Perl Security

```
#!/usr/bin/perl
use strict;

                                              my$s=q:$
                                         /=’;’;%_=map{s/\s
        //g;chomp;/-/;$‘,$’}<DAT     A>; $_{$_}=[$_{$_}=~/\w+
/g]for(’p’,’o’);my$R=sub{my($n,$r)=@_;$r=  ~s/[a-z]/$n
=uc$&/e  ;sub{return(chr(65+index$r,$_[0])   )if!$#_;
  $r=~/.{$_[1]}/;(substr  ($r=$’.$&,ord($_[0])  -  65,
    1),$’=~/^$n/*1)}};@_   =map{&$R(1,$_{$_})}(@{ $_{
      o}       });%_=map{ /./; $&, $’,$’,$&}@{$_{p}};$
           /=       "\n";while(<>) { s/[a-z]/$_=uc$& ;m
             y($        a,$i)=1;$_= $ _ {$_}||$_;for$i(1.
              .$#_){(      $_,$a) =  &{$_[$i]}($_,$a)}for
                ($i=$#_;$i-1;){$_=&{$ _[--$i]}( $_)}$_=$_{$_
                   }||$_/oxeig;print}:; ;$s  =~ s/\s//g;eval
                   $s    __END__;1-EKMF   LGDqVZ  XNTOWYHUSPA
                 IBRCJ;3         -AJDKS IR   UXBL HWTMcQGZNPYF
                VOE;2                 -BDFHJLC  P  RTXvZNYEIWGA
                K                          MUSQO;4-ESOVPZjAYQUI
                                                     RHXLNFTGKDC
                                                MWB     ;r-YRUHQ
                                                  SLDPXN  GOKMIE
                                                    BFZCWVJAT;o
                                                     -1,3,2,4,
                                                      r;p-S
                                                      H,AR
                                                     ,KY
                                                     ;
```

Paul Fenwick

Jacinta Richardson

Perl Security
by Paul Fenwick and Jacinta Richardson

This training manual is maintained by Perl Training Australia. Information about Perl Training Australia’s courses, manuals and course extracts can be found at http://www.perltraining.com.au/.

This is release 1.5 of the Perl Training Australia’s "Perl Security" training manual.

Table of Contents

List of Tables

Chapter 1. About Perl Training Australia

Training

Perl Training Australia (http://www.perltraining.com.au) offers quality training in all aspects of the Perl programming language. We operate throughout Australia and the Asia-Pacific region. Our trainers are active Perl developers who take a personal interest in Perl's growth and improvement. Our trainers can regularly be found frequenting online communities such as Perl Monks (http://www.perlmonks.org/) and answering questions and providing feedback for Perl users of all experience levels.

Our primary trainer, Paul Fenwick, is a leading Perl expert in Australia and believes in making Perl a fun language to learn and use. Paul Fenwick has been working with Perl for over 10 years, and is an active developer who has written articles for *The Perl Journal* and other publications.

Consulting

In addition to our training courses, Perl Training Australia also offers a variety of consulting services. We cover all stages of the software development life cycle, from requirements analysis to testing and maintenance.

Our expert consultants are both flexible and reliable, and are available to help meet your needs, however large or small. Our expertise ranges beyond that of just Perl, and includes Unix system administration, security auditing, database design, and of course software development.

Contact us

If you have any project development needs or wish to learn to use Perl to take advantage of its quick development time, fast performance and amazing versatility; don't hesitate to contact us.

Table 1-1. Perl Training Australia's contact details

Phone:	+61 3 9354 6001
Fax:	+61 3 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058 AUSTRALIA

Chapter 2. Introduction

Welcome to Perl Training Australia's *Perl Security* training module. This is a one day training module in which you will learn about Perl's security related features, and issues that you should be aware of when working with Perl in a security sensitive context.

Course outline

- What is computer security
- Taint checks
- Opening files
- Executing system commands
- Dropping privileges in Perl
- Database security
- Tricks and traps
- Random numbers

Assumed knowledge

This training module assumes the following prior knowledge and skills:

- Intermediate Perl fluency, including a familiarity with Perl variable types and references, file input/output, system interaction and using Perl modules. Some experience of using Perl objects would be useful.
- Basic Unix fluency, including logging in, moving around directories, and editing files

Platform and version details

Security best practices in programming vary depending on operating system. This is because different operating systems have different security models.

This module is taught using Unix or a Unix-like operating system and covers some Unix specific security practices. Much of what is learnt will work equally well with Microsoft Windows and other operating systems; your instructor will inform you throughout the course of any areas which differ.

At the time of writing, the most recent stable release of Perl is version 5.10.0, however older versions of Perl 5 are still common. Your instructor will inform you of any features which may not exist in older versions.

The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appear *`like this`*

Notes and tips appear offset from the text like this.

Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Chapter 3. What is Computer Security

In this chapter...

In this chapter we will discuss what is computer security and why it is important to programmers. We will discuss the C.I.A. model of secure information storage and what kind of attacks can be carried out against our machines. Finally we'll discuss security-sensitive contexts, why we should avoid security through obscurity and how to make your code more secure.

Scope of this course

> A computer is secure if you can depend on it and its software to behave as you expect.
>
> -- Simon Garfinkel and Gene Spafford, Practical Unix and Internet Security

When the media talks about computer security, two common themes re-occur. One is the ever-increasing plague of viruses and worms that are bent on stealing your banking details, bringing down the SCO website, or gaining a new position on the 'most infectious virus' charts. The second theme is the evil cracker, who possesses the almost superhuman ability to launch a nuclear strike armed with nothing more than a PocketPC; especially if popular television programs like *Alias* are to be believed.

While malicious software and individuals are indeed security concerns, this narrow-minded view results in a poor consideration of the other requirements of our overall goal -- which is quite simply to have your system work the way in which you intend.

A comprehensive view of security will include consideration of disaster recovery plans, reliability of hardware, software and communications, staff trustworthiness, physical security, and a range of other factors. There are numerous excellent books on the topic of computer security, and this course does not aim to replace them.

The goal of this course is to discuss computer security and specifically how it relates to Perl. The material covered will help protect your systems not only from malicious attacks, but also from everyday bugs and user errors, from which many undesired operations extend.

This course does not cover aspects of computer security that do not relate to Perl. Considerations such as physical or personnel security, and documents such as disaster recovery plans, are very important for establishing a secure environment. However, they are beyond the scope of this course.

Further reading on a variety of security topics can be found in our *Further Reading* section, in the conclusion of this course. Perl Training Australia also offers consulting services in the fields general security and disaster management.

Target operating systems

Perl runs on a great many operating systems, the distribution of Perl 5.10.0 ships with operating-specific notes for no less than 32 different architectures.

Every operating system has its own unique security concerns, and this course cannot hope to cover them all. When an operating system independent way of achieving a goal is available, this course will aim to present that method. However in some cases no such method will exist. In most of those situations, this course will take a Unix-centric view. This is not because Unix is any more or less secure than any other operating system, but primarily because it is the primary basis of experience for the course authors. Where possible, attempts will be made to discuss options on other operating systems, in particular for Perl running in Microsoft Windows environments.

Why is security important

Security is most important if you wish to keep your job.

Security breaches can cost your organisation money, reputation, respect, productivity and custom. As an individual, a security compromise can interfere with the correct workings of your computer, reveal your personal information, or result in the abuse of your personal resources. Security breaches can also cause you and your company legal issues. If your privacy guarantee is violated and your email address list is sold to spammers you may be liable for compensation. It will be worse if your credit card database is compromised. Even details such as who your clients are, can be valuable information to the right people. For example some newspapers would be delighted to print information about how a prominent politician had been a regular patient in a cancer ward.

It must be noted that in the world of security, prevention is *much* better than cure. The amount of effort required to clean-up after even a minor incident is usually much greater than the effort required to prevent that incident from occurring. More importantly, unless the root of a problem is found and fixed, security incidents can continue to re-occur. As such, it should be no surprise that most texts on computer security focus upon incident prevention. This course is no exception.

There are three main requirements on secure information systems:

- **Confidentiality**: Information must only be accessible by those who are authorised to access it.
- **Integrity**: Data must be accurate and must not change through storage or transfer to another system.
- **Availability**: Data must be available to authorised users upon demand.

These three requirements do not work in isolation. An authentication violation (integrity) can expose data to unauthorised parties (confidentially) and can result in deleted or corrupted information (availability).

What attacks can occur?

As discussed in the previous section, we wish to ensure that our systems meet the CIA requirements of Confidentiality, Integrity and Availability. In the same way that we have three main requirements of a system, it’s possible to group attacks into three corresponding categories.

Table 3-1. Classes of attacks

Requirement Attacked	Description
Confidentiality	Any attack that breaks the confidentiality of the system can be considered a *privacy attack*. This may involve the revealing of personal information, customer records, financial details, private e-mail, or other information that was not intended to be shared. One of the most common targets for privacy attacks are system files and resources. The contents of the Unix `/etc/passwd` or `/etc/shadow` files can be of benefit to an attacker trying to gain greater access to a system. A common privacy attack involves *network sniffing*, which can reveal a great deal of information from un-encrypted connections such as HTTP, SMTP, POP3 or FTP. *SQL injection attacks* may also be used to reveal information.
Integrity	Compromising the integrity of a system can range in scope to being able to affect a few minor pieces, to a complete opening of all privileges to an attacker. While the most commonly reported integrity compromises are those resulting in the execution of arbitrary code by an attacker, there are numerous other examples. A commonly seen attack against poorly designed e-commerce systems would allow an attacker to potentially change the prices for a given order, obtaining goods or services at a discount rate.

Requirement Attacked	Description
Availability	Often the easiest goal to attack, a compromise of availability is also known as *denial of service* attack. Availability of a system may be compromised by numerous means, including consumption of network, disk, memory or CPU resources. Sometimes features that are designed to make a system more secure can be used to execute a denial of service attack. One particular example is the ’feature’ of some systems to deny a user access if they enter their password incorrectly three times in a row, or which deny all traffic from a machine if a portscan is detected. In these cases an attacker can impersonate the user or machine to whom they wish to deny access.

What is a security-sensitive context?

A *security-sensitive context* exists whenever special thought must be given to security. For example, whenever the presence of a program would allow a user or attacker to attempt to compromise one of our goals of *Confidentiality, Integrity, and Accessibility*.

In particular, a *security-sensitive context* exists in the following situations:

- Whenever a program is running with more privileges than the user who executed it. This includes programs with setid privileges, CGI programs and other program triggered by remote events.
- Whenever a program is working with security-sensitive data. This includes client information, credit card details, passwords and other pieces of sensitive information.
- Whenever a program is running with untrusted data. This particularly applies when the data comes from a source other than that of the user executing the program. For example file viewers, e-mail readers, web-browsers, print spoolers, and scheduled tasks all deal with data from untrusted sources.

There are three important things to remember when writing programs which may be run in a security-sensitive context. The first is that elevated privileges should be dropped as soon as possible. Nothing should be done with the elevated privileges that it is possible to do without them.

