your file system, because many backup utilities may not capture the SELinux context information associated with files and directories. If you "`touch /.autorelabel`" and then reboot the system, then during the boot process the system will do a "`restorecon -R -F`" on all local file systems. Doing this during a reboot is recommended because it will be done very early in the boot process before any of the normal OS daemons have started—changing contexts on files and directories used by a process while that process is running can yield weird behavior. Note that doing a `restorecon` on the entire file system can take several minutes, so this isn't something you want to do all the time (at least if you value your uptime numbers).

Alternate Port Number? SELinux Says, "FAIL!"

The scenario:

- You want a dev server on port 8888/tcp
- Change `httpd.conf` and restart
- SELinux terminates your process

The fix:

- Figure out context for `httpd` ports
- Add your new port to this context

Now let's look at how SELinux controls access to socket and port objects. Suppose you decided you wanted to run your web server on port 8888/tcp. After updating your `httpd.conf`, you attempt to restart the server and it gets killed due to an SELinux policy violation (again confirmed via the `audit.log` file).

As was the case with our earlier docroot example, the web server worked fine when it was bound to port 80/tcp. So, there must be some SELinux context associated with 80/tcp that allows the web server to bind to that port. Clearly, this context is not associated with 8888/tcp by default. Can we change that? Of course!

Dealing with Port Contexts

```
# semanage port -l | grep ' 80,'
http_port_t  tcp  80, 443, 488, 8008, 8009
# semanage port -a -t http_port_t -p tcp 8888
# semanage port -l | grep ' 8888,'
http_port_t  tcp  8888, 80, 443, 488, 8008, 8009
```

Cannot reduce default port list without changing and recompiling default policy!

It turns out that we also use the `semanage` command to control port contexts, and you can see from the example on the slide that things are not much different from when we were using it to manage file contexts.

First, we look for port 80 in the output of "`semanage port -l`" so we can figure out the context name we need to associate with our new port 8888/tcp. Note that I'm `grep`-ing for "<*space*>`80`<*comma*>" here so that I match only port 80 in the output, and not ports like 8880, etc.

Once we figure out that the context we want to use is `http_port_t`, we can use "`semanage port -a ...`" to add our new port to the list of ports associated with this context. Frankly, I wish the "`-p`" option took a sane argument like "8888/tcp", but I wasn't consulted when the command was being created. So, you're left with the kind of backward command syntax you see in the example above.

We can confirm our changes by using "`semanage port -l`" again. You'll notice that there are a bunch of other port numbers associated with the `http_port_t` context. If you weren't ever planning on using those ports, might try using "`semanage port -d ...`" to remove those ports from the list, thus further tightening the policy constraints on your web server. And when you try to do that, you see the following:

```
# semanage port -d -t http_port_t -p tcp 8009
ValueError: Port tcp/8009 is defined in policy, cannot be deleted
```

Yep, that's right. Removing these default ports would actually require modifying the source policy! Frankly, this is a bit of a bug.

Lab Exercise

- Troubleshooting SELinux Part II
- How about another helping?

SANS | SEC506 | Securing Linux/Unix 51

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 3, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 3
5. Follow the instructions in the exercise

Policies from Scratch

Our last example gave you a taste of using `audit2allow` and creating some SELinux policy. But what if you weren't just extending the existing policy for an application, but instead had to create an entirely new policy for a brand-new application not currently covered by the default targeted policy. That gets a whole lot more interesting…

Prerequisites

Use the *SELinux Reference Policy*–

- Creates conventions for shared resources
- Provides macros for common policy rules
- Unfortunately, documentation is lacking

Need `selinux-policy-devel` RPM

Also want `selinux-policy` *source RPM* installed

In the bad old days, you had to create raw SELinux policy from scratch. Imagine how awful that must have been. Consider a common shared service like Syslog—not only did you have to create a policy that allows Syslog to do all the things it needs to do (but not allow it to do too much) but then you had to negotiate access to all of the locations in the OS that are shared with other applications (`/etc`, `/var/log`, and so on). Plus, every other application that talks to Syslog would need to share the same set of allow rules to make that happen. Change any single piece and your changes would have to ripple out to the rest of your policy. It's a nightmare.

So, the idea for the Reference Policy was born. The Reference Policy establishes naming and usage conventions for shared resources in the operating system. It also creates macros for common access needs—sending log messages to the Syslog daemon is an example of one such macro. The bad news is that the Reference Policy is still actively being developed, which leads to two problems:

1. It's still changing rapidly. This means that from release to release you may find that policy files need to be updated to reflect changes to the Reference Policy macros. This is particularly a problem when transitioning between major RHEL releases—years of policy development have happened from one release to another.
2. The documentation is sketchy… at best. Really the best way currently to learn about the Reference Policy is to read the source.

So, we need to get a copy of the source. The current development site for the Reference Policy is *https://github.com/TresysTechnology/refpolicy/* and you can find the download link there. But if you're using RHEL, then you're better off getting the source for the version that Red Hat used.

Happily, Red Hat makes source RPMs available for all of the Open Source software included in its releases. You'll find the source RPMs at Red Hat's site or your favorite CentOS mirror in a location like:

http://vault.centos.org/<osvers>/updates/Source/SPackages//selinux-policy-<vers>.src.rpm

When you install the source RPM it unpacks itself into `/root/rpmbuild/SOURCES`. You'll find the files on this path in the virtual machine I gave you at the start of class. The file that actually contains the source code you want to look at is the `serefpolicy-`*<vers>*`.tgz` tar file. There's also `serefpolicy-contrib-`*<vers>*`.tgz` that contains sample policy files you can borrow ideas from. You can unpack these files anywhere in the filesystem that is convenient.

In addition to the files in the source RPM for your reference material, you also must have the `selinux-policy-devel` RPM installed. This RPM includes the sample policy files, but most importantly a `Makefile` that will make the whole compilation process a lot easier.

Our Target: PortSentry

- Daemons started at boot time
- Binary in `/usr/local/sbin`
- Config files in `/etc/portsentry`
- Log files in `/var/log/portsentry`
- Binds to arbitrary network ports
- Talks to Syslog

With the prerequisites out of the way, we're actually going to create a meaningful policy file for a real network service. For our example, we're going to create a policy for the PortSentry HIDS (see the Appendix for more information on installing and using PortSentry). PortSentry is an "interesting" example because it's started at boot time like other system services, has a configuration file in `/etc`, binds to network ports, talks to Syslog, and writes its own log files in a private directory. Basically, it covers all of the different kinds of access you're likely to run into for your own applications, and best of all there's no existing policy module for it. Yay!

The SELinux policy creation process rewards those who follow standard file layout conventions, so we're going to do that. We'll put the `PortSentry` daemon binary in `/usr/local/sbin`, tell it to look for its configuration files under `/etc/portsentry`, and to write its logs to `/var/log/portsentry`. This will minimize the impact of the new software on the existing SELinux policy and make our jobs a whole lot easier.

It also helps to understand the kinds of access your application is going to need before you start crafting policy. You won't be able to predict everything up front, but the more you can "front load" the policy creation process, the less time you'll spend iterating to your final policy. In this case, we know that PortSentry is going to need to talk to Syslog and that it's going to need access to some network ports. However, unlike most applications that will typically only bind to a single port, PortSentry's behavior is to bind to a large number of essentially arbitrary ports. This is going to make the task of crafting SELinux policy in this area a little bit abnormal for our PortSentry policy, at least as compared to the network access sections for other services.

Policy Creation Overview

1. Create working dir, skeleton policy files
2. Install initial policy module
3. Set file contexts for application files
4. Start and use application
5. Capture violations from `audit.log`
6. Update policy as appropriate
7. Repeat from Step #4 until "done"

Before we dive into creating our policy for PortSentry, let's talk about the general process of developing a new policy module. First, you need to create a working directory where you'll be developing your policy file. This directory will have a copy of that `Makefile` from the `selinux-policy` RPM that I mentioned earlier, and it's also where you'll have the various input files you'll be compiling into policy.

You typically start with a minimal policy file where you define the kinds of access you know the application will need. That basic policy gets compiled and loaded into our policy database. Once the baseline policy is loaded, you monitor your `audit.log` file for violations and then use a tool called `audit2allow` to help generate additional rules for your policy based on the `audit.log` entries. You then compile an updated version of your policy module, load it, and repeat the process. You know you have a working policy when you stop seeing alerts in the `audit.log` file.

In case it wasn't already obvious, this is normally a process you would do on a non-production system with SELinux in permissive mode. Permissive mode means that SELinux will allow the application to continue running so you see everything it's trying to do to the system and thus can generate the appropriate rules. Another reason to use a non-production system is that there are points where you're going to want to reboot the machine to both make sure the application is running in the correct context as well as to understand how the application behaves when the system is coming up.

Don't worry. Once you've figured out the policy in your test environment, it can just be installed directly on your production systems (assuming your test environment mirrors your production environment closely).

Initial Setup

```
# mkdir -p /root/selinux/portsentry
# cd /root/selinux/portsentry/
# ln -s /usr/share/selinux/devel/Makefile .
# cp ~sans/Misc/selinux/portsentry/initial-policy/* .
# ls
Makefile  portsentry.fc  portsentry.if  portsentry.te
# make
Compiling targeted portsentry module
/usr/bin/checkmodule:  loading … tmp/portsentry.tmp
/usr/bin/checkmodule:  policy configuration loaded
/usr/bin/checkmodule:  writing … tmp/portsentry.mod
Creating targeted portsentry.pp policy package
rm tmp/portsentry.mod.fc tmp/portsentry.mod
```


Your working directory can live anywhere in the file system you want. I generally keep mine under `/root/selinux`. If you're doing a lot of policy development at your site, consider using a source code control system like Subversion.

Once you have your working directory, copy the `/usr/share/selinux/devel/Makefile` into it, or make a link to this file as I'm doing in the example above. You'll also need a copy of your initial policy files. I've provided some sample policy files in your VMware image and on your course CD, so you can just start with those. We'll look at these files in a lot more detail on the next slide.

When you're ready to compile your module, just run `make`. The `Makefile` we're using takes care of running `checkmodule` and `semodule_package` for you, which is extremely convenient. The `Makefile` also runs `m4` on your policy files to expand the pre-defined macros from the Reference Policy into actual SELinux rules. If you want to see what your policy looks like after the macros have been expanded, look at the file `tmp/portsentry.tmp` in your working directory. There are a lot of comments and blank lines in this file, so if you want to just see the policy statements you can do something like:

```
# grep -v -E '^\s*(#.*)?$' tmp/portsentry.tmp
module portsentry 1.0.0;
require {
…
# grep -v -E '^\s*(#.*)?$' tmp/portsentry.tmp | wc -l
1329
# wc -l portsentry.te
26 portsentry.te
```

As you can see in this case, there's an approximately 50x increase in the number of lines from your macro-based policy file to the final version. This is one of the reasons why it's much nicer to write your policies from the macros in the Reference Policy.

About Policy Files

`portsentry.te`:

- Contains initial straw-man policy
- Save time: do as much as possible up front

`portsentry.if`:

- Put your own custom macros here (if any)

`portsentry.fc`:

- File context definitions go here
- What we did manually with `semanage`

SELinux policy module "packages" are generally created from three different files. The `*.te` file is the main file where you'll put your policy rules. The `*.if` file is where you're supposed to put any macros that you create specifically for your own policy. You could put these in the `*.te` file if you wanted I guess, but it keeps things cleaner to put the macro definitions someplace else.

Finally, the `*.fc` file is used to hold file context definitions that apply to your application. These look a lot like the configuration we did earlier with "`semanage fcontext ...`". The advantage to having these context definitions compiled into your policy package is that they will be automatically loaded into the file context table when you load the policy package and you won't have to muck around with `semanage` yourself. This is one of the reasons the module package format was created in the first place.

The next several pages will cover the contents of our initial straw-man policy files and introduce you to some of the common Reference Policy macros and other syntax elements…

First let's look at our initial `portsentry.te` file:

```
policy_module(portsentry, 1.0.0)

#### Declarations

type portsentry_t;
type portsentry_exec_t;
init_daemon_domain(portsentry_t, portsentry_exec_t)

type portsentry_etc_t;
files_config_file(portsentry_etc_t)

type portsentry_log_t;
logging_log_file(portsentry_log_t);

# You need these lines if you're using an older init-based Linux
#type portsentry_initrc_exec_t;
#init_script_file(portsentry_initrc_exec_t)

#### Policy

files_search_etc(portsentry_t)
files_search_var_log(portsentry_t)

logging_send_syslog_msg(portsentry_t)

miscfiles_read_localization(portsentry_t)
```

First, we use the `policy_module()` macro to declare our policy name and version number. Then we declare a number of context types that will be used by our application. In general, the prefix for all of the types you define should match the policy name from your `policy_module()` declaration—`portsentry_*` in our case. The extra suffixes like `*_exec_t`, `*_etc_t`, and `*_log_t` are also conventions used throughout the Reference Policy (other common types include names like `*_lib_t` , `*_tmp_t`, and `*_content_t`).

For each new type we define, there will typically be some associated macro declarations. For example, `init_daemon_domain()` is commonly used in policy files for daemons that are started at boot time. This macro defines (among other things) a *context transition* that happens whenever `systemd` or `init` runs an executable that belongs to our `portsentry_exec_t` context. The newly started process will be put into the `portsentry_t` context rather than inheriting the default context from the boot system. This transition is important so that we can write policy rules that only apply to our `portsentry_t` processes and not any other processes running on the system.

Other macros that are commonly used with our type declarations include `files_config_file()` and `logging_log_file()`, which generate the appropriate type declarations in your policy file for different types

of files. It's important to note that these declarations don't actually grant any kind of access to these types of files, they merely handle the grunt work of properly declaring the file types. We'll get to access controls in later versions of our policy.

We know our configuration files are going to live under `/etc` and our log files will be written under `/var/log`. In order to be able to even access files in these directories, we need SELinux to give us access to them. That's the purpose of the `files_search_*()` macros. `files_search_etc()` expands to an allow rule that permits read-only access to the `/etc` directory for and programs running in `portsentry_t` context. `files_search_var_log()` is actually not a pre-defined macro in the Reference Policy—we'll be creating it in our `portsentry.if` which I'll be showing you in a moment.

Finally, we end with some macros that will be common to nearly every policy file you write. You'll use `logging_send_syslog_msg()` in the policy file for any application that needs to communicate via Syslog. Similarly, most applications need to access the various language locale files in the system, so you'll want to use `miscfiles_read_localization()`.

I mentioned we needed to define our `files_search_var_log()` macro in `portsentry.if`, so let's take a look at that file next:

```
########################################
## <summary>
##      Search the /var/log directory.
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
#
interface(`files_search_var_log',`
        gen_require(`
                type var_t, var_log_t;
        ')

        allow $1 { var_t var_log_t }:dir search_dir_perms;
')
```

To be honest, I created this macro by copying another similar macro called `files_search_var_lib()` and just replacing "`lib`" with "`log`". Unless you're a whiz with `m4`, I recommend doing the same thing.

Now `gen_require()` is a macro that generates a "`require { ... }`" block to declare the types we're going to be using in our policy rules. Then we have an `allow` rule with some expansions: `$1` corresponds to the argument we give to the macro (`portsentry_t` in our case), and `search_dir_perms` is a macro defined elsewhere in the Reference Policy that corresponds to a list of access permissions. After being run through `m4`, the one line `files_search_var_log(portsentry_t)` gets expanded to the following policy rules:

```
require {
    type var_t, var_log_t;
}
allow portsentry_t { var_t var_log_t }:dir { getattr search };
```

That's about the least complicated macro you're likely to run into, but it gives you a flavor of how the macros are compiled into SELinux policy statements.

Finally, we have our `portsentry.fc` file:

```
/usr/local/sbin/portsentry          --
          gen_context(system_u:object_r:portsentry_exec_t,s0)
/etc/portsentry(/.*)?
           gen_context(system_u:object_r:portsentry_etc_t,s0)
/var/log/portsentry(/.*)?
           gen_context(system_u:object_r:portsentry_log_t,s0)
# You need this line if you're using an older init-based Linux
#/etc/rc\.d/init\.d/portsentry                    --
   gen_context(system_u:object_r:portsentry_initrc_exec_t,s0)
```

In the interests of readability, I was forced to introduce line breaks in the middle of each line. In practice, the `gen_context()` macros would appear as the third column in each line above—in other words, the actual `portsentry.fc` file is only four lines long.

Starting from the left-hand side of each line, the first column looks a lot like the file specification regular expressions we were using "`semanage fcontext ...`" earlier. In the rightmost column, we use `gen_context()` to define what context we want the files and directories to have. Notice that has to fully declare the user context and the role along with the type. The "`,s0`" at the end of each context label is the *security level* as used by MLS. In the default targeted policy everything is declared at "`s0`".

The middle column in the *.fc file syntax is optional and specifies the file type. For example, the first two entries use "--" to specify that the objects being referred to are regular files. "-d" would indicate a directory, "-c" a device file and so on. In two rules, we're using wildcards to specify a directory and all files underneath that directory, so using "--" or "-d" wouldn't work here because it wouldn't cover all cases. In these situations, just leave the second column blank and your rule will apply to all object types that match the given path specifier.

OK, at this point we've got some minimal policy description files, albeit ones that don't really grant our `portsentry_t` processes much in the way of access privileges. But it's enough to get us started. So, we type `make` in our build directory and the `selinux-policy Makefile` automatically generates our `portsentry.pp` file for us. The next slide covers loading this policy package and some other implementation details.

Load Module, Set Contexts

```
# semodule -i portsentry.pp
# semodule -l | grep portsentry
portsentry1.0.0
# semanage fcontext -l | grep portsentry
/etc/portsentry(/.*)?                    all files
system_u:object_r:portsentry_etc_t:s0
/usr/local/sbin/portsentry               regular file
system_u:object_r:portsentry_exec_t:s0
/var/log/portsentry(/.*)?                all files
system_u:object_r:portsentry_log_t:s0
# restorecon -FR /etc/portsentry /var/log/portsentry \
    /usr/local/sbin/portsentry
```


Having generated our initial `portsentry.pp` file, we use "`semodule -i ...`" to load it. We can confirm that the install worked with "`semodule -l`". We can also confirm that the file context definitions from our `portsentry.fc` file was properly loaded using "`semanage fcontext -l`".

However, while the file context table has been updated, the labels on our files and directories have not actually been updated. We have to run `restorecon` as shown on the slide above in order to set our file contexts properly.

Ready? Set? Go!

- Make sure SELinux is in Permissive mode
- Recommend rebooting so that processes are started in proper context via `systemd`
- Start new `audit.log` file
- When system comes up, start exercising new daemons ...

Great, we've now got our initial policy loaded and our file contexts set. Again, the plan is to run the app on a system in permissive mode and capture policy violations from the `audit.log` file. We're going to use a tool call `audit2allow` to help us generate the policy rules we're missing based on the policy violations recorded in the `audit.log`.

A couple of hints before you begin:

1. Create a boot configuration and start the process as it would normally start at boot time—in other words, reboot the system to start the process. If you just run the process from the command line, it may inherit the user context and role from your interactive session rather than using the user context and role that it will have in its production deployment. This difference will be reflected in the alerts in the `audit.log` file and will make it harder when you're trying to generate policy via `audit2allow`.
2. Just before you reboot the system, tell `auditd` to start a new `audit.log` file. You can do this by sending the `auditd` process a `SIGUSR1` signal via "`pkill -USR1 auditd`". If you start with a fresh `audit.log` file each time, then you won't waste time re-creating policy statements from audit.log entries that you've already seen.

After you reboot the system and your process has been started, then go ahead and use the application as normal. In the case of Portsentry, this means hitting the system with some port scans to trigger PortSentry's logging behavior (and possibly other actions). You should also exercise the application by subjecting it to at least one more reboot cycle. There may be behaviors that you get when the application is shutting down and then coming back up that you may not see at any other time.

audit2allow 2 the Rescue!

```
# grep type=AVC /var/log/audit/audit.log |
    grep portsentry | audit2allow -m portsentry

module portsentry 1.0;

require {
   type portsentry_t;
   ...
}

#============= portsentry_t ==============
allow portsentry_t dhcpd_port_t:udp_socket name_bind;
...
```

SANS | SEC506 | Securing Linux/Unix 64

The good news is that the SELinux packages include a little program called `audit2allow` that simplifies policy creation. You just feed your `audit.log` entries into `audit2allow` and it kicks out SELinux policy that would allow the access that's currently preventing your app from functioning. Obviously, you'll want to review the output of `audit2allow` and make sure the rules are not too permissive, but the tool can really help jump-start our policy.

When running `audit2allow`, you must specify a policy module name with "`-m`". You can see that `audit2allow` uses this information to generate a module header at the beginning of the output. In reality, we don't care about the first part of the `audit2allow` output where the module header and "`require`" block sit. What we're really interested in are the "`allow`" rules that come after the comment—these are the rules that we will need to update our policy with.

And Then the Pain ...

- `audit2allow` has no clue about Reference Policy macros
- So, you get to translate manually ... yay!
- Having a copy of the Reference Policy sources pays off ...

Unfortunately, `audit2allow` dumps out raw SELinux policy rules. It has no clue about Reference Policy macros. You could insert the raw rules into the Portsentry policy file we're creating, but in the long term that would end up being unmaintainable.

So, the bad news here is that you have to become an expert at translating raw SELinux policy statements back into Reference Policy macros. The good news is that there are some basic patterns that happen over and over again. So, after you've done this a few times it gets to be pretty rote. But those first few times you're really going to need a copy of the Reference Policy source to work with.

Don't panic! I'm going to work through this example with you to introduce you to how this process works. Stay with me ...

Reference Policy Translation

Find clusters of related rules in output:

```
allow portsentry_t portsentry_etc_t:dir search;
allow portsentry_t portsentry_etc_t:file { read getattr open };
allow portsentry_t portsentry_log_t:dir { write search add_name };
allow portsentry_t portsentry_log_t:file { write read create open };
```

Two choices:

- Steal code from existing policies
- Hunt around with `find/grep`

The first hint is to break the process down into manageable chunks. When you look at our sample `audit2allow` output, it seems like a huge pile of confusing gibberish. But if you stare at it for a while, you'll start seeing some patterns that let you group some of the rules together in clumps like I've done below:

```
allow portsentry_t portsentry_etc_t:dir search;
allow portsentry_t portsentry_etc_t:file { read getattr open };
allow portsentry_t portsentry_log_t:dir { write search add_name };
allow portsentry_t portsentry_log_t:file { write read create open };

allow portsentry_t node_t:tcp_socket node_bind;
allow portsentry_t node_t:udp_socket node_bind;

allow portsentry_t dhcpd_port_t:udp_socket name_bind;
allow portsentry_t echo_port_t:tcp_socket name_bind;
allow portsentry_t echo_port_t:udp_socket name_bind;
...

allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket create;
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create };
```

First, you see some rules related to file and directory access for objects in the `portsentry_etc_t` and `portsentry_log_t` contexts. This is where the daemon is trying to read its configuration files and write to its logs. Then there are some `node_bind` and `name_bind` rules (many of which I didn't show here in order to save space). Finally, there are some miscellaneous rules related to (raw) socket access. The trick is to focus on one chunk at a time in order to break the rule translation process up into manageable pieces.

Clearly, the `portsentry_etc_t` rules are related to the process accessing its configuration files under `/etc/portsentry`. There are lots of other daemons that have configuration files under `/etc` and many of these programs have SELinux policy files already written. So hopefully we can just steal some ideas from those pre-existing policy files.

Let's unpack the tarballs from the directory where we installed the source RPM and look around:

```
# cd /root/rpmbuild/SOURCES
# for f in serefpolicy-*.tgz; do tar zxf $f; done
# cd serefpolicy-*
# ls -d serefpolicy-*
serefpolicy-3.13.1      serefpolicy-contrib-3.13.1
serefpolicy-3.13.1.tgz  serefpolicy-contrib-3.13.1.tgz
# ls serefpolicy-contrib-*/*.te | wc -l
357
```

If you look in the `serefpolicy-contrib-<vers>` directory, you'll lots of sample policy files.

Your best bet is to start with one of the simpler policy files under the services directory. I've found that `soundserver.te` is pretty useful for examples:

```
# grep etc_t serefpolicy-contrib-*/soundserver.te
type soundd_etc_t alias etc_soundd_t;
files_config_file(soundd_etc_t)
read_files_pattern(soundd_t, soundd_etc_t, soundd_etc_t)
read_lnk_files_pattern(soundd_t, soundd_etc_t, soundd_etc_t)
```

The first two lines of output are very similar to declarations we already have in our basic PortSentry policy file. But the third line could be what we're looking for. However, we need to go looking for where `read_files_pattern` is defined in the core Reference Policy so that what can understand exactly what this macro does. Time for some command-line kung fu:

```
# cd /root/rpmbuild/SOURCES/serefpolicy-<vers>
# grep -rl read_files_pattern *
...
policy/modules/kernel/files.if
policy/modules/kernel/corecommands.if
policy/modules/kernel/filesystem.if
policy/modules/kernel/kernel.if
policy/modules/kernel/domain.if
policy/modules/kernel/devices.if
policy/support/file_patterns.spt
```

It turns out that the first file, `policy/support/file_patterns.spt` is the file we want. Here's the declaration for `read_files_pattern`:

```
define(`read_files_pattern',`
        allow $1 $2:dir search_dir_perms;
        allow $1 $3:file read_file_perms;
')
```

This is starting to look a little bit like the "`allow`" rules in our `audit2allow` output, but `search_dir_perms` and `read_file_perms` are themselves reference policy macros that are defined in a different file. If you do another round of "`grep -rl read_file_perms`", you'll send up looking at `policy/support/obj_perm_sets.spt`. In the middle of this file you'll find:

```
#
# Directory (dir)
#
define(`getattr_dir_perms',`{ getattr }')
define(`setattr_dir_perms',`{ setattr }')
define(`search_dir_perms',`{ getattr search open }')
...

#
# Regular file (file)
#
define(`getattr_file_perms',`{ getattr }')
define(`setattr_file_perms',`{ setattr }')
define(`read_file_perms',`{ getattr open read lock ioctl }')
...
```

Great! We seem to have found the macros we're searching for, and many more besides. Putting the macros from the two files together, it looks like `read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)` would expand to two rules that look like this:

```
allow portsentry_t portsentry_etc_t:dir { getattr search open };
allow portsentry_t portsentry_etc_t:file { getattr open read ... };
```

This seems to be a little bit more access than the rules we dumped out via `audit2allow` are saying we need. On the other hand, using the standard Reference Policy macros is a good idea for maintainability, so let's go with `read_files_pattern` here.

If we try and match up our `portsentry_log_t` rules from `audit2allow` against the macros in `obj_perm_sets.spt`, it looks like we need `add_entry_dir_perms` (the smallest permission set that includes `write` and `add_name` capabilities) plus `read_file_perms` and `write_file_perms`. `rw_file_perms` is close but doesn't include `create`. `manage_file_perms` is a little more access than we need. Unfortunately, there's no macro in `file_patterns.spt` that comes close to matching what we need, so instead, I'm going to make a new macro in `portsentry.if`:

```
interface(`rwcreate_files_pattern',`
        allow $1 $2:dir search_dir_perms;
        allow $1 $3:file read_file_perms;
        allow $1 $3:file write_file_perms;
')
```

Note that I have to use "`interface(...)`" to define the macro in `portsentry.if` rather than "`define(...)`" as the Reference Policy does in `file_patterns.spt`. This is due to the different ways in which the two files are compiled into policy.

