

POSIX Permissions and Stateful Firewalls

Created by: Peter A. H. Peterson and Dr. Peter Reiher, UCLA {pahp, reiher}@cs.ucla.edu, with Dr. Tanya Crenshaw, UP {crenshaw@up.edu}

Due: Thursday, October 13th, 2011 11:59PM via CourseWeb

Contents

1. [Overview](#)
2. [Required Reading](#)
 1. [POSIX Permissions](#)
 2. [Criticisms](#)
 3. [sudo and Alternatives](#)
 4. [Software Tools](#)
 1. [adduser, chfn, passwd](#)
 2. [addgroup](#)
 3. [usermod](#)
 4. [chown, chgrp](#)
 5. [chmod](#)
 5. [Firewalls](#)
 6. [Policy Design](#)
 7. [Firewall and Network Testing Tools](#)
 1. [iptables](#)
 2. [nmap](#)
 3. [ifconfig](#)
 4. [telnet](#)
 5. [netcat](#)
3. [Introduction](#)
4. [Assignment Instructions](#)
 1. [Setup](#)
 1. [Saving your work](#)
 2. [Restoring your work](#)
 2. [Tasks](#)
 1. [Home Directory Security](#)
 2. [The Ballot Box](#)
 3. [The TPS Reports Directory](#)
 4. [sudo Problems](#)
 5. [Firewall Configuration](#)
 3. [Tips and Tricks](#)
 1. [Resolving ambiguities](#)
 2. [Using environment variables](#)
 3. [Testing your firewall](#)
 4. [What Can Go Wrong](#)
5. [Submission Instructions](#)

Overview

The purpose of this exercise is to introduce you to filesystem and network access control schemes and the "principle of least privilege" through the use of POSIX filesystem permissions and `iptables` firewalls.

After this exercise, you will:

1. understand the POSIX permissions structure including SUID and SGID bits
2. understand the essence of the `sudo` utility and how to configure and use it securely.
3. be able to apply that knowledge to configure permissions in multiple scenarios, such as:
 1. shared system directories
 2. user home directories and private directories
 3. privileged system directories
 4. unprivileged temporary directories
 5. editing important configuration files
 6. restarting system processes
 7. potential privilege escalation problems
4. understand the basics of stateful firewalls
5. be able to apply that knowledge to configure a basic firewall in Linux using `iptables`.

Required Reading

POSIX Permissions

The Portable Operating System Interface, or POSIX, is IEEE standards family **IEEE 1003.1** and represents an attempt to formalize the API for variants of the Unix operating system. POSIX specifies standards for the kernel APIs (including many internal functions like permissions, threads, networking, and interprocess communication), commands, required utilities and user-level APIs (including many utilities and scripting tools like `awk`, `echo`, `ed`, `sh`, `vi`, basic I/O, and more), and a conformance test that can be run on many platforms. POSIX is very widely disseminated and has been in use for almost 20 years, but is a closed standard (i.e. proprietary). As a result, it is expensive to be certified; this has resulted in many free operating systems such as FreeBSD and Linux being substantially compliant but not officially able to use the moniker "POSIX Compliant". An alternative to POSIX is the [Single UNIX Specification](#), developed by the Austin Group. Unfortunately,

it is also proprietary and expensive.

The expense has not been prohibitive for motivated commercial vendors however, and POSIX has been extremely influential in coalescing and standardizing those elements that are characteristically UNIX-like. Furthermore, POSIX is a standard that influences software design and operating system implementations in seemingly unlikely places -- for example, Windows NT and its derivatives (2000, XP, and beyond) can be "POSIX compliant" with the addition of software packages and the enabling of certain features. This broad influence -- existing before the public explosion of the Internet -- has helped to make software and systems similar in fundamental ways. This, in turn, simplifies many portability and interoperability problems.

One of the enduring legacies of Bell Labs UNIX and the POSIX standard are traditional Unix file system permissions, which are a simple system of permissions stored in the file system and kernel and evaluated to mediate access to system resources. Additionally, since devices are files in Unix, file system permissions also mediate control over many aspects of the operating system.

In most Unix and Unix-like operating systems, the permission system is extremely simple. Each file has `read`, `write`, and `execute` permissions set for each of three groups of users called "access classes". The three classes are `user` (owner), `group`, and `other`; each access class represents one or more users in the following way:

- Users each have unique user identifiers (userids or UIDs, listed in `/etc/passwd`) that only they control.
- Groups are arbitrary collections of users. Each group has its own group identifier (groupid or GID, defined in `/etc/group`). Often, every user on the system has a personal group used for sharing data with a limited set of other users.
 - Example: the group class `cdrom` usually contains a list of all users granted permission to access the system's CD-rom drive.
 - Example: the user `sonny` has a group `sonny` by default; the user `cher` can be added as a member of group `sonny` so that they can share resources between themselves but no one else.
- Finally, the membership of the `other` class for each file is dependent on the user and members of the `group` class -- **the other class does not represent "all users!"** Specifically, the `other` class represents "everyone that is *not* the user (owner) of the file and *or* a member of the file's group class." This fact allows us to express more meaningful permissions by strictly dividing all system users into three non-overlapping sets.
 - A more precise way of stating this is to say that `other` represents the the list of all users on the system, minus the union of the `user` and `group` classes.
 - Example: There are three users on a system: `sonny`, `cher`, and `donovan`. `sonny` owns a file whose group access class is set to the group `sonny`, which contains the user `cher`. The `other` class consists only of `donovan`, because he is not the owner nor in the group `sonny`.

Each access class is encoded by 3 bits. Each bit present grants an additional permission from the set `read`, `write`, and `execute`. When you list files in long form (with the option `-l`), you will see the permissions written out in the leftmost column as a bitstring with the letters `r`, `w`, and `x` in the order `user`, `group`, and `other`:

Here's what it looks like "in the wild":

```
$ ls -alh
-rw-r--r--  1 pahp console  18K May 16  2006 gpl.txt
drwx-----  5 pahp console 512 May 21  2006 john-1.7.2/
lrwxrwxrwx  1 pahp console  25 Sep 13 20:45 labs -> /proj/some/link/
-rwxr--r--  1 pahp console 171 Sep  9 09:18 loadimage.sh*
drwxrwxrwx  1 pahp console 512 Sep 13 20:45 worldwritable/
^ ^ ^ ^ ^ other permissions
| | | '----- group permissions
| '----- user permissions
'----- file type: d=directory, l=symbolic link, - means regular file
```

Here's an example bitstring split into components:

d	rw	rw	rw
type	user	group	other

In the above example, the owner of the file `gp1.txt` has read and write permission because the `owner` part of the bitstring is set to `rw-`. The owner (`pahp`) cannot execute this file, but that doesn't matter because it's just a text file. The group class `console` can read the file, as can the members of the `other` class -- everyone else.

The directory `john-1.7.2` is only accessible by the owner and nobody else. Notice the `d` in the first place; this means that the file is a directory.

The file `labs` is a "[symbolic link](#)" (sometimes called "soft link") to the directory `/proj/some/link/`. Symbolic links are a special kind of file that is a pointer to a disk location (a "pathname" which may or may not exist). Permissions on symbolic links do not work as their `lrwxrwxrwx` permissions might lead you believe.