The second is to get rid of all data, no matter how trivial seeming, as soon as it is no longer required. If you’ve authenticated the user and don’t need their password anymore, clear the string or have the variable go out of scope. If you’ve successfully charged the credit card don’t store the number in your database. By doing these things you make it easier to avoid accidently exposing security-sensitive data to unauthorised viewers.

The third, and most poorly fulfilled, is to never trust your data. Verify everything you get from untrusted sources. Normal testing usually doesn’t find security flaws because normal testing doesn’t usually include the weird and wacky things that attackers might do to find a flaw in your code. Don’t assume that your input obeys the rules you need it to; instead, make it do so.

Part of good security planning is identifying what threats or possible threats exist, and what security goals exist for your system. While this is beyond the scope of this course, it will assist you greatly in the planning, programming and testing stages of your project.

Why does security through obscurity not work?

Security through obscurity (also referred to as *security by obscurity*) is a phrase which refers to a system which would be considered *insecure* if an attacker were to have full knowledge of its workings.

An excellent example of security through obscurity would be a website that has an administrative interface with no authentication requirements, that isn't linked to from any public page. The interface is secure as long as only the authorised people know about it. However, the URL for the interface can get into book marks, proxy logs, or the cache of public Internet terminals. A curious or clever person may be able to deduce the URL from other information.

Another common, and disappointing, example of security through obscurity are encryption systems which proudly proclaim that they use a 'secret, proprietary algorithm'. A good encryption algorithm should be secure even when an attacker knows the exact working of the algorithm. The most secure encryption algorithms are those that are widely known and documented, and continue to be secure despite many years of testing and research.

The label security through obscurity refers to obscuring *how* your programs work. It does not apply to obscuring *what* data your program needs to work. Keeping passwords, private cryptographic keys, and other data, which is required to run your program, secret is just good sense.

In Perl, your source code is generally available to the person attempting to run your program. However, tools do exist to allow your code to be obscured. While these may protect your code from casual inspection, you should not rely upon these programs keeping your source code 'secret'. At some point your file needs to be turned into Perl code in order to run, and a sufficiently skilled and curious user will also be able to turn your file into Perl code for them to read as well.

The InterBase LOCKSMITH account

An example of security through obscurity was the back door found in the InterBase database engine by Borland. Different services in the InterBase system communicated with each other using a 'LOCKSMITH' account, which possesses full access to the security accounts database. Unfortunately, the username and password for this LOCKSMITH account were hard-coded into the system, and consequently the same on every InterBase system before knowledge of the LOCKSMITH account became widely known.

It was possible for an external network user to also login with the LOCKSMITH credentials. As such, access to the InterBase security accounts database was made available. This would allow elevation of privileges, the definition of user-defined functions, and arbitrary execution of code. As this account was hard-coded into the server, it was not possible to delete or remove the account from the system, or even to change the password.

The LOCKSMITH back door remained in the InterBase system until Borland made their code open-source on the 25th July 2000. The account was then discovered by developers of the Firebird project, the open source development branch of the InterBase server.

It could be argued that if Borland did not release their source then this back door into the system would be kept secret and safe from being exploited. Unfortunately, the username and password can be found from running the Unix **strings** command over the binary libraries shipped with InterBase, so it is entirely possible that it was discovered by less ethical individuals before the problem became widely known. There are also numerous examples of back doors being found and exploited in other closed-source systems, including 3com switches (the ’debug’ account), and the Quake server (with a master password of ’tms’).

Implementing undocumented back doors (as opposed to documented and configurable administrator accounts) is one of the oldest examples of security through obscurity, and a practice that should be avoided.

Further information can be found about the LOCKSMITH account in US-CERT Vulnerability Note VU#247371.

How to make your code more secure

Perl is a very permissive programming language, and is designed to "*make simple things easy and hard things possible*". Unfortunately, like most kinds of permissiveness this can lead to increased security risks. In this course we’re planning to cover how you can write your Perl code in a secure fashion. However there are a few simple and straightforward ways in which you can immediately increase the security of your code, and these are true for all programming languages. This section will cover those techniques in more detail.

Review your code with someone else

Code review is a much under-utilised practice. A code review allows other programmers to assist in spotting bugs, increase their understanding of the code and to learn from each other. Authors usually have difficulty effectively proof-reading their own work, whether that work be words or code.

Reviewing code can result in earlier detection and correction of bugs, leading to less defects in production code, and less effort needed to fix them. It can identify places where documentation needs improvement and can ensure that the code complies with any relevant coding standards. Additionally, reviewing code encourages discussion and learning of the appropriate technologies. It helps both the reviewer and the reviewee.

Code can be reviewed for other reasons too, such as maintainability, security, testing and scalability. Obviously we highly recommend keeping these goals in mind while reviewing your code.

Code reviews can be informal such as walking through your code with a friend, or very formal involving checklists and sign-offs, possibly with your entire development team. The more people with input into the review the better, so long as that input is kept concise and to the point. Code review can even be outsourced to another team or even an external body. This is often called code auditing. While auditing can lead to similar levels of code improvement, the code writer tends to learn less through this practice, as they are not as directly involved.

However you achieve it, getting other people to review your code will ensure that your code heads towards being the best it can be. It also increase the chances of correcting errors before they become security problems.

Perl Training Australia proudly provides code auditing/reviewing services. For more information speak to your instructor in the next break or visit our website (http://www.perltraining.com.au/).

Use good programming practices

Regardless of the programming language you work in, there will be a general consensus by the programmers of that language as to what are the *right* things to do. In Perl a fundamental requirement on almost every program is that it starts with:

- `use strict;`
- `use warnings;` (or the `-w` flag).
- a comment detailing what your code does, who wrote it, and when.

It's highly recommended that you run your code in *taint mode* as well, as this enforces an extra level of strictness in checking inputs. We discuss taint mode in the next chapter.

Why use strict and warnings

If you're writing code that's anything longer than a few lines, then it's strongly recommended that you make use of Perl's `use strict` pragma, and the `use warnings` pragma or `-w` switch.

The benefits of using strict and warnings are simple and straightforward. These pragmas ensure that the vast majority of common programming mistakes that can be automatically detected *are* detected.

The `use strict` pragma defends primarily against typographical errors, such as typing `$freind` where one meant `$friend`. The `use warnings` pragma defends primarily against logic and programming errors, such as using an undefined value as if it were defined, printing to a closed filehandle, or treating a string as if it were a number. Using strict also naturally encourages a higher level of encapsulation and better design.

Once your code works cleanly with strict and warnings, only more difficult and time-worthy bugs will remain. There is simply no reason for programmers to waste valuable time debugging a problem caused by a spelling error when modern programming languages are capable of doing this for them.

It is acceptable to turn off strict and/or warnings for some sections of code, but only when the programmer knows exactly why they are doing so. An excellent example would be using `no strict 'refs'` in an `AUTOLOAD` to call subroutines from their names contained in scalar variables.

Other good rules

The Perl programming community has also created the following rules, some based on good rules from C and other earlier programming languages:

- Comment your code;
- Test your code;
- Restrict your variables to the smallest possible scope (by declaring them with `my`) and avoid global variables whenever you don't need them;
- Use meaningful variable names;
- Be consistent;
- Just because you *can* do something does not mean that you *should*. Aim for your code to be clear and obviously correct, rather than using obscure features;
- Don't reinvent the wheel - someone has probably already made a better one than you can. Refer to CPAN first;
- Use `CGI.pm` or another equally tested and prominent module for CGI parameter parsing;

- Use `DBI` for working with databases;
- Talk to people about the problem you have to work on, they may have already solved it or know someone who has.

Abiding by these rules will make you a better programmer. It will also make reviewing your code much easier and effective. Of course, it's sometimes acceptable to bend and break the rules, but you must know why the rules exist, and have a good reason for doing so.

New and self-taught programmers

Some new Perl programmers and some self-taught Perl programmers find abiding by these rules an exercise in frustration. Particularly the use of `strict`. As a result they often end up at logger-heads with much of the Perl community who helpfully suggest these practices. These rules are promoted to make your life easier, not harder. Following the above rules will save you time and energy in the long run and let you be a more productive programmer.

Keep up to date

Keep learning. A one day course cannot cover all the things you need to do to make your programs secure. Perl itself keeps changing and you need to understand what these changes are and why they were made. New modules are added to CPAN regularly and some of them may be more secure than those you chose last month or last year.

Keep your version of Perl patched and up to date. Security issues with Perl are discovered and patched. Sometimes these patches may be written for older versions of Perl and other times you can only solve the issue by upgrading. If nothing else, keep track of new Perl releases and read the `perldelta` file that comes with each, to see what has changed since the previous revision.

Keep you system up to date. There's very little point in writing secure Perl programs if your system itself has security problems. The moment an attacker can edit your script any security it may have offered is lost.

Security is a moving target. Years ago we didn't have exploits taking advantage of buffer overruns, now we do. Years ago we thought than any sufficiently large prime number was good material for cryptography, now we know that there are certain properties of primes we need to avoid. As security gets more sophisticated so do the attacks. The only way you can properly ensure that your code is secure is to keep up to date on what security means.

Chapter summary

- A program is secure if it behaves as expected.
- Any defect in a program is a security risk
- Security is important because breaches can cost money, good will, reputation, productivity and custom.
- The three requirements on secure information storage and processing are confidentiality, integrity and availability

- An example attack against confidentiality is SQL injection.
- An example attack against integrity is changing prices on a poorly designed e-commerce system
- An example attack against availability is a denial of service attack
- A security-sensitive context exists whenever a program is running with more privileges than the user who executed it, and whenever a program is working with security-sensitive data.
- Security through obscurity doesn’t work because the moment your secret gets out your security is dramatically reduced
- To make your code more secure use code reviews, good programming practices and keep your system and knowledge up to date.

Chapter 4. Taint Checks

In this chapter...

This chapter will talk about why taint checks exist, how to turn them on and how to use them. We will also cover some current taint traps that you might run into.

The importance of validating input

We all know the importance of validating our input. The old saying, "Garbage In, Garbage Out", reflects the truism that computer programs cannot provide meaningful output unless they are provided with meaningful input. When working securely, validating our input is more important than just being able to provide the correct results; it's also important to ensure that unexpected input does not cause our program to work in unintended and dangerous ways.

We'll begin by examining two ways in which the data supplied to our program can be used to potentially manipulate its behaviour unexpectedly. This data may come from a user at the keyboard, a network connection, via a web-browser to a CGI script, or could be the contents of a file on the disk. Regardless of how we obtain our data, it's important that we validate it correctly.

Sending user data to an application

One of Perl's strengths is that it makes it easy to call external applications. This makes Perl a popular choice with system administrators and system integrators. Perl can invoke an external program in many ways including `system`, `exec` and backticks (including `qx//` and `readpipe`) and also via calls to `open`, which we will discuss in the next section.

The code below shows an example where information is naively read from an untrusted user before being fed to a command.

```
#!/usr/bin/perl -w
# DON'T USE THIS CODE
use strict;
use CGI;

my $username = CGI->param("username");

print "Content-type: text/plain\n\n";
system("/usr/bin/finger $username");
```

In the above code we take a username from the user (in this case using Perl's `CGI` module, but it could be from any source) and pass that to the operating system's **finger** command. The **finger** command displays information about a user, such as their idle status, last time they checked mail, and any other information they wish to reveal about themselves. What we want the code to do is print out some information about the user in `$username` if they exist. Unfortunately, what we have written is actually much more permissive than this.