Now we can update our `portsentry.te` file with the following macros:

```
read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)
rwcreate_files_pattern(portsentry_t, portsentry_log_t,
portsentry_log_t)
```

So much for the file access rules, now let's tackle all those `name_bind` and `node_bind` entries. There are a lot of these but they're all generally the same, just for different ports. Let's try searching through our policy source directory for `name_bind` and see what turns up:

```
# grep -rl name_bind *
policy/flask/access_vectors
policy/modules/kernel/corenetwork.if.in
policy/modules/kernel/corenetwork.te.in
policy/modules/kernel/corenetwork.if.m4
policy/mls
```

Those `.../kernel/corenetwork.if.*` files look promising. If you scan through those files you find lots of macros referring to `name_bind` and `node_bind`. But remember we need to allow Portsentry to bind to pretty much any port it wants to, so the macros like `corenet_tcp_bind_all_ports()` seem most like what we need. Since we apparently need both `name_bind` and `node_bind` access to both TCP and UDP ports, the following collection of macros seems most apropos:

```
corenet_tcp_bind_all_ports(portsentry_t)
corenet_tcp_bind_all_nodes(portsentry_t)
corenet_udp_bind_all_ports(portsentry_t)
corenet_udp_bind_all_nodes(portsentry_t)
```

The only lines we haven't dealt with from the `audit2allow` output are the following:

```
allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket { read create };
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create listen };
```

Turns out, there aren't really any macros in the Reference Policy to help us with these. There are certainly socket-related macros that are supersets of the "`{ bind create listen }`" and "`{ read create }`" access lists, but since we're allowing Portsentry to bind to any port on the system, we're probably better off only allowing the minimum access to these ports that we can. So, we're just going to shove these lines into our policy file verbatim.

Thus our "final" `portsentry.te` file looks like this:

```
policy_module(portsentry, 1.0.1)

#### Declarations

type portsentry_t;
type portsentry_exec_t;
init_daemon_domain(portsentry_t, portsentry_exec_t)

type portsentry_etc_t;
files_config_file(portsentry_etc_t)

type portsentry_log_t;
logging_log_file(portsentry_log_t);

# You need these lines if you're using an older init-based Linux
#type portsentry_initrc_exec_t;
#init_script_file(portsentry_initrc_exec_t)

#### Policy

files_search_etc(portsentry_t)
files_search_var_log(portsentry_t)

read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)
rwcreate_files_pattern(portsentry_t, portsentry_log_t, portsentry_log_t)

corenet_tcp_bind_all_ports(portsentry_t)
corenet_tcp_bind_all_nodes(portsentry_t)
corenet_udp_bind_all_ports(portsentry_t)
corenet_udp_bind_all_nodes(portsentry_t)

allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket { read create };
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create listen };

logging_send_syslog_msg(portsentry_t)

miscfiles_read_localization(portsentry_t)
```

New Rules ... Now What?

- Update your `*.te` file (change policy version number!)
- Build and load new policy
- Start new `audit.log`, reboot system, exercise daemon, check `audit.log`...
- Keep going until `audit.log` warnings are gone

Awesome! We've got a new `portsentry.te` file to try out, this one labeled with version marker 1.0.1. Having saved the changes to the file in our build directory, we run `make` again and install the new `portsentry.pp` file. "`semodule -l`" should show that the installed module version is now 1.0.1.

At this point, it's really a question of "lather, rinse, repeat." Start a new `audit.log` file with "`pkill -USR1 auditd`" and reboot the machine. Fire some more port scans at the box to let Portsentry do its thing. Monitor your `audit.log` for any new warnings (I think our new policy file is complete and you won't hit any additional warnings, but you never know). If any warnings show up, follow the same procedure we just went through to find the appropriate macros to update your policy file, and so on.

And Finally ...

- Let the app run for a while to catch all possible behaviors
- Try testing in Enforcing mode for a while too ...
- Eventually you arrive a complete policy
- Now push same policy out across multiple systems

In general, it's also probably a good idea to let the application run for a while just in case there are occasional behaviors of the application—weekly cleanup tasks, for example—that you didn't catch during your testing. It's also a good idea at some point during this "burn in" period to put your test system into enforcing mode as a final check that your policy is working as expected.

Eventually, you'll have high confidence that your policy is correct and complete. At this point, you can push the policy out across your enterprise. Assuming your systems are running the same OS version, you should just be able to copy the `*.pp` file out to each machine and load it with "`semodule -i`". Don't forget to do a `restorecon` to set the right contexts on the files and directories associated with the application.

If you have different OS versions running around, you'll need to recompile a `*.pp` file for each different version you're running—but again beware of changes to the Reference Policy macros between OS revisions.

Summary

- Don't just blindly turn SELinux off
- Security benefits and helps you understand applications
- Pointers to additional docs in notes ...

Is enabling SELinux the right choice for your environment? All too often sites don't make an informed choice on this issue—they just disable SELinux immediately because they don't understand it. Hopefully, the material in this course will clear up some of the fear, uncertainty, and doubt around SELinux and let you make an honest, intelligent decision for your organization. Maybe I've even sparked your interest in SELinux a bit.

SELinux does have the promise of providing much higher levels of security. But we're only going to work out the kinks in the default policy if people actually start using it. Creating a policy for an application also teaches you a lot about how that application interacts with the operating system (similar to having to build a directory for an app to `chroot()` into, really).

More documentation on SELinux:

A good intro to SELinux: *http://www.billauer.co.il/selinux-policy-module-howto.html*

Excellent How-To from the CentOS project: *http://wiki.centos.org/HowTos/SELinux*

The Fedora Wiki: *https://fedoraproject.org/wiki/SELinux*

Reference Policy development site: *https://github.com/TresysTechnology/refpolicy/wiki*

Lab Exercise

- SELinux capstone exercise
- Earn your SELinux merit badge!

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 4, Exercise 4, so navigate to *.../Exercises/Day_4/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 4
5. Follow the instructions in the exercise

DNS and BIND

Some of the material in this section was excerpted from a larger tutorial on administering DNS and Sendmail. If you are interested in more detail on DNS and Sendmail administration (as opposed to security) issues, you can download the PDF for full course book for this tutorial from:

http://www.deer-run.com/~hal/dns-sendmail/

The *DNS and BIND* and *Sendmail* books published by O'Reilly and Associates also make useful references. There is also the BIND *Administrators Reference Manual* published by the ISC *https://www.isc.org/downloads/bind/doc/*

Another good source for DNS security configuration examples is the "Secure BIND Template" document available at *http://www.cymru.com/Documents/secure-bind-template.html*

DNS/BIND Topics

- Introduction to DNS/BIND
- Split-horizon ("split-brain") DNS
- Configuring BIND
- DNSSEC

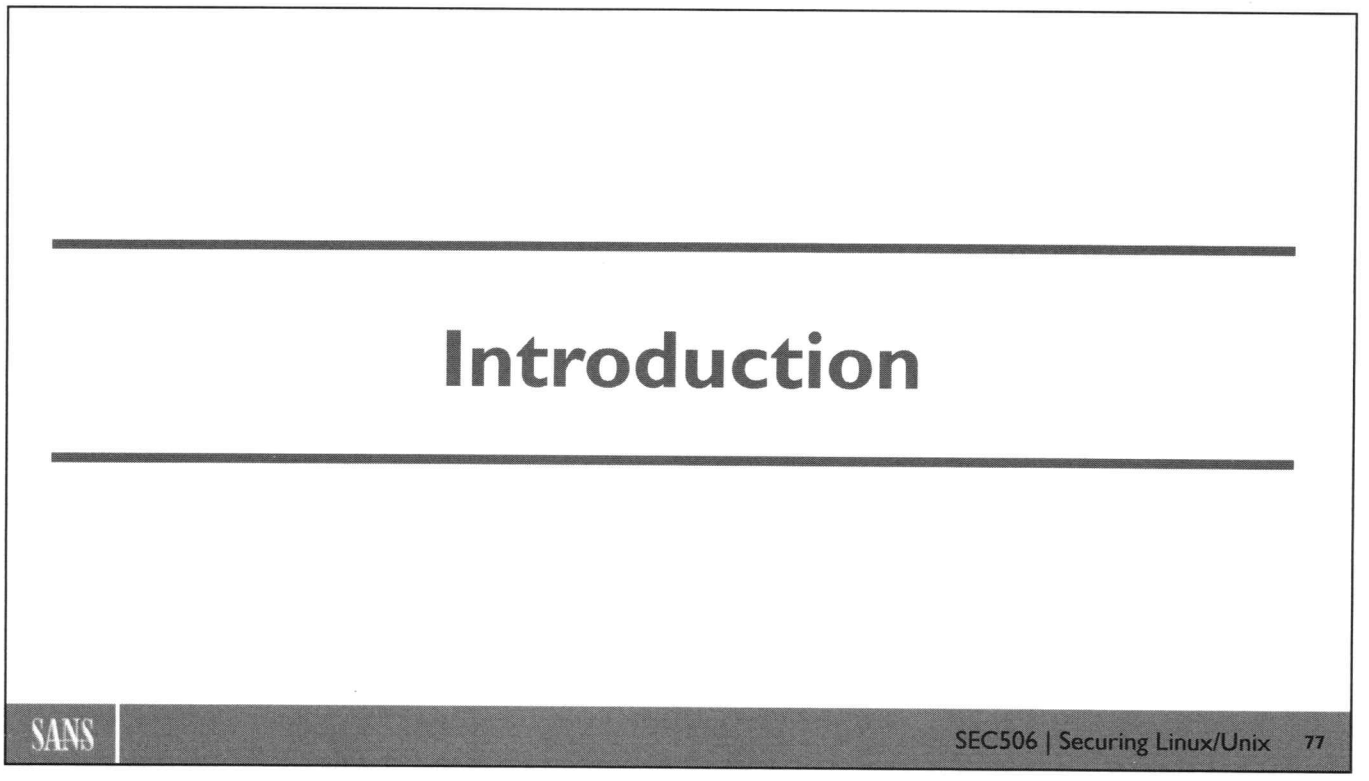
The *Introduction* is a quick introduction to the Domain Name Service and BIND plus an overview of common vulnerabilities in past and present DNS and BIND implementations.

Split-Horizon DNS discusses the theory behind presenting one version of your DNS information to the outside world and a completely different view internally—why and when this is useful and some architectural issues with such a configuration.

Configuring BIND presents specific configuration examples of the DNS architecture introduced in the *Split-Horizon DNS* section, and introduces many of the new security features in recent versions of BIND.

At this point, you'll have the opportunity to get some hands-on experience setting up BIND to run without superuser privileges and in a `chroot()` ed environment.

We'll finish up the module with a section on *DNSSEC*, which is the only real defense against some of the DNS security vulnerabilities we'll be discussing in the *Introduction*.



The *Introduction* is a quick introduction to the Domain Name Service and BIND plus an overview of common vulnerabilities in past and present DNS and BIND implementations.

What Are DNS and BIND?

- The Domain Name Service maps host names to IP addresses and vice versa
- BIND is the reference standard DNS implementation, maintained by the ISC
- Global database
  - Distributed
  - Hierarchical

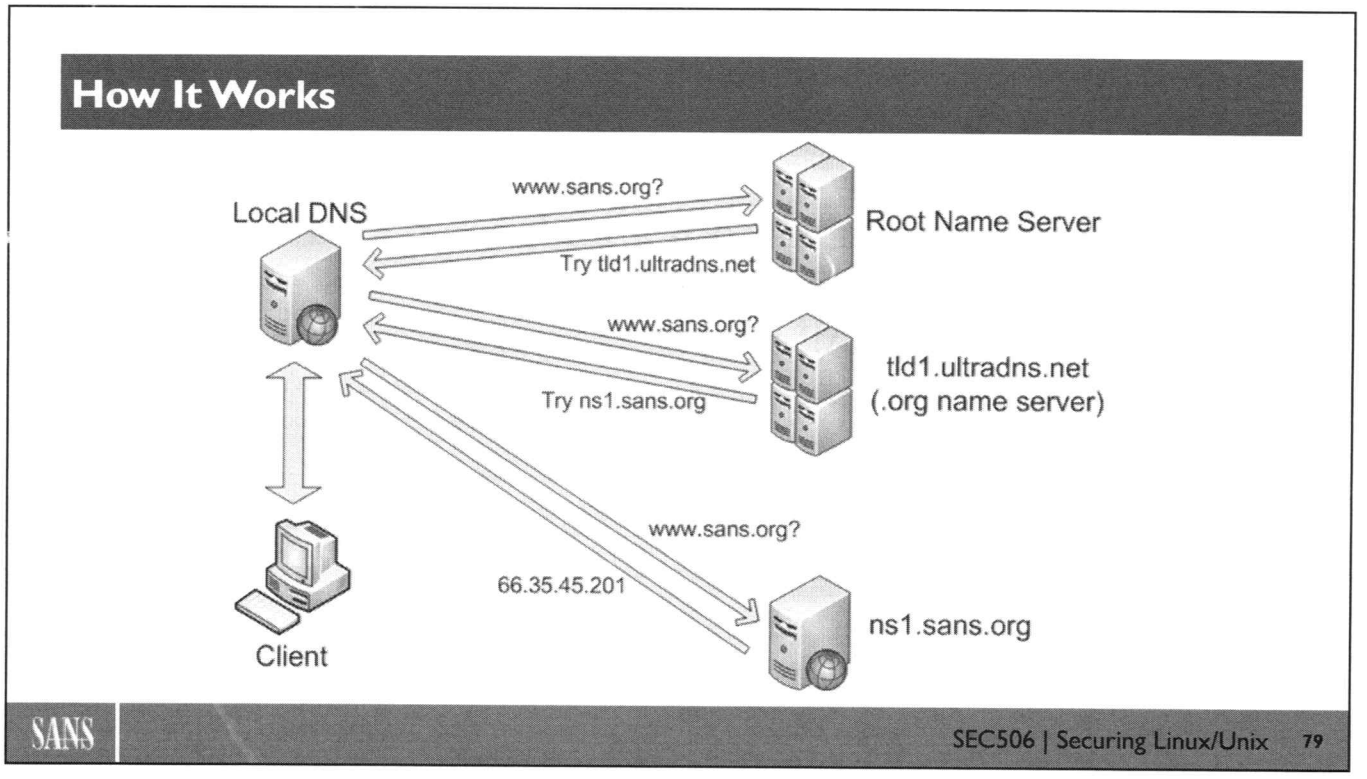
The Domain Name Service (DNS) is the mechanism that Internet hosts use to determine the IP address which corresponds to a given hostname. For example, if your Web browser wishes to reach the home page for the SANS Institute it must first determine the IP address for `www.sans.org`. This IP address is then used as the destination address in the packets which your client sends to communicate with the remote server. Conversely, the Web server at `www.sans.org` will receive the IP address of your machine as the source of these packets and will most likely attempt to determine the hostname which corresponds to this IP address. Again, DNS is the mechanism which `www.sans.org` will use to make this determination.

DNS can contain more information than just hostnames and IP addresses—for example, DNS can store information about the type of machine and OS platform (HINFO records), general "free-form" information about machines (TXT records), and many other types of data. In particular, DNS usually provides both internal and external machines with information about how e-mail is to be routed to a given organization (MX records).

BIND stands for Berkeley Internet Name Daemon. From its creation, BIND has been the reference implementation for DNS on the Internet and is the basis for the DNS implementation provided with most modern operating systems. For much of the 80s and 90s, BIND was developed and maintained by Paul Vixie. When Paul co-founded the Internet Software Consortium, BIND was one of the first applications to be supported by this new group. You can download the BIND sources for free from:

```
http://www.isc.org/products/BIND/
```

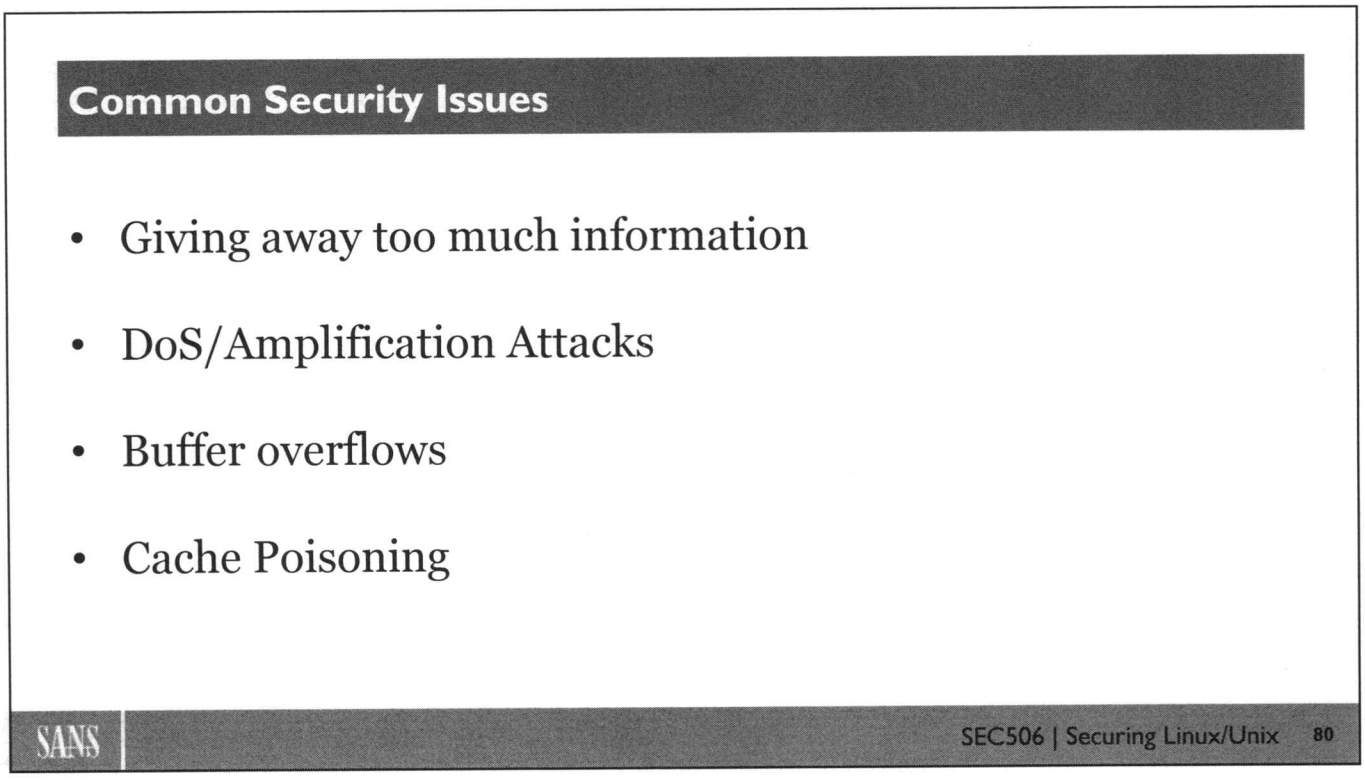
DNS is fundamentally a distributed database system—each organization maintains its own local information. These distributed collections of information are linked in a hierarchical fashion, which is more easily demonstrated pictorially… *(see next slide)*



Let us suppose that your local client wishes to learn the IP address of `www.sans.org`. Your client contacts a local name server which has been configured on the local client by the administrator (either statically or via DHCP, etc.). Your local DNS server actually does all of the work required to resolve the IP address and then will hand the result back to the client.

The local name server first attempts to contact one of the several *root name servers* that have been deployed on the Internet and asks the question, "What is the IP address of `www.sans.org`?" The root name servers don't know the answer, but they give you a *referral* to another name server that has more information—in this case, they hand back a list of host names and IP addresses for the name servers that handle the `.org` top-level domain. Your local name server then asks one of the `.org` servers for the IP address of `www.sans.org` and receives *another* referral—this time to the list of name servers for `sans.org`. Finally, your local name server asks one of the `sans.org` servers for the IP address of `www.sans.org` and actually gets the answer back. Your local DNS server will *cache* this information for some period of time (so that other machines at your site which ask for the same information don't put extra load on the Internet DNS infrastructure) and also hand the result back to your client.

Note that in order to be able to contact a root name server to start this process, your local name server must be statically configured with the names and IP addresses of the available root name servers. This information is maintained by the InterNIC and downloaded by the administrator into a static file on the local name server.



Since DNS is a public networked database, one significant risk is that you give away too much information—information that can be used by people who wish to attack your systems and networks. Specific examples are given in the upcoming slides.

Because DNS is so critical for the operation of individual networks and the Internet as a whole, it's also a popular target for denial-of-service type attacks. Aside from DoS danger to your own infrastructure, badly configured DNS servers also provide attackers with an opportunity to "amplify" their attacks against other targets.

BIND has historically had buffer overflow problems in various releases. Some have led to root compromise attacks; others have simply been denial-of-service type attacks. The best defense against these attacks is to stay up to date on the version of BIND you are running, though the *Running a Name Server* section suggests how to configure BIND to run in a `chroot()`ed environment, which can help protect you in the event of an exploitable buffer overflow.

Cache poisoning occurs when a name server has been tricked into believing erroneous information from some external source. Sometimes this occurs by accident, but most often it is used by attackers who wish to embarrass an organization or exploit trust relationships based on hostname/address information. More on this shortly.

Too Much Info – *Public/Private*

Other organizations don't need to know that much about you:

- Addresses of your public servers
- How to route e-mail to you

Other info is useful to your internal users—*and attackers!*

Your DNS database contains information about all of the machines in your organization. In particular, machine names and other information (HINFO and TXT records) may help an attacker locate machines which are most critical to the functioning of your organization or which can be easily targeted for attack—for example, a machine called `proxy.yourdomain.com` might be interesting to an attacker wishing to penetrate the interior of your network or anonymize their attacks on other Internet hosts.

Generally, the outside world needs to know very little information about your network. At a minimum, you need to advertise hostnames and IP addresses of a limited set of "public" servers: name servers, e-mail servers, Web and FTP servers, etc. In a perfect world, an organization would be able to present one set of information to the outside and reserve a full, rich set of information for internal consumption—this is the theory behind *split-horizon DNS* which will be covered in an upcoming section.

Too Much Info – *BIND Version*

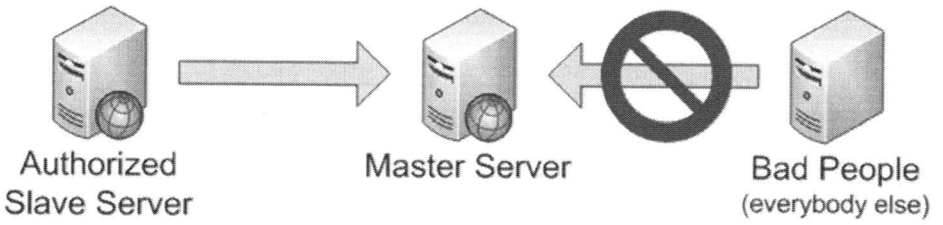
```
% dig @ns1.sans.org version.bind txt chaos

; <<>> DiG 9.10.4 <<>> @ns1.sans.org version.bind txt …
; (1 server found)
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34912
;; flags: qr aa rd ra; Ques: 1, Ans: 1, Auth: 0, Addit: 0
;; QUESTIONS:
;;      version.bind, type = TXT, class = CHAOS

;; ANSWERS:
VERSION.BIND.    0       TXT     "9.10.4"
[…]
```


It is possible to query a running name server and retrieve the embedded version number string from the remote machine. Since there are known vulnerabilities in most older releases of BIND, this information can help an attacker target your machines. As we will see, administrators are able to easily change the version number string to fool attackers.

Too Much Info – *Zone Transfer*

Backup name servers transfer DNS info from master servers

Attackers will do the same to gather info about your network



Block Unauthorized Zone Transfers!

Generally, each organization runs one master DNS server and one or more slave servers for redundancy. Periodically, the slaves must contact the master and download any updates to the local DNS database—this is referred to as a *zone transfer*. By default, nameservers running BIND allow any remote system to perform a zone transfer—whether that system is a legitimate name server for that domain or not. Zone transfers can even be requested from the slave name servers for your domains.

Attackers often attempt zone transfers in order to gather information about your local network. If they succeed, then they have instantly gotten all of the information about your internal hosts and networks with very little effort. Of course, a split-horizon DNS configuration can limit the amount of information an attacker will receive, but it is still a good idea to prevent unauthorized hosts from downloading your zone databases. In a later section, we will see how to configure BIND to restrict zone transfers, but it is also a good idea to block this activity at your firewall as well.

A useful resource for teaching people about the dangers of open zone transfers is DigiNinja's ZoneTransfer.me project: *http://www.digininja.org/projects/zonetransferme.php*

DoS and Amplification Attacks

DNS is a popular DoS target

Misconfigured DNS servers used to "amplify" DoS effects:
- Arbitrary recursive queries
- Open cache queries

"Amplifiers" often end up DoS-ed as well

Attackers have learned that DNS is often a weak link in an organization's infrastructure. For example, and organization may spend millions on a worldwide, redundant web services infrastructure. But if the attacker can knock all of that organization's DNS servers off-line, then nobody will be able to resolve the IP addresses of those web servers, and they'll be effectively knocked off the Internet. So, apply the same care to building out your DNS infrastructure and use the same DoS prevention techniques you would use with your web services.

What's become particularly prevalent lately is attackers leveraging poorly configured DNS servers run by various organizations on the Internet to "amplify" their DoS attacks:

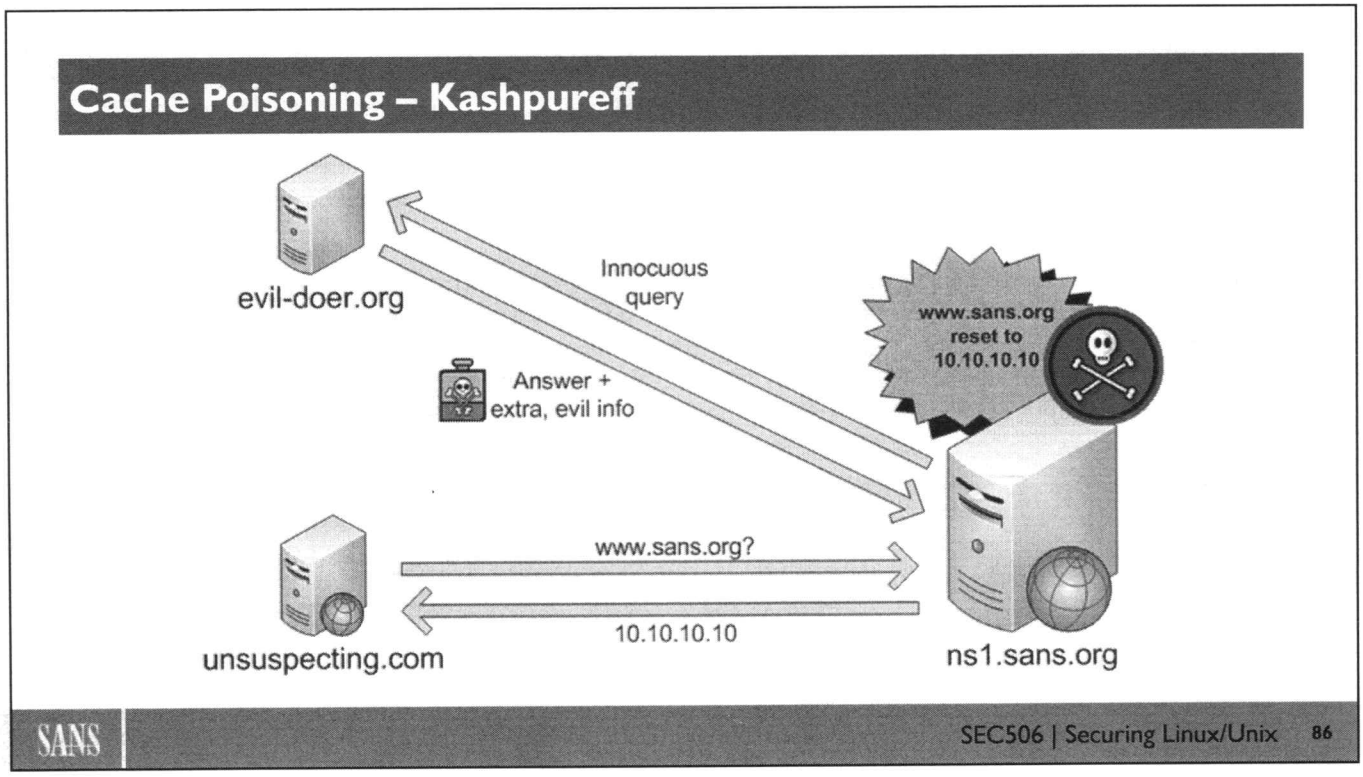
- Normally a name server receives a request from an external name server, responds with the best information it has and does no further work-- this is a *non-recursive* (sometimes referred to as an *iterative*) *query*. However, client resolvers (and sometimes other name servers as we'll see in a moment) generally do *recursive queries* when communicating with their local name server—that is, they rely on their local name server to do all the work required to look up a given piece of information (including contacting root name servers and name servers at other organizations).