In addition to the 9 bits required to store the `user`, `group`, and `other` permissions of a file, there are three kinds of "additional permissions" that are rarely used but are extremely important to understand. The additional permissions are known as the *setuid bit*, *setgid bit*, and *sticky bit*. Each special bit has a different function depending on where it is used:

When the the special bits are used on regular files:

- The `setuid` bit on an executable file means that the file will run as the `userid` of the file's *owner* as opposed to the `userid` of the user executing the file. This is done to allow users to perform tasks that temporarily require the user to be someone else, such as changing passwords or restarting a service. Any program with the `UID` bit set must be carefully written so as to block all misuse. Innumerable vulnerabilities have stemmed from `setuid root` programs with security holes that allowed users to execute other commands as `root`.
- The `setgid` bit on an executable file is like the `setuid` bit, except that the process gains the effective user of the file's *group*, not its owner or the user executing the file.
- The "sticky bit" was used in the olden days to tell the kernel to keep an executable's image in memory so that it would not have to be reloaded from disk. This was commonly done with programs such as editors that were used regularly but had a significant load time. Modern systems use the "sticky bit" for other uses.

When the special bits are used on directories, they have different meanings altogether. Search online to discover what those meanings are on Linux systems.

Every file has permissions of some kind set for the each of the three classes, using the three types of basic permissions: `read`, `write`, and `execute` (and the three

special bits). These permissions have a default value (based on the system umask) when the file is created. Permissions can be changed with command line utilities such as `chmod` and `chown`, which are discussed [later in this document](#).

These permissions are fairly self explanatory when applied to files, but when applied to directories their meanings are **not always intuitive**. In particular, permissions set on a directory **do not always apply to that directory's contents**, and the concepts of "read", "write", and "execute" as they pertain to directories is not obvious. (It helps to think of a directory as a file which contains links to other files, rather than as a nested series of containers -- experiment on DETER to determine how they really interact!)

Furthermore, while the special bits are commonly overlooked, they have important meanings with **incredible significance** in terms of security. You are encouraged to play with these permissions and read other materials in order to fully understand how they are interpreted.

Criticisms

Traditional Unix permissions date back to the early 1970s. While simple and inexpensive to implement and evaluate, they have long been criticized for being out of date and woefully inexpressive compared to the power of other access control systems. Other systems (such as Microsoft Windows and other ACL models for Unix) have more than three access groups, nested groups, explicit deny lists, the ability to specify write permissions for changing, creating, and deleting (with Unix permissions they are largely the same), the ability to consider arbitrary state in evaluating the ACLs (such as time of day, network address, etc.), the ability to make files invisible to unprivileged users, and many other features that traditional Unix permissions lack.

Another important and oft-criticized fact is that Unix permissions do not exactly reflect the current state of the permissions configuration on the system. For example, while Unix checks the file permissions at the moment a file is *opened*, it only checks those permissions *once* for each open system call. This means that the permissions are only tested when you *first open the file for reading, writing, or executing*, **not** each time you access or modify the file's contents via an open file descriptor. If a user opens a file, and immediately thereafter all permissions on that file for that user are removed, the user with the open file descriptor will *still* have access to the resource according to the permissions set *when the file was opened*, even if the *current* permissions deny reading, writing, or execution.

Furthermore, the traditional Unix permissions and authentication system assigns group membership and other credentials at login time and *never* checks them again -- this means that system password, login permission, group membership details in `/etc/group` and other details changed *after* you log in are **not** reflected in your processes until you log out and log back in or the process with those credentials is restarted. For example, this means that any change to a user group is fully applied only when all affected processes have been restarted. Another example is that if an attacker steals a password and logs in to a system before that account is disabled or the password is changed, the attacker will still have that level of authentication until he or she logs out.



This is an example of a *time-of-check vs. time-of-use* (TOCTOU) problem.

These design decisions were made to limit the space occupied by and CPU cycles spent evaluating system permissions. As long as the semantics of the system (including the above criticisms) are fully understood, it can be an effective, acceptably secure solution for many purposes. In fact, in some ways, its simplicity and even naivete make it more transparent and easily understandable when compared to more complex systems. For these and many other reasons, the traditional Unix file permissions are still in wide use, over 40 years after their initial deployment.

sudo and Alternatives

Some applications simply require stronger guarantees, finer granularity, or other features that the traditional permissions cannot express. For those applications, users have many alternatives. `sudo` is one simple extension to traditional Unix permissions that has become very popular.

sudo is a setuid root application with its own ACL (stored in `/etc/sudoers`) that specifies, with fine granularity, tasks that users and groups can perform as root. For example, `sudo` could be used to allow a user to act as root in order to kill processes with a specific name. The user would otherwise have no additional privileges. This is an example of a perfect use for setuid root programs -- granting strongly-constrained privileges to unprivileged users.

However, `sudo` is a double edged sword. On the one hand, it greatly enhances the expressiveness of Unix permissions without actually changing the permissions system. On the other hand, if improperly configured, it offers **easy access to root**. For more information on `sudo` and the `sudoers` ACL format, please see the manpages for `sudo` and `sudoers` (the configuration file), or other `sudo`-related material online (in particular regarding `sudo` exploits).

Beyond `sudo`, two broader and more revolutionary alternatives for Linux permissions include SELinux and grsecurity, while other options exist including LDAP, Novell eDirectory, and other permissions systems for other operating systems. (We won't be using any of these.)

Software Tools

adduser, chfn, passwd: add users to a system

`adduser` is the tool available for adding users to the system. To create a new user, execute:

```
$ adduser username
```

`adduser` will copy files from the `/etc/skel` directory to become the new homedir of the new user in the `/home/` directory. You can specify a different home directory or automatically add the new user to groups; those options can be found in the `adduser` manpage.

`adduser` does not set any `finger` information for the user (this is not strictly necessary anyway), but the tool `chfn` will do that:

```
$ sudo chfn jimbo
Changing finger information for jimbo.
Name []: ...
```

By default, the user is created with a "locked out" account with no password set. To set the password, use the command `passwd`:

```
$ sudo passwd jimbo
Changing password for user jimbo.
New Unix password:
Retype new Unix password:
passwd: all authentication tokens updated successfully.
```

`passwd` will complain if it doesn't like the password you enter, but will accept anything with enough prodding.

Once an account is created, you can log into it in a number of ways:

1. If logged in to the system where you created the account, you can execute `ssh newuser@localhost` to reconnect locally as the new user with the new password.
2. If local, you can also use the `su` command `su newuser` to change to that user. This does not always update all access credentials, however. This method can also be used to become root by entering `sudo su -`.
3. If logged in to `users.deterlab.net`, you can `ssh` to the node as the new user.

groupadd: add groups to a system

`groupadd` adds a new group to the system with a unique ID (by default). Example:

```
$ groupadd newgroup
```

See `man groupadd` for more information.

usermod: modify a user

`usermod` can be used to modify many details of a preexisting user account. For more information, see `man usermod`.

chown, chgrp: change ownership of a file

`chown` stands for **change ownership** and is (unsurprisingly) used to change the owner and group of a file.

The syntax is very simple. To change the owner of a file, execute:

```
$ chown newowner filename
```

To change the owner and group classes of a file, execute:

```
$ chown newowner:newgroup filename
```

`chgrp` stands for **change group** and works very similarly to `chown`. To change the group of a file, execute:

```
$ chgrp newgroup filename
```

Recursive and other options exist; see `man chown` or `man chgrp` for more information.

chmod: change the mode of a file

`chmod` stands for **change mode** and is used to change the permissions mode of a file. Earlier, we discussed how the POSIX ACL has three access classes. User (or owner), group, and other (or world). The permissions mode for each access class can be changed by the `chmod` command.