This code does not check that the username provided by the user is sensible. It could be similar to any of the following:

- `pjf` - a valid username. Everything works as intended.

- `''` (the empty string) - which would display all the users logged in to the system.
- `user@another.host` - a user on a different host. This could be used to exploit **finger**'s ability to make network connections to probe or otherwise access a third-party machine.
- `fred; rm -rf /etc/;` - in which case we print out the result of fingering the user `fred` and then proceed to delete all of the files in `/etc/`.

Of the above examples, we only intended for the first choice to be possible. By not checking our user's input we've allowed a situation where they can execute arbitrary commands on our system, as well as being able to contact the systems of others.

Using user data in a call to open

Perl's `open` command is sometimes a lot more powerful than programmers realise. This means that we really should think carefully about how we use it. Unfortunately, code which can be simplified to the following is all too common:

```
#!/usr/bin/perl -w
# DON'T USE THIS CODE
use strict;
use CGI;

my $filename = CGI->param('filename');

chomp $filename;
open(FILE, "/home/test/$filename")
      or die "Failed to open /home/test/$filename for reading: $!";
```

By not specifying the mode for opening the file, which we do by putting an explicit < in the `open` call, we run the risk of the user being able to specify their own mode to use when opening the file. Since our file starts with a path (`/home/test/`), they can't specify the traditional modes such as < or >. However, the user can specify a filename ending with a pipe, such as:

```
../../bin/cat /etc/shadow |
```

to instead run the command **/bin/cat /etc/shadow** and to read the results of this executed command. It doesn't take much imagination to realise that this could be used to do much worse than simply *reading* the shadow password file.

Obviously letting the user specify their own mode is a bad idea as it effectively gives the user shell access to your system. In a situation where the user should have no shell access to your machine (for example if your program is a CGI script) this is a huge security violation. The goal of integrity is compromised.

Even if we specify that we want to open a file for reading there remain some issues:

```
#!/usr/bin/perl -w
# DON'T DO THIS
use strict;
use CGI;

my $filename = CGI->param('filename');

chomp $filename;
open(FILE, "<$filename") or die "Failed to open $filename for reading: $!";
```

This code still allows the user to open any file on the operating system they want rather than the files in the current working directory. The user can do this by passing in a value such as:

`../../etc/passwd` as their filename. There are other tricks a clever attacker can use, which we'll discuss later on the chapter on opening files.

The fundamental issue

The issues we're seeing in the above examples are unintended consequences. These occur for two reasons: we're not validating the user's input, and we're using functions that can allow the shell to be invoked.

Perl has a three-argument version of `open`, and multi-argument versions of `system` and `exec`. These alternate versions are stricter in how they process arguments, but we can still encounter serious problems if we just throw data at them without thinking about it first.

Good programming practice says that we should always validate our program's input. Values should be checked to be within bounds and should be similar in form to what we expect. However, retrospectively applying this good programming practice can be both tedious and error-prone. User input can be left unchecked by accident resulting in cases like the above.

Fortunately, Perl provides us with a useful tool which keeps track of unchecked user data, called taint mode. Any data that originates from outside the program, such as environment variables, user input, command line arguments, or data from from files is considered *tainted*. If this tainted data is passed to `system`, `exec`, `open` and many other functions, Perl throws an exception, which (if uncaught) will cause your program to die with an error. This helps us avoid the aforementioned unintended consequences. Taint is discussed in detail in the remainder of this chapter.

What is taint mode?

Perl has a special mode of operation known as *taint mode*. When in this mode, Perl will consider any data from the user, environment, or external sources (such as files) to be considered *tainted*, and unsuitable for certain operations. The philosophy behind taint is as follows:

> You may not use data derived from outside your program to affect something else outside your program -- at least, not by accident.
>
> -- the `perlsec` documentation

Tainted data is communicable. This means that the result of any expression containing tainted data is also considered tainted. Just because the data has been reversed, mangled, and converted into a uuencoded string doesn't mean it's considered to be *clean*. Tainting is applied at the *scalar* level, meaning that an array or hash may contain some elements that are tainted, and some that are clean.

Data that is considered tainted cannot be used to do any of the following:

- Execute system commands
- Modify files
- Modify directories
- Modify processes
- Invoke any shell
- Perform a match in a regular expression using the `(?{ ... })` construct
- Execute code using string eval

Trying to use tainted data for any of these operations results in an exception:

```
Insecure dependency in open while running with -T switch at insecure.pl line 7.
```

The list of functions in *Programming Perl, 3rd Ed* (Larry Wall et al), chapter 29, provide notes as to which functions return tainted data, and which functions will refuse to accept tainted data.

Enabling taint mode

Taint checks are automatically enabled when Perl detects that it's running with differing real and effective user or group ids -- which most commonly occurs when the program is running setid.

Taint mode can also be explicitly turned on by using the `-T` switch on the shebang line or command line.

```
#!/usr/bin/perl -wT             # Taint mode is enabled
```

It's highly recommended that taint mode be enabled for any program that's running on behalf of someone else, such as a CGI script or a network daemon. It's also a good idea if you're using data from a source you don't completely trust, such as processing e-mail or externally generated documents. Once taint checks are enabled, they cannot be turned off.

Using taint checks are often a good idea even when we're not in a security-sensitive context. This is because it strongly encourages the good programming (and security) practice of checking your arguments before using them.

Taint and the environment

Even with clean data, there are many environment variables that can result in your program doing something unexpected when interacting with external commands. For example, an attacker running a setuid program could manipulate the PATH variable to try and have their own code executed, rather than the system defaults.

In order to run an external command, Perl requires that the following environment variables are set to untainted values:

- `PATH` - the directories searched when finding external executables.
- `IFS` - Internal Field Separator; the characters used for word splitting after expansion.
- `CDPATH` - a set of paths first searched by **cd** when changing directory with a relative path.
- `ENV` - the location of a file containing commands to execute upon shell invocation.
- `BASH_ENV` - similar to `ENV` but only comes into effect when Bash is started non-interactively (eg. to run a shell script).
- `PERL5SHELL` - (Windows only), the shell Perl should use instead of the default `command.com` or `cmd.exe` under Windows.
- `DCL$PATH` - (VMS only) additional directories to search under VMS.
- `TERM` - The terminal type being used. Perl does not insist on this being untainted, only that it contains only letters, numbers, underscores, dashes and colons.

Even your command is not using the shell, Perl will err on the side of caution and refuse to run it if any of the above variables are tainted (except where otherwise noted). Trying to run with a tainted environment will produce a message similar to:

```
Insecure $ENV{PATH} while running with -T switch at insecure.pl line 4.
```

The best way to avoid encountering these errors is to set these values yourself. For the most part this means the start of your script will look similar to:

```
#!/usr/bin/perl -wT
use strict;

delete @ENV{qw(IFS CDPATH ENV BASH_ENV PERL5SHELL)};
$ENV{PATH} = "/usr/bin/:/usr/local/bin";
```

Before Perl 5.10.0 and 5.8.8, the `PERL5SHELL` environment variable was not checked for tainted data. You should *always* manually check (or preferably clear) the `PERL5SHELL` enviornment on Windows systems.

PERL5LIB, PERLLIB, PERL5OPT

The `PERL5LIB` and `PERLLIB` environment variables can be set to tell the perl interpreter where to look for Perl modules (before it looks in the standard library and current directory). These can be used instead of including `use lib "path/to/modules"` or similar techniques in your code.

The `PERL5OPT` environment variable can be set to tell the perl interpreter which command-line options to run with. Only the `-[DIMUdmtw]` switches are allowed in `PERL5OPT`.

These environment variables are silently ignored by Perl when taint checking is in effect.

Setting `PERL5OPT` to `-T` will enable taint-mode but ignore all other switches. `-T` anywhere else in the `PERL5OPT` line is silently ignored.

Using taint and laundering data

Not allowing data from external sources to be used to alter things outside your program is a great protection against malicious attacks or accidental mistakes. However, it's also very difficult to write a useful program under such restrictions, so it's important to have a way to launder or clean data. There's only one built-in way of doing this, and that's by using a regular expression.

Perl considers data extracted using capturing parentheses to be 'clean'. This data can either be read from the match variables `$1, $2, $3 ...`, or out of the expression in list context, like so:

```
my ($user,$host) = /^(\w+)\s+([\w-]+)$/;
```

Passing your data through a regular expression does not guarantee that it's safe to use. However it does at least force you to think about it first. There's nothing to stop you from bulk-untainting data with an expression like `/(.*)/s`, but doing so is extremely trusting of your data, and certainly not recommended.

A commonly seen mistake is to forget to check that the regular expression match succeeded. Where the match variables `$1, $2, $3...` are concerned, this could result in a previous match result being used.

It is recommended that you *always* untaint data by using your regular expression in list context, as it guarantees that undefined values will be used on failure.

Regular expressions are notoriously difficult to write correctly, and this is even more true when working in a security-sensitive context. As such, here are a few tips:

- Try to write an expression that matches the entire input string, using either `^` and `$`, or preferably `\A` and `\Z`. Regular expressions try *really hard* to match, so without matching the entire string it's possible to match a substring contained in bad input by accident.
- It's *much* safer to specify a list of characters that are allowed, rather than bad characters that should be filtered out. Adding another character to an allowed list is relatively painless, whereas failing to detect a dangerous meta-character can be disastrous.

Examples using taint

At the start of this chapter we discussed two examples of how user data is often provided to the file system unchecked. Here we show how we can untaint the user data to ensure that we're only getting the kind of data we want to use.

Sending user data to an application

```
#!/usr/bin/perl -wT
use strict;
use CGI;

delete @ENV{qw(IFS CDPATH ENV BASH_ENV PERL5SHELL)};     # clean ENV variables
$ENV{PATH} = "/bin:/usr/bin";                            # set a safe PATH

my $username = CGI->param("username");
die "Missing username\n" unless $username;

# Check that the username looks like the kind of username we want.
# Usernames must start with a letter and can contain letters, numbers and
# underscores.

my ($safe_user) = ( $username =~ /^([A-Za-z]\w*)$/ );

print "Content-type: text/plain\n\n";

if ($safe_user) {
    system("/usr/bin/finger $username");
} else {
    print "Sorry, that’s not a valid username\n";
}
```

Unlike the example at the start of this chapter a malicious (or merely curious) user cannot use this program to run arbitrary programs on our system.

Using user data in a call to open

```
#!/usr/bin/perl -wT
use strict;

delete @ENV{qw(IFS CDPATH ENV BASH_ENV PERL5SHELL)};  # clean ENV variables
$ENV{PATH} = "/bin:/usr/bin";                         # set a safe PATH
```

```
my $filename = <STDIN>;
chomp $filename;

# Check that the filename looks like the kind of filename we want.
# Filenames must start with a number or a letter and can contain letters,
# numbers, _, . and -

unless( $filename =~ /^([A-Za-z0-9][\w.-]+)$/ ) {
        die "Invalid filename: $filename";
}
$filename = $1;

open(FILE, "<", "/home/test/$filename")
        or die "Failed to open /home/test/$filename for reading: $!";
```

As in the previous example, our untainting of the user’s data ensures that the user cannot back-track up the file system tree and access files outside of `/home/test`.