  Generally, only machines that you own should be making recursive queries via your name server. If you allow outsiders to do recursive queries via your name servers, you make it possible for attackers to force your name server to make queries on their behalf. The attacker then forces your name server to query the name server they're trying to bring down, adding your name server's bandwidth and resources to the DoS attack on the target, and incidentally making your name servers appear as a source of malicious traffic that should be blacklisted.

  Also allowing anybody to make recursive queries at your name servers makes it easier for attackers to get your name server to query the attacker's maliciously configured name servers. This can assist attackers in performing buffer overflow attacks and cache poisoning attacks against your name server.

- Up until BIND 9.4.1, the default behavior for BIND was that anybody could query your name servers for information in their cache. This was commonly used by attackers to amplify DoS attacks. The attacker would spoof a DNS query using their target as the source IP address of the bogus query. The query itself is typically a request for the list of root name servers—a very small query that results in a relatively large response. Thus, the attacker can reflect thousands of these small queries through various open servers and have a very large impact on the target machine spoofed in the source.

Aside from being a nuisance to the rest of the Internet, if your name server is used as an amplifier you may find that the bogus queries from the attacker are coming in so fast that your own name server may be DoS-ed or at least significantly degraded. You may also find yourself being blacklisted by various organizations and unable to communicate with large chunks of the Internet. So, I'll be showing you some useful configuration settings to block this sort of malicious activity.



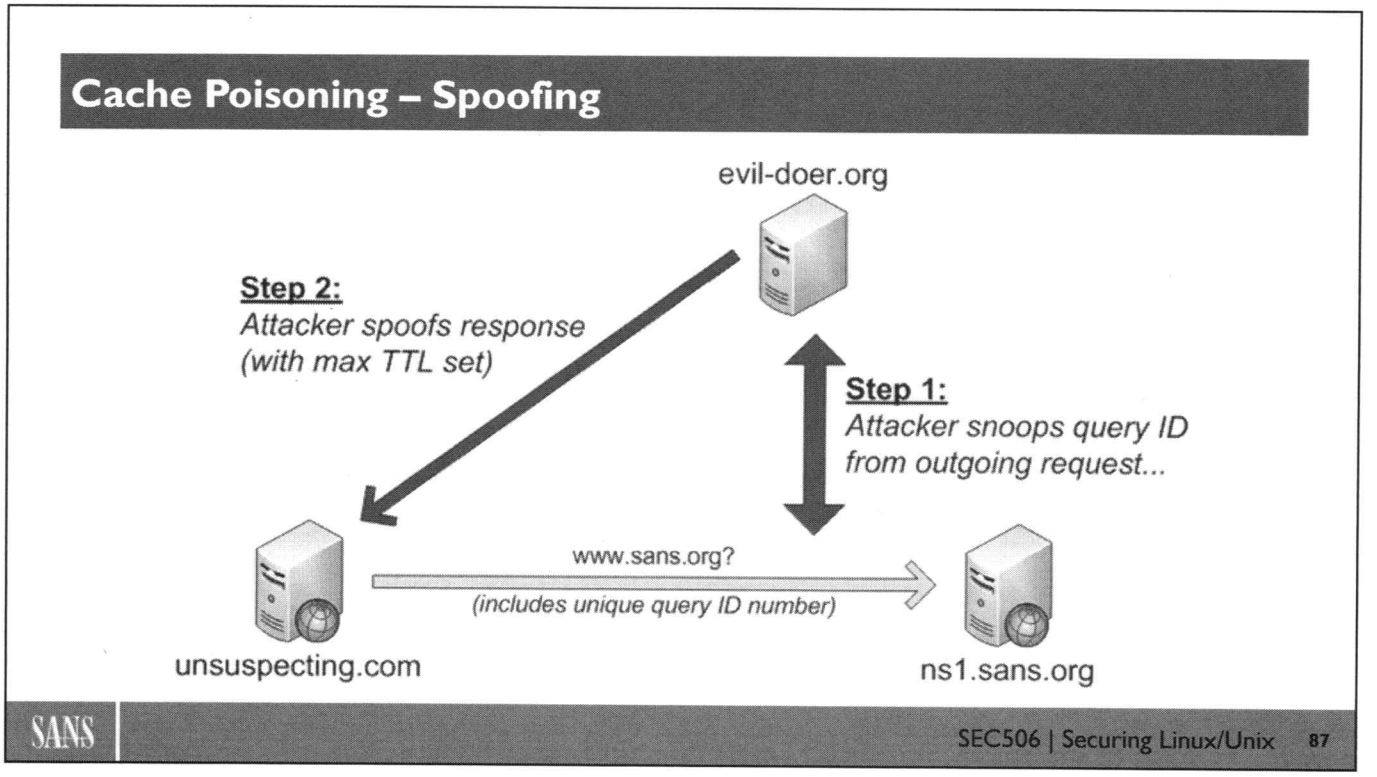
The classic cache poisoning attack described on this slide is usually referred to "Kashpureff style" cache poisoning after Eugene Kashpureff who used this attack in 1997 to poison the root name servers in order to advertise his alternate "AlterNIC" top-level domains. The underlying vulnerability in BIND was fixed soon after this attack and so this form of cache poisoning is no longer effective.

The attack is triggered when a vulnerable name server makes a query against some evil name server owned by the attacker. The attacker often triggers this initial query externally by connecting to some service on the vulnerable name server (or one of the hosts which use the vulnerable name server as their local name server)—this can cause an IP address to hostname lookup against the evil name server.

The vulnerable name server makes an innocent query against the evil name server and the evil name server hands back the proper response *plus* some extra information. Older versions of BIND would put this extra information in their cache—potentially overwriting information that they had learned from their own local zone databases.

In the example above, the evil name server has poisoned the cache on `ns1.sans.org`, convincing this machine that the IP address of `www.sans.org` is `10.10.10.10` (an invalid IP address). When any other machine queries `ns1.sans.org` for the IP address of `www.sans.org` they will get the invalid address and be unable to reach the real Web server for sans.org. Imagine, however, that the attacker had used an address which corresponded to a hard-core pornographic site (embarrassment) or an IP address which corresponded to a website owned by the attacker (phishing or man in the middle attack).

Note: the idea for the design of this slide comes from Judy Novak (thanks Judy!)



The next generation of cache poisoning attacks was really just a simple spoofing attack. DNS responses are essentially unauthenticated, so if an attacker can create and send a legitimate-looking response before the "real" name server that was queried can respond, then the host doing the lookup will happily accept the forged response containing bogus information.

The only problem from the attacker's perspective is that each DNS query goes out with a unique query ID number—the attacker's bogus response must include this ID. So, the attacker must typically first sniff the outgoing request and then incorporate the stolen query ID number into the bogus response. Obviously, the attacker must also forge the source IP address of the bogus response to make it appear that the packet comes from the remote name server that was queried. The attacker will also set the "time to live" (TTL) value in the response to be the maximum possible value so that the name server receiving the spoofed information will cache it for as long as possible.

Early releases of BIND used very predictable query ID numbers—making it possible for attackers to spoof responses without first sniffing the outgoing query. More recent releases of BIND have better randomization algorithms, though problems with these algorithms have been disclosed as recently as June 2007 (so stay up-to-date!).

Zodiac is a tool which actually implements this sort of DNS spoofing. For more information on Zodiac, see: *http://www.darknet.org.uk/2008/07/zodiac-dns-protocol-monitoring-and-spoofing-tool/*

Better Ideas for Better Exploits

- Attacker can force DNS servers to make predictable queries
- Attacker can force stream of queries
- Sends multiple responses, guessing query ID more likely
- What if attacker spoofs *glue records* instead of answers? *[Kaminsky, 2008]*

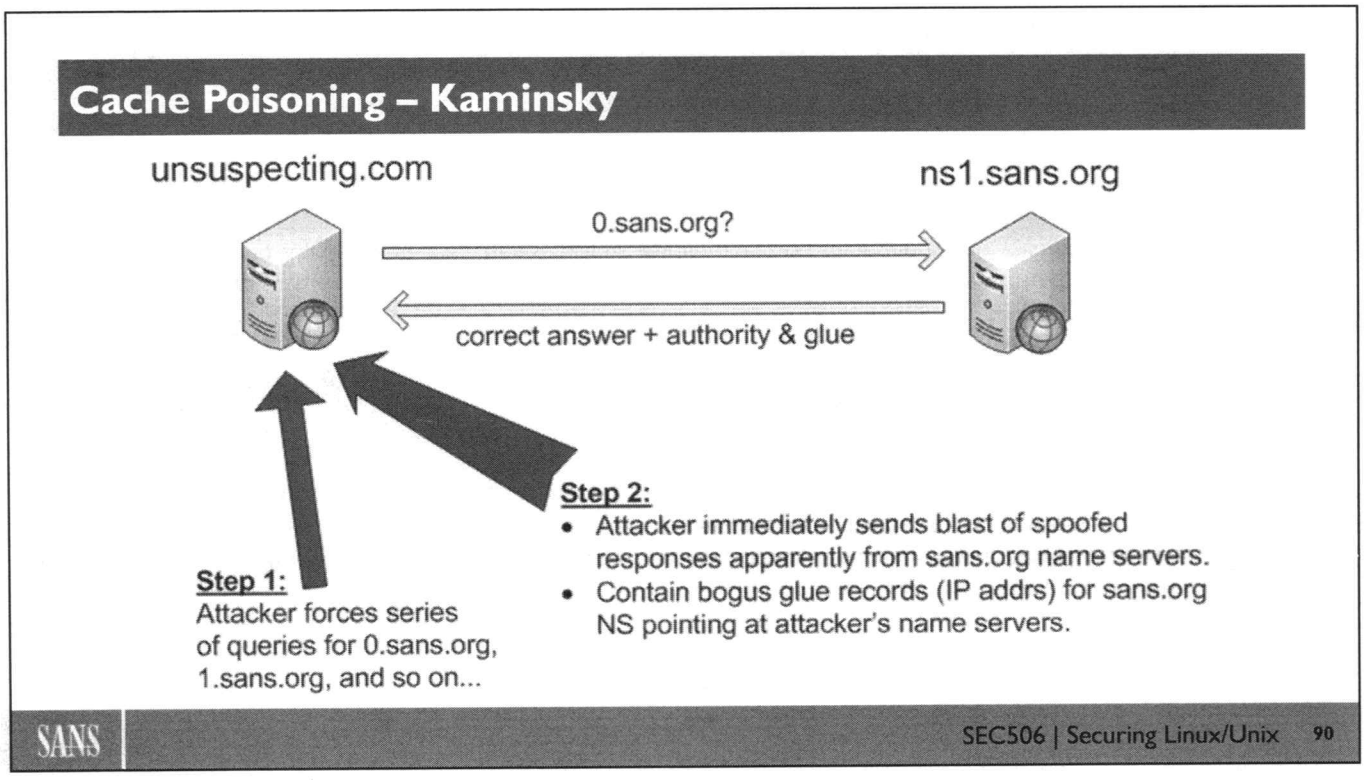
In the decade from 1998-2008, attackers continued to advance the state of the art in DNS attacks in several ways (Dan Kaminsky does a nice job of summarizing all of this research in his Black Hat slides from the 2008 conference, *http://www.slideshare.net/dakami/dmk-bo2-k8*):

1. Attackers can often cause remote name servers to make queries for information specified by the attacker, whether because the remote site is running open name servers (more on this later) or because the attacker can force another server (mail server, web server, etc.) or client (via XSS or similar mechanism) at the remote site to make a query. This means the attacker doesn't have to wait passively for queries to spoof and it means the attacker has a very good idea of which Internet name servers are going to be involved in answering the query.
2. If the attacker can force a remote name server to make one query, they can usually cause that server to make multiple queries back-to-back. This is typically done by having the remote server query a series of names like `0.sans.org`, `1.sans.org`, `2.sans.org`, etc. (you need to keep changing the hostname because the remote DNS server will cache the results each individual query, whether positive or negative). This means the attacker can create a constant stream of queries to try and spoof, just in case their first attempt fails.
3. The spoofing attack described on the previous slide relies on sniffing the query ID out of the outgoing query—if the attacker were just guessing, they'd only have a 1 in 64K chance of hitting the right query ID. But what if the attacker sent 64K spoofed responses, each with a different query ID? Then they'd be guaranteed to get a hit, right? Of course, they probably don't have the time or the bandwidth to generate that much traffic, but they can improve their odds by sending lots of spoofed responses with random query IDs.
4. But in 2008, Dan Kaminsky had an even bigger brainwave. Every DNS response includes an "authority" section listing the NS records for the domain. Included with those NS records are so-called "glue records"—the IP addresses of the NS records in the authority section so that you can actually query them.

> Dan realized that if your spoofed response included bogus glue records that used the IP addresses of the attacker's evil name server, then when the spoofing attack worked, *all future DNS queries* that the poisoned host makes for the domain would come to the attacker's name server. The attacker effectively "owns" the entire domain from the perspective of the affected name server.

In fact, as Kaminsky points out in his presentation, the attacker could even try to do this one level up the DNS hierarchy and try to override a site's list of name servers for a top-level domain like `.com`! In other words, all of your sites queries to *any domain in* `.com` would end up going to the attacker's DNS servers. The potential for mischief here cannot be overstated because it potentially enables attackers to impersonate critical infrastructure like the Microsoft patch servers, certificate authority root servers, banking websites, mail servers, etc. *all over the Internet*. This is a huge security issue.

What's fascinating to me about the Kaminsky attack is that it took somebody *30 years* (if you count from the time that DNS first appeared on the Internet) to realize that this attack was possible. You have to believe that there are plenty of other such critical exploits out there that we simply haven't thought of yet. Scary.



Kaminsky developed his own tool called "DNSRake" for performing the exploit he described in his Blackhat talk. The ideas from the talk were eagerly latched onto by other "security researchers" and further refined and weaponized. There is a Metasploit module that implements a Kaminsky-style attack.

The basic concept of the attack is straightforward:

1. Figure out a way to get the remote name server to query for a name like `0.sans.org`
2. Immediately send a large number of spoofed responses using various random query IDs that purport to come from the `sans.org` name servers (there will typically be multiple DNS servers for any given domain and the attacker can't be sure which one the remote name server is going to choose, so they have to send spoofed responses from all of them) with the bogus glue records for the `sans.org` name servers which point to the attacker's name servers.
3. Query the remote name server for what it thinks are the name servers for `sans.org` ("`dig @ns.unsuspecting.com sans.org ns`"). If the attacker sees their spoofed name servers in the reply, then the attack was successful. If the remote name server is behind a firewall and cannot by queried by the attacker, then the attacker just monitors the debugging output from their name servers and look to see if they start getting lots of query traffic from `unsuspecting.com` for hosts in `sans.org`.
4. If the initial attack doesn't succeed, the attacker just starts over at Step #1 with a new hostname like `1.sans.org` (repeat as necessary until success).

Typically, it takes an attacker less than 10 minutes to poison a vulnerable name server.

A good write-up of the Kaminsky attack can be found at

http://www.unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html

The "Fix" Doesn't Solve the Problem

"Fix" is to randomize source ports:

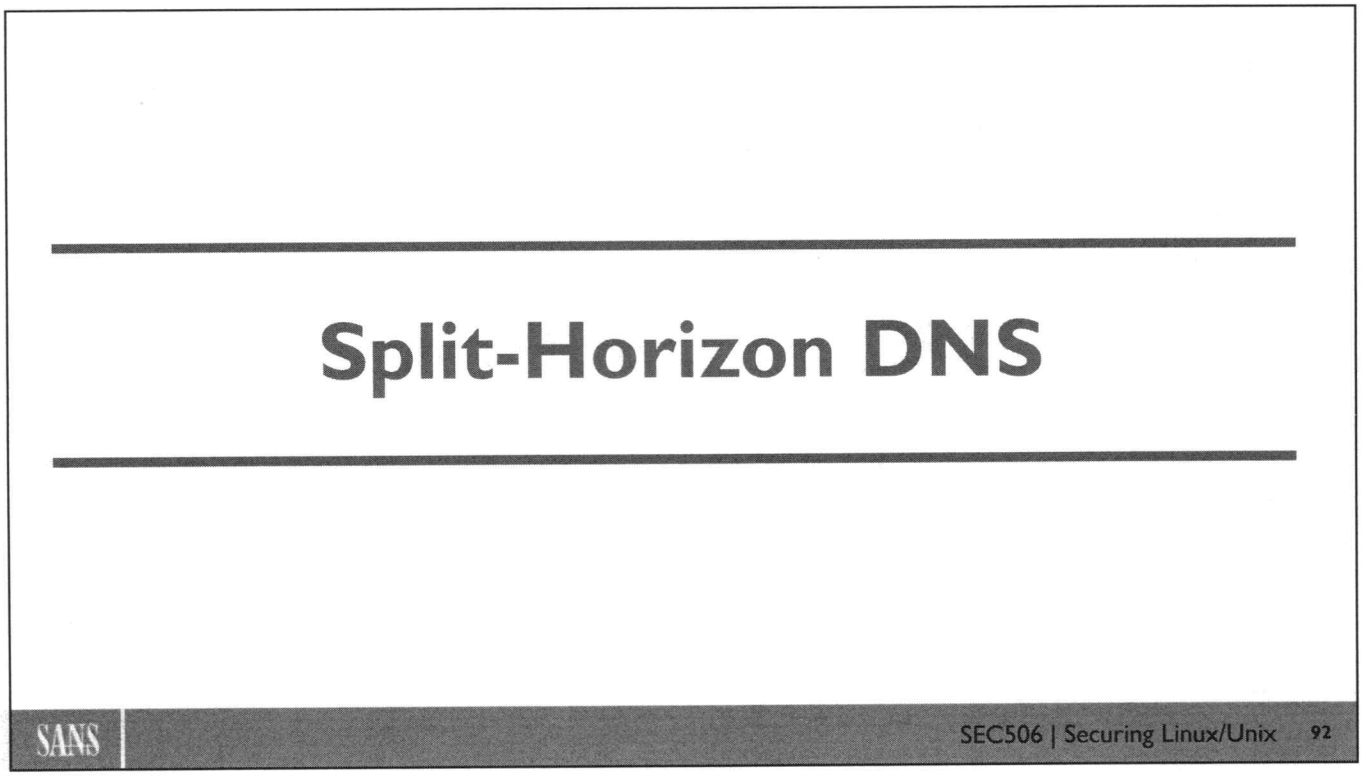
- Adds additional 16 bits of entropy (plus 16-bit query ID)
- Doesn't fix problem, just makes attackers' job hard enough to prevent easy exploits

Real fix is DNSSEC, but rollout has been complicated...

The ISC and other DNS software providers have "fixed" the problem by using random source ports for outgoing queries. This means that in addition to having to guess a random 16-bit query ID for their spoofed response, the attacker also has to guess a random 16-bit destination port. 32 total bits of randomness is deemed sufficiently difficult so as to make the attack impractical—if it takes about 10 minutes to perform the attack without random source ports, then you'd expect it to take about 16 months to perform the attack against 32 bits of randomness. Assuming you have an IDS or some other way to detect the attack in progress, you should be able to shut it down before the attacker succeeds.

What's disturbing, however, is the idea that somebody might modify the Zodiac tool to sniff the outgoing query ID and source port and then fire in a spoofed referral at the name server that originated the query. In fact, such an attack would also know which of the top-level name servers was being queried, so the attacker would know which of the servers the spoofed response should appear to originate from. It requires the attacker to have the network access to snoop the outgoing queries, but it would be a very nasty exploit.

The true fix for all of these DNS spoofing attacks (Kaminsky, Zodiac, etc.) is to implement DNSSEC, which creates a "chain of trust" throughout the DNS hierarchy and uses digital signatures to validate DNS responses. More on this later.



Split-Horizon DNS discusses the theory behind presenting one version of your DNS information to the outside world and a completely different view internally—why and when this is useful and some architectural issues with such a configuration.

What Is Split-Horizon?

- One version of DNS for outside, keep different DNS inside
  - Outside gets "bare-minimum" information
  - Inside sees complete set of DNS records
- Implies running two different sets of name servers
- Can be combined with DNS proxying

As discussed earlier, giving away too much DNS information to the outside world can help attackers map your networks or choose vulnerable or otherwise "interesting" machines to target for initial attacks. *Split-horizon* (sometimes called *split-brain*) *DNS* is a DNS configuration where an organization presents one set of DNS information to external organizations and reserves a second, separate set of DNS information for internal use. This is generally done by maintaining two different collections of name servers: an "external" set which publishes the limited amount of DNS information that external organizations need to interact with your company, and an "internal" set which holds your complete, rich set of DNS information. Note that while the separate DNS zone databases are generally maintained on two different sets of physical machines (and this is the configuration we shall describe in this course) it is possible under BIND v8 to run to different name servers with different zone databases on the same machine.

When your internal name servers wish to resolve external host names they must contact root name servers and name servers at other Internet-connected sites. This can open up your internal name servers to attack from the outside. For this reason, many organizations that run split-horizon DNS also employ a sort of DNS proxying (*slave forwarding name servers*) to "hide" their internal name servers completely from the outside world.

Digression: IP-Based Auth ...

- Remote server receives IP source address
- Server does a DNS lookup to map IP address to hostname
- Server then looks up hostname to get IP address
- If addresses don't match, access is denied

One common simple form of authentication used by some Internet servers (often FTP daemons, and services protected with TCP Wrappers) uses a combination of reverse (IP to hostname) and forward (hostname to IP) DNS lookups. A remote server will take the IP address it receives as the source of a connection and reverse that address to a hostname. Looking up this hostname yields an IP address. If the original address received as the source of the connection doesn't match the IP address retrieved from DNS, then the remote server assumes the connection comes from an attacker trying to spoof IP addresses and/or DNS information and will not allow the remote machine to connect.

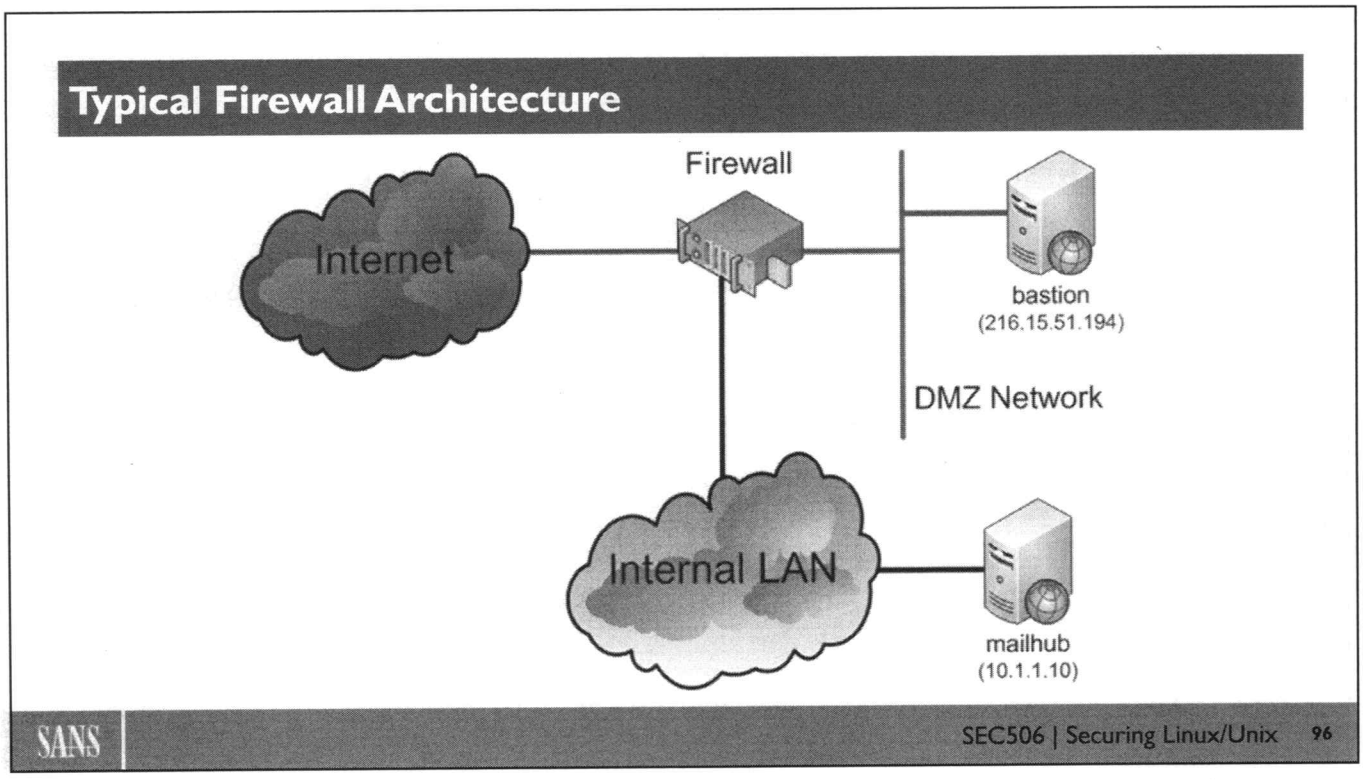
... Split-Horizon Implications

- Must advertise all hosts with Internet routable IPs
- Thus, split-horizon useful only with NAT or proxy servers
- Could use dummy records if all hosts must be advertised

The implication of this form of authentication is that sites should maintain both forward and reverse DNS information for all hosts that are capable of connecting directly to the Internet. This information must be made available in your "external" DNS database in a split-horizon configuration.

Unfortunately, if all of your hosts can connect directly to the Internet, that means you have to advertise information about all of your machines in your external split-horizon database—largely nullifying the usefulness of a split-horizon configuration. On the other hand, if your organization uses Network Address Translation (NAT) or proxy servers, you generally only have to advertise information about your NAT pool or proxy servers and split-horizon is useful.