There are two ways to use `chmod`; one is an *absolute numeric mode* and the other is a *symbolic mode*.

chmod: absolute -- setting the permissions explicitly

In the absolute mode, `chmod` takes a file mode in 3 or 4 digits, each of which represents the absolute permission mode for one access class expressed in octal, where each number is a sum of the permission bits. Each permission has a unique value: `read` permission is 4, `write` permission is 2, and `execute` permission is 1. These values represent the position of the permission in a 3-bit value.

For example, the mode `777` means "full permissions" because all bits are set in each access class. Similarly, `000` means "no permissions" because all bits are unset in each access class.

The 3-digit mode `777` is the equivalent to the 4-digit mode `0777` where the leading 0 represents the "special" permission class of `setuid`, `setgid`, and sticky. Likewise, the modes `000` and `0000` are equivalent and represent the absence of permission. (The owner of the file and root still have the ability to change the file's mode by virtue of their ownership and superuser status.) Finally, if the 3-digit mode is used, `chmod` always assumes that the special access class is 0. Therefore, if you set an `suid-root` file to mode `777`, `chmod` assumes that you meant mode `0777`, which would take away the special permissions. This is consistent if you remember that octal modes represent *absolute permissions* and *the special group class is assumed to be 0 if a 3-digit mode is provided*.

The following is a table to help calculate permission modes:

chmod absolute values

special	user	group	other
---------	------	-------	-------

s	g	t	r	w	x	r	w	x	r	w	x
---	---	---	---	---	---	---	---	---	---	---	---

4	2	1	4	2	1	4	2	1	4	2	1
---	---	---	---	---	---	---	---	---	---	---	---

For example, full access is `0777`, which represents:

special user group other

s g t r w x r w x r w x

4 2 1 4 2 1 4 2 1 4 2 1

0 7 7 7

There are no special bits set, so the special octal digit is 0, while all three bits in each other class are set. Each set of bits totals 7 (4 + 2 + 1), so the mode is 0777.

Another example is mode 2755, which represents `setgid` (2), plus "full access" for user (4 + 2 + 1) and write and execute (4 + 1) access for both the group and other class.

special user group other

s g t r w x r w x r w x

4 2 1 4 2 1 4 2 1 4 2 1

2 7 5 5

Absolute modes are applied like this:

```
$ chmod 0755 somefile.sh
```

Absolute mode is great for setting things to be exactly what you want, or imposing a radically different order onto a file or directory, but it's not very good for adding the sticky bit to a file. For that kind of work (or if the octal modes just confuse you) the symbolic mode is well suited.

chmod: symbolic -- more user-friendly

The symbolic mode works pretty much the way you would expect. To add the `execute` bit to the `user` class, you would execute a command like this:

```
$ chmod u+x somefile.sh
```

Or to add `execute` to all three classes:

```
$ chmod ugo+x somefile.sh
```

To make a file `setuid`:

```
$ chmod u+s somefile.sh
```

To remove all permissions:

```
$ chmod ugo-rwxsgt somefile.sh
```

Some people are so put off by the absolute mode that they never learn it -- you'll find with experience that both methods of setting permissions can be expeditious depending on the situation. More information is available in the `chmod` manpage.

Regardless of how you set permissions, it is critical that you use the "principle of least privilege" and only grant the privileges that are necessary for proper operation.

Firewalls

Stateless Firewalls

In the late 1980s, the Internet was just beginning to grow beyond its early academic and governmental applications into the commercial and personal worlds. The [Great Internet Worm](#) in November of 1988 infected around 6,000 hosts (roughly 10% of the Internet) in the first major infection of its kind and helped to focus research and awareness on securing computers from unauthorized access. It was in this environment that the first firewalls were written about and developed at Digital Equipment Corporation (DEC) and Bell Labs (AT&T).

The first functional firewalls inspected individual packet headers without regard for established connections, other packets, or their contents. These kind of firewalls became known as "packet filters" because they literally filtered the packets one by one according to a set of criteria, not unlike a quality control inspector on an assembly line. For TCP and UDP, these criteria could be reduced essentially to the source and destination addresses and ports in the packet header. For example, a packet filter could reject or drop any packets destined for port 23 (telnet) on host 10.10.10.10 from any address other than 10.10.10.11. This kind of filter could rapidly and inexpensively inspect and classify packets without using much space (although they were not very "smart").

Unsurprisingly, simple packet filters are not adequate for many applications, such as the [File Transfer Protocol](#) (FTP), because these protocols open additional connections on random ports that can not be anticipated or recognized by the firewall since it does not understand or consider the state of any connection.

This kind of simple "packet filter" ultimately became known as a "stateless firewall".

Stateful Firewalls

"Stateful firewalls" arrived not long after "stateless firewalls". Stateful firewalls keep tables of network connections and states in memory in order to determine if a packet is part of a preexisting network connection, the start of a new and legitimate connection, or an unwanted or unrelated packet. This kind of firewall can recognize, for example, that a new connection on a random high port from a host with a preexisting FTP connection is a related connection and should be allowed. Another difference is that while a stateless firewall will allow all packets from acceptable hosts to an open port, a stateful firewall can be configured to allow packets to that port only if a legitimate TCP connection (or some other protocol) has already been established in some acceptable way. Understanding protocol state essentially gives stateful firewalls vastly more criteria in deciding whether to accept or reject a packet, which translates into finer granularity.

The cutting edge of firewall design today is what is called an "application-layer firewall", which is a firewall that performs "deep packet inspection". This means that the firewall is capable of looking not just at the header of the packets and the state of the connection, but at the payload of the packet in context of what the application processing the packets will do. For example, an application-layer firewall could be used to block Java applets from HTTP traffic by inspecting the packets and stripping Java code or dropping the packets entirely. In order to do this, it must understand what applet code looks like within the payload portion of any HTTP traffic stream. An application-layer firewall essentially has total control over the network stream, although this control comes at a significant expense in terms of CPU time and software complexity.

Most firewalls in use today lie somewhere between the stateful firewall and the application-layer firewall. These firewalls function essentially as a stateful firewall, but may understand enough of a few applications to perform some application-layer tasks. It is also common to couple a primarily stateful firewall (such as `netfilter/iptables`) with separate application layer firewalls for individual applications.

Firewall Policy Design

People imagine many different things when they hear the term "[firewall](#)" in the context of computer networking. Some envision an impenetrable wall of flame [*at least I did --ed.*]. A Hollywood screenwriter might envision Harrison Ford battling kidnappers. A mechanic might envision the wall between the engine and passenger compartment of a car. Yet mysteriously, every firewall is illustrated as a boring, red brick wall, typically with no fire in sight.

Actually, the brick wall isn't that strange -- the name "firewall" comes from the brick walls in buildings placed to stop the spread of a fire from one area to another. But no matter who you are or what you see in your minds eye, the conventional wisdom is that firewalls are used to "keep the bad stuff out," whether you're protecting your desktop PC at home, your office LAN, or the Pentagon. However, those of us in the field of computer security often see firewalls more as a means of keeping things *in* rather than keeping them *out*.