Capturing without laundering

Sometimes we wish to extract data using a regular expression, but we don’t wish to untaint it. Perl provides us with the `use re 'taint'` pragma that allows us to disable the untainting effect of regular expressions. The `use re 'taint'` pragma is lexically scoped, meaning it lasts from where it’s invoked to the end of the enclosing block, file, or package.

It’s considered good practice to `use re 'taint'` at the start of any file or package, then then use `no re 'taint'` to explicitly mark sections of code where untainting is to occur. This makes the auditing of untainting expressions much easier, and also avoids the problems of accidental untainting of data.

```
#!/usr/bin/perl -wT
use strict;
use re 'taint';

delete @ENV{qw(IFS CDPATH ENV BASH_ENV PERL5SHELL)};
$ENV{PATH} = "/bin:/usr/bin";

# Step 1 -- Read the contents of a legacy configuration file consisting
#           of 'key = value' pairs, one per line.

my $CONFIG = "/etc/example.conf";

my $conf_fh;
open($conf_fh,"<",$CONFIG) or die "Cannot open $CONFIG - $!\n";

my %config;
while (<$conf_fh>) {
        # Naively grab a key and value pair.  Note that this does not
        # untaint these values, as "use re 'taint'" is in effect.

        # However you should note that our keys *do* become
        # untainted.  Perl hash keys are not true scalars, and
        # 'forget' their taintedness.

        my ($key, $value) = /^(\S+)\s*=\s*(\S+)/;
        $config{$key} = $value;
}

# The values in our %config hash are still tainted.  This is a good
# thing, as our regular expression did not do very much in the way of
# testing for bad values.  If there are variables which we wish to use
# in dangerous operations, then we can explicitly untaint them.

{
        # Prepare to untaint some of our variables.
        no 're' taint;

        # Our phone number can only consist of digit characters.
        $config{phone} =~ /^(\d+)$/ or die "Bad phone number in config\n";

        $config{phone} = $1;    # This value is now untainted.
}

# Then later...

if ($DISASTER) {
        system("/usr/sbin/send_page",$config{phone},"It's a disaster!");
}
```

Using a regular expression is not the only way to untaint data. Because hash keys are just strings, and not *true* scalars, they "forget" they were derived from tainted data. You should be mindful of this whenever you're working with hashes in taint mode.

Taint and locales

When using locales in Perl, the result of any operation that uses locale information is considered tainted. One item of particular note is that since character classes like `\w` are locale dependent, you must explicitly define character classes (eg, `[A-Za-z0-9_]`) in regular expressions where you want untainting to occur.

When Perl's locale features are not in effect, character classes and other functions work as expected.

Taint traps

Taint mode is an excellent safety belt, but it should be kept in mind that it doesn't stop you from making mistakes, it only provides a chance that Perl will catch the problem before doing something foolish.

One thing to be particularly careful of are things which taint mode does *not* protect you against. This section discusses these in more detail.

Output functions

Perl does not do any sort of checks when data is written for output, either using `print`, `syswrite`, or any other output function.

The primary thing to note here is that while taint affords a level of protection against *directly* using untrusted data for dangerous operations, it does nothing to protect against the recording or outputting of this data.

A taint-enabled program could read tainted data and store that in a file, or pass it to another program which does not adequately valid its input. Likewise, taint does nothing to prevent the accidental disclosure of sensitive information.

Perl's DBI module provides some excellent mechanisms to provide taint checks when data is retrieved from or sent to your database. This is discussed in more detail in the chapter on database security.

Opening a file for reading

Due to historical reasons, Perl does not perform taint checks when opening a file for reading. There are of course obvious risks associated with this, we almost certainly do not wish to allow opening of the file `../../../../etc/shadow`, however even if our input is tainted, Perl will still allow this operation to occur.

If a program uses the two-argument version of `open`, it may also be possible for an attacker to gain access to an existing file, using the notation `<&=N` (where `N` is a file descriptor number or filehandle) to open an existing filehandle for reading.

The following code demonstrates the use of this syntax to gain access to an existing filehandle.

```
#!/usr/bin/perl -wT
# DON'T DO THIS
use strict;

# $filename will be the first argument on the command-line.

my $filename = shift or die "No argument supplied\n";

open(FILE, "<$filename") or die "Cannot open $filename: $!\n";

while (<FILE>) {
        print;
}
```

When an argument of `&=0` is supplied (which you may need to escape from your shell), the above program will echo any input from `STDIN`. If the program had other filehandles open for reading, the same syntax could be used to gain access to these resources.

An effective solution to prevent this mistake is to always use the three-argument version of open:

```
        open(FH,"<",$filename) or die ...;
```

The three-argument open always treats the filename *literally*, and is discussed in greater detail in the chapter on opening files securely.

In perl 5.6.1 and earlier, opening a file for reading and writing (using the mode `+<`) didn't die when using a tainted filename. This is because Perl considered this as opening the file for reading, even though it was then possible to also write to the file.

Invoking a subroutine via symbolic reference

Unless `use strict` or `use strict 'refs'` is in effect, Perl allows a simple string to be used as a *symbolic reference* to a subroutine name. This has limited usefulness in common code, and if you're using `strict` then Perl will disallow this practice.

If this functionality is allowed, you should be aware that Perl does not check to ensure that the symbolic reference is untainted. The following code operates with no tainting exceptions:

```
#!/usr/bin/perl -wT
# DON'T DO THIS

my $sub = shift;

&$sub();
```

The above code would allow the user to invoke any defined subroutine, including those in other packages (if the input were in the form '`Package::subroutine`'). This is a dangerous and error-prone practice regardless of your security considerations, and unless there's a requirement to allow unfettered access to your entire running program, a hash listing allowable subroutines is a preferable alternative.

```
#!/usr/bin/perl -wT
use strict;

my %allowed_subs = (
        order   => \&order,
        test    => \&Test::order,
        print   => \&print_results,
);

my $sub = shift;

exists( $allowed_subs{$sub} ) or die "Action not allowed\n";

$allowed_subs{$sub}->();
```

The use of a hash above provides a well-defined set of subroutines that can be called. It also provides the flexibility to insert anonymous subroutines into the hash, which may prove to be convenient.

If your program *does* have the requirement to provide unfettered access to untrusted users, then your code can be made more secure by removing the CPU from any machine upon which it is to run.

Method calls

Perl *always* allows a symbolic name to be used when making method calls. This is regardless of whether the symbolic name is tainted. An example is:

```
My::Package->$method();    # Doesn't complain if $method is tainted.

$my_object->$method();     # No checks against $method here, either.
```

Of special note is that it is possible to call methods from *other classes* using this technique.

Perl does not allow symbolic references to built-in *functions*, only to subroutines.

In a security-sensitive context, careful consideration should always be made before using any symbolic subroutine or method calls.

Calling dbmopen

The `dbmopen` function has been largely superseded by the `tie` function, and the `DB_File` group of modules.

It should be noted that `dbmopen` is not documented to check its arguments for tainted data. As of Perl 5.8.8 and 5.10.0, taint checks *do* occur, but it is unclear if this has always been the case.

Use of `dbmopen` should be treated with caution, as unchecked data can be used to open, create, or overwrite a file.

Using DB_File

It should be noted that `DB_File` does not do taint checking on any of its arguments. Tainted data may be used in creating and opening databases. As this data does not use the shell the primary security dangers are as follows:

- Access to databases other than those intended
- Creation of new databases
- Overwriting of existing files which were not databases

More information about filesystem abuse can be found on the chapter on opening files securely.

Multi-argument system or exec

Older versions of Perl would allow tainted data to be passed to a *multi-argument* call to `system` or `exec`. Perl 5.8.0 deprecated this behaviour, producing a warning, and more modern and future versions of Perl will explicitly forbid tainted arguments being passed to any form of `system` or `exec`.

Taint and unicode

In versions of Perl prior to 5.8.4, Perl could sometimes forget the UTF8 status of tainted strings. If you are using both unicode and taint, then it’s recommended that you use Perl 5.8.5 or later.

Improvements on taint

Sometimes it would be nice to know which values are tainted. This would allow us to test for tainted data in our own code when required. Fortunately, the `Scalar::Util` module comes standard with Perl and provides a `tainted` function. This function returns true if the data is tainted and false otherwise.

We can use this function to write a wrapper around the `open` function to ensure that tainted filenames can never be used:

```
#!/usr/bin/perl -wT
# An example of writing a wrapper around open to always refuse tainted
# filenames.
use strict;
use Scalar::Util qw/tainted/;
use Carp;
```

```
my $filename = <STDIN>;
chomp $filename;

# We should untaint $filename here.

my_open(my $filehandle, "<", $filename) or die "Failed to open $filename: $!";

# do something with the data from our file
my @lines = <$filehandle>;

#######################

sub my_open {

        # Comaplin if any of our arguments are tainted.

        foreach my $arg (@_) {
                croak "Insecure argument passed to my_open" if tainted($arg);
        }

        my ($filehandle, $mode, $filename) = @_;

        return open($_[0], $mode, $filename);
}
```

In Perl 5.8.x the special variable `${^TAINT}` can be used to determine is perl is running in taint mode. In Perl 5.6.0 there is no easily reliable way to detect if the program is using taint mode without interacting with the operating system in some way. The common practice of examining `$^X` cannot be recommended as it is possible to replace `$^X` with untainted data.

To learn more about `Scalar::Util` read **perldoc Scalar::Util**.

Chapter summary

- Many programs pass user data directly to the system without first validating it. This can lead to unintentional side-effects if the user enters specially crafted data.
- Using taint forces the programmer to process the data before using it to open files, execute system commands and invoke the shell.
- To enable taint mode use the `-T` switch.
- To untaint user data it must be captured through a regular expression.
- To capture data through a regular expression without untainting it, use the `re 'taint'` pragma.
- Perl's regular expression character classes are locale dependent.
- Taint mode does not do any sort of checks to data written as output.
- Tainted data can still be used in filenames of files to be opened for reading.
- Perl does not check to ensure that symbolic references are untainted.
- `dbmopen` does not necessarily check that its arguments are untainted.

- Newer version of Perl will explicitly forbid tainted arguments being passed to any form of `system` or `exec`.
- The `Scalar::Util` module provides a function (`tainted`) which can be used to check if a value is tainted or not.

Chapter 5. Opening files

In this chapter...

Opening files is an everyday operation, and Perl makes this operation relatively easy. However, there are a number of considerations that should be taken into account when working with files, particularly when creating new ones, or opening files for writing. We cover these issues in this chapter.

Using the wrong open

One of the most common security mistakes in Perl code is not using `open` safely. For a new programmer wanting to open a file, **perldoc -f open** is a daunting amount of text.

Maybe they turn to the Camel book, or read **perldoc perlopentut**, or use their favourite search engine to find a solution. Either way the first code they find will tend to look like this:

```
my $filename = get_input_from_user();

# Open the file for reading
open(INFO, $filename) or die "Can’t open $filename: $!";
```

The *wrong* open.