Note that even if you have to advertise information about all of you internal hosts, you don't need to tell the outside world your HINFO or TXT records, so split-horizon can be somewhat useful. It is also sometimes useful to present one set of mail routing information (MX records) to the outside world and have a different configuration internally.



Here is a diagram of a simple, but fairly typical firewall configuration in use at many organizations today. The organization has a multi-legged firewall which connected the external Internet to both a semi-private de-militarized zone (DMZ) network and a private internal corporate network. The DMZ network is where an organization would put its Web and FTP servers and any other machines that the outside world needed to reach—and for purposes of this example a machine called `bastion` which will be the "external" DNS server for our split-horizon example (and also the external e-mail relay for the extra Appendix on *Sendmail* configuration). On the internal network, there is a machine `mailhub` which acts as the primary server for the "internal" DNS (and will be the internal mail server in the Appendix).

Note that combining DNS and e-mail services on a single server is *not* recommended practice since a security vulnerability in one service will quickly become an issue for the entire system. I'm combining services in this example simply to reduce the number of machines in play and make things less confusing. But in the real world, you should run your DNS servers and e-mail servers on separate machines (or at least separate VM instances).

Most Restrictive Assumptions

All inbound/outbound traffic must stop on DMZ

- Internet hosts can't talk to internal machines
- Internal hosts can't reach Internet hosts
- `mailhub` can reach `bastion`
- `bastion` has a hardened OS
- `mailhub` may be hardened

For purposes of example, we shall assume the most restrictive possible firewall configuration—assuming you have a working configuration for this environment, you can easily adapt the configuration for your own (less restrictive) firewall setup. In particular, we assume that hosts on the Internet can only route packets to hosts on the DMZ network and that the DMZ is the only network that is allowed to communicate with the internal corporate network. Internet machines are not allowed to communicate with "internal" machines and vice versa.

It's a good idea to spend effort removing dangerous services (read: everything not absolutely needed for mail and DNS transport) from the `bastion` host. Everybody in the world will be trying to break into this system. Given the chance for e-mail address (stack smashing) attacks which can work on inside machines even through the bastion host, tightening down your internal mail servers is also probably a good idea.

DNS – Goals

`bastion` is DNS for external hosts:

- Contains limited zone information
- MX records to force mail to *bastion*

`mailhub` is internal name server:

- Contains richer set of information
- Internal-only subdomains may exist
- Must proxy outgoing DNS requests

Again, the idea is that the external DNS for your organization will contain the bare minimum necessary for your organization to successfully conduct business on the Internet. It should not advertise internal hostnames, hardware or operating system information (HINFO and TXT records), and the like since this information could be useful to crackers trying to penetrate your network.

The internal DNS information at your site will contain complete information for your domain including useful host aliases, subdomain information, and system information. This DNS information will only be visible to hosts owned by your organization.

However, since the internal DNS servers are unable to talk to the Internet, we must arrange some sort of DNS proxy solution to resolve external names like *www.sans.org*.

Configuring BIND

Configuring BIND presents specific configuration examples of the DNS architecture introduced in the *Split-Horizon DNS* section, and introduces many of the new security features in recent releases of BIND.

bastion-named.conf

```
options {
    directory "/etc/namedb";
    version "like nothing you have ever seen";
    allow-transfer { 207.90.181.1; 207.90.181.2; };
    allow-recursion { 10.1/16; 216.15.51/24; };
    allow-query-cache { 10.1/16; 216.15.51/24; };
    rate-limit { responses-per-second 10; };
};

zone "." {
    type hint;
    file "named.ca";
};
```


This is the new BIND configuration file syntax that was actually introduced in BIND v8 (and is still used in BIND v9). The Syslog-NG developers essentially copied the BIND `named.conf` syntax for the `syslog-ng.conf` file, so you'll see a lot of similarities in the configuration file syntax between the two programs.

The `options` block applies globally to the rest of the configuration. The `directory` statement means that all other filenames are relative to `/etc/namedb` (e.g. `/etc/namedb/named.ca`). In fact, `directory` actually changes the value of `DESTRUN`, the default working directory for BIND.

As demonstrated earlier, it is possible for outsiders to query your running name server and find out what version of BIND you are running. Since certain versions of BIND have known vulnerabilities, you want to hide what version your name servers are running. The `version` option allows the administrator to specify an arbitrary string instead of the actual BIND version number. In this example, the attacker will be able to guess you're running at least BIND v8 (the first release where the `version` option appeared), but nothing more specific than that. Theoretically, you could change the version string to be a valid version string for an earlier version (e.g. `"4.9.7"`) to confuse attackers.

On your master DNS server, you should use the `allow-transfer` option to restrict zone transfers to only those machines which are legitimate secondary servers for your domains. If you do not specify an `allow-transfer` list, then any server on the Internet may do zone transfers from your server.

But as we discussed earlier, slave DNS servers are also capable of servicing zone transfer requests, so we need to specify an `allow-transfer` list on these servers as well. What list should you use? Some sites simply configure the same canonical list of servers on all machines, so any machine will be willing to provide zone transfers to any of the others. You also have the option of specifying `"allow-transfer { none; }"` on your slaves to completely prevent zone transfers from these machines. Personally, I like to put just the IP address of the master server in the `allow-transfer` block on my slaves—that way as the administrator of the master server I can request zone transfers from my slaves when I need to confirm that they've got the latest version of the zone files, but I still prevent unauthorized machines from doing zone transfers.

Earlier, we discussed the problem of allowing anybody to make recursive queries and/or cache queries at your DNS server—you'll most likely end up being used as an amplifier in a DNS DoS attack and the performance of your own DNS servers may be significantly affected. The `allow-recursion` and `allow-query-cache` settings allow you to specify a list of netblocks that should be allowed to perform these functions against your name server. In general, you should only list the netblocks of networks that are under your control and never allow external networks to make these sorts of queries.

In fact, if you have a name server that *only* exists to respond to queries from the outside world—as is often the case if you're doing split-horizon DNS—you may not need to allow either recursive queries or cache queries to anybody. In this case, you can use a configuration like:

```
allow-recursion { none; };
allow-query-cache { none; };
```

In our case, however, we are expecting to use the bastion server as a proxy for DNS requests from the internal network, so we need to allow recursion and cache queries from these networks.

BIND 9.9 introduced the `rate-limit` option to further reduce the opportunities to use BIND for DDoS attacks. For more details on tuning and additional options, see *https://kb.isc.org/article/AA-00994/0*

Note that `version`, `allow-transfer`, and `allow-recursion`, `allow-query-cache`, and `rate-limit` options are most appropriate for your external name servers. You can apply them to your internal name servers as well, but you may find this more of an administrative hassle than anything. Aside from having to maintain the lists of IP addresses in the `allow-transfer` and `allow-recursion` options, it is occasionally useful to be able to query your own internal name servers and find out what version of BIND they're running (so you know which ones to upgrade).

The latest version of the `named.ca` file is available from the InterNIC as

```
https://www.internic.net/domain/named.root
```

Alternatively, you can actually just run "`dig @a.root-servers.net . NS >named.ca`" to produce the necessary file.

More of named.conf on bastion

```
zone "sysiphus.com" {
    type master;
    file "sysiphus.hosts";
};

zone "51.15.216.in-addr.arpa" {
    type master;
    file "sysiphus.rev";
};
```


Now, for each zone, we specify whether this name server is the `master` server for the domain (sometimes referred to as a *primary* server by DNS administrators) or a `slave` (*secondary* server). Then we give a filename (again, relative to `/etc/namedb`) where the zone data can be found.

The `in-addr.arpa` domain is where you maintain the mappings between IP addresses you own and hostnames. Read the domain from right to left—we are saying that bastion is the master server for all addresses in the `216.15.51.0` network.

Note that `sysiphus.com` is an example domain used in this course. Be sure to replace it with your local domain name when you get back to the office!

bastion-sysiphus.hosts

```
; Local domain configuration for external DNS
$TTL 1d
@     IN SOA bastion.sysiphus.com. hostmaster.sysiphus.com. (
      2017050100    ; Serial - year/month/date/revision
      1d            ; Refresh from server
      1h            ; Retry after failure
      7d            ; Expire data
      2h )          ; Negative cache TTL

@                IN      NS      bastion.sysiphus.com.
                 IN      NS      ns1.alameda.net.
                 IN      NS      ns2.alameda.net.
                 IN      MX 10   bastion.sysiphus.com.
*                IN      MX 10   bastion.sysiphus.com.
```


The `SOA` ("Start of Authority") record defines global properties for your domain and lists contact information for the domain (`hostmaster.sysiphus.com` instead of `hostmaster@sysiphus.com` since `@` is a restricted character).

Generally speaking, the values in the `SOA` record above are appropriate for sites that don't make a great deal of DNS updates—your external DNS should be relatively static. Don't forget to increment the serial number value every time you make changes to your DNS information (here we're using a date-based serial number: the last two digits are provided in case you make more than one update on a given day).

You should always have at least two advertised name servers for your domain, just in case the network partitions. If these hosts can be situated on two different major service provider networks, so much the better. Note that the name servers listed above belong to the author's (former) ISP—please do not advertise these name servers as slave servers for your domain!

The `MX` ("Mail Exchanger") records are sending all mail for `sysiphus.com` and `*.sysiphus.com` to `bastion`. Wildcard `MX` records are dangerous if you're not using split-horizon DNS, but administratively convenient unless you have some automated mechanism for maintaining your DNS tables.

sysiphus.hosts

```
; host info below

bastion     IN    A          216.15.51.194
ns          IN    CNAME      bastion
mail        IN    CNAME      bastion

server      IN    A          216.15.51.195
www         IN    CNAME      server
ftp         IN    CNAME      server
```


Here are some standard forward address records that might appear in your external DNS information. A `CNAME` (*canonical name*) record is a mechanism for setting up an alias for a given machine. Note that we use `CNAME`s to associate functional names with specific machines—we could just as easily have configured multiple `A` records all pointing at the same IP address. While religious debate runs hot and heavy about whether `CNAME`s or `A` records should be used in these situations, the reality is that *it fundamentally doesn't matter*—there is no substantial technical reason to prefer one over the other.

bastion-sysiphus.rev

```
; Reverse address lookups for external DNS
$TTL 1d
@     IN SOA bastion.sysiphus.com. hostmaster.sysiphus.com. (
      2017050100    ; Serial - year/month/date/revision
      1d            ; Refresh from server
      1h            ; Retry after failure
      7d            ; Expire data
      2h )          ; Negative cache TTL

@                IN       NS       bastion.sysiphus.com.
                 IN       NS       ns1.alameda.net.
                 IN       NS       ns2.alameda.net.
195              IN       PTR      server.sysiphus.com.
194              IN       PTR      bastion.sysiphus.com.
```


This file is very similar to the `sysiphus.hosts` file except that it contains `PTR` records instead of `A` and `CNAME` records. Note that the leftmost column of the `PTR` record entries reverses the order of the octets of the hosts' addresses. The trailing dot at the end of the `PTR` record is vitally important.

mailhub- named.conf (1)

```
options {
    directory "/etc/namedb";
    forwarders { 216.15.51.194; 216.15.51.194; };
    forward only;
};

zone "." {
    type hint;
    file "named.ca";
};
```


Remember that we are assuming our internal name server is unable to reach Internet-connected hosts for DNS information. The `forwarders` lines cause the internal name server to send all external DNS queries to `bastion` to be resolved. In the event the query fails, the local server would normally attempt to contact a remote name server, but the "`forward only`" directive prevents this behavior.

Note that we are listing the IP address of `bastion` twice in the `forwarders` directive. The timeout on `forwarders` queries is actually shorter than the timeout on the normal DNS queries made by the external machine `bastion`. Hence, we list the IP address of `bastion` twice so that the local server will retry in the event it fails to get a response to its first query.

In the "real world" of course, you would probably have multiple external name servers to forward queries to in order to provide some redundancy. If you had name servers A, B, and C to list in your `forwarders` directive, you are probably better off listing their IP addresses `{ A; B; C; A; B; C; }` rather than `{ A; A; B; B; C; C; }`. After all, the most likely reason to not get a response to your first query is that machine A is down, so you want to try another name server first before re-trying server A.

All other configuration statements in this part of the file are the same as they were on the `bastion` host.

mailhub- named.conf (2)

```
zone "sysiphus.com" {
    type master;
    file "sysiphus.hosts";
};

zone "1.10.in-addr.arpa" {
    type master;
    file "sysiphus.rev";
};
```


This section of the file defines the zone files for the internal copy of the `sysiphus.com` zone and the `in-addr.arpa` domain for the internal network (remember the zone file names are relative to `/etc/namedb`). The main drawback to split-horizon DNS is that some information is necessarily duplicated between the internal and external versions of the `sysiphus.com` zone—primarily address information for the external "public" servers. Some sites choose to develop scripts which automatically dump out the "external" zone information from the internal zone files in order to avoid mistakes where one copy of the information is updated and not the other.

mailhub– named.conf (3)

```
zone "eng.sysiphus.com" {
    type slave;
    file "eng.zone";
    masters { 10.1.64.254; };
};

zone "corp.sysiphus.com" {
    type slave;
    file "corp.zone";
    masters { 10.1.128.2; };
};
```


In this section of the file, we see some internal domains that are not known to the outside world. In this case, `mailhub` is a `slave` server of these domains and gets its zone information from some other internal `master` nameserver.

Remember the zone file names are relative to `/etc/namedb`.

Lab Exercise

Running BIND `chroot()`ed

You'll learn this better if you do it by hand

Make sure:

- SELinux is in Permissive mode
- You ignore pre-configured `chroot()` area

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 5, Exercise 1, so navigate to *.../Exercises/Day_5/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 1
5. Follow the instructions in the exercise

DNSSEC

Earlier when we were talking about different kinds of DNS spoofing attacks, I mentioned that the "best" fix for these problems was to implement DNSSEC. But I also said that DNSSEC is "hard" on many levels. So, I'm going to show you how DNSSEC works and give you some tools for implementing it in your organization.

Before we begin, let me say that the information in this section is based on the DNSSEC implementation BIND 9.4.x and later, and discusses functionality introduced in v9.7.x and v9.9.x. If your OS includes an older version, you'll need to upgrade in some fashion—possibly by building BIND from the source distribution.

Good Idea In Theory

All DNS responses digitally signed

"Trust" comes through DNS hierarchy:
- Parent domains sign keys for children
- Keys for root zone distributed out-of-band

Should defeat all current spoofing attacks

The basic idea behind DNSSEC is reasonably straightforward:

- You generate a key pair and use it to sign your zones.
- You pass your public key back up the DNS chain to the owner of your TLD—for example, if we're `sysiphus.com`, then we give a copy of our key to the registry for `.com`.
- The TLD uses DS (Delegation Signer) records to publish your public key (actually the key fingerprint) and then signs those records with its key.
- Similarly, the root zone maintainers use DS records for all of the TLDs and sign those keys with the key for the root zone.
- Everybody manually downloads the key for the root zone from central authority and verifies the key using something like a PGP signature. Think of this as being similar to the current process of locally caching the names and IP addresses of the root name servers.

Once this infrastructure is in place, then you should be able to verify DNS responses because there will be a "chain of trust" from the root of the DNS hierarchy. Attackers will be unable to spoof DNS responses unless they manage to steal the private key for some zone (so we'll need to rigorously protect those keys).

Current Implementation? Well ...

Hassle for local zone admins:

- Zone signing: security vs convenience
- By default, keys recycle every 30 days

Political/technical issues at the root:

- TLD issues already contentious, now you're adding PKI...
- Need process for registrars to manage keys

The original DNSSEC implementation required administrators to manually sign the entire zone file for every update, even if that update was only for a single record. This architecture allowed you to keep your signing keys off-line for better security but was a burden on administrators. BIND 9.7.x introduced the ability to keep the signing keys on your name server and sign zones on-the-fly. The downside is that an attacker who compromises your name server can now steal your keys.

By default, in the current implementation, the signatures you generate are only valid for 30 days. Furthermore, the recommendation is that you generate a new key to sign your zones at each 30-day interval (this is referred to as a *rollover* in most DNSSEC documentation). DNS caching comes into play and you typically end up having multiple keys in play per zone to make everything work out. More on this later.

Beyond the problems for individual domains, there are all kinds of political and structural issues when it comes to implementing DNSSEC at the root and in the TLDs. Consider that all of the TLD maintainers and registrars are going to have to work out a scheme to get keys propagated up from the people who are registering zones and shoved into the appropriate zone databases, and then get those TLD zone databases signed. That's a big infrastructure issue for these companies.

Also, *anything* you do at the top of the DNS hierarchy is going to offend somebody it seems. Other countries already feel that the US has too much control over the DNS infrastructure and there is an ongoing political battle about managing keys in the top-level domains.

Understanding Key Types

Zone Signing Key (ZSK):

- Ephemeral key used to sign zone records
- This is the one that cycles every 30 days

Key Signing Key (KSK):

- Long-lived "trust" key used to sign ZSKs
- This is the key your parent domain signs

There end up being two different types of keys running around when you start using DNSSEC. First, there are the *Zone Signing Keys (ZSK)*. These are the keys that are actually used to sign your zone files. But these are also the keys that you should recycle every 30 days. It would be awful if you constantly had to be rolling these keys over and feeding the new keys back up to your TLD registrar. In fact, it would be totally unworkable in practice.

So, the recommended strategy is to also create a *Key Signing Key (KSK)*. As the name implies, you use the KSK to sign your ZSKs. You pass the public portion of the KSK up to your TLD registrar and that's what gets signed and put into DS records in the TLD. Basically, you're setting up a "chain of trust again"—the world trusts your KSK because it's signed by the TLD, therefore they can trust your ZSKs which are signed by your KSK, and therefore they can trust the individual zone records which are signed by your ZSK. The KSK is supposed to last for a long time—more like the multi-year lifetimes of SSL certificates.

So, why bother with all of this? Why can't we just keep the ZSKs around for longer? I assume that the thought process is that with sufficient access to a variety of DNS queries and responses it might be possible to eventually mount an attack on the private portion of the ZSK. So, the ZSK is cycled before such an attack would be computationally feasible. Since the KSK is only used to sign a limited number of ZSK records that change relatively slowly, there's less material an attacker could use to break this key, prolonging its useful lifespan.

Signing Your Zones

1. Generate KSK and ZSK(s)
2. Include new keys in zone file
3. Generate signed zone files
4. Update `named.conf` file

First, we're going to cover the basic process of manually creating signed zone files. This includes generating keys and incorporating them into your zone files and then actually signing all the records in the zone file. Then there will be some tweaks you have to make in your `named.conf` file to turn on DNSSEC and use the new signed zone file.

Once we have the signed zone files all squared away, we'll look at other issues like joining into a "chain of trust" with the rest of the DNS hierarchy and also issues that come up when you have to "rollover" your ZSK. More on this in just a moment.

Step 1: Generating Keys

```
# dnssec-keygen -r/dev/urandom -f KSK \
                        -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+21946
# mv Ksysiphus.com.+005+21946.key Ksysiphus.com.ksk.key
# mv Ksysiphus.com.+005+21946.private Ksysiphus.com.ksk.private
```

- Use `/dev/urandom` unless you feel like waiting...
- Generate ZSK by dropping "`-f KSK`"
- Make two ZSKs: "`curr`" and "`next`"

The `dnssec-keygen` command (comes with BIND) is used to generate both ZSKs and KSKs. The only difference is whether or not you include the "`-f KSK`" option on the command line. I'll also note that by default, `dnssec-keygen` uses `/dev/random` to generate entropy, which can take a long time for large keys. I generally use `/dev/urandom`, which only generates pseudo-random numbers, but which is "good enough" in practice.

`dnssec-keygen` creates file names like `K`*<domain>*`.`*<gibberish>*`.*` (the `.key` file is your public key, the `.private` is the secret half—protect this file!). I personally prefer to rename my files, replacing *<gibberish>* with a useful name that indicates which key is which. Notice that `dnssec-keygen` outputs "basename" of the file it's creating, without the `.key` or `.private` extensions.

For reasons that will become clear when we start talking about key rollover, you should actually create two ZSKs at this point. One will be our "current" ZSK which we use to sign our zone files. The other will be the "next" ZSK we use when it's time for the rollover. So, your commands for creating and renaming these keys might look like:

```
# dnssec-keygen -r/dev/urandom -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+17565
# mv Ksysiphus.com.+005+17565.key Ksysiphus.com.curr.key
# mv Ksysiphus.com.+005+17565.private Ksysiphus.com.curr.private
#
# dnssec-keygen -r/dev/urandom -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+11811
# mv Ksysiphus.com.+005+11811.key Ksysiphus.com.next.key
# mv Ksysiphus.com.+005+11811.private Ksysiphus.com.next.private
```

Step 2: Update Zone Files

```
# cat >>sysiphus.hosts
$include Ksysiphus.com.ksk.key
$include Ksysiphus.com.curr.key
$include Ksysiphus.com.next.key
^D
```

- Make sure you use the `*.key` files
- If keys in separate directory, use correct path names

If you look at the `*.key` file that's generated by `dnssec-keygen`, you'll see that it's actually formatted as a `DNSKEY` record:

```
# cat Ksysiphus.com.ksk.key
sysiphus.com. IN DNSKEY 257 3 8
AwEAAQFyb9DzHm2siAd62P+6jtaqdRp26ZPn3cYHrVEL5f5noqBLoVHh
5EuCazU1OOccErdLZu/ZPEyOl2fL3Qv+QqeUpuG1CJpaZhfZkYHuM98Y
ZxmoV3SGOc4YGJci7knEddnzpfz6MgUSMehRfKCV9csr8GQ7QwmTF5KK
JVcZNdu+6lR6z1Ku8Aym9uywD4j9s88Xy0ym/ZFttDDPq6HQsnPWZabH
JYwndasEawKIn3tQhS9oGHDqsHOODk373ETDBwhvMYtrXDxYfmLTikI8
ptFCqGZVW1GdayHpqklTNe3EI4naGzDUgwG0pBmhEiDJk0fzcRkVwDl...
```

You need to include these records in your zone file so that they can be signed with the rest of your zone. You could cut'n'paste them directly into your zone files, but since the ZSKs will be changing regularly, keeping them in separate files is actually a lot easier—especially once you start automating the rollover process. The zone signing process automatically sucks the contents of the included files into the final, signed zone file so you end up with only one file you need to copy into the directory used by your named process.

If you have a lot of signed zones, you'll find that keeping the keys and the zone files in the same directory become too cluttered. Personally, I keep all my key files in a separate directory, so my directives look more like "`$include ../keys/Ksysiphus.com.ksk.key`", etc.

Step 3: Signing Zone Files

```
# dnssec-signzone -o sysiphus.com -k Ksysiphus.com.ksk \
      sysiphus.hosts Ksysiphus.com.curr
sysiphus.hosts.signed
```

- Yes, you have to do this every time you make a zone update
- Warning: signed zone is MUCH larger than unsigned zone
- Also creates files for upstream registrars

Once you have your keys and you've updated your zone file with the "`$include`" statements, you're ready to use `dnssec-signzone` to produce a signed zone file. The "`-o`" option specifies the "origin", i.e. the name of the domain you're signing. Use "`-k`" to specify the KSK—notice that we're using the "basename" of the file without the `.private` extension. Then you specify the filename to work on and the ZSK that should be used to sign the file (again, use the "basename" without the extensions). `dnssec-signzone` outputs the name of the signed file—by default, this is *<filename>*`.signed`, where *<filename>* is the original filename specified on the command line.

Remember that you need to *regenerate* the signed zone file every time you make updates to the `sysiphus.hosts` file. Yes, this sucks. We'll look at some ways to automate this in just a moment.

The other bit of bad news is that your signed zone file can be more than 10x larger than your unsigned file. This translates to much higher memory usage in your name servers. Maybe this isn't a big deal for a small company with just a few zones. But for an ISP, hosting provider, or a TLD, this can be a huge issue.

Aside from the signed zone file, `dnssec-signzone` also generates a couple of other files. The `keyset-`*<domain>*`.` file contains a `DNSKEY` record for the public half of the KSK for the zone. The `dsset-`*<domain>*`.` file contains `DS` records. Normally you would pass this information upstream to your TLD registrar for them to include in their zone files and sign with their key.

Step 4: named.conf Updates

```
options {
        ...
        dnssec-enable yes;
        dnssec-validation yes;
};

managed-keys {
        . initial-key 257 3 8 "AwEAAagAIKlV...
};

zone "sysiphus.com" {
        type master;
        file "/master/sysiphus.hosts.signed";
};
```


OK, you've got your signed zone file. Now what?

All of the name servers (primary *and* secondary) will need to add the `dnssec-enable` and `dnssec-validation` options in their `named.conf` file. On the primary server, you also will typically have to change your `zone` declaration to use the signed file instead of the old, unsigned version.

At this point, you've established what the docs generally call an "island of trust." You've got DNSSEC enabled in your name servers and your publishing a signed zone file. But nobody else trusts your responses until you push your key info up through your registrar and get it signed. You also need to add the key for the root zone via the `managed-keys` block. You can obtain the necessary key from *https://www.iana.org/dnssec/files*

But how can you validate that DNSSEC is actually working? There are two mechanisms you can try:

1. "`dig +multiline +cd +dnssec +trace sans.org`" and you'll see all the DNSSEC validation records in the output. Try other domains to see what things look like when there is no DNSSEC enabled.
2. Alternatively, you could add this to your `named.conf`:

```
logging {
  channel dnssec_logs {
            syslog local4;
            severity debug 3;
  };
  category dnssec{ dnssec_logs; };
};
```

Configure Syslog to put `local4` logs into a separate log file, `HUP` your name server, and try those `dig` queries again. You'll see all the gory detail of the DNSSEC traffic.

Internal Testing Only

```
options {
        ...
        dnssec-enable yes;
        dnssec-validation yes;
};

trusted-keys {
        sysiphus.com. 257 3 8 "AwEAAQFy...RI24IcH";
};
```

SANS SEC506 | Securing Linux/Unix 119

You can also set up hard-coded trust within your environment for testing purposes. Since you have control over these machines, you can manually copy the public half of your KSK to each system and directly load it into the `named.conf` file on these systems.

As you can see on the slide, the `named.conf` configuration directive is `trusted-keys`. This is where you list the public info for the KSK of any zone(s) you want to be able to validate. You can pull this information out of the `K<domain>.ksk.key` file we created with `dnssec-keygen` earlier. These entries are long and cumbersome, so if you end up with a lot of them you'll probably put them in another file and just "`include`" them into `named.conf`.

Once you've got `trusted-keys` configured and turned on `dnssec-enable` and `dnssec-validation`, your resolver will start validating responses from the zones you configured in `trusted-keys`. It will also *stop* accepting unsigned responses from these zones (and will obviously drop responses with incorrect signatures).

Key "Rollover"

Signatures only good for 30 days

Should use new key each interval

DNS caching issues:
- Can't just remove old ZSK
- Publish new ZSK *before* signing can start
- Need to do this well before 30-day limit!

This is why we made two ZSKs

At this point, we've seen how to sign a zone file and join in the DLV experiment. However, the real fun starts after the first 30 days and we need to do a key rollover in our zone(s). Actually, you'd better start this process well in advance of the 30-day limit.