In one sense, these are two sides of the same coin -- but how you design something is (often unconsciously) directly related to how you view the problem, and this can lead to very different design choices when developing a firewall. The goal of "keeping things out" is by definition, exclusively concerned with keeping external attackers "outside" the system, with no regard for what is inside that is worth protecting, and without considering threats (intentional or unintentional) that are *already* inside, like malicious or foolish employees. This is only half the picture. In contrast, "keeping things in" by definition concerns itself with what is "inside" like sensitive data, privileged access, etc., and encourages the designer to consider *all* threats -- both internal and external -- against the protected resource.

Practically speaking, these two goals often result in different default policies. The goal of "keeping things out" often results in a policy that by default allows anything not considered to be a threat. This is called a **default allow** policy, and the classic example of this kind of firewall allows **all** outbound traffic, but only allows "untrusted" inbound traffic to special services, such as a web server (which is then responsible for its own security). This is better than nothing, but is hardly secure. If an attacker can trick someone inside into opening a [trojan horse](#), the malicious software can exploit the liberal egress policy by making connections to a malicious host on the Internet, which can be used to send messages to the now-compromised system. Incidentally, this is how the firewalls on most home routers are designed.

On the other hand, the "keeping things in" policy usually results in a policy that by default *denies everything*, and allows only what is necessary for the proper functioning of a system. This embodies the principle of "[least privilege](#)" and in the context of a firewall is called a **default deny** policy. A firewall configured this way allows only the handful of things that are strictly required. This limits inbound traffic as before, but also only allows outbound traffic to carefully chosen targets. For example, this might only allow outbound traffic to a secured mail server, ssh server, and the few web servers required for an employee to accomplish their job. This drastically limits the means by which traffic can enter *or* leave the network, and if an employee executes a trojan as in the last example, that malicious software will not be able to contact its evil master because the malicious Internet host will almost certainly not be in the list of allowed outbound connections.

The obvious downside to a "default deny" firewall policy is maintenance and inconvenience -- it is harder to install in the first place, and any new network service or traffic type on the network must be explicitly allowed or it will not function. Allowing all outbound traffic significantly cuts down on this kind of maintenance -- at the cost of security.

Firewall and Network Testing Tools

iptables: set and clear rules in netfilter

[iptables](#) is actually the user space tool for administering the `netfilter` functions and tables in the Linux kernel, but the entire `netfilter` and `iptables` package is commonly referred to simply as `iptables`. `iptables` has several built-in tables of rules (such as `filter` and `nat`), several built-in "chains" (which are sets of network traffic including the built-in INPUT, OUTPUT, and FORWARD for inbound, outbound, and routed traffic), a set of powerful loadable modules of matching stateful filters, the typical set of stateless criteria (such as source, destination, and interface), and a set of targets that represent what to do with a matching packet. These options allow sophisticated firewalls to be defined.

`iptables` can be intimidating and confusing at first glance even for veteran sysadmins, but especially to users who are not used to configuring firewalls at all or are used to configuring firewalls through a GUI. `iptables` expressive plugins further complicate the syntax. A typical `iptables` command looks something like this:

```
$ iptables -t filter -A INPUT -m state --state NEW -p tcp -s 192.168.0.1 --dport 23 -j REJECT
```

Upon closer inspection, `iptables` is revealed to be merely a command whose arguments define a single rule for packet filtering based on a number of possible criteria. `iptables` takes those arguments translates them one command at a time into priority-ordered filter rules in the Linux kernel. Thinking of `iptables` as a command with arguments can help demystify `netfilter` and the process of designing firewalls with `iptables` -- let's break down the above `iptables` command and translate it into English:

iptables command arguments

command/argument translation

<code>iptables</code>	<i>We're going to use the <code>iptables</code> tool to insert a new rule into <code>netfilter</code>.</i>
<code>-t filter</code>	<i>This rule is going to go in the <code>filter</code> table, which is the built-in packet filtering table. This rule will apply only to:</i>
<code>-A INPUT</code>	<i>packets that have been put into the <code>INPUT</code> chain either by the kernel or by some previous rule and which:</i>
<code>-m state -- state NEW</code>	<i>represent a new connection,</i>
<code>-p tcp</code>	<i>are Transmission Control Protocol (TCP) packets,</i>
<code>-s 192.168.0.1</code>	<i>are from the host 192.168.0.1,</i>
<code>--dport 23</code>	<i>and are destined for port 23.</i>
<code>-j REJECT</code>	<i>Reject any matching packet. Processing of all packets matching this rule will instantly jump to the built-in target <code>REJECT</code>, which means that the packet will be rejected by the kernel with some kind of network error message.</i>

A few other examples:

```
$ iptables -p tcp --syn --dport 23 -m connlimit --connlimit-above 2 -j REJECT
```

This rule (from `man iptables`) allows 2 telnet connections per client host. Note that this rule uses the `connlimit` matching module, and rejects additional connections.

```
$ iptables -A INPUT -i lo -j ACCEPT
$ iptables -A OUTPUT -o lo -j ACCEPT
```

These rules accept any inbound or outbound traffic on the internal loopback network device (an internal, logical network adapter the kernel uses for network communication internal to the computer) regardless of state, protocol, source, or destination address. The `-i lo` and `-o lo` arguments specify the "input interface" and "output interface" the packet arrived on.

```
$IPTABLES -t filter -A INPUT -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
$IPTABLES -t filter -A OUTPUT -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
```

These rules accept all INBOUND and OUTBOUND traffic regardless of interface, address, port or protocol. They use the `state` matching module, but accept all `NEW`, `RELATED`, and `ESTABLISHED` packets (which is basically all traffic). This rule is basically like having no firewall at all!

NEW, RELATED, ESTABLISHED

Think of your firewall as a security checkpoint in a big office building. There's usually two lines -- one for people with IDs, and one for people without IDs. If someone already has an ID, they can skip the long line, and go through. If they don't, they have to wait to get an ID card or visitor's pass. This is analogous to the distinction between **NEW** traffic versus **RELATED** or **ESTABLISHED** traffic (which you usually see together). Traffic marked **NEW** doesn't have an ID badge yet, because it is the first packet of a new stream of traffic. On the other hand, a packet of a **RELATED** or **ESTABLISHED** stream is part of something that by definition has already come through the firewall in the past. In other words, the firewall has already given that stream a "badge" (which is really an entry in an internal firewall data structure).

Among other things, this means that firewalls are typically structured so that the first section passes all accepted **RELATED,ESTABLISHED** traffic first, and then carefully allows only certain kinds of **NEW** traffic. Why do it in that order?

While this brief introduction to `iptables` should point you in the right direction, there are other features of `iptables` not included here that you may want to use for the exercise. There are many HOWTOs, tips, and tutorials online in addition to the `iptables` manpage; the exercise manual assumes that in order to complete the `iptables` exercise, you will need to [do some research](#) on your own.

nmap: network mapping port scanner

[Nmap \(homepage\)](#) is a very popular "[port scanner](#)" that can be used to determine what kind of services are running on a remote or local host, perform [OS fingerprinting](#), and many other tasks. Nmap is capable of performing many tasks in a "stealth mode" designed to not raise the suspicion of the victim, but some tasks require more obvious techniques.