The 2-argument version of open

If the type of operation (reading, writing, appending) is not given to an `open` call, then Perl will open the file for reading. That’s why the above example works.

However, when programming with security in mind, this feature should never be used. The two argument open is arguably the most difficult to understand function in perl, and unless your arguments have been checked with exterme prejudice, you may find your file opening in a very unexpected way.

Here is an example:

```
# DON’T DO THIS

open(FILE,$filename) or die "Could not open $filename for reading - $!\n";
```

Even though the programmer wants to open the file for reading, if the `$filename` variable were to begin with a >, then we would open the file for writing. If the `$filename` were to begin or end with a pipe (`|`) then we’d be opening a *command* for execution!

Perl’s regular 2-argument open is very magical, meaning it performs a lot of extra processing to its arguments, and can be used to do a great many things. In fact, the documentation for `open` that accompanies Perl 5.8.8 is no less than 374 lines long -- the largest documentation set for any Perl function. The two-argument open is too magical to use safely in secure programs.

Many of the extra features provided by Perl’s two-argument `open` are not widely known. For example, the special construct <`&=N` (where `N` is a number, or a name of a filehandle) allows an existing open filehandle to be duplicated and returned. In a similar vein, Perl’s two-argument `open` considers the special filename '-' to refer to either <STDIN> or <STDOUT> depending upon if it is opened for reading or writing.

While these features may be useful in day-to-day programming, they are extremely undesirable when working with untrusted user input. It can be possible to duplicate an existing stream of data when the programmer intends to open a file.

A better open - the 3-argument version

To avoid these problems, it is recommended that the three argument version of `open` be used, like so:

```
open(FILE, '<', $filename) or die "Could not open $filename - $!\n";
```

The three argument version of `open` requires us to explicitly provide the operation we wish to perform, in this case opening the file for reading. Furthermore, the three argument version of `open` treats the filename *literally*. Unlike the two argument version of `open`, any special characters in the filename, including leading and trailing spaces, are considered to be part of the filename.

To use a pipe with this version of `open` the following modes can be used:

- `-|` (minus pipe): interpret the filename as a command which perl will be reading from.
- `|-` (pipe minus): interpret the filename as a command to which perl will be writing to.

```
use autodie qw(open);

# Rot13 our text when we print it out
open(my $rot13_fh, '|-', '/usr/games/rot13');

print {$rot13_fh} "This text will have each character rotated by 13 places\n";

# Run strings over the kernel and then read a line from it.
open(my $kernel_fh, '-|', 'strings /vmlinuz');

print scalar(<$kernel_fh>);
```

Further problems with open

When opening a file for writing in a security-sensitive context, there are a few other things we need to consider.

Symbolic links

Most Unix-like operating systems have a concept of a symbolic link. These are small files which point or link to other files or directories. When you open a symbolic link, it's the file that the link points to that gets opened.

Symbolic links are an extremely useful feature of the file system, but their existence also presents some risks, and these come in the form of *symbolic link attacks*. Put simply, a *symbolic link attack* involves placing a symbolic link in a place where an attacker knows or suspects a file will be opened, and pointing that symbolic link to another location of their own choosing.

When we’re opening a file, we need to make sure that it hasn’t been substituted for a symlink to a file we don’t want to be opening. If we’re not careful we could clobber an existing file or print out information we didn’t intend. Let’s see an example attack:

> Old versions of **lpr**, a command on Unix systems to schedule files for printing, used a spool directory for storing files to be printed. The **lpr** command would only allow one thousand files in its spool directory, and would clobber and replace the contents of older files without first removing the original. **lpr** would commonly run with root privileges.
>
> The older versions of **lpr** had another ’feature’ as well, which was the ability to place a symbolic link to a file to be printed into the spool directory. This was very useful with large files or when the disk was getting full, as it didn’t require large amounts of copying or space. **lpr** would ’do the right thing’ and check that the user was submitting a file which they had read access to before creating the link. However this did not prevent an attack from being possible. In fact, two separate attacks could be performed using **lpr**’s handling of symlinks.
>
> Firstly, any file on the system could be printed. An attacker would submit one of their own files for printing. **lpr** would check the file was readable by the user, and create a symlink to this file. However if the original file was removed and then replaced with another symlink before the printing occurred, then the linked file would be printed. This new link could point to any file on the system, including ones that the attacker could not read.
>
> A second, and more dangerous attack would begin the same way, but involved scheduling 999 dummy files after scheduling the first for printing, and then switching the original destination file with a symlink. Because the **lpr** would overwrite the contents of its spool files when the spool was full, the 1,000th file printed would *overwrite* the target file pointed to by the attacker. Once again, this could be any file on the system.
>
> Both of these attacks required files to remain in the spool directory long enough to be abused. This reliance upon timing is known as a *race condition*, as the end results depend upon a race depending upon which events happen first. In this particular example the race condition was very easy to exploit, simply printing a large file (or jamming the printer, if physical access was possible), would ensure that the no other files would be removed from the spool for some time.
>
> These bugs, and exploits to abuse them, were found in 1991, and affected a variety of operating systems, including SunOS, BSD, A/UX, and most other systems using the BSD **lpr** subsystem.

Most implementations of **lpr** no longer provide this "functionality".

If you wish to ensure your code is as secure as possible it’s important to ensure that you are not chasing a symlink when you open a file unless you expect to be doing so. If you do wish to handle symlinks, it’s important to ensure that untrusted data can’t be used to trick your system.

The obvious (but wrong) way to ensure that you don’t chase a symlink is the following:

```
# DON’T DO THIS!
if(-l $file) {
        die "Cannot open $file, it is a symbolic link.";
}

open( FILE, ">", $file ) or die "Failed to open $file: $!";
```

However this suffers from a race condition which we’ll talk about next.

Avoiding race conditions

A race condition exists when the result of several interrelated events depends on the ordering of those events, but that order cannot be guaranteed due to nondeterministic timing effects.

Programming Perl -- Larry Wall, Tom Christiansen and Jon Orwant

In the last section we saw an example of when a *race condition* could occur when we used a file-test operator prior to opening the file:

```
# DON’T DO THIS!
if(-l $file) {
        die "Cannot open $file it is a symbolic link.";
}

open( FILE, ">", $file ) or die "Failed to open $file: $!";
```

In this case the programmer is *assuming* that the file in `$file` will not change between the test and the file open. However on a multi-tasking machine this is not guaranteed. A lucky attacker could change the file between the test and the call to `open`. It’s a *race* to see who gets to the file first.

To solve this race condition we need to make sure nothing can occur between the file test and open. That is, we wish to be *atomic*. We need to ensure that our open function does the file test for us.

Perl’s `open` isn’t the solution here. Instead, we have to get closer to our operating system by using the `sysopen` function.

An even better open - using sysopen

Perl’s low-level open function is called `sysopen`, and it provides access to an interface very similar to C’s `open(2)` call. Knowledge of the `open(2)` system call is very handy when dealing with `sysopen`.

The great advantage of using `sysopen` is it allows for the programmer to explicitly detail how the file should be opened. Exactly one of the following modes must be used:

- `O_RDONLY`: open the file for reading;
- `O_WRONLY`: open the file for writing, does not truncate the file;
- `O_RDWR`: open the file for both reading and writing;

Additional flags can also be passed. Some of the commonly seen flags are documented below:

Table 5-1. Optional flags for use with sysopen

Flag	Meaning
O_CREAT	Create the file if it does not already exist. The owner of the file will be set to the effective user-id of the process. The group-id will be set to either the effective group-id of the process, or the group-id of the parent directory, depending upon the mode of the parent directory. You should not use this flag when opening a file for `O_RDONLY`.

Flag	Meaning
O_EXCL	When used with `O_CREAT` this causes the call to `sysopen` to fail if the file already exists. This also causes `sysopen` to fail if the filename is that of a symlink, regardless of where the symlink is pointing. When not used with `O_CREAT`, the behaviour is undefined, but usually has no effect. This is an important flag for secure programs, as it allows for a file to be created without the possibility of damaging an existing file, or accidently chasing a symlink. `O_EXCL` may not work under some file systems, particularly network file systems such as NFS.
O_TRUNC	If the file already exists and is a regular file, then it will be truncated. This should not be used when opening a file as `O_RDONLY`, as its behaviour is not well defined. The file will not be truncated if `O_CREAT|O_EXCL` is specified -- an existing file will not be harmed. Using `O_TRUNC` with `O_CREAT|O_EXCL` is redundant, as a freshly created file will always have a zero length.
O_APPEND	The file is opened in append mode. Before each write, the file pointer is positioned at the end of the file. While this is an atomic operation (with some limitations) on local files, it can result in in race conditions and corruption when multiple processes are accessing a file on a network file system such as NFS which does not natively support appending.
O_SYNC	Open the file for synchronous I/O. Writing to the file will cause your process to block until the data has been physically written to disk. This flag is unlikely to be honoured on a network file system.
O_NOFOLLOW	This causes the operation to fail if the file specified is a symbolic link. This is only supported on some operating systems (most notably Linux and FreeBSD) and is *not* considered portable. The combination of `O_CREAT|O_EXCL` should be used instead when creating files.

Examples of using sysopen

```
use Fcntl;
use autodie qw(sysopen);

# Opening file for reading
sysopen(my $fh, $infile, O_RDONLY);

# Opening a file for writing, truncating file if it exists
sysopen(my $fh2, $outfile, O_WRONLY|O_CREAT|O_TRUNC);
```

Creating files safely

One of the risks when working with privileges is clobbering or unintentionally writing to an existing file. This could be a harmless mistake, or it could be part of something more sinister, such as a *symlink attack* described earlier in this chapter.

A common mistake is to use one of Perl's file-test operators to check a file before opening it. Here's the (incorrect) example we've seen that tries to ensure we don't chase symlinks:

```
# DON'T DO THIS!
if(-l $file) {
        die "Cannot open $file it is a symbolic link.";
}

open( FILE, ">", $file ) or die "Failed to open $file: $!";
```

Unfortunately, as we've discussed, this code contains a *race condition*. The file on the file system may not be a symbolic link (or may not exist) when the file-test is made, but there's an opportunity for the file to change between that file-test and the call to `open`. Worse still, this sort of problem is often difficult to catch in testing.

The solution to opening files safely is to use an atomic operation, which `sysopen` provides. In the case where we wish to create a new file safely, we can use the following code:

```
use Fcntl;
use autodie qw(sysopen);

# Open a new file for writing.
sysopen(FH, $file, O_WRONLY|O_CREAT|O_EXCL);
```

The combination of `O_CREAT|O_EXCL` specifies that we wish to create a new file rather than open an existing one. If the file already exists, or is a symlink, then the call to `sysopen` will fail.

New file permissions

In most circumstances we wish the user running our process to specify the permissions used when creating files. A user may be working as part of a team, and may wish for their new files to be readable and writable by their group. In these common circumstances, Perl 'does the right thing' and respects the user's `umask`.