DNS caching combined with DNSSEC forces us to be a lot more deliberate in our rollover actions. You can't just drop your old ZSK and replace it with a new one. If we did this, we'd run into problems in two ways:

1. It's possible that another site could have cached one of your DNS records with the signature from the old ZSK, but have timed out the DNSKEY record that corresponds to the old ZSK. This can force a situation where the remote side queries your name server for the DNSKEY record for the new ZSK. Since the cached signature doesn't match the new ZSK, the cached record is marked invalid on the remote side and SERVFAIL is returned. The remote name server will *not* try and look up the invalid record again, so this situation results in denial-of-service.
2. The opposite case is also possible—namely the remote site is caching the DNSKEY record for your *old* ZSK and then tries to look up a record in the current zone that's signed by the new ZSK. Again, failure and denial-of-service is the result.

The fix for problem #1 is to keep the DNSKEY record for the old ZSK around for some period of time after you've switched over to use the new ZSK to sign the zone. At least remote sites will still be able to find the old ZSK to validate cached signatures. The fix for problem #2 is to publish the new ZSK well before you start using it to sign zones. That way it will be cached in remote name servers before the rollover.

So now you understand why I had you create two ZSKs earlier. When it's rollover time, the "`curr`" key becomes the "`old`" key, and the "`next`" key becomes "`curr`". When I do this swap, I also create a new key to be the "`next`" key. So, I normally actually have three ZSKs running around for each zone. Just to be extra safe, I also generally cycle keys every 14 days. Really, your timeout window is something more like "(30 days) —(maximum time to live)", but cycling the keys faster than that gives you some leeway with broken resolvers that cache information longer than they should.

Let's See That In Code ...

```
# rename curr old Ksysiphus.com.curr.*
# rename next curr Ksysiphus.com.next.*
# dnssec-keygen -r/dev/urandom \
    -a RSASHA256 -b 2048 sysiphus.com
Ksysiphus.com.+005+22785
# rename +005+22785 next Ksysiphus.com.+005+22785.*
# echo \$include Ksysiphus.com.old.key >>sysiphus.hosts
# dnssec-signzone -o sysiphus.com -k Ksysiphus.com.ksk \
    sysiphus.hosts Ksysiphus.com.curr
sysiphus.hosts.signed
```

"Hmmm, we should automate this ..."
"Oh really? You think?"

Here are the commands you would use to implement the rollover strategy I described on the last slide. First, the "curr" files get renamed to "old" and the "next" files are renamed to "curr". Then I generate a new ZSK and call it the "next" key. We actually don't yet have a $include directive in our zone file for the "old" key, so I add that.

Finally, I sign the zone file with the "curr" key, just as we did before. However, this time the "curr" key was what was formerly the "next" key.

Even if you do this once in a blue moon, it's likely that you'll make a typo and mess something up. And there's no way you're going to do this every two weeks by hand. So, you automate the process, but really this doesn't have to be that difficult…

Simple Automation Strategy

Main automation points:

1. Generating initial KSK and ZSKs
2. Handling regular rollover cycle
3. Signing zones during regular updates

Wrote `genkeys` script to handle #1-2

`Makefile` handles #3 (automatically calls `genkeys` to do #1)

Really there are three things we need to automate here: generating the initial keys for a zone, rolling the ZSKs on a regular basis, and automating the zone file signing process for both rollovers and for normal zone updates. I've included a couple of tools on your course DVD to make this easy.

First, there's the `genkeys` script. If you call it with a domain name as a command-line argument, then it will generate new keys for that zone—creating a KSK if one doesn't exist as well as cycling the ZSKs or creating new ones if they don't exist. If called with no arguments, it will look in its working directory (defined on the command line or in the script itself) for any keys that are 14 days old or older and cycle the keys for the associated domain. This latter mode means that `genkeys` suitable for calling from `cron` on a daily basis to automatically cycle your keys.

I've also included a `Makefile` for the directory where you keep your arezones. It looks for zone files named *<domain>*`.db` and creates `../signed/`*<domain>*`.signed` files. It will even call `genkeys` automatically if there are not currently any keys for the domain.

Similarly, `genkeys` will automatically run a "`make install`" in your zone files directory every time it ends up rolling keys for any of your zones. "`make install`" copies the newly signed zone files into place and `HUP`s your `named` process.

When the `Makefile` calls `dnssec-signzone`, it uses the "`-N unixtime`" option to update your SOA serial number automatically, so you no longer need to worry about that. It needs to be able to do this automatically so that `genkeys` can handle ZSK rollovers without human intervention. The only problem is that the "`unixtime`" (number of seconds since Jan 1, 1970) may be smaller than your current serial number. See this URL for more information on how to roll your SOA serial number over to a smaller value:

https://fatmin.com/2011/01/21/bind-zone-file-serial-number-reset/

Summary

- Important to be able to trust DNS
- Admin interface is improving
- Coming soon?

We really need to get DNSSEC implemented for security reasons. Unfortunately, there are some technical, administrative, and political hurdles that need to be overcome before this happens. But if nobody wants to use DNSSEC because it's too hard for sites to implement then we're never going to be able to tackle the bigger issues.

Lab Exercise

- DNSSEC and you
- Experiment with keys and zone signing

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 5, Exercise 2, so navigate to *.../Exercises/Day_5/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 2
5. Follow the instructions in the exercise

Apache Web Server

This section covers basic security configuration information for the Apache Web server (by far the most popular Web server on the Unix platform).

This section does not attempt to address security issues with server-side application environments like Tomcat, WebSphere, etc. Consult appropriate vendor documentation in these areas. You may also find the following "Tech Tips" from the CERT/CC useful when doing server-side programming:

http://www.cert.org/tech_tips/malicious_code_mitigation.html

Biggest Security Problems

Poorly written CGI/PHP scripts and other "server-side" code

Buffer overflows

Poor configuration practices:
- Running servers with privilege
- Running "helper apps" with privilege
- Overlapping Web root with other data/apps
- Programs, libraries, config files in doc root

Before we start talking about how to secure Apache Web servers, it's useful to look at some of the most common ways Web servers are compromised.

One common issue is not a Web server security issue per se, but rather the fault of badly written (or in some cases maliciously written) code that is run by the Web server. Complete coverage of secure server-side programming could take an entire week-long course (or more), but the short answer is disable server-side execution if you're not using it and make sure to carefully review any server-side code you are running. Also, take steps to protect the software that's being run by the server and keep it out of the normal document trees (more on this later).

The Apache Web server has had its share of buffer overflow problems. Keeping up-to-date on software releases is the primary defense here. Of course, you may be vulnerable to a previously undiscovered exploit in the current code, so some sites will run their Web servers in a `chroot()` environment (the Apache Benchmark document at *CISecurity.org* provides an example). However, `chroot()`ing a complex Apache install with lots of CGI scripts and other server-side executables can be difficult.

Most of the other common Apache exploits come down to poor administration and configuration. There should never be a reason to run the Web server or any other apps related to the server (like CGIs, MySQL servers, etc.) with root privileges—all of these processes should be running as unique unprivileged UIDs. Another common mistake is to overlap the Web document root with the data area of another server—for example, sites that mix their Web docs with an anonymous FTP upload/download area. This opens a window for security problems in one service to impact the other, or possibly sets up a way to turn small vulnerabilities in each service into a massive problem for the machine as a whole. Another common mistake is to put executable code and/or server configuration files into the document root (some sites have accidentally put server password files and site certificates in vulnerable locations!). This may allow external attackers to view the source code for your apps (making it easier to spot vulnerabilities in the software).

apache.org Compromise

Jan 2001 *apache.org* compromise is a perfect example:

- Web root and anon FTP area overlapped
- Uploads to FTP area handled insecurely
- Web server allowed server-side execution
- MySQL server running as root

Result was cute defacement, could have been much worse ...

The January 2001 break-in at `apache.org` is a good example of the kinds of configuration mistakes we were discussing on the previous slide:

- At the time of the incident, the `apache.org` server's Web root and FTP server area overlapped.
- Furthermore, the FTP area allowed files to be uploaded with insecure permissions.
- The attackers were able to upload a PHP document that allowed them to execute arbitrary commands via the Web server.
- The attackers also uploaded a network listener daemon, which they triggered via their PHP document that gave them a shell (as an unprivileged user) on the box.
- Once on the box, they investigated the system and found that the MySQL server was running as root and was accessible from their account. A username and password for accessing the server were found in one of the CGI scripts on the box.
- They were able to coerce the MySQL server into writing arbitrary files (though it wouldn't overwrite existing files). The attackers put a "create set-UID shell" script into `/root/.tcshrc` and waited for one of the admins to `su` to root. Game over.

I've made a copy of the full description of the incident (from the perpetrators themselves) at

http://deer-run.com/~hal/apache-Y2K-defacement.txt.

Perhaps the most humorous part of the write-up is the comment from the attackers: "We didn't wanted [sic]to use any buffer overflow or some lame exploit." Nice to see they have some professional standards…

The Rest of This Section

- Configuring Apache
- SSL Configuration
- **`mod_security`**

The rest of this section is devoted to looking at the security-related issues around running an Apache Web server. This is not a complete course on how to administer Apache, just enough to allow you to talk to your server admins and content developers about the "right" way to run things.

Configuring Apache covers the basic Apache server configuration directives that can be used to improve security.

SSL Configuration looks at how to turn on SSL support in Apache and also covers topics like getting server certificates.

Finally, we'll take a quick look at *mod_security*, an Apache add-on module that can be used as a Web Application Firewall (WAF).

Configuring Apache

In this section, we'll look at some basic "best practices" to use when configuring your Web server. Many of the topics covered here are also covered in the Apache "Security Tips" document available from *http://httpd.apache.org/docs/2.2/misc/security_tips.html*

All configuration directives discussed in this section are set in the standard `httpd.conf` configuration file used by Apache.

Basic Server Config

```
ServerName www.sysiphus.com
UseCanonicalName on
Listen 0.0.0.0:80              # default

User www                       # create special
Group www                      # ditto

ServerAdmin webmaster@sysiphus.com
ServerSignature off
ServerTokens Prod              # hides version
```


Setting the Web server's internal `ServerName` may not seem important from a security perspective, but using a single canonical name can be important with sites that are password protected. Basically, this is because browsers cache authentication information by hostname (and one other piece of information, the authentication "realm name", which we will discuss when we talk about password authentication in Apache)—if the Web server doesn't use a consistent canonical name, then you may end up prompting users several times for password access to the same information. The `UseCanonicalName` directive tells the Web server to always use the value of `ServerName` when constructing so-called "self-referential" URLs so that naming will be consistent throughout your website.

We'll talk about the `Listen` directive on the next slide. `User` and `Group` determine what privileges the Web server should run with. Create a specific UID and GID for your Web server (don't use `nobody` because other services use that UID) and *never* run Web servers as root.

`ServerAdmin` is the e-mail address of the person who should be contacted in case of problems—this address appears in various error messages by default. `ServerSignature` controls whether or not a footer message (which happens to contain the Apache server version number) should be appended to server-generated error messages and other server generated docs. Generally, we'd like to hide the server version by turning this off. Similarly, `ServerTokens` controls the format of the server header returned on HTTP requests. The most minimal setting is "`Prod`" which causes on the word "`Apache`" to be displayed (without version number, etc.)—many automated worms trigger on the Apache version number presented here, so you can slow down infection by hiding this information. If you want to hide that you're running "`Apache`", you'll actually have to hack the `httpd.h` file in the source distribution to change the values of the `SERVER_BASE*` variables and recompile. If you're going to this extreme, though, you'll also want to change the default error documents, etc. because the format of these messages gives away the fact that you're using Apache.

Quick Aside: Local Servers

- Developers may need local Web servers
- Security admins worry about potential security disaster with that many servers
- Consider having server bind to localhost:

```
ServerName localhost
Listen 127.0.0.1:8080
```

- Now only local developer sees server on their machine

`Listen` sets the IP address and port number which the server is listening on. By default, the Web server will try to listen on port 80 for all of the machine's IP addresses (represented as the special address `0.0.0.0`), but the admin may choose to bind the server to a specific address if desired. This could allow the admin to bind the server to a special "protected" interface on the system (and this is also used for setting up Web servers on "virtual" IP addresses for hosting multiple sites on the same physical machine). But there's also another use for the `Listen` directive.

In Web development shops, it's pretty common for every developer to have their own local Web server(s) running for development and testing. That's a lot of Web servers on a network—any one of which might be running some untested code which contains a security vulnerability. Not to mention the fact that the Web server software itself might have a buffer overflow or other remote exploit. While the network used by your developers is probably protected by your corporate firewalls and security infrastructure, local security admins may worry about having so many essentially "self-supported" Web servers running out there.

One option is to use the `Listen` directive to bind the server only to the "loopback" address on the system—address `127.0.0.1`. The developer using the system can still "see" their local Web server by pointing their browser at "`http://localhost/`", but nobody outside of the system will be able to access the Web server. This seems like a reasonable compromise for everybody concerned.

Start at the Top

- Default policy should block all access:

```
<Directory "/">
    Options None
    Order allow,deny
    Deny from all
    Satisfy all
    AllowOverride None
</Directory>
```

- Override settings for specific dirs as needed

The next step in server configuration is to define access controls for the various directories served by the Apache software. "Best practice" is to first define a general "default deny" stance for the entire file system as we do here. Later, we will configure access to specific directories that are actually part of the Web server's document trees.

If we first set a default policy of allowing no access and then have to explicitly enable access for Web server directories, this can help prevent "accidental" misconfigurations where material ends up getting into the Web server tree that doesn't belong there. This "default deny" policy also helps protect the server against attacks (like the recent "directory traversal" attack reported in Aug 2002) which allows external attackers to "escape" from the document trees and walk the file system of the local server.

So here we're setting options for the directory "`/`", or the root of the file system on downward. These options are put into a `<Directory...></Directory>` "container" which specifies the specific physical directory structure the options apply to. The alternative is to use `<Location...></Location>` which specifies a URL path rather than a physical directory. Frankly, `<Location...>` is confusing—best to stick with the literal path names on your system and use `<Directory...>`.

The next fifteen slides or so cover what all of these configuration options mean in great detail …

Setting Options Field

- Options can be any/all of the following:

```
Indexes
FollowSymlinks/SymlinksIfOwnerMatch
Includes/IncludesNOEXEC
ExecCGI
MultiViews
```

- "All" and "None" are also supported
- Some options have surprising results ...

Options defines a list of the optional behaviors that are defined for the Web server on a given directory structure. Generally, the Options directive is written as a space separated list of the options you want turned on:

```
Options Indexes Multiviews FollowSymlinks
```

Any option not specifically listed is disabled.

However, an alternate form is to list options with +/- to specifically enable/disable certain options:

```
Options +Indexes +Multiviews +FollowSymlinks -Includes -ExecCGI
```

It turns out the second form shown above can have some unexpected consequences when Options is set on both a parent directory and one of its subdirectories—when +/- is used for all options, then the *union* of the Options for parent and subdirectory is used. Best to use the first form.

"Options All" sets all options *except MultiViews* which must be set explicitly (this appears to be due to the "organic" nature of the growth of Apache—MultiViews is a relatively recent development). "Options None" disables all options (per our "default deny" stance on the previous slide).

Of all of the options shown here, only MultiViews has no real security implications (MultiViews enables "server negotiated" content—typically used to select multiple versions of the same Web document translated into different languages). But let's take a look at the other options right now…

Indexes Option

- Accessing a web directory usually shows `index.html` file
- If no `index.html`, server returns "404 Not Found"
- If `Indexes` is turned on, directory listing is displayed
- May not want this–
  - Can show "private" files
  - Give away docroot structure

Normally when you view a URL like `http://www.sans.org/info/`, the Web server displays a file called "`index.html`" in the "`info`" subdirectory of the Web server document root (actually, the file name doesn't have to be `index.html`—the file name can be set by the server administrator). If this file is not found, then by default the Web server will display the ever-popular "`404 Not Found`" error.

However, if `Indexes` is turned on for a directory, the Web server will instead show a directory listing of the other files and subdirectories in the given directory. Sometimes this is useful—particularly if you're using your Web server as a mechanism for sharing files. However, it can also be a problem—if somebody deletes the `index.html` file or forgets to create it in the first place, then you may be exposing files and directories that you didn't want to expose. It may also allow remote users to navigate into sections of your document tree that they shouldn't know about (though honestly if the information is that sensitive it either shouldn't be on the Web server in the first place or at least should be password protected).

Generally, it's best to disable `Indexes` unless you're sure you're going to need it. This has the best chance of preventing "accidental" errors which expose your doc root to scrutiny.

Symlink Options

`FollowSymlinks`: Web server will follow symlinks–
even out of normal doc tree
`SymlinksIfOwnerMatch`: link owner and file owner must match

- Disable `FollowSymlinks` to improve security, but has performance impact
- `SymlinksIfOwnerMatch` doubles performance impact

When `FollowSymlinks` is enabled, then the Web server will simply "open" any symbolic links in the doc root without question and display the contents of the file pointed to by the symbolic link. This means that content maintainers could accidentally create a symlink which points to a critical system configuration file and expose the contents of that file to the world. Now, of course, this file must still be readable by whatever user the Web server is running as, but this still may be something of a concern.

So, the first impulse is to turn `FollowSymlinks` off. However, doing this has an unexpected performance impact. The server must now check each "file" in the doc root before opening it, just to verify that the file is not, in fact, a symlink. This means an extra system call before each and every file opened by the Web server—which amounts to a lot of extra work on a busy server. For this reason, many sites choose to leave `FollowSymlinks` enabled. You can scan your document trees periodically for symlinks (perhaps even verifying that the links don't point out of the doc tree), or even run the Web server `chroot()` ed if you're really concerned.

Another option is `SymlinksIfOwnerMatch`, which means only open the file pointed to by the symlink if the owner of the file and the owner of the symlink are the same UID. This was designed to prevent Web developers from accidentally pointing to root-owned system configuration files. Of course, this now means that the Web server potentially has to do *multiple* system calls on each file open—one to check for a symlink, and then additional calls to get the name of the file the link points to and the owner of that file.

Still, if your Web server is not heavily loaded, just don't enable `FollowSymlinks` at all.

Includes Option

- Allows "server parsed HTML" capability
- Has some real advantages:
  - Include standard headers/footers
  - Add limited dynamic content w/o using CGI
- Beware `#exec` option, however
- Consider using `IncludesNOEXEC`, which disables `#exec`

`Includes` turns on "server parsed HTML" capability (usually indicated by files with the "`.shtml`" extension). This is a simple way to get bits of dynamic content into otherwise static HTML files without going to the trouble of writing a full CGI or PHP program. Another use for server-parsed HTML is to automatically include certain standard header/footer information in many different documents. For example, some sites include a copyright statement at the bottom of each page—updating the copyright date in a single file is much easier than having to modify all of the Web pages on your site.

However, server parsed HTML includes the `#exec` command which runs a specified command and substitutes the output of that command into the document (like the `date` command to display the current date/time on the page). This opens the door for a badly written document to execute a command which has unexpected consequences on the system. As an alternative, you can set `IncludesNOEXEC` which enables server parsed HTML, but disables `#exec`.

As an aside, we mentioned that server parsed HTML files are generally distinguished by the "`.shtml`" extension. This is accomplished by setting up a special "handler" for files with this extension:

```
AddType text/html .shtml
AddHandler server-parsed .shtml
```

Another option, however, is to set "`XBitHack on`", which causes any HTML file that has the execute bit set to be treated as a server parsed HTML file. On the plus side, this means that outsiders won't know that the file is server parsed HTML because you no longer need the "`.shtml`" extension. On the minus side, it's easy to mess things up and forget to set the execute bit (or lose the bit when copying files from one directory to another) or to set this bit on files which shouldn't have it.

Enabling CGI Access

- Avoid using "`ExecCGI`" which scatters CGI scripts throughout doc trees
- Restrict CGIs to a tightly controlled area:

```
ScriptAlias /cgi-bin/ "/var/apache/cgi-bin/"
<Directory "/var/apache/cgi-bin/">
        Options None
        AllowOverride None
        Order deny,allow
</Directory>
```


There are two ways to enable CGI scripts with Apache—a "right way" and the "wrong way." The "right way" is to restrict CGIs to one (or at worst a small number) of tightly controlled directories which are not part of the normal document tree. Only a few people should be authorized to add CGIs to this directory, and then only after close scrutiny of the program and some serious testing. CGI directories are created with the `ScriptAlias` directive—in the example on the slide we're saying that `http://www.ourdomain.com/cgi-bin/foo` means "run the program `foo` from the directory `/var/apache/cgi-bin`". Our "default deny" stance that we set up on the root of the file system means that we need to explicitly allow access to this CGI bin directory using the configuration commands in the `<Directory...></Directory>` container.

The alternative to setting up restricted CGI bin directories with `ScriptAlias` is to use the `ExecCGI` option on directories in the document root. Typically, `ExecCGI` needs to be combined with an `AddHandler` directive so that the server can recognize CGI programs:

```
AddHandler cgi-script cgi pl sh
```

Now any file that ends with the extension `.cgi`, `.pl`, or `.sh` will be executed as a CGI script.

The problem with `ExecCGI` is that now anybody with access to the doc root can drop in a CGI script that could contain a vulnerability that compromises the system. If there are a lot of people providing content to the Web server, the situation can get out of control quickly. The Apache "Security Tips" document recommends using `ExecCGI` if and only if:

- You trust your users not to write scripts which will deliberately or accidentally expose your system to an attack.
- You consider security at your site to be so feeble in other areas, as to make one more potential hole irrelevant.
- You have no users, and nobody ever visits your server.

IP-Based Access Control

- Use "`Allow from`" and "`Deny from`" ACLs with:
  Hosts – use host name or IP address
  Networks – use "net/mask" or CIDR notation
  Domains – use domain name
- Must be used with `Order` directive–the confusing part ...

Now let's look at how to control access to Web directories based on the IP address of the remote machine.

Apache uses "`Allow from`" and "`Deny from`" to permit/deny access based on IP address. You can specify host names or IP addresses of individual machines, or even entire networks using either network/mask syntax (e.g. `192.168.1.0/255.255.255.0`) or CIDR notation (`192.168.1.0/24`). You can even specify domain names. Note however that using host names and domain names depends on attackers not being able to spoof or corrupt the local DNS server—best to use IP addresses whenever possible.

By the way, you can also use the special keyword "`all`" ("`Deny from all`", "`Allow from all`") as a wildcard.

If "`Allow from`" and "`Deny from`" were the entire story, then things would be simple. However, there's an extra complication because of Apache's `Order` directive—more on this on the next slide…

The `Order` Directive

- Just remember that `Order` controls:
  - Eval order for `Allow/Deny` commands
  - Default behavior if no matching `Allow/Deny`

"`Order Allow,Deny`"
- Process `Allow` statements, then `Deny` rules
- Default is *access denied*

"`Order Deny,Allow`"
- Do `Deny` statements first, then `Allow` rules
- Access is *allowed* if no match

What's confusing about the `Order` directive is that it actually controls two essentially unrelated parameters. First, the `Order` directive controls the order in which `Allow/Deny` statements are processed—which is sort of intuitive given the name "`Order.`" However, `Order` also defines a *default* policy for connections which don't match one of the explicit `Allow/Deny` statements given by the administrator—and, of course, the default policy is different depending on how the `Order` statement is written.

Basically, the administrator has two options. "`Order Allow,Deny`" says to process the "`Allow from`" statements first, and then all of the "`Deny from`" statements. However, "`Order Allow,Deny`" also means that the default is to drop any connections that don't have an explicit matching "`Allow from`" (in other words, "`Order Allow,Deny`" is the standard *default deny* stance).

The alternative is "`Order Deny,Allow`". Here the "`Deny from`" statements are processed first, and then the "`Allow from`" statements. The default for "`Order Deny,Allow`" is to permit any connections that aren't explicitly blocked by "`Deny from`" (making this the *default permit* stance).

Confused yet? Let's try the example on the next slide…

An Example

- Suppose we had the following

```
Allow from sysiphus.com
Deny from foo.sysiphus.com
Order Allow,Deny
```

- What happens with various connections?
- Now reverse the `Order` statement—what happens?

Given the example shown on the slide, let's consider a few different scenarios:

- If a connection comes in from the host `foo.sysiphus.com` it will be denied—first the "`Allow from`" is consulted (access would be permitted because `foo.sysiphus.com` is part of `sysiphus.com`), and then the "`Deny from`" (which explicitly blocks `foo.sysiphus.com`).
- Connections from any other host in `sysiphus.com` would be allowed—these would match the first "`Allow from`" and be allowed, falling through the "`Deny from`" directive because they don't match.
- Connections from all other hosts would be blocked—these connections don't match either the "`Allow from`" or the "`Deny from`", so the implicit "deny everything" caused by "`Order Allow,Deny`" takes over and drops these connections.

OK, that was easy! Now suppose we changed the `Order` line above to "`Order Deny,Allow`" and left everything else the same:

- Now all of a sudden, connections from `foo.sysiphus.com` are *allowed*—the "`Deny from`" is now consulted *first* (which would normally block the connection), but then the "`Allow from`" is checked (which allows connections from anything in `sysiphus.com`—including `foo.sysiphus.com` which otherwise would be blocked).
- Anything else in `sysiphus.com` is also allowed because of the "`Allow from`".
- Other hosts are also *allowed*—remember that our default policy is now "permit everything" because we now have "`Order Deny,Allow`".

Frankly, it's hard to imagine how the Apache folks could have messed this up any worse. They should have simply emulated the same sort of ACLs used by TCP Wrappers and have done with it, but they didn't.

Granting Access to Doc Root

```
DocumentRoot "/var/apache/docs"

<Directory "/var/apache/docs">
    Options Indexes SymLinksIfOwnerMatch \
            IncludesNOEXEC MultiViews
    Order deny,allow
    AllowOverride None
</Directory>
```


OK, with all of that out of the way, let's look at how we might allow access to our Web server document tree. Remember that we implemented a "default deny" stance on the root file system, so we have to explicitly enable access to our `DocumentRoot`.

We do this by setting options on the appropriate directory in another `<Directory...></Directory>` container—these settings will override the settings for the "`/`" directory we set up earlier. First, we set our `Options` directive to include the options we want on our doc root. Next, we need to allow access from all hosts on the Internet—setting "`Order deny,allow`" means that our default access control policy is to allow ali access.

The `AllowOverride` option will be discussed later in this section.

Password Access

- Directories can be password protected
- Web is stateless – username/password resent for each page
- Standard is "Basic Auth" – passwords are sent in the clear
- Alternative is "Digest Auth" – but there are issues

In addition to IP address based access controls, portions of your Web space can also be password protected. When entering a protected document space for the first time, the user is prompted for a username/password (this is generally referred to as "HTTP Basic Auth"). What the user doesn't see is that this same username/password is transmitted over and over again on each and every request for a document from that section of the Web tree—the browser caches the username/password and simply resends it without prompting the user (when the browser is killed and restarted, the user must re-enter the password). The bad news is that the username/password travels over the network in the clear and can be easily sniffed. So, you should only use HTTP Basic Auth in combination with SSL encryption.

The alternative to Basic Auth is "Digest Auth"—Digest Auth transmits an MD5 hash of the user's password (plus other data) rather than the password itself. This prevents the password from being sniffed directly, but brute-force attacks are possible to recover the password from the MD5 hash. However, Digest Auth generally requires that passwords are stored in clear text on the server side, which is a problem if your web server is ever compromised. There has also historically been a problem with general browser support for Digest Auth, but all of the major browsers should support it at this point. For more details on issues around Digest Auth, see

http://en.wikipedia.org/wiki/Digest_access_authentication#HTTP_digest_authentication_considerations

To enable Digest Auth in Apache, you must configure the server with `--enable-digest` and recompile. Further information on configuring Digest Auth can be found at:

http://httpd.apache.org/docs/current/mod/mod_auth_digest.html

Managing Password Files

- Format of Apache password files is:

```
username:password[:ignored]
```

- Don't use `/etc/shadow` file!
- Use `htpasswd` command to manage Apache password file
- Keep password files out of doc trees!!!