Nmap is incredibly powerful, but the basic functionality of the application is easy to use:

```
$ sudo nmap yahoo.com
```

```
Starting Nmap 4.20 ( http://insecure.org ) at 2007-09-22 21:33 PDT
Warning: Hostname yahoo.com resolves to 2 IPs. Using 216.109.112.135.
Interesting ports on w2.rc.vip.dcn.yahoo.com (216.109.112.135):
Not shown: 1694 filtered ports
PORT      STATE SERVICE
25/tcp    open  smtp
80/tcp    open  http
443/tcp   open  https
```



```
25/tcp open  smtp
80/tcp open  http
443/tcp open https
```

Nmap finished: 1 IP address (1 host up) scanned in 29.990 seconds

```
$ sudo nmap www.somehost.edu -P0
Starting Nmap 4.20 ( http://insecure.org ) at 2007-09-22 21:34 PDT
Interesting ports on dns.somehost.edu (33.xx.111.1):
Not shown: 1677 filtered ports
PORT      STATE SERVICE
53/tcp    open  domain
80/tcp    open  http
443/tcp   open  https
2048/tcp  open  dls-monitor
2049/tcp  closed nfs
2053/tcp  open  knetd
2064/tcp  closed dnet-keyproxy
2065/tcp  open  dlsrpn
2067/tcp  open  dlswpn
2068/tcp  open  advocentkvm
2105/tcp  open  eklogin
2106/tcp  open  ekshell
2108/tcp  open  rkinit
2111/tcp  open  kx
2112/tcp  open  kip
2120/tcp  open  kauth
2121/tcp  open  ccproxy-ftp
2201/tcp  open  ats
2232/tcp  open  ivs-video
2241/tcp  closed ivsd
```

Nmap finished: 1 IP address (1 host up) scanned in 32.385 seconds

See the Nmap manpage or online documentation for advanced features.

ifconfig: configure Linux network devices

[ifconfig](#) is the network interface configurator in Linux. It is most commonly used by users to see network addresses and statistics, but can also be used to enable and disable interfaces, set configuration options such as network addresses, and more.

For the purposes of this exercise, `ifconfig` will be used to determine what network addresses are running on what interfaces.

To see the current interface configurations:

```
$ ifconfig

eth0      Link encap:Ethernet  HWaddr 00:00:5A:00:01:B3
          inet addr:64.81.0.256  Bcast:64.81.40.255  Mask:255.255.255.0
          inet6 addr: fe80::200:5aff:fe00:1b3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1826346 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1887951 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:691689933 (659.6 MiB)  TX bytes:1037280707 (989.2 MiB)
          Interrupt:58

eth1      Link encap:Ethernet  HWaddr 00:13:D4:04:44:CA
          inet addr:10.10.10.10  Bcast:10.10.10.255  Mask:255.255.0.0
          inet6 addr: fe80::213:d4ff:fe04:44ca/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1165519 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1549057 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:428484191 (408.6 MiB)  TX bytes:1780325755 (1.6 GiB)
          Interrupt:50

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:5808042 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5808042 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6895907043 (6.4 GiB)  TX bytes:6895907043 (6.4 GiB)
```

Note that `eth0`'s address is 64.81.0.256, while `eth1`'s address is 10.10.10.10. This means that the two interfaces are on different networks.

telnet: cleartext remote shell

[TELNET](#) (TELe-NETwork) is a cleartext remote terminal protocol. On its face, telnet is very simple; the user issues commands over a TCP socket, and the server replies with the results of those commands and waits for more input. In practice, this is complicated with various network and terminal emulation layers. Still, telnet is one of the simplest and oldest network protocols still in use. Due to its cleartext nature and low level access to the system, telnet is incredibly insecure -- it was common in the past for system administrators to log in as root using telnet on a hub network connection that could be sniffed by any sufficiently prepared attacker.

Thanks to the advent of Secure Shell (ssh), active use of telnet servers has died off except for some specialized uses. One place where telnet lives on is debugging ASCII-based network services. For example, web pages can be retrieved by telnetting to HTTP servers, and emails can be sent by telnetting to SMTP servers.

Telnetting to a suspected open port is still one of the fastest ways to see if a service is available or reachable.

Here are a few sample uses of telnet:

here are a few sample uses of telnet:

```
$ telnet yahoo.com 80

Trying 66.94.234.13...
Connected to yahoo.com.
Escape character is '^]'.
GET /
...

<html><head> ...[web page data] ...
</body>
</html>

Connection closed by foreign host.

$ telnet mailserver.net 25
Trying 216.0.1.1...
Connected to mailserver.net.

Escape character is '^]'.
220 mailserver.net ESMTP Postfix (Ubuntu)
HELO sender.net
250 sender.net
MAIL FROM: me@sender.net
250 Ok
RCPT TO: me@mailserver.net
250 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
Subject: test mail message
test message
.

250 Ok: queued as 152CFB8802A
^]

telnet> Connection closed.

You have mail in /var/mail/me
```

For this exercise, you will use `telnet` to test if a TCP port is open on a remote host. Telnetting to an IP and port (see above) should return a "connected" message if it is possible to connect to a running server.

netcat: a network swiss army knife

netcat (often `nc` on some systems) is a Unix utility for creating and using TCP and UDP sockets. In a very simplified way, netcat is like a telnet client and server without any built in protocol or terminal emulation. Another way of putting it is that netcat is the bare essentials for creating a TCP or UDP socket and client, with hooks for using standard in and standard out for IO.

There are too many cool uses of netcat to describe here. For the purposes of this exercise, we'll use netcat to create "fake" TCP or UDP servers that we can use to test firewall configurations.

Creating fake TCP/UDP servers with netcat

Starting a fake listening server (a program that will accept connections on a port) is as simple as running:

```
$ sudo nc -l 80          # we need sudo because 80 is a privileged port
```

... to start a listening TCP socket on port 80. Then, from the another host, you can either use telnet or nc to connect to the server you just started on the the first host. You should be able to type in one window and see output in the other if the network pipe is open.

Testing UDP services is exactly the same -- you can use netcat for that, too. You need to start a listening UDP process on the receiving side, and a sending process on the sending side. If you are testing UDP traffic from a client to a server, you can do something like this:

```
[server]$ nc -u -l 10000    # listen for UDP traffic on port 10000
```

Then on the client, do something like this:

```
[client]$ nc -u server 10000  # connect to server via UDP on port 10000
```

After establishing the connection, enter some data from standard input (probably your keyboard). Input on the sender should appear on the receiving terminal. Hit ^C to close the programs. UDP is of course an unreliable network protocol, so it's possible that there will be errors in the text file.

You can do any of this in reverse to test the connection from a server to a client. If you're able to transmit data, then the firewall is allowing communication.

Introduction

You are Wilbar Memboob, the Security Administrator of [FrobozzCo](#) ("*You name it, we make it*"). You are looking to hire a new system administrator to replace the guy you just fired -- unfortunately, even though his resume looked great on paper and he interviewed well, once you put him at a console he clearly had no idea what he was doing.

To him, proper permissions meant that the application worked; if it was secure then that was just icing on the cake. You should have done something when the other admins starting calling him "Triple-7" -- a.k.a. "wide open". But it was when he changed the company fileserver directory permissions to "ugo+rwX" and the projected budget was "released" early -- including a list of the employees being laid off -- that he signed his own pink slip.

To keep this from happening again, you've created a little test for your new applicants to take. Unfortunately, before you can grade the applicants, you need to create an

To keep this from happening again, you've created a true test for your new applicants to take. Unfortunately, before you can grade the applicants, you need to create an answer key.

Assignment Instructions

In the assignment portion of this exercise, you're going to create an answer key for your "hiring exam." You'll need to swap in your nodes and complete the exercises below.