However, there are also circumstances where we want to create files that have more restrictive permissions. Encryption keys, personal information, passwords (encrypted or plain), and other sensitive information falls into this category. To do this, we can explicitly pass a set of permission bits to `sysopen`.

```
#!/usr/bin/perl -wT
use strict;
use Fcntl;
use autodie qw(sysopen);

# Setup a basic set of accounts for our new application.

# Application-specific passwd file.
my $file = "/usr/local/etc/application/passwd";

# Note the extra argument to sysopen.  Without this, the permissions
# use the default mode of 0666, which then may be modified by our
# umask.
```

```
sysopen( my $fh, $filename, O_CREAT|O_WRONLY|O_EXCL, 0600 ) ;
```

Setting our file permissions at file creation is preferred over trying to change them afterwards using `chmod`, as it avoids the race condition where another process may open the file before the permissions are changed.

Writing to files safely

When writing to files that already exist we need to consider where they are located. If the file is located in a directory which other users have access to, such as `/tmp` then we should be very careful.

The most common symlink attacks are made against programs that use predictable filenames in world-writable directories, especially `/tmp`. By placing a symlink where a file is going to be created, an attacker could trick a process into reading or writing a file it didn't mean to.

If you want to create a new file, then `O_CREAT|O_EXCL` is the way to go about it. If you wish to write to an existing file, and your system supports it, you can use `O_NOFOLLOW` to ensure we're not chasing a symlink. However this directive is not portable across all systems. A better methodogly is to drop privileges (which is discussed later in this course), or to use temporary files (which is discussed later in this chapter).

To open an existing file for writing, without clobbering the contents, we can do the following:

```
use Fcntl;
use autodie qw(sysopen);

# Open an existing file for writing
sysopen(my $fh, $filename, O_WRONLY);
```

This will open the file for writing, and any changes made will overwrite the existing content. This is ideal if we wish to manully seek to a fixed-length record or overwrite, or if we wish to lock the file before truncating and re-writing it.

If we wish to open an existing file for writing and clobber the contents immediately, we can use the following code:

```
use Fcntl;
use autodie qw(sysopen);

# Open and clobber an existing file.
sysopen(my $fh, $filehandle, O_WRONLY|O_TRUNC);
```

Because `O_CREAT` is not specified in these examples, they cannot be used to create new files, only open existing ones.

Safely opening temporary files

Opening a temporary file is a very common operation. In line with Perl's design of making "simple jobs easy, hard jobs possible", opening a temporary file securely in Perl is a very easy task.

In many situations, there's no need to have a temporary file with an actual *name*. If a file is temporary, and is only to be manipulated by the current process and its children, then it's possible to use that file without referring to the file system at all.

The lack of name has numerous advantages. The file is automatically cleaned up when the last filehandle to it is closed. It’s also possible to keep very tight controls on what can access that file, as it’s not accessible via the regular file system.

Creating an anonymous file in Perl version 5.8.0 and later is a very simple operation using `open`:

```
use autodie qw(open);

open(my $tmp_fh,"+>",undef);
```

Using an undefined filename indicates to Perl that an anonymous temporary file is desired. This can be written to and read from just like a normal file, however you will need to use the `seek()` function to read the contents of the file once you’ve written to it.

If you need more portability or control, the `File::Temp` module can be used to safely create temporary files:

```
use File::Temp ’tempfile’;

my $fh = tempfile() or die "Could not open temp file - $!\n";
```

The `File::Temp` module provides an excellent cross-platform interface for working with temporary files, and contains a number of additional safety checks to ensure that files are created in a secure fashion. The `File::Temp` module also provides ways of securely creating temporary directories, and safely unlinking (deleting) files.

Sharing anonymous files with other processes

On most unix systems, it is impossible to guarantee that a filehandle is associated with a given filename. After a file has been opened, the original filename can be moved, renamed, or even deleted! As a result, the recommended way of working with temporary files is to create anonymous files and use those. By never using, or even knowing, the file’s name, numerous race conditions that abuse the filesystem can be avoided. However, there is a major drawback to this technique: how do you give access to your file to another process, when that file does not have a name?

Through fork and exec

On systems where Perl can execute a `fork()` file-descriptors will be shared with child processes. This means that if you’re simply forking a child process there is no special work that needs to be done to share a filehandle to an anonymous file. However, the problem occurs if you wish to start a new process, including using `exec()` after a fork.

Most commands expect to start with only three open filehandles, namely `STDIN`, `STDOUT` and `STDERR`. These always remain open over an `exec()`, so if we’re intending for the temporary file to be attached to one of these handles, the problem becomes simple to fix. Simply `fork`, adjust filehandles, and then `exec`:

```
#!/usr/bin/perl -wT
use strict;
use File::Temp ’tmpfile’;
use Fcntl ’SEEK_SET’;

use autodie qw(seek open);

# Set our path to make sure its safe
$ENV{PATH}="/bin";
```

```
my $cmd = "cat";
my @args = qw(/etc/motd);

# Create a temporary file
my $tmpfile = tmpfile() or die "Could not open temporary file";

# fork the process
my $pid = fork();

# Instead of the following, we could have also used autodie qw(fork);
defined($pid) or die "Could not fork child - $!\n";

if ($pid) {
        # Parent, wait for the child to finish.
        $pid == wait() or die "Where did my child go?\n";

        # Print out what was written
        seek($tmpfile,0,SEEK_SET);
        while (<$tmpfile>) { print; }
} else {
        # Child process...

        # Get our file’s number
        my $fileno = fileno($tmpfile);

        # Redirect STDOUT to the temporary file
        # We use ’+>’ to match the original file mode.
        open(STDOUT, "+>&$fileno");

        # Do things and then finish
        exec($cmd,@args);
        die "Could not exec - $!\n";    # Should never be reached.
}
```

We use `fileno` to obtain the file number for using in `open`’s special ’duplicate filehandle’ mode. It’s also important that we `seek` back to the start of the file after our child process has finished, so we can actually read our data.

The above example could have also used `pipe()` for communication between parent and child.

When the process wants an actual file

Sometimes we wish to share an anonymous file with a process that actually wants to use the file as a real file with a real name, rather being a new place to send `STDIN/STDOUT/STDERR`. An example of this may be an editor (such as **vi**), or any other program that expects a filename on the command-line. You would this this would be impossible to accomodate with anonymous files, but it’s actually quite easy.

If your system supports the special files `/dev/fd/X` to refer to an existing filehandle in the current process, then these can be used in lieu of a real filename. However, we need to be careful to ensure that the file actually remains open across a call to `exec`, and to do this we need to use the low-level `fcntl` call.

Here’s an example of using **vi** to edit an anonymous temporary file:

```
#!/usr/bin/perl -wT
use strict;
use File::Temp ’tmpfile’;
use Fcntl qw(F_SETFD SEEK_SET);
use autodie qw(fcntl seek system);

# Set our path to make sure its safe
$ENV{PATH} = "/usr/bin:/bin";

# Create a temporary file
my $fh = tmpfile() or die "Could not open temporary file - $!\n";

# Make sure this file will not be closed when we execute another command
fcntl($fh,F_SETFD,0);

# Get our file number
my $fileno = fileno($fh);

# Call vi and give it our temporary file; autodie will generate
# an error if vi dumps core, is zapped by a signal, or exits oddly.
system("/usr/bin/vi", "/dev/fd/$fileno");

# Seek to the start of the file...
seek($fh,0,SEEK_SET);

# Here’s what was written.
while (<$fh>) { print ; }
```

The `File::Temp` module always creates files with the close-on-exec flag set. As such, it’s essential that this be unset before running our command.

File locking

Race conditions can and most commonly do occur without any sort of malicious intent, and are usually one of the most difficult types of bugs to diagnose and fix. This section deals with how to avoid conflicts when multiple processes need to deal with shared resources, particularly files.

A very common race condition occurs when multiple processes are trying to access a common file. For example if one process is writing to a file while another is reading from it, then it is uncertain whether the data read will be the old data, the new data, a mix of the two, or possibly none at all!

This race condition can generally be fixed by having the processes lock the file before accessing it. To do this, we can use Perl’s in-built file locking features.

`flock` is Perl’s portable file-locking mechanism, and works on most operating systems (and produces a fatal error on those which it does not). The locks set by `flock` are advisory only, which means that a process that chooses not to use `flock` can (and will) ignore any locks in place. `flock` can only lock entire files, not individual records. Depending upon your setup, `flock` may or may not work over NFS.

```
# using flock
use Fcntl ’:flock’;             # import LOCK_* constants

flock(FILEHANDLE, LOCK_EX);     # exclusive (write) access
flock(FILEHANDLE, LOCK_SH);     # shared (read-only) access
```

As `flock` only works on *filehandles*, instead of filenames, you have to open the file first *before* you try to lock it. It’s important to make sure that you open the file for writing, if you intend to write to it, and that you don’t clobber the contents of the file when doing so. This is a good use of `+<`. Closing a locked file releases any locks the process holds upon it. This is good because it means that if your process exits unexpectedly all locks it held are released and other processes may then go forward with their locks.

In the following example, we're locking a file before re-writing it. The exclusive lock stops any other process from holding a lock on the file while we perform our operations.

```
use Fcntl ':flock';             # import LOCK_* constants
use Fatal qw(open close truncate flock);

# Open file for read and write
open my $file_fh, "+<", $file;

# Lock the file for writing (exclusive lock)
flock($file_fh, LOCK_EX);

# At this point we have exclusive access to the file.
# Wipe previous process' details
truncate($file_fh, 0);

# Write to the file, or perform other operations as needed here...
print {$file_fh} $data;

close $file_fh;             # Closing the file releases the lock as well.
```

`flock` will wait indefinitely until the lock is granted, however it can return early if interrupted by a signal or other event. It's important to ensure that flock returns *true* to be sure that you have the lock you requested. It is possible to make `flock` *non-blocking* as follows:

```
use Fcntl ':flock';             # import LOCK_* constants

flock(FILEHANDLE, LOCK_EX | LOCK_NB);     # non-blocking exclusive lock
flock(FILEHANDLE, LOCK_SH | LOCK_NB);     # non-blocking shared lock
```

All attempts to get a *non-blocking* lock return immediately with either *true* for success (the lock was obtained) or *false* for failure (the lock was not obtained).

For an excellent introduction on using `flock`, the slides from Mark Jason Dominus' *File Locking Tricks and Traps* make excellent reading. They can be found at http://perl.plover.com/yak/flock/.

Chapter summary

- The two argument version of `open` is almost always the wrong open to use.
- The three argument version of `open` prevents us from allowing user data to specify the mode for opening the file.
- A race condition may occur whenever the ordering of events can affect the outcome of a situation.
- Due to the issues that symbolic links can cause, careful consideration should be taken on systems where symbolic links may occur.
- `sysopen` can be used to only open files under certain circumstances. Most notably this can be used to avoid symbolic link attacks when creating files.

- When creating private files, you should specify the desired permissions of those files at creation time.
- Perl makes it very easy to open anonymous temporary files. Using temporary files allows processes to write to these files when required and remove them without fear that another process can access them.
- Perl's temporary files can be shared with other processes as well.
- Some race conditions can be avoided by using file locks and not using Perl's file test operators before accessing the same file.

Chapter 6. Executing system commands

In this chapter...