To password protect part of the Web tree, we first need a password file. The format of Apache password files is simple—just the username, a colon, and an encrypted password string. Anything after the encrypted password is ignored by Apache—so it's possible to simply use a copy of the system `/etc/shadow` file (you'd have to use a copy because `/etc/shadow` wouldn't normally be readable by the Web server user). This is a terrible idea for two reasons: (1) you'd have a copy of `/etc/shadow` on your system that was readable by a non-root user, and (2) users should use a different password for Web access and login access because Basic Auth insists on retransmitting the username/password all the time. It's also important to put the Apache password files someplace *outside* of the Web document tree (nothing is worse than finding your Web password file indexed on an Internet search engine). As we'll see, Apache allows the administrator to specify the location of the password file anywhere in the file system.

The `htpasswd` command provided with the Apache distribution allows the admin to add/remove users and change passwords. The syntax is "`htpasswd [-ms] <pwdfile> <username>`" where `<pwdfile>` is the path to the Apache password file, and `<username>` is the name of the user being added/changed. By default, `htpasswd` uses old-style DES56 password encryption. The `-m` option tells `htpasswd` to use MD5 encryption, and `-s` uses SHA-1 (the same password file may have different passwords using all three types of encryption).

So, here's how to add user "`hal`" to `/var/apache/etc/passwd` with MD5 password encryption (the same command is also used to change the password for user "`hal`"):

```
htpasswd -m /var/apache/etc/passwd hal
```

In this mode, the admin will be prompted for the password for the user's new password. `htpasswd` also supports "batch mode" where passwords can be entered on the command line (good for scripts):

```
htpasswd -mb /var/apache/etc/passwd hal NewP8ssw0rd
```

Of course, this means that the user's password would be at least momentarily visible via `ps`.

Sample Directory Config

```
<Directory "/var/apache/docs/private">
    AuthType Basic
    AuthName "Private Info"
    AuthUserFile "/var/apache/etc/passwd"
    Require valid-user
</Directory>
```

- "Require" directive is *critical*
- Can optionally use Berkeley DB files for performance

Here's how to actually configure password based access control on part of the doc tree. "AuthType Basic" says that we should use HTTP Basic Auth (as opposed to "AuthType Digest" for Digest Auth).

"AuthName" is a string that is used as part of the username/password prompt by the browser—in this example, the prompt on the pop-up box would read something like "*Enter username/password for Private Info.*" Actually, recalling our earlier discussion of ServerName and UseCanonicalName, AuthName is really the authentication "realm name" that the client browser uses (along with the server name) to remember the username/password credential for the protected area of the remote website. In other words, two password protected directories on the same Web server that used two different values for AuthName would require the user to enter a username and a password for each specific directory. If the two directories used the same value for AuthName, then the user would only be prompted once.

"AuthUserFile" is the location of the Apache password file—again, put this file someplace *outside* your document tree. Different directories can use different password files or all share the same file. Be careful here—if you configure multiple directories with the same AuthName, then also configure them to use the same AuthUserFile. Otherwise, you could run into problems when the files get out of synch and the user's password that's valid in one file is not valid in the other.

The Require directive says that the user must enter a valid username/password before being given access. If you forget to put the Require directive into the config file, then the username/password box will pop up but *any username and password will allow access!* The reason the Require directive exists at all is that it's also possible to say "Require hal sally bob", which would only grant access if the username were one of the three listed users (and obviously only if the correct password for the given account was entered as well).

One problem with standard flat text password files is that as the number of users grows, finding a particular user in the file takes longer and longer. And again, remember that the username/password verification has to be done on each and every document access from the restricted area. For performance reasons, it's possible to create your password files as DBM or Berkeley DB style database files to enable faster lookups. You must reconfigure

Apache with `--enable-auth_db` or `--enable-auth_dbm` flags and recompile before this is possible, however. Once the server has been properly built, simply use `AuthDBMUserFile` instead of `AuthUserFile`. The Apache distribution comes with the `dbmmanage` program for managing user accounts in DB or DBM style database files. Again, for more info see:

`http://httpd.apache.org/docs/current/mod/mod_authn_dbm.html`

Access by Password *or* Address

```
<Directory "/var/apache/docs/locals-only">
    AuthType Basic
    AuthName "Local Info"
    AuthUserFile "/var/apache/etc/passwd"
    Require valid-user
    Order allow,deny
    Allow from 192.168.0.0/16
    Allow from 127.0.0.1
    Satisfy any
</Directory>
```


Now it's possible to discuss the "`Satisfy`" directive we saw in an earlier example. `Satisfy` simply controls the relationship between the various forms of access controls defined in a particular part of the doc tree. This includes IP-based access controls and password-based access controls. Basically, the choice is either "`Satisfy all`", which means that both the IP based restrictions must be satisfied and a proper username/password entered, or "`Satisfy any`" which means that either the IP address must match or a correct username/password must be used.

This latter case is useful when you want to have a section of your Web tree that's only visible to "local" people. In the example on the slide, our IP-based access controls allow access from the `192.168.0.0` network—since "`Satisfy any`" is defined, then as long as the connection comes from a host on this network, a password will not be required (in fact, the prompt will never even appear since the IP address of the source of the connection is checked first). Connections from other IP addresses will cause the password popup box to appear and force the user to enter a valid password.

What's AllowOverride Do?

➢`AllowOverride` allows configuration in doc trees

PRO:

- Delegate administration to various groups
- Reconfigure server without restarting

CON:

- Security policy spread throughout doc tree
- Performance hit

The last configuration command to discuss is the `AllowOverride` option. Enabling `AllowOverride` allows the owner of the document tree to create a configuration file in the doc tree itself which overrides settings in the `httpd.conf` file (usually these files are named `.htaccess`, but this is customizable as we'll see).

On the plus side, this allows the Web server administrator to delegate administration to the owner of the document tree. For large Web environments, this may be the only manageable solution. Another advantage is that these configuration files are read every time a document is accessed from the given document tree, so any changes made to the configuration in these files takes effect immediately without having to restart the Web server (changes to `httpd.conf` require at least a `HUP` to the running daemon).

This is actually also a problem as well. Since the Web server has to look for these per-directory configuration files when `AllowOverride` is set, and since these files then need to be parsed by the Web server when found, this causes a performance hit for each and every file access. In fact, if `AllowOverride` were set on `/var/apache/docs`, and the file being accessed were in `/var/apache/docs/local/config`, the Web server would actually have to look for override files at each level of the tree from `/var/apache/docs` on downward. This is a real problem for deeply nested trees.

The other problem is that the security policy for the Web server is now spread throughout the document tree. This not only makes it difficult to know how the server or a particular part of the document tree is configured, it also makes it easier for somebody to make a mistake which opens up a vulnerability on the machine. And since the mistaken configuration directive in hidden away in some random file in the doc tree, it could go unfound for a long time.

AllowOverride Syntax

`AllowOverride` can be "`All`" or "`None`", or list from:

- `Options` – Change Options settings
- `Limit` – Address-based access controls
- `AuthConfig` – Password auth configuration
- `Indexes` – Directory listing "look and feel"
- `FileInfo` – Document and language control

To allow setting the `Indexes` option, use the `Options` override, not `Indexes`!

The Web server administrator actually has some level of control over what configuration settings may be overridden in the per-directory config files. `AllowOverride` can be set to "`None`", which disables overrides entirely, or "`All`", which basically means that any configuration option can be overridden. However, the administrator may also choose to only allow overrides for certain groups of commands, as defined by the five options shown here.

"`AllowOverride Options`" means that the per-directory config files can override any of the `Options` set in `httpd.conf` for that directory (it also allows the `XBitHack` option for server parsed HTML to be turned on here as well). The `Limit` override allows the per-directory config file to set "`Allow from`", "`Deny from`", and "`Order`" to set IP-based access controls. `AuthConfig` allows the per-directory config file to set all of the `Auth*` and `Require` directives used for password-based access controls.

`Indexes` and `FileInfo` don't particularly have any security-related impact. "`AllowOverride Indexes`" just allows local control over how directory indexes appear but doesn't allow the local config file to change the value of the `Indexes` option—the Web server administrator needs to set "`AllowOverride Options`" to allow this to happen.

Multiple override options may be specified with a single `AllowOverride` directive—simply enter a space-separated list of the desired overrides (example on the next slide).

Using AllowOverride

```
AccessFileName .htaccess

<FilesMatch "^\.ht">
        Order allow,deny
</FilesMatch>

<Directory "/var/apache/docs/locals-only">
    AllowOverride Limit AuthConfig
</Directory>
```


`AccessFileName` defines the name of the override file recognized by the Web server. Again, the default is usually `.htaccess`, but you might want to choose a different standard for your Web servers to try and prevent external attackers from stealing these files out of your doc tree.

Whatever name you end up choosing, it's important to prevent these files from being displayed by your Web server. Here we're using the a `<FilesMatch...></FilesMatch>` container to deny access to any file on our Web server named "`.ht*`". Another example of this useful feature would be to block access to backup files created by common editors (like `*~` files created by Emacs, or `.bak` or `.sav` files):

```
<FilesMatch "*(\~|\.bak|.sav|.orig)$">
    Order allow,deny
</FilesMatch>
```

Now we can set `AllowOverride` for our "locals only" document tree that we set up in the previous example. No further configuration would be required in `httpd.conf`—the contents of the `<Directory...></Directory>` container in our previous example would simply go into the `.htaccess` file in the `/var/apache/docs/locals-only` directory.

SSL Configuration

Earlier we mentioned that it may be useful to combine SSL encryption with HTTP Basic Auth to protect passwords which are being used to access private websites. More commonly, though, SSL is used to protect sensitive data that is flowing between Web browser and Web server— credit card numbers, personal information, etc. This section looks at how to set up SSL support in Apache and also talks about how to get a server certificate for real Internet e-commerce.

Keys and Certificates

- SSL uses public/private key encryption
- Every SSL-ready server needs a key pair
- *Certificate*: server's public key signed by Certificate Authority
- For e-commerce, cert must be signed by a "recognized" CA

SSL is based on public/private key encryption. Each server that wants to support SSL communications needs a public/private key pair. Browsers expect that the server's public key has been signed by a recognized certificate authority (CA)—this signed public key is referred to as a *server certificate*.

For purely internal use, it's possible to set up your own CA or use dummy certificates that have been signed by a fake CA. This is particularly useful for test environments where you don't want to go to the trouble and expense of acquiring a commercial-grade cert. When a Web browser accesses a site that has a dummy certificate, a little pop-up appears warning the user that the certificate is invalid and asking if they want to continue talking to the site. Most users click "yes" and accept the cert without thinking about it.

The reality, though, is that for public e-commerce type activity you really need to get a site certificate from one of the few CAs that are recognized by the popular Web browsers currently in use. This process is described more fully on the next slide…

Getting a Certificate

- Generate key (protect this file!):

```
openssl genrsa [-rand list:of:files] \
    -out server.key 4096
```

- Generate Certificate Signing Request:

```
openssl req -new -key server.key -out server.csr
```

- Transmit CSR (+ money) to CA to receive cert file (`.crt`)

The first step in getting a certificate is to generate a key pair for your Web server using the `openssl` command from the OpenSSL distribution. "`openssl genrsa`" generates a standard RSA-type key which is used for standard Web browser SSL connections. The `-out` option specifies the file where the key is placed and `4096` is the key length. On systems which do not have a built-in `/dev/random` device, the `openssl` command needs some large amount of apparently random bits to generate the key—the `-rand` option can be used to specify a colon-separated list of files which can be used to generate pseudo-random values. The usual tactic is to use log files, the system kernel, etc. but not that this approach doesn't really produce truly random values (log files generally use only alphanumeric characters, binaries contain lots of nulls, etc.).

The resulting `server.key` file contains the server's private key, so it's critical that you protect this file. Anybody who steals that key file could impersonate your server and/or act as a man-in-the-middle and compromise SSL sessions between your server and remote browsers. The `openssl` command gives you the option of encrypting this file with a password, but if you do that, then your Web server will not be able to start automatically without this password being provided in some fashion (you could code it into the Web server boot script, but this doesn't seem much better than not encrypting the file at all). Since the Web server starts with root privileges (it needs to bind to port 80 after all), the `server.key` file should be mode 400 owned by root.

Once you've generated the server key, you need to use the `openssl` command again to generate the certificate signing request (CSR) that you will be transmitting to the certificate authority. A list of "recognized" CAs can be found at:

http://httpd.apache.org/docs/2.2/ssl/ssl_faq.html#realcert

There are also cheaper options (like *GoDaddy.com*) that sell what are referred to as *chained certificates.* Basically, these services have their own signing key from the top-level certificate authorities and resell certificates that

they've signed with their signing key. Chained certificates "work"—they don't impact the quality of SSL security from your website and users won't receive annoying pop-up warnings about certificates—and they're much cheaper. You will need to enable the `SSLCertificateChainFile` option in your Apache configuration file, however—the instructions you get from the certificate provider will explain how this is done.

Along with the CSR file, you'll also be sending along a quantity of cash money and some proof of the legitimacy of your business. This magic certificate signing rain dance can actually take up to several weeks depending on the CA you choose, so factor that into your timelines. Eventually, you'll get back a signed certificate (`.crt`) file which is what your Web server requires (along with the `server.key` file). Note that your certificate needs to be renewed every year or two (more money).

Creating a Self-Signed Cert

Same procedure as requesting a commercial certificate:

- Generate key
- Generate CSR

Just sign the CSR with your own key:

```
openssl x509 -req -days 730 -in server.csr \
    -signkey server.key -out server.crt
```


The first two steps of creating a self-signed cert—creating the key and CSR—are identical to the steps we saw on the previous slide. The only difference is, instead of transmitting your CSR and some money to some external CA, you just use the key you generated to sign the certificate yourself.

If you're going to need multiple dummy certificates for some internal application, you might consider actually creating your own internal CA. That way all certs can be signed from a consistent root. Furthermore, you could load this internal root certificate into the trusted certificate store on your user desktops, thus eliminating the browser pop-ups about using "untrusted" certificates.

This page has information on installing self-signed certs into the "trusted root certificates" store on Windows machines:

> *http://blogs.technet.com/sbs/archive/2008/05/08/installing-a-self-signed-certificate-as-a-trusted-root-ca-in-windows-vista.aspx*

The next page shows a complete command-line session where I created a self-signed certificate for *www.deer-run.com* ...

```
$ openssl genrsa -out server.key 4096
Generating RSA private key, 4096 bit long modulus
..........+++
.....................................................+++
$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Oregon
Locality Name (e.g., city) []:Eugene
Organization Name (e.g., company) [WWW Pty Ltd]:Deer Run Associates
Organizational Unit Name (e.g., section) []:
Common Name (e.g., YOUR name) []:www.deer-run.com
E-mail Address []:webmaster@deer-run.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
$ openssl x509 -req -days 730 -in server.csr \
                        -signkey server.key -out server.crt
Signature ok
subject=/C=US/ST=Oregon/L=Eugene/O=Deer Run Associates/CN=www.deer-
run.com/emailAddress=webmaster@deer-run.com
Getting Private key
$ ls -l
total 12
-rw-r--r-- 1 hal hal 1314 2009-08-26 00:12 server.crt
-rw-r--r-- 1 hal hal 1058 2009-08-26 00:11 server.csr
-rw-r--r-- 1 hal hal 1675 2009-08-26 00:09 server.key
```

Configuring SSL

```
Listen 443
            ...
<VirtualHost _default_:443>
    ServerName www.sysiphus.com:443
    DocumentRoot "/var/apache/docs-private"
    ...
    SSLEngine on
    SSLCertificateFile /var/apache/ssl/server.crt
    SSLCertificateKeyFile /var/apache/ssl/server.key
    SSLCipherSuite ...
    ...
</VirtualHost>
```


Once you've got your `server.key` file and corresponding `server.crt` file, you need to enable SSL support in `httpd.conf` and tell the Web server where these files are installed (again, keep them *out* of the doc trees). The usual configuration (at least for sites with only one Web server) is to have an `httpd` which is listening on port 80 for unencrypted requests and on port 443, which is the standard port for SSL communications (after all, there's no point in shipping around image files via SSL because the encryption adds extra overhead and burns extra CPU cycles on the Web server). There's no problem with having multiple `Listen` directives in `httpd.conf`—the Web server will bind to all of the requested addresses/ports.

In this configuration, SSL support is generally enabled with a `VirtualHost` definition (`VirtualHost` is also used for supporting multiple different websites via a single Apache instance). Here we're defining a virtual host which is listening on port 443 on the default list of addresses that the server is also using for its port 80 bindings. The document root for this virtual server is a different set of documents than those used for unencrypted traffic (it is permissible to use the same document root, but this seems fraught with peril). Note that the security configuration we set up elsewhere (IP-based access controls, password protection, etc.) still applies to the `VirtualHost` directive, so you don't need to repeat all of that again inside of the `VirtualHost` declaration, but you will need to allow access to the new `docs-private` directory since our stance is still "default deny".

We do need to enable SSL (`SSLEngine On`) and tell the server where to find the `server.key` and `server.crt` files. Again, theses files—particularly the `server.key` file—should be only readable by root. The files are read by the master Apache process, which runs with root privileges, so restricting access to these files should not be a problem.

There are many, many other SSL options that need to be included, but you can just copy these directly from the sample files provided with the Apache distribution.

Starting Apache w/ SSL

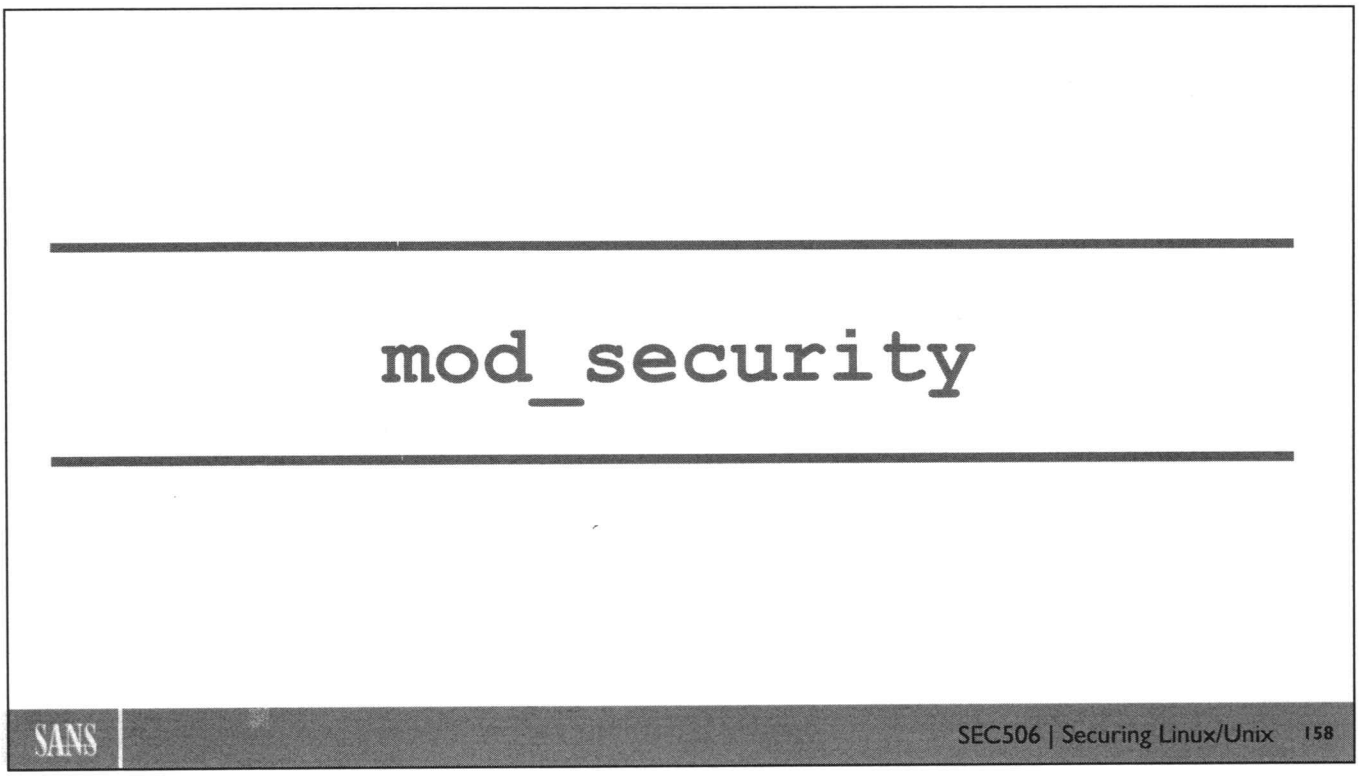
- Server needs to be started with `-DSSL`:

```
/var/apache/bin/httpd -DSSL \
     -f /var/apache/conf/httpd.conf
```

- "`apachectl startssl`" accomplishes the same thing
- Don't forget to update your boot scripts!

Once you've got `httpd.conf` all set up, you must still start the Apache server with the `-DSSL` command line argument as shown on the slide. You can also use the `apachectl` program—"`apachectl startssl`"—which accomplishes the same thing. Don't forget to update your boot scripts so the Web server will be started correctly when the system reboots.

Again, it's sort of a pain that starting up SSL requires a special command line flag. Why can't the settings in `httpd.conf` be enough? If you don't want to have to bother with the whole "`-DSSL`" nonsense, go into your httpd.conf file and simply remove all of the `<IfDefine SSL>...</IfDefine>` tags. Once these tags are gone, the configuration settings between the tags (i.e., all of the normal SSL startup code) will be executed even if "`-DSSL`" is not set on the command line.



`mod_security` is an Open Source Web Application Firewall (WAF) module that can be hooked into Apache. Given that PCI DSS and other security standards are starting to require WAFs, `mod_security` can be a nice way to meet these requirements inexpensively.

We don't have time to go into great detail about the `mod_security` rule language, but I wanted to at least cover the basics of getting `mod_security` installed so that you can start testing its capabilities on your own.

What Is mod_security?

Open Source Web Application Firewall
- Pattern match "signatures" of bad traffic
- Can deny access or just log
- Extensible functionality with Lua scripting

Deployment options
- Hook directly into Apache instance(s)
- Use in reverse proxy to protect other servers

At its simplest, `mod_security` is a pattern matching engine that allows you to match "signatures" of various kinds of malicious web traffic. You can also extend these rulesets to do more complicated checks using the Lua scripting language.

`mod_security` is deeply embedded in Apache via the Apache API and can filter not only the incoming HTTP request headers, but can also do deep content inspection (including uploaded file data), and even scan *outgoing* content from the web server to make sure you're not accidentally emitting information you shouldn't like PII and PCI. When you first start using `mod_security`, it's recommended that you run it in "detect only" mode so that it merely logs possible security events—false positives are always a risk. Once you've convinced yourself that `mod_security` isn't going to interfere with your normal web traffic, you can switch over to enforcing mode where `mod_security` blocks access when it detects potentially malicious content.

There are two standard deployment options for `mod_security`:

1. Hook it directly into your web servers that are serving your content. This works fine as long as you have a relatively small number of servers and they're all running some recent version of Apache.
2. Set up an Apache instance with `mod_security` as a reverse proxy between incoming connections from the Internet and your web server "farm". This allows one system to protect a large number of web servers, including non-Apache web servers (i.e., IIS).

If you're looking for a commercial solution that utilizes `mod_security`, Trustwave sells COTS appliances based on `mod_security` and hosts the `mod_security` development effort.

What Are the "Core Rules"?

- "Core Rules" block many common web attacks
- Good starting point, good learning tool
- Bundled with `mod_security`
- Start with `DetectionOnly` mode first!

`mod_security` comes with a bundled set of rules which are called the "Core Rules". This is a fairly extensive set of `mod_security` rules that have been developed over time to detect common sorts of malicious traffic and known exploits. They're also a good overview of the kinds of functionality available in `mod_security`, and serve as an example of how you can write your own rules.

The Core Rules have been developing over many years now, and are quite well tested and tend to have a low false-positive rate. Still, remember that when you're first getting started with `mod_security` and the Core Rules to only run in "detect only" mode until you're sure that `mod_security` isn't going to interfere with the normal functioning of your web app.

The Bad News

Not a standard Apache module

May need to build from source

Dependencies:

- XML libs (common on Linux)
- Lua (optional, shared libs required)
- Perl Compatible Reg Ex library (`libpcre`)
- Apache apxs utility (`*-devel` package)

The only rub here is that `mod_security` isn't included with the core Apache distro—generally, you have to download the source code (*http://www.modsecurity.org/download/index.html*) and compile it yourself.

If you want to include support for the Lua scripting language in your rules, then you also need to download the Lua source (*http://www.lua.org/download.html*) and build it. Note that to use Lua with Apache and `mod_security`, you need to build the embedded Lua interpreter as a shared object, which is not supported by the default Lua build system. Please see the following URL for more information on creating shared objects for Lua:

http://lua-users.org/lists/lua-l/2006-10/msg00091.html

If you need `mod_security` to be able to parse "text/xml" content being handled by your web server then you will also need the libxml2 library. This library is commonly installed on Linux systems, but you may have to build it yourself on other platforms. Source code is available from *http://xmlsoft.org/downloads.html*

When it's time to build `mod_security`, you'll need a copy of the Perl Compatible Regular Expression library, `libpcre`. This is commonly available as a package in most Linux distros, but make sure you have both the library itself and the `*-devel` package that contains files that will be used by the `mod_security` build. Similarly, you'll need to install the `*-devel` package associated with your Apache software so that the `mod_security` build will be able to find the `apxs` program required to build Apache modules.

Getting It Working

- Install Core Rules files wherever
- Modify settings in Core Rules config file
- Apache config directives:

```
# Optional functionality
#LoadFile /usr/lib/libxml2.so
#LoadFile /usr/local/lib/liblua.so
LoadModule unique_id_module modules/mod_unique_id.so
LoadModule security2_module modules/mod_security2.so
Include core_rules/*.conf
```


Once you've built and installed the actual `mod_security` module (along with all of its various dependencies), the next step is to install the Core Rules as your initial ruleset. You'll find the Core Rules configuration files in the "`rules`" subdirectory of the `mod_security` source distribution—the `*.conf` files in this directory contain the basic Core Ruleset, and the `*.conf` files in the "`rules/additional_rules`" directory are alternate versions of some of the basic rules and implement a more restrictive ruleset.

Copy the Core Rules `*.conf` files into some directory—I generally install them under my Apache configuration directory in a directory of their own called `core_rules` (`/etc/httpd/core_rules` on most standard Apache installations). You then need to edit the `modsecurity_crs_10_config.conf` file that contains basic settings for `mod_security`. In particular, pay attention to these config settings:

- `SecRuleEngine` —the default setting is "`On`", but you want to set this to "`DetectOnly`" so that `mod_security` is not in enforcing mode initially
- `SecUploadDir`, `SecDataDir`, `SecTmpDir` —these are set to `/tmp` initially, but I recommend setting these to a directory that only your Apache user has access to. I will typically create `/var/cache/mod_security/{upload,data,tmp}` and make them mode 700 and owned by your web server user

Be sure to look at the other settings in `modsecurity_crs_10_config.conf` and make sure they're appropriate for your environment and web applications.