Setup

1. If you don't have an account, follow the instructions in the [introduction to DETER](#) document.
2. Log into DETER.
3. Create an instance of this exercise by following the instructions [here](#), using `/share/education/PermissionsFirewalls_UCLA/permissions.ns` as your NS File.
 - In the "Idle-Swap" field, enter "1". This tells DETER to swap your experiment out if it is idle for more than one hour.
 - In the "Max. Duration" field, enter "6". This tells DETER to swap the experiment out after six hours.
4. Swap in your new lab.
5. After the experiment has finished swapping in, log in to the node via ssh.

As mentioned in [DETER Instructions](#), changes to DETER nodes are lost when the nodes are swapped out. This means that you must manually save your work between working on nodes in order to keep it. However, **this exercise** includes experimental scripts to help you save and restore your work.

Saving your Work


After completing the POSIX and firewall-related exercises on the `server` node, cd into the `/root` directory and execute the script `/root/submit.sh`; like this:

```
$ ./submit.sh
```

...it will make a tarball of all the relevant files and their permissions, including the `/root/permissions-answers.txt` and `firewall.sh` script you have updated. **You must copy or move the created tarball into your group directory, otherwise it will be lost upon swapout.**

Restoring Your Work

The **experimental** script `/root/restore.sh` on the `server` node, takes as input the path to a tarball created by `submit.sh` described above and restores the files to their proper locations on the system. This includes all the files you are asked to create or change in the first part, as well as the `firewall.sh` script for the second part.


 **WARNING:** These scripts do **not** back up all arbitrary changes you may have made to the node (e.g., changing a peripheral configuration file), and it does not "merge" system files with submission files -- *it only restores submission files copied by submit.sh*. You shouldn't need to change anything else, but see `submit.sh` and `restore.sh` to see exactly what those scripts do, and do not delete any of your submission tarballs so that you can go back to an earlier version if need be.

To use the `restore.sh`, copy a tarball created by `submit.sh` into the `/root/` directory and execute this command:

```
$ ./restore.sh username-permissions-123123123.tar.gz
```

You will be asked if you want to automatically overwrite files, and if you want to selectively restore some files and not others. The options are self-explanatory.

Finally, if you don't trust the scripts, you can always make your own backups into your group directory and restore them by hand if you prefer.

 **NOTE:** You do not need to run the submit and restore scripts with `sudo`. However, if you use `sudo` to run `submit.sh`, your tarball will be named after the `root` user. This is OK -- just run `sudo ./restore.sh root-permissions-2134234243.tar.gz`.

Tasks

Your task is to create an answer key for the following test:

FrobozzCo Permissions and Firewall Test

This test is a part of the application process for the administrator position at [FrobozzCo](#). You will be presented with a number of tasks and questions; the tasks must be executed on the `server` node of the DETER experiment we have created for this purpose (you can use the `client` node for testing), while the questions must be answered in a plain text file that will be submitted with the results of your tasks. See below for further instructions.

Part 1: POSIX File Permissions and 'sudo'

Instructions

The following section includes a number of permissions, firewall, and file creation exercises. This test will be performed on a live network testbed with a full complement of standard utilities and editors. In addition, you are fully encouraged to use the Internet, man pages, help screens, and any other resources available to you in the execution and answering of these problems.

Please read and use these [disambiguation rules](#) for setting the correct permissions. Also, make sure that you test your answers -- POSIX permissions are simple in theory, but in practice many combinations have counterintuitive effects!


Your answers will consist of an archive, created by the `submit.sh` script described above. The archive will contain:

1. A copy of the necessary server resources containing the results of your file creation and permissions modifications. You will be graded only on the correctness of

1. A copy of the necessary server resources containing the results of your file creation and permissions modifications. You will be graded only on the correctness of your answers, not their elegance nor how efficiently you came to them.
2. Your written answers to the numbered short answer questions. These should be written in the file `/root/permissions-answers.txt` on the server system using an editor (such as `vim`).
3. Your firewall script.

See below for instructions on [submitting](#) your answers.

Home Directory Security

 **Note:** Admins -- members of the `wheel` group have `sudo` access. However, unless instructed to do so, use only standard UNIX ACLs -- don't give user accounts `sudo` permissions or consider the `sudo` access the 'wheel' group already has. Of course, *you* need to use `sudo` to create the accounts, edit root files, etc... this is exactly what `sudo` is for.

Your server needs two home directories, the usual `/home` and also `/admins` for your team. Normal home directories are private, while the homedirectories in `/admins` will be publicly viewable and somewhat collaborative.

1. Create the `/admins` directory.
2. Create the group `emp` (see `man groupadd`).
3. Create the normal (i.e., non admin) user accounts `larry`, `moe`, and `curly`, and the admin user accounts `ken`, `dmr`, and `bwk`.
 - Note that several files from `/etc/skel` are copied into each new homedir.
 - Use `passwd` to set their passwords to whatever you like (password security is not important for this test).
4. make sure you specify the home directories for the admins as being in `/admins` (see `man adduser`).
5. add the non-admin accounts to the `emp` group and add the admins to the `wheel` group by editing `/etc/group` or using `usermod`.
6. add `larry`, `bwk`, and `dmr` to `ken`'s group.
7. add `moe`, `dmr`, and `ken` to `bwk`'s group.
8. add `curly`, `ken`, and `bwk` to `dmr`'s group.
 - Admins are **not** a part of the `emp` group.
9. The permissions on the home directories already exclude prying eyes by default. Set the permission mode on `/home` itself so that normal users can't list its contents (but can still access their home directories) but so that members of the `wheel` group have full access to the directory (without using `sudo`).
10. By default, the owner of a homedir is set to be the user and the group class of their homedir. Set the permission modes recursively on the individual homedirectories in `/admins` (see `man chmod`) so that:
 - owners have *full access*
 - group users (users who are in the group associated with a user's home directory) can read and create files in that homedir
 - Set it so that any other users can read and execute any files
 - Files created by a group member in that homedir should be set to the homedir owner's group. (Hint: Look up what the SUID and SGID bits do on directories.)
 - Homedir permissions for the normal users should use the default permissions.
 - Example: `larry` is in `ken`'s group. `larry` can create files in `ken`'s homedir, and those files are owned by `larry`, but are assigned to `ken`'s group rather than `larry`'s group. `moe`, not in `ken`'s group, can only read and execute files.

The Ballot Box

All regular employees use this directory to submit votes for "employee of the month".

1. Create the `/ballots` directory.
2. Set the permissions on `/ballots` so that it is owned by root and users can write files into the directory but cannot list the contents of the directory.
3. Furthermore, set it so that members of the `wheel` group have no access (not including `sudo`).
4. **Short Answer 1:** Is there any way that employees can read the ballots of other users? If so, what could be done to stop this? If not, what prevents them from reading them?
5. **Short Answer 2:** What does the 'x' bit mean when applied to directories?

The TPS Reports Directory

Admin employees submit TPS reports in this partially collaborative directory.


1. Create the `/tpsreports` directory.
2. Create the `tps` user.
3. Set the permissions on `/tpsreports` so that it is owned by `tps` and that members of the `wheel` group have full access to the directory, but so that no one else has access to the directory.
4. Furthermore, configure it so that files written into it are still owned by the `wheel` group, but so that one member of `wheel` cannot delete another member's files.
5. **Short Answer 3:** Which users on the system can delete arbitrary files in the `/tpsreports` directory?

5. **Short Answer 3:** Which users on the system can delete arbitrary files in the `/tmp/reports` directory?