In this chapter we will look at how we can more safely call system commands from Perl.

Using system and exec

As discussed in our chapter on *Taint checks*, Perl's `system` command invokes the shell by default. By injecting shell meta-characters into arguments used by `system`, it can be tricked into performing unintended behaviour. For example:

```
system("/usr/bin/finger $username");
```

If `$username` contains something like `root; cat /etc/passwd` then on a Unix system both the **finger** and **cat** commands will be executed.

The same thing would happen if we replaced `system` with `exec`, or Perl's backticks operator (which can also be written as qx{}).

The problem here is that the one-argument version of `system` passes its argument to the shell. The shell then translates metacharacters (for example expanding `*` to be the contents of the current working directory) and runs the command. If there is more than one command then each command is run. Finally, the shell returns the return code of the last command.

Multi-argument system/exec

In Perl version 5.6.1 a second form of `system` and `exec` were introduced. These versions allow a command to executed along with a list of arguments to pass that command.

An important feature of the multi-argument version of `system` is that on Unix systems it does not invoke the shell.

At the time of writing, the multiple argument version of `system` may invoke the shell under Windows on all known versions of Perl (including 5.8.8 and 5.10.0) if the command cannot be found by searching the system's `PATH` environment variable.

Replacements for `system` that never use the shell, even on Windows systems, can be found on the CPAN. Notably, the `IPC::System::Simple` provides a taint-aware, drop-in replacement for both `system` and backticks.

The remainder of this chapter assumes that you are working on a Unix system, or using `IPC::System::Simple` when calling `system`.

The multi-argument version of `system` is used like this:

```
system("/usr/bin/finger", $username);
```

In the finger example above, `$username` will be treated literally. Therefore if `$username` contained something like `root; cat /etc/passwd` then we would attempt to find the user information for the user: `root; cat /etc/passwd`, who probably doesn't exist.

This is also the case for exec:

```
exec("program", @args);
```

This still doesn't mean that passing untrusted data to `system` is a good idea; bad data can do other things besides from tricking the shell, and the application you're calling may invoke the shell anyway as part of its work. Like most techniques described in this course, the multiple-argument versions of `system` and `exec` (and `capture`, which we'll see shortly) are an extra safeguard against making a mistake.

Between Perl versions 5.6.1 and 5.8.0 inclusive, the multiple argument versions of `system` and `exec` would accept tainted data. This has been deprecated, and in Perl 5.8.1 and later pass tainted data to any `system` or `exec` results in an insecure dependency error.

Replacing backticks

Perl's backticks operator (which can also be invoked with `qx()`) allows a command to be executed and its output captured. Commands executed in backticks are always passed to the shell, and as such are inherently unsafe. If you are running in taint mode, Perl will not allow you to use tainted data in combination with backticks.

If you wish to read the output of a command, without passing it via the shell, then this can be done without additional modules in Perl 5.8.x by using a multi-argument piped open:

```
# This only works in 5.8.0 and above.

# Open a pipe from 'gunzip -c $file' and read the results
# into $data.

my $data;
open(PIPE,"-|","gunzip","-c",$file) or die $!;
binmode(PIPE);
{
        local $/ = undef;       # Slurp mode.
        $data = <PIPE>;
}
close(PIPE);
```

The multi-argument open doesn't exist in Perl 5.6.x and cannot be used. The alternative, which is opening the special file `-|` makes it difficult to tell if the command actually ran successfully, and the complexity of code requried can increase the chances of mistakes.

In all cases, the `IPC::System::Simple` module from the CPAN can be used to provide a safe and easy to understand backticks replacement:

```
use IPC::System::Simple qw(capture);

my $data = capture('gunzip','-c',$file);
```

The `capture` command will throw an exception if passed tainted data, used with any tainted environment variables that Perl 5.10.0 considers dangerous, or if the command terminates abnormally. See the `IPC::System::Simple` documentation for further details.

If your code is running with additional unix privileges, and you wish to run a command with lesser privileges, you can combine the above technique with `Proc::UID` which is discussed later in this manual:

```
use Proc::UID qw(drop_uid_temp restore_uid $RUID);

drop_uid_temp( $RUID );

my $data = capture('gunzip','-c',$file);

restore_uid();
```

General guidelines for backticks and system

Unless you actually want to invoke the shell, don't use backticks. Even then, backticks and system don't provide an easy way to examine `$?` to determine if your command executed successfully. If you care about security enough to read these notes, you probably want to know if your command was zapped with a signal, or bailed out with an unusual exit value. `IPC::System::Simple`'s `capture()` provides not only a safer backticks (and `system()`), but provides better diagnostics when things go wrong.

Even if you are using the multi-argument commands from `IPC::System::Simple`, you should still be mindful of shell and meta-characters. Many commands are known to invoke the shell to do their work, or pass data to other commands where characters can have special meanings. You may even be calling a command that contains its own bugs, or your data may contain a null-byte (see the tricks and traps chapter for more information).

Where you absolutely must use external commands, make sure you deny all characters by default and only allow characters which you know to be safe. Alphanumerics and underscore are generally without meaning on most systems, but even innocent characters like hypens can be dangerous if they appear at the start of an argument (which may turn what you thought was a filename into a switch).

Finally, don't use backticks, `system()` or `capture()` where Perl provides a built-in way of doing the same thing. For example, use `unlink` instead of `system("rm",$file)`; use `readdir` or `glob` instead of `my @dirs = 'ls'`. The Perl built-ins are not only more secure, but they also run faster than starting a whole new process to do your work.

Chapter summary

- The multi-argument versions of `system` and `exec` by-pass the shell on Unix systems, but can still invoke the shell on Windows systems.
- There is no way to arrange for backticks to by-pass the shell, however the `IPC::System::Simple` module from the CPAN provides safer alternatives for backticks and `system()`.
- If you must invoke system commands, be extra picky about the characters you allow, and always deny-by-default.

Chapter 7. Dropping privileges in Perl

In this chapter...

In this chapter we discuss how privileges can be handled in your Perl code. We will cover programs with elevated privileges (that is those running with more privileges than the user who invoked the script), and how to implement privilege separation using the `ops` pragma and `Safe` compartments. We will touch briefly on `chroot` and discuss virtual machines.

Privileges the Unix way

All modern Unix systems have a concept of a user ID (uid) and group ID (gid) which define the privileges of the running process. Most modern Unix systems also have concepts of *real*, *effective* and *saved* uids (and gids). These need to be discussed further to properly appreciate how privileges work on a Unix system, and how we can manipulate them in Perl.

Throughout this discussion, anything that applies to the *user ID (uid)* also applies to the *group ID (gid)* unless stated otherwise.

Most programs run with the the privileges of the user who executed it. However by setting the *setuid* (or setgid) bit on a file, it can be started with the additional privileges of the *owner* of the file. A good example of a setuid program is the unix *passwd* command, which needs to be able to use root privileges to modify the `/etc/passwd` and `/etc/shadow` files.

When a setuid program starts, it begins with the *effective uid* of the owner, but the *real uid* of the person invoking the command. Examining the real and effective user IDs is how Perl determines if it's being invoked setuid.

Almost all operations performed by your program are done with the privileges of the *effective uid*. With our example of the Unix `passwd` program, its operations will be performed as root by default.

Most modern Unix variants will allow swapping of the real and effective user IDs. This allows a process to perform a number of operations with privileges, and then drop its privileges back down to that of the invoking or unprivileged user.

It's possible for a process to swap its real and effective user IDs multiple times during its execution. This allows for the good programming practice of only enabling elevated privileges for the operations that require it. However, it also comes with the risk that a compromise to the program will allow for re-elevation of privileges. Well-designed programs will drop their privileges permanently as soon as it is practical to do so.

Many modern Unix operating systems also have the concept of a *saved uid*. This is the effective uid when the process was started. For example, when our `passwd` program starts it would have a saved and effective uid of root, and a real uid of the user changing their password. A process can change its effective uid to either its saved or real uids, which makes it simpler for a setuid program to swap between the user running the file and the user who owns the file.

Unfortunately, *Perl has no concept of the saved uid*, and this results in a serious problem. It's very difficult in Perl to fully and *irrevocably* drop the privileges of a process. Almost all systems that have a saved uid at the very least allow a process to change its effective uid back to that of the saved uid.

It's good practice to drop privileges permanently once finished with them. This is a common paradigm for otherwise unprivileged daemons (such as apache) running on privileged ports (those with numbers less than 1024).

Fortunately, CPAN provides a module that allows for the manipulation of the three standard Unix uids. This module, called `Proc::UID`, provides a consistent interface to manipulating the various Unix uids and gids, as well as standard ways to either temporarily or permanently drop privileges.

Using Proc::UID

The `Proc::UID` module provides a consistent and logical interface for Unix uid manipulation. It has been designed with the following goals in mind:

- The interface should be easy to understand
- Mistakes should be difficult to make

`Proc::UID` actually provides a few different ways to manipulate unix privileges, however the preferred interface thinks of privilege transitions as logical operations, rather than manipulating the uids directly.

For the latest information about `Proc::UID` check its documentation on CPAN.

The preferred (logical) interface

In 2002, Hao Chen, David Wagner, and Drew Dean presented a paper called “Setuid demystified” at the Usenix Security conference. In this paper they explained that there were three logical security related operations that privileged scripts were likely to perform. These are to temporarily drop privileges, to regain those privileges and to permanently drop privileges.

`Proc::UID` provides these operations with the names suggested in the paper. This is the preferred interface to Unix-style privileges as it is clear, self-documenting and difficult to get wrong.

- **`drop_uid_temp($uid)`** and **`drop_gid_temp($gid)`** :

  These functions allow a program to temporarily drop its privileges to the given `$uid` or `$gid`. They have the effect of setting the effective uid/gid to the supplied argument, and the saved uid/gid to the previous effective uid/gid. The real uid and gid are not changed.

- **`restore_uid()`** and **`restore_gid()`** :

  These functions allow a program to restore privileges previously dropped with `drop_uid_temp` or `drop_gid_temp`. They set the effective uid/gid to the saved uid/gid. The real uid and gid are not changed.

- **`drop_uid_perm($uid)`** and **`drop_gid_perm($gid)`** :

  These functions allow a program to permanently drop its privileges to the given `$uid` or `$gid`. They guarantee that the real, effective and saved uids/gids will be set to the argument supplied.

All of `Proc::UID`’s preferred functions will throw an exception if the operation fails. This can be captured using an `eval` block if desired.

The preferred interface is used as follows:

```
use Proc::UID qw(:logical);
```

```
drop_uid_temp($new_uid);        # Temporarily drop user privileges

restore_uid();                  # Restore user privileges

drop_gid_perm($new_gid);        # Permanently drop privileges
drop_uid_perm($new_uid);
```

The variable interface

`Proc::UID` does not export its variables by default. However these can be accessed by importing them by calling `Proc::UID` with the `:vars` import tag or by requesting them specifically. Changing the values of these variables will change the actual privileges of your programs, or throw an exception if the operation fails. The variables `Proc::UID` provides are:

- **`$RUID` and `$RGID`** :

  The real uid and gid of the process. Nominally the same as `$<` and `$(`.
- **`$EUID` and `$EGID`** :

  The effective uid and gid of the process. Nominally the same as `$>` and `$)`.
- **`$SUID` and `$SGID`** :

  The saved uid and gid of the process. There is no equivalent special variable in Perl.