The slide shows the configuration directives to add to your `httpd.conf` file (or other configuration directories) in order to enable `mod_security`. You only need to load `libxml2` and `liblua` if you're planning on using that functionality. `mod_security` requires `mod_unique_id`, so make sure to load that module if you haven't already. Finally, we load the `mod_security` module itself and then suck in all of the Core Rules `*.conf` files with an `Include` directive.

Testing

Restart Apache

Enter "<script>" into a web form:

```
[25/Aug/2009:14:13:08 --0700]
[www.deer-run.com/sid#8e318][rid#90ca8][/][2]
Warning. Pattern match
 "(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b..." at
 ARGS:email.
[file "/…core_rules/…40_generic_attacks.conf"]
 [line "102"] [id "950004"] [msg "Cross-site
 Scripting (XSS) Attack"] [data "<script"]
 [severity "CRITICAL"] [tag "WEB_ATTACK/XSS"]
```


Once you've made the appropriate configuration settings, restart Apache. You should immediately see Apache create two new log files in your normal logging directory: `modsec_audit.log` and `modsec_debug.log`. The `modsec_audit.log` file contains full details of the offending requests, and the `modsec_debug.log` file contains differing levels of detail depending on your debug level setting in `modsecurity_crs_10_config.conf`.

Now the question is whether `mod_security` is actually doing something. Surf on over to some web form that's hosted by your web server and enter "`<script>`" in one of the fields. When you submit the form, you should get output like this in your `modsec_debug.log` file (I've artificially introduced some line breaks in the output below for readability; in the normal log file this would be one big long line):

```
[25/Aug/2009:14:13:08 --0700]
[www.deer-run.com/sid#8e39198][rid#90cab28][/login/][2]
Warning. Pattern match
"(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)|application\b\
W*?\bx(?:java|vb))script|c(?:opyparentfolder|reatetextrange)|get(?:special|
parent)folder|iframe\b.{0,100}?\bsrc)\b|on(?:(?:mo(?:use(?:o(?:ver|ut)|down
|move|up)|ve)|key(?:press|d ..." at ARGS:email.
[file "/etc/httpd/mod_security/modsecurity_crs_40_generic_attacks.conf"]
[line "102"] [id "950004"] [msg "Cross-site Scripting (XSS) Attack"] [data
"<script"] [severity "CRITICAL"] [tag "WEB_ATTACK/XSS"]
```

As you can see, you get information about the server name and page URL, followed by info about the rule that matched, which `*.conf` file it's in, etc.

Further Reading

- Blog has lots of useful articles
- FAQ and Reference Manual available
- *jwall.org* has policy creation and audit viewer tools
- "Securing WebGoat" definitely pushing the envelope ...

If `mod_security` is interesting to you and you want to learn how to write your own rules, a good place to start is with the Core Rules themselves. The `mod_security` blog (*http://blog.spiderlabs.com/modsecurity/*) has many useful articles about various aspects of rule creation, modification, maintenance, etc. There's also the `mod_security` documentation page (*http://www.modsecurity.org/documentation/index.html*) which has links to the `mod_security` FAQ and Reference Manual. This page also has links to various `mod_security` documents written by the community.

Over at jwall.org are some other mod_security related tools. There's a graphical tool for creating mod_security policies and another tool for interpreting the audit logging from mod_security, plus a few other smaller tools.

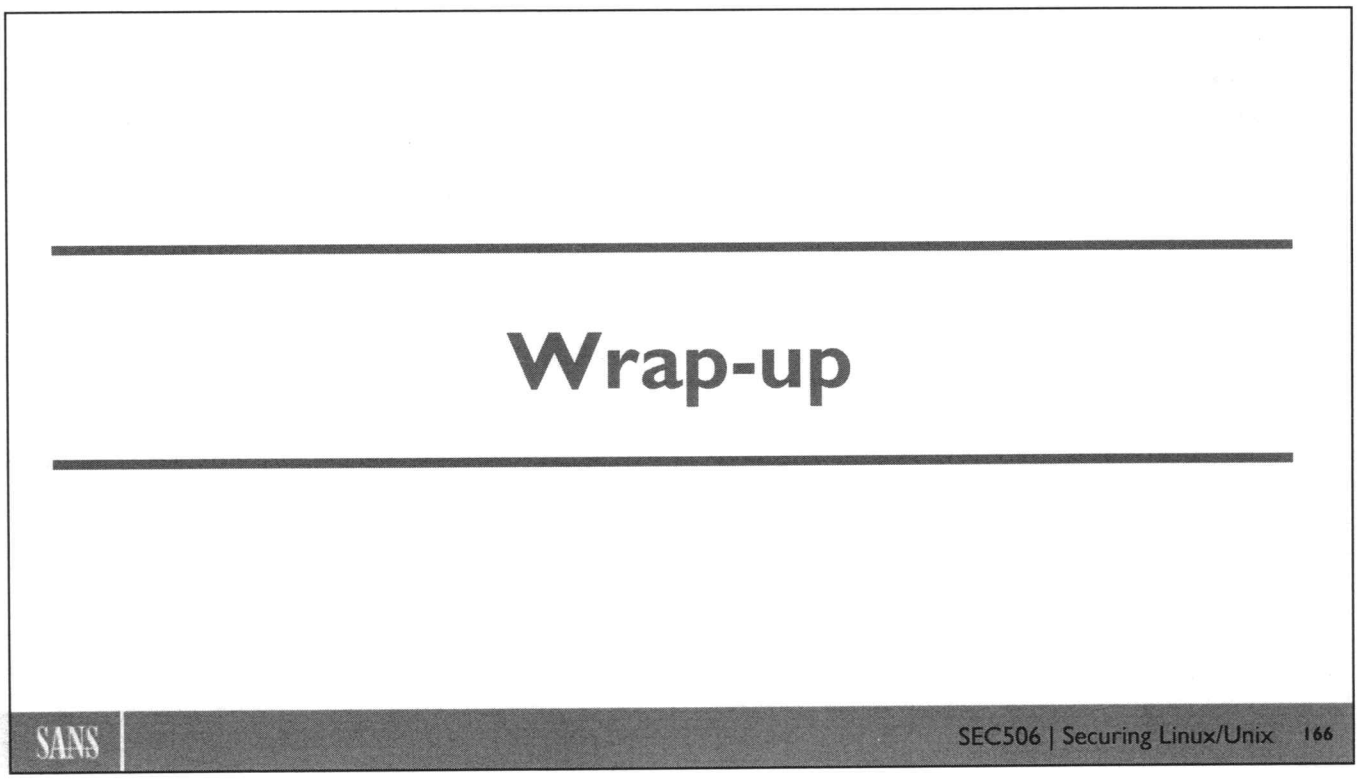
Google sponsored a project during one of their "Summer of Code" events that uses `mod_security` to mitigate the security vulnerabilities in the OWASP WebGoat application. Considering that WebGoat was an application that was deliberately *designed* to be insecure (it's a tool for learning about commonly perpetrated web vulnerabilities), actually making it reasonably secure with `mod_security` is something of an achievement. There are lots of crazy `mod_security` rules in this project. More info at:

http://www.owasp.org/index.php/Category:OWASP_Securing_WebGoat_using_ModSecurity_Project

That's All for Now!

- The material presented here is enough to "get you going"
- Really only scratches the surface, though
- Lots of good docs at *www.apache.org*
- Keep up-to-date on new releases!

While this has been a quick introduction to the basic settings for Apache security, there's a lot of additional information available on general Web server configuration issues. Most of this documentation is linked off of `www.apache.org`, so take some time to peruse this site. It's also important to check this site regularly to make sure you're up-to-date on the latest Apache releases because most new versions are driven by security-related bug fixes.



This page intentionally left blank.

That's It!

- Final Q&A?
- Please Fill Out Your Surveys!

Hal Pomeranz
hal@deer-run.com
http://www.deer-run.com/~hal/
@hal_pomeranz on Twitter

Lab Exercise

- Fun with Apache
- Practice your security configuration chops

Exercises are in HTML files in two places: on the course USB media, and in the home directory of the SANS user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. This is Day 5, Exercise 3, so navigate to *.../Exercises/Day_5/index.html* and open this HTML file
4. Click on the hyperlink which is the title of Exercise 3
5. Follow the instructions in the exercise

Appendix A

Sendmail

Some of the material in this section was excerpted from a larger tutorial on administering DNS and Sendmail. If you are interested in more detail on DNS and Sendmail administration (as opposed to security) issues, you can download the PDF for full course book for this tutorial from:

```
http://www.deer-run.com/~hal/dns-sendmail/
```

The *DNS and BIND* and *Sendmail* books published by O'Reilly and Associates also make useful references.

Agenda

- Common Security Issues
- Configuring and Running Sendmail
- Running Unprivileged

Common Security Issues is a review of security problems that have historically plagued Sendmail (some of which are appropriate to other Mail Transfer Agents (MTAs) as well).

Configuring and Running Sendmail presents specific configuration examples on how to configure Sendmail to operate securely in the firewall environment we introduced in the *DNS and Bind* portion of this course.

Running Unprivileged discusses where and how to run Sendmail as a non-root user to reduce the impact of future vulnerabilities.

Common Security Issues

Common Security Issues is a review of security problems that have historically plagued Sendmail (some of which are appropriate to other Mail Transfer Agents (MTAs) as well).

Historical Perspective

- Forging e-mail
- Back doors
- Dangerous aliases
- Addressing attacks
- Privilege Escalation/root Compromise

At a high-level, these are the typical problems which have caused heartburn for Sendmail administrators everywhere. Each of these problems is discussed in more detail in upcoming slides. Note that many of these problems are not specific to Sendmail—other MTAs suffer from similar vulnerabilities.

Forging E-mail

```
% telnet localhost 25
[...]
220 buck.deer-run.com ESMTP Sendmail 8.13.8/8.13.8; ...
mail from: santa@northpole.com
250 santa@northpole.com... Sender ok
rcpt to: hal@deer-run.com
250 hal@deer-run.com... Recipient ok
data
354 Enter mail, end with "." on a line by itself
From: Santa@Northpole.com
To: hal@deer-run.com
Subject: Ho Ho Ho

You've been a naughty boy this year!
.
250 KAA06575 Message accepted for delivery
```


The SMTP protocol, unfortunately, makes it very easy to forge e-mail. Simply connect to port 25 on any mail server and compose a bogus message as shown above. Note that the source of the connection will be reported in the mail headers of the resulting message and this information will also go into the mail server's log files, so tracking the source of such forgeries is possible.

The message above is merely humorous. However, there have been several successful attacks where outsiders have forged e-mail from System Administrators in order to dupe users into sending passwords (sometimes credit card information) to an external address. Be sure to explain to your users that you would never send them any e-mail requesting passwords or other private information since as the administrator you can change such information at will.

Note that when you connect to a running Sendmail daemon, it reports its version number in the default SMTP greeting. Like BIND, there are attacks which target a specific version of Sendmail, so it is generally a bad idea to advertise this information. As we will see later, Sendmail provides a mechanism for changing this default greeting.

Back Doors

- Sendmail v5 allowed admins to obtain root shell via 25/tcp
- One of the propagation schemes used by the Morris Worm
- The first CERT Advisory (CA-88.01) warns about this

Early versions of Sendmail allowed remote administrators to connect to a running Sendmail daemon and enter "debug" (sometimes "wizard") mode and get a root shell on the remote machine (the Internet was a much friendlier place in those days). Since essentially zero authentication was required, this was a huge hole.

As a matter of historical interest, the very first CERT Advisory (from 1988!) warns of this vulnerability, though in fact the hole was known for years before the advisory. This hole was one of the primary channels which the Morris Worm used to move around the Internet.

Dangerous Aliases

```
# decode: "|/usr/bin/uudecode"
```

Sendmail executes aliases as root

Various aliases can be dangerous

- Overwrite arbitrary files
- Create a set-uid shell
- Give a remote root shell

Sendmail allows administrators to create aliases which pipe mail to programs. Some of these aliases are dangerous and/or the programs themselves are buggy and easily compromised. Since Sendmail executes these aliases with root privilege, very bad things can happen.

The classic example is the `decode` alias (still provided on many operating systems, though usually commented out, thank goodness) which pipes any message sent to this alias to the `uudecode` program. uuencoded files specify the pathname of the file to be unpacked, and an external attacker can use this mechanism to overwrite any system file (like the `passwd` file).

Note that normal users can create `.forward` files which pipe e-mail to programs. Sendmail executes these programs at the privilege level of the given user, but this is still enough privilege for an external attacker to, say, create an `authorized_keys` file in the user's home directory and then log into the machine as that user (possibly later using another attack to get a root shell).

Sendmail ships with a program called `smrsh` (SendMail Restricted SHell) which can be used to safely execute programs in a `chroot()`ed environment. However, we will not cover how to use `smrsh` in this course.

Addressing Attacks

```
'foo@bar.com; rm -rf /*'
```

Requires address be interpreted by a Unix command shell:

- Sometimes triggered through aliases
- Check inputs from Web pages!

Another classic Sendmail attack is to send in an e-mail address which contains embedded shell commands. When this e-mail address is handled by a program—for example, used as the reply address on a later invocation of Sendmail—the embedded commands are triggered. The above example is a classic denial-of-service attack, but other attacks can be used to create set-UID shells or modify system files.

The explosion of Web-based forms which collect e-mail addresses has produced another avenue for injecting these dangerous e-mail addresses into people's systems. Be sure to carefully check all Web inputs before using that information as part of any command!

Another form of addressing attack is to create a message with a very, very long e-mail address in order to trigger a buffer overrun…

root Compromise

Sendmail daemon runs as root

- *Popular "break root" vector because so many systems ran Sendmail by default*

Pre v8.12, `sendmail` binary set-UID to root

- *Popular "privilege escalation" path for attackers*

Again, the fundamental problem here is that Sendmail is running as root. If a remote buffer overflow attack is successful, then an attacker can possibly get a root shell on the remote machine, or at least overwrite a file like the password file to accomplish the same thing. At the end of this section we'll discuss how to run Sendmail as a non-root user in some cases to help mitigate these issues.

Aside from running as root, `sendmail` was historically installed as a set-UID root binary. There is a class of Sendmail attacks which manipulate the Sendmail binary (via command line options and sometimes by manipulating the environment) to cause Sendmail to overwrite system files, create set-UID shells, et al. Generally, these attacks allow local, non-privileged users to increase their privilege level. These attacks are often exploited by attackers who have gained unprivileged access to your machine and who want a root shell.

The reason Sendmail has historically been installed set-UID to root is so that the `sendmail` process had the privileges to write outgoing e-mail from users and processes on the system into Sendmail's mail queues. Starting with v8.12.x, Sendmail now has a separate mail queue for outgoing message submission which is group writable by the special `smmsp` group. So as of v8.12, the `sendmail` binary is normally installed set-GID to `smmsp`, rather than set-UID to root. This is a huge win from a security perspective.

Not only is it critical to stay up to date on Sendmail releases, it is important that you completely eradicate the old binaries when you upgrade. Old set-UID binaries lying around can still be a vector for an attacker.

Other Options

Postfix
http://www.postfix.org/

Qmail
http://www.qmail.org/
http://cr.yp.to/qmail.html

Exim
http://www.exim.org/

Many attacks against Sendmail have succeeded because it runs as root and is a set-UID root binary. In response to these architectural issues, several other alternate MTAs have sprung up in past years. Of course, Sendmail v8.12.x is now following suit and trying to reduce its use of root privilege.

The major architectural difference between these alternate MTAs and earlier Sendmail versions is that the above mail systems generally break up their functions into multiple separate programs, most of which do not run with superuser privilege. In fact, the only portion of a mail system which generally needs to run as root (and be set-UID to root) is the function which appends e-mail to user mailboxes—this is typically a very simple piece of code and can be exhaustively analyzed for security problems. The process which listens for e-mail, which manages the queue, and which relays mail to other systems does not need root privilege and can often be `chroot()` ed besides.

There's also, of course, an ongoing religious battle between adherents to the various MTA cults. "My MTA is faster than your MTA!", etc., etc.

Configuring and Running Sendmail

Configuring and Running Sendmail presents specific configuration examples on how to configure Sendmail to operate securely in the firewall environment we introduced in the *DNS and Bind* portion of this course.

The 99.9% Case

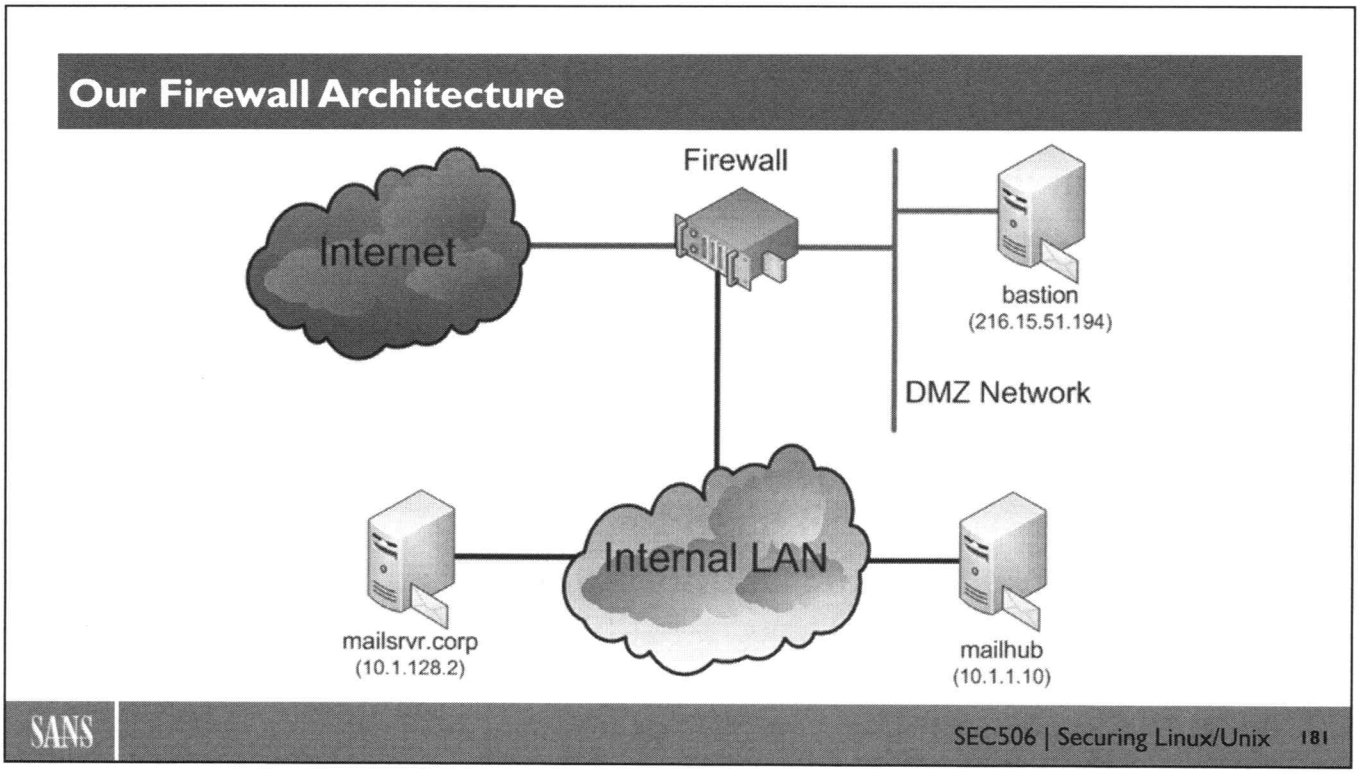
➢*Most hosts don't need a mailer daemon*

- Sendmail daemon does two things:
  - Listens for *incoming* e-mail
  - Process the queue of unsent messages
- If you're not a mail server, then no incoming e-mail
- Queued messages can be handled via safer config

Recall our discussion from earlier in the curriculum. The primary purpose of the Sendmail daemon (or whichever mail daemon you prefer) is to listen on 25/tcp for incoming e-mail from outside the machine. Programs on the system that are sending e-mail out of the machine normally invoke the `sendmail` binary directly from the disk, rather than communicating with the running Sendmail daemon on the system. So, if the machine isn't acting as a mail server, then you should shut off the daemon so that it's not reachable by other hosts on the network—this will protect you the next time a remote exploit is found in Sendmail. This is probably the biggest Sendmail security win because it allows you to focus your attention on the really important machines in your e-mail architecture—the servers—and not worry as much about what's happening on the "client" machines.

See our discussion from earlier in the curriculum on how to manage the outgoing mail queues from your e-mail "client" machines, or read the following articles:

```
http://www.deer-run.com/~hal/sysadmin/sendmail.html
http://www.deer-run.com/~hal/sysadmin/sendmail2.html
```



Now let's talk about how to create working Sendmail configuration for mail servers. Here's the firewall architecture picture we introduced in the *DNS and BIND* section. Note that we have added another machine to the picture which is the mail server for one of our theoretical internal-only subdomains.

Mail Routing - Goals

Inbound mail:

- Goes to `bastion` first
- Is immediately forwarded to `mailhub`
- No local delivery on `bastion`

Outbound mail:

- Relayed from internal hosts to `bastion`
- `bastion` delivers to remote domain

The external mail gateway will be used as a mail proxy for both inbound and outbound mail. Mail coming in from outside your organization has to stop on the `bastion` host and then be forwarded into your internal network. No local delivery will ever take place on the `bastion` host itself, thwarting many popular Sendmail-based attacks.

Since we are assuming that no internal hosts can directly reach hosts on the Internet, all outbound mail has to find its way to your `bastion` host, which in turn will deliver the mail to its final destination.

Given Sendmail's history of security problems, you may feel the risk of running Sendmail on your external mail server is too great. You may wish to investigate running an alternate MTA (QMail, Postfix, Exim), at least on your external relay. On the other hand, the Sendmail sources have probably received much more scrutiny than the QMail or Postfix sources, and perhaps may actually be more secure than the newer alternatives. Also, later in this module, I'll show you how to run Sendmail as a non-root user on your external mail relay servers, which further helps reduce the impact of future Sendmail vulnerabilities.

Creating Sendmail Configs

As of v8, Sendmail config files built from macro definitions

Use "m4" to create actual config files:

```
m4 myfile.mc > myfile.cf
cp myfile.cf /etc/mail/sendmail.cf
```

Recommend collecting macro files together in one place

The rest of this section deals primarily with creating appropriate Sendmail configuration files for machines in our example network architecture. In the bad old days, creating Sendmail configuration files required learning a cryptic configuration language. Sendmail v8 made matters easier by defining a set of macros which allow administrators to easily define their Sendmail configurations with a set of (reasonably) English-like declarations.

So, first, the administrator creates a file containing the macro settings that correspond to their desired Sendmail configuration ("`myfile.mc`" in the example on the slide)—we'll see plenty of examples of these files in the upcoming slides. These macros are processed and expanded by the "`m4`" program (a very old program from the early days of Unix) into the standard Sendmail configuration language. Note that `m4` simply spits its output to the standard output, so this output must be redirected into a file. The resulting file should then be copied into place as `/etc/mail/sendmail.cf`, which is Sendmail's default configuration file location.

The standard Sendmail macro definitions can be found in the "`cf`" subdirectory of the Sendmail distribution. For any reasonably large site, you will probably end up creating lots of custom Sendmail configuration files. I find that it's usually a good idea to collect up all of these different configuration files into a single location, and keeping them in a subdirectory of the "`cf`" directory seems to work well. Typically, I'll name this directory for the site's local domain name ("`cf/sysiphus.com`" for example) and name the individual files in this directory for the machine they apply to ("`bastion.mc`", "`bastion.cf`", etc.).

Config for bastion

```
include(`../m4/cf.m4')
OSTYPE(`linux')

define(`confSMTP_LOGIN_MSG', `$j mailer ready at $b')
define(`confPRIVACY_FLAGS',
       ``noexpn,novrfy,noverb,noetrn'')
define(`LOCAL_SHELL_PATH', `/dev/null')
define(`confMAX_HOP',`50')

MASQUERADE_AS(sysiphus.com)
FEATURE(`mailertable', `hash -o /etc/mail/mailertable')
define(`MAIL_HUB', `mailhub.sysiphus.com')
MAILER(smtp)
```


When you `telnet` to port 25 on a machine running Sendmail, you see a greeting string which includes the version of Sendmail. Advertising which version of Sendmail you are running is a bad idea since it can help attackers target your system with specific exploits. The new `SMTP_LOGIN_MSG` we set looks like

```
220 bastion.sysiphus.com ESMTP mailer ready at <date>
```

which doesn't even advertise that we're running Sendmail.

The `PRIVACY_FLAGS` setting turns off the `EXPN` and `VRFY` commands and prevents outsiders from gaining information about users in your domain. We're also disabling the VERB command which can allow attackers to gather information about your internal mail architecture. ETRN allows remote users to request that your mail server run its queue looking for any queued mail for the remote site. While this is useful in an ISP environment, it could also be used as a potential denial-of-service so we disable it here.

Changing `LOCAL_SHELL_PATH` to `/dev/null` prevents the local Sendmail daemon from executing any aliases which pipe mail to a program. This should never be a factor anyway since all locally delivered mail should be forwarded to `mailhub` by the MAIL_HUB directive (see below).

`MAX_HOP` is the number of times a given piece of e-mail can be forwarded before Sendmail kicks the e-mail back to sender-- this is used to prevent mailing loops, but the default value (18) may be too low if your mail has to jump through a lot of hoops to get through your firewall.
`MAIL_HUB` forces all mail for the local machine to be forwarded to `mailhub` (nobody from the outside world should be sending mail to `bastion` but cron and other system processes may generate e-mail). The `mailertable` feature (which we'll cover in more detail in the next couple of slides) is a way of creating a simple database of special-case e-mail routing rules for the e-mail being sent to users in our local e-mail domains. All other mail that isn't explicitly handled by a rule in our `mailertable` database–i.e., outgoing mail destined for non-local domains–will be delivered normally via the SMTP mailer.
`MASQUERADE_AS` causes all mail originating from this host to appear to be from `user@sysiphus.com` rather than `user@bastion.sysiphus.com`.

/etc/mail/mailertable

```
corp.sysiphus.com       smtp:mailhub.sysiphus.com
eng.sysiphus.com        smtp:mailhub.sysiphus.com
sysiphus.com            smtp:mailhub.sysiphus.com
```


The `mailertable` database is just a simple mapping of e-mail domain names to machines that should receive e-mail for the given domain. First, we create a text file called `/etc/mail/mailertable`. The left-hand column gives a domain name and the right-hand column specifies a mailer to use and a destination host.

In our case, all we're trying to do is fire all e-mail for our local e-mail domains into the `mailhub` machine for further routing, so our rules are all essentially identical. The nice thing about this configuration, however, is that the only internal IP address the `bastion` host needs to know now is the address of the `mailhub` machine. You could simply put this address in the `/etc/hosts` file on the `bastion` host and then this machine wouldn't even have to look at your internal DNS infrastructure.

Although I'm not going to show you the Sendmail configuration for the `mailhub` machine, it's likely that the `mailhub` also uses a `mailertable` to handle routing e-mail within our hypothetical corporate environment. Of course, the rules in the `mailertable` on the `mailhub` machine would look much different from the rules we're using here on the bastion host.