6. **Short Answer 4:** Is there any way that non-wheel employees can read files in the `/tmp/reports` directory?

7. **Short Answer 5:** What do '0' permissions mean for the owner of a directory? What privileges do they have over files in that directory?

sudo: Editing Configuration Files

 For the following three short answer questions, assume that your answers are unrelated; that is, your answer for question 6 does not impact the answer for questions 7 or 8.

All members of the `wheel` group do system administration on the server. Because of this, they all have full sudo access to `root` privilege with the `/etc/sudoers` directive `%wheel ALL=(ALL) NOPASSWD: ALL`. The "NOPASSWD" means they don't have to enter a password upon sudo invocation.

The employee `larry` is the webmaster for the system, so he has some extra privilege compared to the other employees. For example, `larry` has sudo access to the command

```
/usr/bin/vim /etc/httpd/conf/httpd.conf
```

... with the directive

```
larry ALL=(ALL) /usr/bin/vim /etc/httpd/conf/httpd.conf
```

... so that he can update the configuration of the webserver.

Short Answer 6: Is this safe? Why or why not? If it is not safe, is there a better way to give `larry` this access? If it is safe, how do you *know* it is safe? (Hint: search online for common sudo issues.)

sudo: Restarting System Processes

As a part of his webmaster duties, `larry` often requests the `wheel` group to restart the Apache server with the command `/etc/init.d/httpd restart`. It would make everyone very happy if there was a secure way to let `larry` restart the Apache server (but not give him any other access).

Short Answer 7: Assuming the init script `/etc/init.d/httpd` has no unknown vulnerabilities, is it safe to grant `larry` sudo access to the command `/etc/init.d/httpd restart`? If this is not secure, explain why.

POSIX and sudo: Two Wrongs Make a Much Bigger Wrong

Carefully examine the permissions on the server. Look at `/etc/group`, `/etc/sudoers`, and the files in the `/admins` and `/home` directories (and their subdirectories).

Short Answer 8: Is there some way that `moe` or `curly` could subvert this system to gain `root` privileges? If not, how do you know this is true?

Hint: Consider what happens during the UNIX login process -- the time between when the user enters the correct password and when they can interact with their shell.

Part 2: Firewall Configuration

The test server has a totally permissive firewall installed -- it accepts all inbound and outbound traffic from all ports, protocols, addresses, interfaces, and states. This is basically like having no firewall at all.

Your task is to configure the firewall according to the principle of "least privilege". This means that it should be maximally restrictive while still permissive enough to allow a strictly defined set of tasks. While some of these rules can *also* be configured in the server software (this strategy is called [defense in depth](#)), we want you to implement the rules in iptables **only** -- do not reconfigure the underlying software.

The firewall has been copied into the directory `/root/firewall/` along with a script called `extingui.sh` to "put out the fire" and clear all the rules in case you make a mistake. The firewall is **not** enabled by default -- to enforce the rules, execute:

```
/root/firewall/firewall.sh
```

... as the `root` user or using `sudo`:


```
sudo /root/firewall/firewall.sh
```

This will load the rules and start enforcing them. To make sure that you are removing all iptables rules, you should run `extingui.sh` in **between every invocation of `firewall.sh`** or rules might "stick around" which can be very confusing if you are trying to debug the system. This can be done like this as `root` (or with `sudo` as above):

```
/root/firewall/extingui.sh
```

If you make the server inaccessible with broken rules, don't worry -- you can reboot the node in the DETER console, and since the firewall is *not enabled by default*, you can log in in order to fix it. Your files will still be on the experimental node **as long as you don't swap out the experiment**. (Of course, you can permanently save your files in your home directory.)

Finally, only your final product is evaluated -- not the number of times you have to reboot the server. You should expect to lock yourself out a few times. ☐

 Your experimental nodes have at least two networks. The first is a control network between your node and `users.deterlab.net`. This is the network you use to connect to your nodes from `users`. Your nodes also have an "experimental" network that connects all the machines in a given experiment. For this project, your experimental network connects `server` and `client`

The firewall only needs to limit the experimental network interface (the interface with the 10.1.x.x network) and should **not** ever limit the control network (192.x.x.x as of this writing) or you may cut yourself off from the node. The experimental interface is one of `eth0-ethN` and you can determine which using the command `ifconfig`

and looking for the 10 x x x interface. Be warned that the specific interface used may change with a reboot or different experimental node

and looking for the `client` interface, be warned that the specific interface used may change from a release to different experimental nodes.

See below for instructions on using environment variables to define the experimental interface in your `firewall.sh` script.

Here's what the firewall needs to do:

1. **passively ignore** any traffic inbound to the interface that says it's coming from the server itself (obvious spoof attempt). The server uses the localhost loopback device `lo` for internal traffic, so it should never see incoming traffic from its own IP on the experimental network interface. (See test case 11, below)
2. Allow all *established* traffic on the experimental network interface. Established or related traffic is traffic that is part of *previously accepted* new connections.
3. Accept **new** connections on the experimental network (10.1.x.x) of the types listed below:
 1. Inbound TCP connections to the OpenSSH, Apache, and MySQL servers on their standard ports. (Test cases 1, 3, 5)
 - The MySQL server should **only** accept connections from the **client** host.
 2. Inbound UDP connections to the **server** ports 10000 to 10005 from the host **client**. (Test case 8)
 3. Inbound ICMP ping requests and replies. (Test case 6)
 4. Outbound TCP connections to any OpenSSH, SMTP, and Apache (on standard ports). (Test cases 2, 4)
 5. Outbound UDP connections to the ports 10006 to 10010 on host **client** from the **server**. (Test case 9)
 6. Outbound ICMP ping requests and replies. (Test case 7)
4. **passively ignore all other traffic**. (Do not allow it or respond to it in any way.) (Test case 10)

There are many online resources and tutorials for iptables configuration -- feel free to use them. Be aware, however, **not all tutorials emphasize the principle of least privilege** and may give you overly permissive advice! In order to properly configure the firewall, you must consider the basic ways the firewall can differentiate traffic and allow only the specific types you require to properly function.

Please read the helpful advice in the next section for configuring and testing your firewall.

Tips and Tricks

This section includes important rules and tips for making sure that your answers are correct.

1. Rules for resolving filesystem permission ambiguities:

Permissions should *always* be set to reflect the **least privilege** required to fulfill the requirements. In POSIX permissions, every bit set represents *less security*, so we want to set as few bits as possible. Resolve any ambiguities with this cumulative list of guidelines:

1. Files you are instructed to create are to be owned by `root` unless otherwise specified. Doing the work as `root` should do this automatically. (You can become `root` by executing `sudo su -`)
 - Exception: Files in homedirs should be owned by the homedir owner (`useradd` should do this by default for boilerplate files like `.bash_login`, etc).
2. When setting ownership of a file, set the group class to the same thing as the user (owner) class unless otherwise specified.
 - Example: Create the directory `foo`. Set `www-data` as the owner of `foo`.
 - Answer: Set `www-data` to be the user (owner) and group class of `foo`.
 - Example: Create the directory `xyzzzy`. Set the group class to `www-data`.
 - Answer: Set the group class to `www-data` but the user class should still be set to `root`.
3. Files whose permissions you are *not* otherwise instructed to change should stay at their default.
 - Example: Create directory `baz`. Create directory `quux` and set its permissions to be wide open.
 - Answer: Don't change `baz`'s file mode. It should be owned by `root` (as per #1). Set `quux` permissions to `0777`.
4. For any files whose permissions you *are* instructed to change, unspecified permissions are **always** assumed to be 0 (no access). In other words, instructions for setting file permissions implicitly include all access groups (even those not explicitly mentioned).
 - Example: Set directory `foo` so that its owner has full access.
 - Answer: The correct mode is `0700`, since no group or other permissions were specified and we assume that they are 0.
5. Permissions are assumed to be for all classes unless specified.
 - Example: Remove all permissions from `/dev/hdc`.
 - Answer: Set `/dev/hdc` to mode `000` (root still has access of course).
 - Example: Set read and write permissions on `/etc/motd`.
 - Answer: `/etc/motd` should be set to mode `0666` -- all classes can read and write.
6. When granting permissions, `setuid`, `setgid`, and sticky bits are **never** granted unless specifically required to solve the problem. These bits are special and must be

When granting permissions, `setuid`, `setgid`, and `sticky bit` are never granted unless explicitly required to solve the problem. These can be special and must be required by the nature of the question or be otherwise mentioned to be granted.

- Example: Set a directory so that owner has full access and it's group class members can create files in it.
 - Answer: The correct mode is `0730` because a user (in this case the group user) requires both write and execute to create files. The question didn't specify anything requiring `setuid` or `setgid` so they are not enabled.
 - Note: the `other` class in this example has no access because the permission is assumed to be `0` (as per #4).
 - Note: In #3 above, see that "wide open" for the directory `quux` meant `0777` **not** `4777` because `setuid` was not specified or required.
7. If any tasks are not possible with the standard POSIX permissions available in this exercise, explain why.

2. Use Environment Variables in your firewall.sh!

Your firewall script is only supposed to limit the traffic on the "experimental network" interface, as opposed to including the "control network" of DETER. If you block the control network, you're likely to lose connection to your node, or shut off networked file systems, etc.

Unfortunately, different DETER nodes (the physical computers you are given) bring up their networks on different interfaces (and in general you can't control which nodes you get). This means that on one node, the experimental network might be on `eth0`, and on another node, it might be on `eth4` (or any other `ethN`). This makes writing your script difficult because it is not 100% portable from one node to another.

However, we can use an environment variable to substitute in the right interface. In the `firewall.sh` script, there is a variable declaration:

```
ETH="eth0"
```

You can use this variable with the token `$ETH` -- the shell will substitute in its value at runtime. Use `ifconfig` to make sure that `ETH` is set to the right value for your experimental node (or update it). For example, use `ETH` in a hypothetical `iptables` command like this:

```
iptables -A INPUT -i $ETH -j ACCEPT
```

This way, you only need to update the `ETH` variable if your interface changes, rather than every `iptables` call that specifies an interface.


3. How to test your firewall


Testing your firewall is easy; you just need to make sure that the allowed services are allowed, and that things that should be denied are denied. To do that, we'll use a few tools like `telnet`, `netcat`, and others.

You may also have noticed that this experiment swaps in two nodes instead of one. One will be called `client` and the other will be called `server`. `server` is the node with the firewall and resources you want to protect, but you can use `client` to check to see if the firewall is doing its job. You can also use `client` as a target to see if the server's outbound rules are functioning properly, using tools such as `nmap`, `telnet`, `nc`, and others in the [network tools portion of this document](#).

The following tests have been provided by Dr. Tanya Crenshaw from the University of Portland. While these test cases cannot guarantee a perfect firewall, they should help you understand if your firewall is meeting the guidelines.

Firewall Test Cases

 **c\$** indicates tests run from client node and **s\$** indicates tests run from the server node

 The test file "hi.txt" was created on the client using by executing: `c$ echo "hi" > hi.txt`

#	Rule	Test	Result
1	Allow inbound traffic to the OpenSSH port.	<pre>c\$ telnet server 22 Trying 10.1.1.3... Connected to server. Escape character is '^]'. SSH-1.99-OpenSSH_4.2</pre>	SUCCESS: A connection on port 22 is established using the telnet tool.
2	Allow outbound ssh traffic (established).	<pre>See well-formed reply from ssh server in Test #1: 'SSH-1.99-OpenSSH_4.2'</pre>	SUCCESS: The ssh server running on the server node replies to telnet's connection request with the SSH version number.
3	Allow inbound traffic to http traffic.	<pre>c\$ telnet server 80 Trying 10.1.1.3... Connected to server. Escape character is '^]'. </pre>	SUCCESS: A connection on port 80 is established using the telnet tool.
			SUCCESS:

```

4 Allow outbound http traffic (established).
  c$ nc -q -l server.MyExp.MyClass.isi.deterlab.net 80 < hi.txt
  ...

```

```

5 Allow inbound and outbound MySQL traffic
  c$ [client]$ telnet server 3306
  Trying 10.1.1.3...
  Connected to server.
  Escape character is '^]'.
  K#HY000Host 'client-link0' is not allowed to connect to this MySQL serverConnection closed by foreign host.

```

```

6 Allow new inbound ping traffic.
  c$ ping server
  PING server-link0 (10.1.1.3) 56(84) bytes of data:
  64 bytes from server-link0 (10.1.1.3): icmp_seq=0 ttl=64 time=0.412 ms

```

```

7 Allow new outbound ping traffic.
  s$ ping client
  PING client-link0 (10.1.1.2) 56(84) bytes of data:
  64 bytes from client-link0 (10.1.1.2): icmp_seq=0 ttl=64 time=0.302 ms

```

```

8 Allow inbound UDP traffic on ports 10000:10005
  s$ nc -u -l 10000
  hello
  c$ nc -u server 10000
  (type 'hello')

```

```

9 Allow outbound UDP traffic on ports 10006:10010
  c$ nc -u -l 10006
  hello
  s$ nc -u client 10006
  (type 'hello')

```

```

.. Disallow all c$ telnet server PORT

```

The Apache Web Server running on the server node replies to the client's badly-formed http request with a well-formed http reply.
 SUCCESS: Connect to the MySQL server running on the server node using the telnet tool. MySQL server responds with an error indicating that outbound traffic is allowed.

SUCCESS: Ping request from client is replied to by server. Note that request is new traffic, while reply is established traffic.

SUCCESS: Ping request from server is replied to by client. Note that request is new traffic, while reply is established traffic.

SUCCESS: Traffic is allowed into the server on port 10000. Run the same test for 10001:10005.

SUCCESS: Traffic is allowed out of the server on port 10006. Run the same test for 10007:10010.

SUCCESS: The full test is to run a bash script on the client-side and server-side which attempts telnet on every other port aside

from those

10 other traffic `s$ telnet server PORT`

not addressed above, and also tests UDP, etc. A good firewall design should ensure that all traffic is dropped except that which is explicitly accepted.

11 Prevent spoofing Straightforward test unknown

NA

What can go wrong

1. Your firewall cuts off your access to the node.

If you have misconfigured your firewall, and it "locks you out," you can try to reboot your experimental nodes using the DETER interface. If that does not work, you will need to swap your nodes out and back in again, but beware that swapping your nodes out will destroy any work you have not backed up in your home directory.

Make sure you are using the environment variable to define the interface you are restricting -- this will help keep you from getting "locked out." See the [tips and tricks](#) section for more information.

Submission Instructions

Submit a tarball created by `submit.sh` to your instructor. You **must** use a tarball created by this script so that it will correctly save the permissions of the directories.