How To Create DB Files

Sendmail sources include `makemap`

```
% cd sendmail-8.x.x/makemap
% sh Build                    (lots of output)
% /bin/su
Password:
# cp obj.<arch>/makemap /etc/mail
# cd /etc/mail
# vi mailertable             (per previous slide)
# ./makemap hash mailertable < mailertable
#
```


Our `mailertable` text file must be converted into a Unix binary database file so that Sendmail can use the information. Your operating system probably already includes a program called `makemap`, which performs this conversion. If your OS doesn't come with `makemap`, the Sendmail distribution includes the `makemap` source code. As you can see, building the software is trivial. I usually install `makemap` in `/etc/mail` if it's not already included with my OS.

Using the `makemap` program is also straightforward. First, you specify the kind of database file to create—here we're using a standard "Berkeley DB" `hash`-style database, which is typical for Linux and other Open Source OSes (older proprietary Unix systems like Solaris and HP-UX may use "`dbm`" instead of "`hash`" here). The second argument is the "base name" of the database file: for a `hash` style database, `makemap` will actually create a file called "`mailertable.db`" (`dbm` style databases will produce two files, `mailertable.dir` and `mailertable.pag`). `makemap` wants to get the contents of our `mailertable` text file on its standard input, so we're just using simple shell redirection to make that happen.

Note that every time you update the text file you must re-run the `makemap` program to rebuild the database file. You may want to create a `Makefile` or shell script which does this task automatically.

relay-domains File

- Examples of "good" relaying:
  - Inbound e-mail to local users
  - Outgoing e-mail from local users
- Everything else is "bad"
- Sendmail uses `relay-domains` config:

```
sysiphus.com
eng.sysiphus.com
corp.sysiphus.com
216.15.51
10.1
```


While it's the job of the bastion host to relay e-mail into and out of our hypothetical company, we have to be careful not to allow too much. E-mail from the Internet to our local e-mail domains is fine, as is outgoing e-mail from our users to other sites on the Internet. However, any other relaying would mean that we were accepting e-mail from outside sites and relaying it to non-local users. This is referred to as being a *promiscuous* or *open relay*. When your mail servers are an open relay, spammers will use them to forward their junk e-mail to other sites. Not only is this a waste of your bandwidth and resources, but it also means that your mail servers end up getting blacklisted by much of the Internet with the result that most sites won't even accept legitimate e-mail from you.

Sendmail controls relaying via the `/etc/mail/relay-domains` file. First, you should list all local e-mail domains you want to pass incoming e-mail for–in our case, we want to make sure that `bastion` passes along e-mail for our internal domains. The `relay-domains` file should also contain the IP address ranges of machines that our mail server will relay *outgoing* e-mail for (you don't want to rely on e-mail domains for this because the spammers could just forge the sender e-mail address to relay e-mail through your mail servers). In this case, we're adding our internal address space to the `relay-domains` file so that our internal users can send mail back out to the rest of the Internet.

Config for mailsrvr.corp

```
include(`../m4/cf.m4')
OSTYPE(`linux')

define(`confMAX_HOP',`50')
define(`confSMTP_LOGIN_MSG', `$j mailer ready at $b')

FEATURE(promiscuous_relay)
FEATURE(accept_unqualified_senders)
FEATURE(use_cw_file)

define(`SMART_HOST', `mailhub.sysiphus.com')
MASQUERADE_AS(corp.sysiphus.com)
MAILER(smtp)
```


Here's an m4 macro file for `mailsrvr.corp.sysiphus.com`, which will demonstrate some of the different Sendmail features that might be appropriate for *internal* mail servers in your environment.

`promiscuous_relay` and `accept_unqualified_senders` turn off some default Sendmail anti-spam checks. While these features should *never* be disabled on your external mail relay because they help prevent the flow of spam, many sites find that disabling these features makes life easier on their internal mail servers. For example, `promiscuous_relay` tells your mail server that it's OK to relay e-mail from one part of your company to another—without having to mess with the `relay-domains` file or other local configuration. Similarly, `accept_unqualified_senders` means that the mail server will accept e-mail from broken mail clients that simply show the sender's address as a username ("`bob`") without a domain qualifier. The `MASQUERADE_AS` directive will make sure that these e-mails get the correct domain name appended to them before they're sent on to their final destination.

The `use_cw_file` directive tells Sendmail to look in the file `/etc/mail/local-host-names` for a list of domains (one per line) that are to be considered local (prior to Sendmail v8.10, the file was called `/etc/mail/sendmail.cw` which is how the `FEATURE` gets its name). In this case, the local-host-names file will only contain "corp.sysiphus.com", but this feature is particularly useful if you accept mail for a lot of domains and is generally easier than editing the `sendmail.cf` file to update this information. `SMART_HOST` says to forward all e-mail that isn't local to this machine to `mailhub` for further processing. This means that `mailsrvr.corp` will send all mail to `mailhub` that isn't specifically destined for `corp.sysiphus.com` (whether that's outgoing e-mail or e-mail for another subdomain like `eng.sysiphus.com`). This means the `corp.sysiphus.com` mail admins can have an easier time because they don't have to configure and maintain all of the different e-mail routes for the company, but the downside is that `mailhub` now becomes a single point of failure. Consider deploying multiple machines in parallel for redundancy.

`SMART_HOST` is essentially the logical inverse of the `MAIL_HUB` directive we used on `bastion`.

Running Unprivileged

Running Unprivileged discusses where and how to run Sendmail as a non-root user to reduce the impact of future vulnerabilities.

Do I Have to Run as `root`?

Why Sendmail runs as **`root`**:

- Bind to port 25/tcp
- Perform "local delivery" for users
- Read and write config files/queues

But ...

- Give up privs after binding to 25/tcp
- No local delivery on `bastion`
- Change perms on files and queue

Sendmail runs as root for several reasons:

- Only processes running as root are allowed to bind to network ports below 1024, at least on Unix machines (on Windows systems, for example, any user can bind to any port). Since the MTA daemon needs to bind to port 25/tcp, it needs to be root.
- When Sendmail is performing local delivery, it needs to be root so that it can append e-mail to different user's mailboxes.
- Various Sendmail configuration files and directories, like the aliases database and the mail queue directory, are only accessible to the root user. Sendmail must run as root to access this information.

However, there are workarounds for all of the above issues—at least on the `bastion` host:

- If you look at other servers on Unix that need to bind to low port numbers—BIND, for example—the daemon will run as root only for as long as it needs to bind to the low port number. Thereafter it will give up root privileges and run as an unprivileged user.
- We are not doing any kind of local delivery on the `bastion`, so appending mail to user mailboxes is not an issue for us.
- While you don't want many of Sendmail's critical files and directories to be readable by normal users on the system, that doesn't mean those files and directories have to be owned by the root user. We can create a special user just for Sendmail, and make everything owned by that user instead.

It's actually pretty straightforward to get all this setup and working on your external relay servers—you could even use this configuration on your internal relay servers as long as you're not doing local delivery there. There's just a little bit of system reconfiguration and then a new configuration directive in your Sendmail macro definition file. Just follow along with me on the next two slides.

System Configuration

Create new **sendmail** user/group

Shut down Sendmail daemon

Reset permissions on critical dirs:

```
chown -R sendmail:sendmail /var/spool/mqueue
chgrp -R sendmail /etc/mail
chmod -R g+r /etc/mail
chmod g+s /etc/mail
```


First, create a new user ID and group ID that your Sendmail will run as. *Do not* use the special "`smmsp`" user and group already in use by Sendmail. I will often create a user and group called "`sendmail`" with UID and GID 24 (`smmsp` is usually UID and GID 25):

```
groupadd -g 24 sendmail
useradd -u 24 -g 24 -M -d /var/spool/mqueue \
    -s /dev/null sendmail
```

Now before you start messing around with ownerships and permissions of various files and directories it's a good idea to shut down the running Sendmail daemon so that it doesn't get confused while you're in the middle of making your changes. There are really two critical directories for the MTA process: the queue directory and the `/etc/mail` configuration area. The queue directory should be owned by whatever user and group you created to run the MTA daemon as ("`sendmail`" and "`sendmail`" in our example). The permissions on this directory don't need to be changed—it's already mode 700 so only root can access the files in there, we're just changing which user is allowed to peek.

You still want the `/etc/mail` directory and the files in it to be owned by root, because you only want legitimate system administrators to be messing around with these files. But we need to make sure that the files in this directory are at least readable by the Sendmail daemon when it's running unprivileged. So, what we're going to do is change the group ownership in the directory and its contents to our "`sendmail`" group, and then use "`chmod -R g+r /etc/mail`" to give group read permissions to everything in the directory. Actually, the files in there are probably already group readable but it never hurts to be sure.

The last "`chmod g+s /etc/mail`" command adds the so-called "set-GID" bit onto the `/etc/mail` directory. On Unix systems, if set-GID is set on a directory, then any new file created in that directory will automatically inherit the group ownership of the directory (as opposed to the group membership of the user that creates the file, which would be the default). In this way, we can make sure that any newly created files end up with the proper ownerships—like when you're rebuilding the `mailertable` database with the `makemap` program.

RUN_AS_USER

```
include(`../m4/cf.m4')
OSTYPE(`linux')
define(`confRUN_AS_USER', `sendmail:sendmail')
define(`confTRUSTED_USERS', `sendmail')
define(`confSMTP_LOGIN_MSG', `$j mailer ready at $b')
define(`confPRIVACY_FLAGS',
       ``noexpn,novrfy,noverb,noetrn'')
define(`LOCAL_SHELL_PATH', `/dev/null')
define(`confMAX_HOP',`50')
MASQUERADE_AS(sysiphus.com)
FEATURE(`mailertable', `hash -o /etc/mail/mailertable')
define(`MAIL_HUB', `mailhub.sysiphus.com')
MAILER(smtp)
```


Sendmail allows you to specify an alternate user and group to run as with the `RUN_AS_USER` configuration option. As you can see, we're specifying the "`sendmail`" user and group we created per the instructions on the previous slide. The `TRUSTED_USERS` setting adds our `sendmail` user to the list of users who have special administrative privileges—this setting eliminates some spurious error messages from your Sendmail logs when the aliases database and other files are owned by the `sendmail` user.

If you look at the running Sendmail process after you set this option, you may be surprised to see that it's still running as root—this is expected behavior. The master process always runs as root, but when a new SMTP connection comes in the master process must `fork()` a copy of itself to handle the incoming connection. The "child" process will run as the unprivileged user and group you specify with the `RUN_AS_USER` option. So, while the master Sendmail process itself runs as root, outsiders will only ever be able to communicate with unprivileged child processes.

Anyway, once you've run your new macro definitions through `m4`, copy the resulting config file out to your external server(s) and install it as `/etc/mail/sendmail.cf`. Then restart Sendmail. You can test the configuration by using `telnet` to connect to port 25 ("`telnet localhost 25`"). If you get a process listing in another window, you should see a `sendmail` process running as the `sendmail` user. Close the `telnet` session to make this process go away.

Summary

Security thoughts:

- Disable Sendmail daemon on most systems
- Run unprivileged on pure relay servers
- Change greeting string to hide version
- Disable other features where possible

E-mail routing:

- `MAIL_HUB` for preventing local delivery
- `SMART_HOST` for outbound mail
- `mailertable` for everything else

We've covered a whole bunch of different Sendmail configuration options, so it's probably a good idea to summarize things before leaving this topic. The most important step for Sendmail security is to disable the Sendmail daemon on all of the machines you own that are not acting as mail servers (which should be 99% of the machines in your environment). Machines which are acting purely as e-mail relay servers (no local delivery) can be configured to run Sendmail as an unprivileged user to help mitigate future vulnerabilities. Actually, once we have Sendmail running as an unprivileged user, it should be reasonable to run the daemon `chroot()` ed as well, but that configuration is beyond the scope of this course.

On your external mail servers particularly, it's a good idea to change Sendmail's default greeting string to hide what version of Sendmail you're using. You may also be able to disable other features—like setting `LOCAL_SHELL_PATH` to disable aliases that pipe e-mail to other programs or turning off SMTP commands like `EXPN`, `VRFY`, `VERB`, and `ETRN`.

The trickiest part about modern firewall architectures is getting your e-mail to route properly in and out of your organization. The `mailertable` feature can be used to create special-case routing rules for most circumstances, and the `SMART_HOST` feature can be used by internal servers to forward e-mail to a central machine for proper routing. `MAIL_HUB` is useful for making sure that local delivery never happens on your external mail relay servers.

Virus Protection

- Filtering out viruses and phishing scams also critical
- MIMEDefang can be added into Sendmail via "Milter"
- Plug ClamAV or other anti-virus into MIMEDefang
- Can also add SpamAssassin, Razor, etc. for spam control

While not strictly related to the security of Sendmail, protecting your organization from e-mail-borne viruses and phishing scams is obviously important. While configuring this functionality is outside of the scope of this course, a few pointers are in order.

Most of the solutions in this area for Sendmail make use of the so-called "Milter" (a corruption of "mail filter") interface that appeared in Sendmail v8.11. Basically, the Milter spec defines an API for plugging external modules into Sendmail that allow you to filter messages as they're passing through your server.

MIMEDefang is a Milter that really just acts as a framework for other modules. For example, MIMEDefang has hooks to do virus scanning with the Open Source ClamAV scanner on incoming e-mail (MIMEDefang also has hooks for many of the popular commercial anti-virus scanners as well). Recent versions of ClamAV will even recognize certain phishing scams in addition to standard e-mail worms and viruses.

In addition, MIMEDefang has hooks for all of the popular spam control solutions like SpamAssassin, Razor, and others. It's a pretty effective and comprehensive solution and relatively easy to install.

The best place to start is the MIMEDefang "how-to" at *http://www.mickeyhill.com/mimedefang-howto/*. Specific instructions for installing ClamAV can be found at *www.clamav.net*. The SpamAssassin project page is *spamassassin.apache.org*.

Appendix B

PortSentry

More details on Portsentry configuration, if the discussion in the SELinux section made you curious.

Simple Model

- PortSentry daemon binds to unused network ports
- Any traffic hitting these ports is logged
- PortSentry daemon can also:
  - "Null route" the source of the packet
  - Put block rules into local firewall config
  - Run an arbitrary command

Portsentry is really a very simple-minded tool (simple solutions are sometimes the best). Basically, you run the Portsentry daemon, which binds to a list of ports that you specify. These should be ports where you don't normally expect to see any traffic. If Portsentry sees traffic on these ports, then it's probably somebody running a port scanner against you or probing around for some backdoor infection on your machine.

Portsentry will log the unexpected access and can optionally take action. Built-in actions include inserting a route into the system's local routing table so that any packets your host might try to send back to the remote machine just go to a non-existent router and never leave the network. Portsentry can also interact with an IP filter or IP Tables firewall or with TCP Wrappers to block all future IP traffic from the remote system. Since most port scans are precursors to an attack, rapidly responding to the scan and blocking future traffic can be a big win.

Portsentry also allows the administrator to define an arbitrary command that will be run—either in addition to one of the built-in blocking actions or instead of those actions. This could, for example, allow you to update the firewall rules on your network-layer firewall rather than just the local system.

Of course, all of this firewall talk leads to an important point. If your system is already protected by a network-layer or host-based firewall, then you may be already blocking all of the traffic that Portsentry would alarm on. On the other hand, Portsentry can provide a second level of verification that your firewalls are actually working properly, or monitor network traffic on the relatively wide-open networks *behind* your firewalls, or be used on systems that need to be "exposed" to hostile network environments.

Problem with Active Response

- Might block legitimate traffic due to:
  - Scans from spoofed source IPs
  - Accidental probes from the confused
- Tune ports and "trigger sensitivity" via config file
- Docs discuss why this is less likely than you might think

SANS | SEC506 | Securing Linux/Unix 197

If you configure Portsentry to automatically block traffic after a probe from some remote system, there is a danger that you end up triggering a false alarm and blocking traffic from somebody you actually want to talk to. For example, somebody might try to reach your web server on port 80, but you're only doing HTTPS traffic on port 443. If you've got Portsentry watching port 80, you may end up blocking a lot of people unintentionally. Or an attacker who figures out that you're using Portsentry might scan you with the spoofed IP address of a machine that you really want to talk to (like your log server) in order to mess with you.

Obviously, being sensible about the ports you monitor with Portsentry is critical to reducing your false alarm rate. Portsentry also allows you to wait until you reach a certain number of probes from a system (say 2 or 3) before triggering a response.

The `README.stealth` file in the Portsentry source distribution also makes some good arguments why scans from forged IP addresses are not really likely. Basically, the arguments boil down to:

- Attackers usually want to get the responses to their port scan, so using a forged address doesn't help them.
- Using forged addresses for decoy scans only increases the amount of time required to do the scan as well as increasing the total amount of network traffic, making the scan easier to spot.
- More and more sites are implementing egress filtering which prevents spoofed packets from even leaving their networks.

Your mileage, as always, may vary. Perhaps you will want to run Portsentry in "log only" mode until you're convinced that your false alarm rate is acceptable.

Another PortSentry Issue

- Monitored ports appear as "live" to **`netstat`**, **`nmap`**, ...
- Can create false-positives when security scanners are run
- Large number of open ports may make host a target

When PortSentry is running on your machine, it binds to the list of ports you specify just like any normal network service. This means the ports in your list show up as "active" network ports in the output of netstat or lsof. And when the machine is scanned from the outside with a port scanning tool like Nmap, the ports show up as "live" in the scan output.

This causes all kinds of consternation when you run host-based or network-based security audits. "Why are all those ports open on your machine?" "Well, they're not really open, per se..." You get the idea.

If the bad guys are scanning you with Nmap and they find a machine with dozens of open ports, which machine do you think they're going to try to attack first? This may become a nuisance.

Linux "Stealth Mode"

- PortSentry on Linux can listen on ports via raw sockets
- Ports now invisible to **`netstat`**, **`nmap`**, ...
- Can detect half-open and other "stealth" type scans
- Not portable to other systems

In an effort to deal with this problem, PortSentry can take advantage of a peculiarity of the Linux raw socket interface. The Linux kernel allows PortSentry to listen on specific network ports via raw sockets, rather than "binding" to ports in a traditional sense. This means that the ports being monitored don't show up as "active" in the output of `netstat`, `lsof`, and `Nmap`.

What's more interesting is that when Portsentry is listening via raw sockets, it can see various "stealth" scanning modes like half-open connection scanning, SYN-FIN scans, etc. that would not normally be visible to normally bound network sockets. Basically, if Portsentry is bound to the socket via the traditional mechanism, it will only see full three-way handshake type connections on its TCP ports.

Unfortunately, this raw socket approach is not portable to other operating system platforms where raw sockets behave differently than under Linux. A portable approach would be to have Portsentry put the interface into promiscuous mode and use something like `libpcap` to capture traffic. Unfortunately, this would also hurt performance on the system.

Installation Notes

Set in **`portsentry_config.h`**

- Config file location
- TCP Wrappers config location
- Syslog facility/priority
- Number of hosts to "remember"

Also have to set paths in **`Makefile`**

Build with "**`make`** *<target>*"

PortSentry doesn't come with a `configure` script, so certain parameters must be set manually.

The `portsentry_config.h` file contains several configuration settings that get hard-coded into the binary. `CONFIG_FILE` is the path to Portsentry's configuration file, and `WRAPPER_HOSTS_DENY` is the path to the TCP Wrappers `hosts.deny` file on the system. Additionally, you can define what Syslog facility to use and what level to log at–I prefer that PortSentry logs to `LOG_AUTH`, rather than the default which is `LOG_DAEMON`. You can also set the number of hosts that PortSentry should keep state for (the default is 50 but feel free to set this higher).

Unfortunately, there are also paths in the `Makefile` that need to get set as well, like the installation directory.

Once you're all done, you just run "`make <target>`", where *<target>* is something like "`linux`" or "`solaris`". "`make`" with no arguments displays the list of available targets.

Basic Configuration

Pathnames set in **`portsentry.conf`**–

- **`IGNORE_FILE`**
  List of IP addrs to ignore–don't ignore too much!
- **`BLOCKED_FILE`**
  IP addresses that have been blocked during this session
- **`HISTORY_FILE`**
  Permanent running history of blocked hosts

Set **`RESOLVE_HOST="0"`** to turn off reverse DNS queries

Once you've got the binary built, a "`make install`" will copy the binary and some sample config files into the installation directory you specified in the `Makefile`. The default configuration file for Portsentry is the `portsentry.conf` file in this directory. The sample config file that comes with Portsentry is full of comments to help you along. First, let's take a look at some of the basic settings in the file.

`IGNORE_FILE` (usually `$INST/portsentry.ignore`) is a list of IP addresses or networks in CIDR notation that Portsentry should simply ignore. The usual tactic is to list 127.0.0.1, 0.0.0.0, and the IP addresses of the local interfaces on the machine in this file. You might be tempted to list your entire "internal" network range, but this means you won't detect attacks from infected machines on your own networks.

`BLOCKED_FILE` is a temporary file that lists the hosts that have been blocked by the current instance of the Portsentry daemon. This file is started from scratch every time you restart the daemon. On the other hand, `HISTORY_FILE` is the chronological history of the traffic that the Portsentry daemon has detected and responded to.

If you don't trust DNS and/or your machine is seeing a lot of traffic, you may want to set `RESOLVE_HOST="0"` to disable reverse DNS lookups.

Configuring Ports

Set both **`TCP_PORTS`** and **`UDP_PORTS`**

General tips:

- Avoid ports in use by legitimate services
- Monitor low ports to detect scans quicker
- Monitor ports of well-known back-doors
- Monitor ports of disabled services
- Port lists provided with PortSentry definitely need tuning!

The next step is to tune your initial lists of ports that PortSentry should be monitoring. There's one configuration variable for TCP ports and another for UDP ports. PortSentry has an internal limit of 64 ports that it is willing to monitor–actually 64 ports each for TCP and UDP. You can increase this value by changing the `MAXSOCKS` line in `portsentry.h`, but 64 ports are probably more than enough for most uses.

The most important factor to consider when setting up your port lists is to avoid ports where you have network services already running or expect to see legitimate network traffic. This will help you avoid false alarms.

Other than that, make sure you monitor several low-numbered ports so you can quickly detect trivial port scans that start at port 1 and work their way up. You also want to have ports scattered throughout the entire 65K port range so that you have good coverage. Adding ports for popular back-door Trojans (31337, 5554, etc.) is also a good idea. If you've specifically disabled a service because of security concerns (e.g., `telnetd`) then you might want to monitor for people probing for that service.

Note that the port lists in the default `portsentry.conf` file are not well tuned. They contain ports like DHCP (68 UDP), the RPC portmapper (111 TCP and UDP), NFS (2049 TCP and UDP), and X Windows (6000 TCP) that are surely going to trigger false alarms on many systems.

Response Options

Potential probe attempts always logged

Three different response options:

- **KILL_ROUTE** – adjust routing table or firewall rules
- **KILL_HOSTS_DENY** – block via TCP Wrappers
- **KILL_RUN_CMD** – any other arbitrary command line

Any or all of these may be undefined

Portsentry will always log probes on the ports it's monitoring via Syslog. This functionality cannot be shut off.

However, you can control what actions Portsentry will take when it sees a probe. The `KILL_ROUTE` option is usually set to a command that either adjusts the local system's routing table to black hole traffic back to the host sending the probes or which puts a block rule into the local host-based firewall config to block all traffic from the host. Generally, blocking with a host-based firewall is preferred to "null routing" the host because the firewall actually blocks incoming traffic from the host whereas the routing solution still allows the bad guy to send "packets of death" to your machine.

`KILL_HOSTS_DENY` is the TCP Wrappers syntax you want to go into `hosts.deny` to block the host. Of course, you need to be careful that there's no permit rule in `hosts.allow` that might allow the traffic through (remember `hosts.allow` is checked before `hosts.deny`). Again, using a host-based firewall is more effective.

Finally, `KILL_RUN_CMD` is an arbitrary command line that will be run either before or after the blocking action. The variable `KILL_RUN_CMD_FIRST` controls the order—"1" to fire `KILL_RUN_CMD` before the block actions, and "0" to fire it after. You might have `KILL_RUN_CMD` send a piece of e-mail to your pager about the block, or send an SNMP trap to a network monitor, etc. You could also have `KILL_RUN_CMD` retaliate against the remote machine in some way, but this is not recommended.

All three of the action variables can use the syntax `$TARGET$`, `$PORT$`, and `$MODE$` in the command line. These are replaced with the IP address, port, and protocol (`tcp` or `udp`) of the original probe, respectively.

Tuning Your Response

Use **BLOCK_TCP** and **BLOCK_UDP** to control response:
"0" – do nothing except log
"1" – trigger all defined responses
"2" – only run **KILL_RUN_CMD**

SCAN_TRIGGER is number of probes to provoke response

One way to prevent Portsentry from taking action is to simply not set the three "action variables" discussed on the previous slide. You can also use `BLOCK_TCP` and `BLOCK_UDP` to control the Portsentry behavior to different types of scans. The default setting is "0", which means to log probes only. A setting of "2" will cause logging to happen and only `KILL_RUN_CMD` to be triggered—useful if `KILL_RUN_CMD` modifies the configuration of some other machine but you don't want to make local updates. A setting of "1" triggers all defined actions.

Normally even a single probe packet will cause Portsentry to start firing. However, you can set `SCAN_TRIGGER` to the number of packets you're willing to receive before firing. A setting of "2" will help eliminate a lot of false alarms. Note that there's really no difference between a setting of "0" and a setting of "1"–both will cause Portsentry to fire on the first packet received.

Warning Banner

- Optional **PORT_BANNER** is message that is displayed once PortSentry triggers
- Does not work in "stealth mode"
- Discourages attackers or enrages them?

You may optionally set `PORT_BANNER` with a message that will be displayed when an IP address that has been "blocked" by Portsentry attempts to connect to one of the ports that Portsentry is monitoring. Of course, if Portsentry has configured your local firewall to drop all traffic from the host, there's no point in doing this. On the other hand, if you're not doing active response with Portsentry ("log only" mode), this can put the person probing the system on notice that you've detected their activity. Note that `PORT_BANNER` is only displayed if Portsentry is bound to the ports in "normal" rather than "stealth" mode.

Be careful that your `PORT_BANNER` message is not overly antagonistic. Something simple like "`All unauthorized access is monitored and reported`" is sufficient. "`D00d! You sur3 are a l8m3 hax0r!`" is probably only going to provoke a more extreme response.

Starting PortSentry

- Start two daemons (one TCP, one UDP):

```
portsentry -tcp
portsentry -udp
```

- Or if you're using "stealth mode":

```
portsentry -stcp
portsentry -sudp
```

- Don't forget boot script for automatic startup

Once you've got your configuration file all squared away, it's time to actually start the PortSentry daemon. Actually, you start two daemons–one to monitor the TCP ports you've defined and one to monitor the UDP ports. The command line invocations for the daemons are shown on the slide.

If you decide PortSentry is useful, then you'll probably want to create a boot script so that it's started automatically when the system is rebooted.

Final Thoughts

- Like AIDE, PortSentry requires some tuning work up front
- Functions best on machines that are not "well-advertised"

http://sourceforge.net/projects/sentrytools

As with any sort of IDS, Portsentry will require some tuning for your specific environment. Mostly this involves tweaking the port lists so you don't get buried by false alarms.

The README files in the Portsentry documentation point out that running Portsentry on a very "public" machine like your external web server, name server, etc. is probably not a useful thing because these machines are getting scanned all the time. Your best bet is to have it running on an out of the way machine so you detect indiscriminate scan behavior, and on critical servers where you can very tightly define "expected" network activity.

Portsentry can be obtained from the URL shown on the slide.