The variable interface is used as follows:

```
use Proc::UID qw(:vars);

print "My saved-uid is $SUID, effective-uid is $EUID ",
      "my real-uid is $RUID\n";
```

The functions to drop privileges permanently have no variable equivalent. While the statement

```
$EUID = $RUID = $SUID = $unprivileged_uid;
```

has logically the same effect, it is slower and results in privileges being dropped one at a time. This is less reliable, and may result in failures on some systems.

A minor caveat

In the above section we mention that some of `Proc::UID`'s variables are *nominally* the same as Perl's special variables for the same purpose. It's important to note that there may be situations where they do *not* give the same values.

`Proc::UID` *always* queries the operating system for the actual value being requested. However, in its Perl uses instead *cached* values for its privileges. This means that should a piece of code (usually written in C or XS) modifies the process' privileges, then Perl may report *wrong* values for its uids and gids. `Proc::UID` automatically updates the cached values of Perl's built-in special variables when it is used.

A further difference to the variables provided by Perl and those by `Proc::UID` is that an unsuccessful attempt to change one of Perl's special variables will be ignored. An unsuccessful attempt to change one of `Proc::UID`'s variables will result in an exception being thrown.

Where possible, using `Proc::UID`'s special variables are preferred over using the built-in ones from Perl. However even better than that is using the `drop_uid_*` and `restore_uid()` functions from `Proc::UID`.

The variables from `Proc::UID` can conflict with those from Perl's own `English` module. However you can always use the fully qualified names for `Proc::UID`'s variables, eg `$Proc::UID::EUID`.

Functional interface

`Proc::UID` supplies a number of functions to manipulate privileges directly. These should only be needed if you really need them, the `drop_uid_*` and `restore_uid` interface is easier to understand and more difficult to get wrong.

- **`getruid()` and `getrgid()`** :

  Returns the value of the real uid and gid.
- **`geteuid()` and `getegid()`** :

  Returns the value of the effective uid and gid.
- **`getsuid()` and `getsgid()`** :

  Returns the value of the saved uid and gid.
- **`setruid()` and `setrgid()`** :

  Sets the value of the real uid or gid to `$uid` and `$gid` respectively.
- **`seteuid()` and `setegid()`** :

  Sets the value of the effective uid or gid to `$uid` and `$gid` respectively.
- **`setsuid()` and `setsgid()`** :

  Sets the value of the saved uid or gid to `$uid` and `$gid` respectively.

Privilege Separation

One of the most secure ways of dealing with privileges in any language is using the paradigm of *privilege separation*, trying to completely separate code that requires additional privileges from the rest of the program. This allows the privileged code to be more carefully modelled, reviewed and audited. All other operations can be pushed out into the unprivileged sections thus limiting the damage from a potential compromise.

One of the most famous examples of privilege separation is that employed on the *OpenSSH* project. By using privilege separation, it was possible to model the privileged section of code using a finite state automata, and rigorously test against this model. The amount of code running with privileges was substantially reduced, and the unprivileged program could be run in a very restricted environment, as its need to access privileged resources was reduced or eliminated.

While a full discussion on privilege separation is beyond the scope of this course, it's possible to create an unprivileged child that can still communicate with its parent using the `IPC::Open2` module:

```
use IPC::Open2;
use Proc::UID qw(:funcs $RUID);

# Drop privileges back to our real uid.
drop_uid_temp($RUID);

# Run our command without privileges
my ($readfh, $writefh);
my $pid = open2($readfh, $writefh, "/path/to/command");

# Re-establish privileges.
restore_uid();

# $readfh can be used to read from the program’s STDOUT, and
# $writefh can be used to write to the program’s STDIN.
```

Dropping code privileges with the ’ops’ pragma

Perl provides the ability to voluntarily ’give up’ the ability to perform certain operations. Indeed, it’s possible to revoke the privileges of your program to use just about any Perl construct (opcode).

The ability to revoke opcodes is an extremely powerful one, and is surprisingly easy to use with a simple Perl pragma. Opcodes are collected into logical groups (eg, subprocesses or mathematical functions). The following is an example of how to revoke the right to perform operations that include opening files and ’dangerous’ operations, which include `syscall`, `dump`, and `chroot`.

```
#!/usr/bin/perl -wT
use strict;
no ops qw(:dangerous :filesys_open);

# I can no longer open files.  However, I can still use ones that
# are already open.
```

While the `no ops` pragma is the most easy to use, in some ways it is like setting a list of characters that are not allowable in filenames -- it’s easy to forget some.

If you want to be very explicit and restrictive in what operations are permitted by your program, you can use the `use ops` pragma. Anything that’s not mentioned in the `use ops` line is forbidden.

```
#!/usr/bin/perl -wT
use strict;

# Allow ’safe-enough’ opcodes, but not dbmopen, since we don’t
# need or want it.

use ops qw(:default !dbmopen);

# Perl is now sand-boxed.
```

The use of the `ops` pragma does not prevent your program from compiling or executing any code written *before* it comes into effect. Nor does it prevent any compiled C code that your program may use from doing naughty things.

The protections afforded by the `ops` pragma should not be relied upon as your sole layer of safety. However this can be considered a form of defensive programming that can act as an extra safety belt and ensure your programs stay within their defined roles.

Safe compartments

Working with the principal of least privileges, it can be preferable to be able to break a larger program into smaller, specialised components. In the case where these components must act with different privileges, then the idea of privilege separation (discussed earlier) is most preferable.

The `Safe` module is designed to allow a Perl program to be broken into separate compartments, within the same process, each with its own function and ability to perform certain actions.

The `Safe` module is sometimes touted as a way to run 'untrusted' or un-reviewed 3rd party code, however this cannot be recommended. If you think code may be hostile, *you should not be running it*.

`Safe` is most useful to enforce compartments and sandboxing on your own trusted code.

`Safe` compartments provide:

- a new namespace; by changing the root of the namespace, code inside the compartment is prevented from referring to variables and code outside it.
- an operator mask; this allows the programmer to allow and refuse different Perl operators.

The only variables shared with `Safe` compartments, by default, are `$_` and `@_`. This sharing is necessary to ensure that subroutines can receive their arguments and Perl operators that use `$_` by default still work. These variables are shared both with the code calling the `Safe` compartment and between compartments. You should be mindful of these variables being shared when working with `Safe` compartments.

Using Safe

To use the `Safe` class we need to do the following things:

- Create one or more compartments
- Specify which opcodes we wish to deny/permit
- Share any desired variables and code with the compartment
- Execute the desired code in our compartment
- Retrieve any values from the executed code that we wish to use

Creating a compartment

A compartment can be created as follows:

```
#!/usr/bin/perl -wT
use strict;
use Safe;

# Create our new compartment
my $compartment = Safe->new( "some_namespace" );
```

The argument to `new` is optional and allows us to specify the root namespace to use for the compartment. If this argument is not provided the default of `Safe::Root0` is used and the trailing number is incremented for each new compartment.

The namespace of the compartment can be determined by calling the `root` method on the compartment:

```
my $namespace = $compartment->root();
```

Permitting and denying opcodes

Rather that specifying operators directly, such as `open`, `rand` and `goto`, we specify the Perl opcodes. These opcodes are the internal representations of these operators used by Perl. We have to specify these because it is during the compilation to these opcodes that `Safe` is able to influence the compiler and refuse undesirable opcodes.

Newly created compartments start by allowing everything in the `:default` optag. This tag comprises of the tags:

```
:base_core :base_mem :base_loop :base_io :base_orig :base_thread
```

these in turn comprise of Perl opcode names, for example:

```
base_loop:
        grepstart grepwhile
        mapstart mapwhile
        enteriter iter
        enterloop leaveloop unstack
        last next redo
        goto
```

The `:base_core` tag attempts to provide all the opcodes required for basic Perl functionality without including opcodes which can be abused by bad code. For example, the opcodes in `base_loop` while appearing to be innocuous can be used in a resource attack - to consume all available CPU time.

To find out what each optag includes and why those opcodes are not part of `:base_core` read **perldoc Opcode**.

Opcodes can be permitted using the `permit` and `permit_only` methods. The `permit` function adds the listed opcodes to the allowed list in addition to any already allowed opcodes. The `permit_only` only allows the provided opcodes, any already allowed opcodes are removed.

```
#!/usr/bin/perl -wT
use strict;
use Safe;

# Create our new compartment
my $compartment1 = Safe->new( );
my $compartment2 = Safe->new( );

# Allow compartment1 to call: atan2 sin cos exp log sqrt and rand srand as
# well as the basic stuff
$compartment1->permit( :base_math );

# ONLY allow compartment2 to call the :base_core and :base_loop opcodes.
$compartment2->permit_only( :base_core, :base_loop );
```

Opcodes can be denied by using the `deny` and `deny_only` methods. `deny` denies the listed opcodes from being used although other opcodes may still be permitted. The `deny_only` method denies only the listed opcodes, all other opcodes will be permitted.

```
# This compartment can call the math ops but not srand
$compartment1->permit( :base_math );
$compartment1->deny( srand );
```

Sharing code and variables

The opcodes specified are *only* permitted/denied during the compartment's compilation. If you pass your compartment a reference to a subroutine which is allowed to use forbidden opcodes this can still be executed. This can be done with the `share` method:

```
# Create our new compartment
my $compartment = Safe->new( );

# Let our compartment use our safe_open subroutine
$compartment->share('&safe_open');

# Later...
sub safe_open {
        my ( $mode, $path, $filename ) = @_;

        # vet arguments ...
        my $filehandle;
        open ($filehandle, $mode, "$path/$filename")
                or die "Failed to open "$path/$filename";

        return $filehandle;
}
```

The `share` method takes a list of variables and subroutines to share with the compartment. These variables must not be lexical variables (that is they cannot have been declared with `my`) and are assumed to have come from the calling package. Sharing is very similar to exporting using the `Exporter` module.

To share symbols from another package the `share_from` method can be used. This method takes the name of the package that the symbols are in and an array reference of the symbol names. For example if we wished to allow the following values from our `Utilities.pm` modules we could write:

```
use Safe;
use Utilities;

# Create our new compartment
my $compartment = Safe->new( );

# Let our compartment use the following symbols from our Utilities module
$compartment->share_from('Utilities',
        [qw/&safe_open $Verbose @paths %warnings/]
);
```

Sharing functions which use string `eval` for any reason (such as to set certain values) can lead to potential security issues. This is because the string eval will be eval'd within the namespace of the compartment.

By evaling code within the namespace of the compartment the compartment gains control of any values set in those evals. For example, consider a function `permissive` in a package `pkg` which is compiled outside the compartment but shared with it. Assume the compartment has the root package name of `Root`. If `permissive` contains an eval statement like `eval '$var = 1'` then when `permissive` is called from within the compartment (by any means) the eval will set `$Root::pkg::var` *not* `$pkg::var`.

Using `Safe`'s `share` method can be rather restrictive when it comes to sharing subroutines. The shared subroutines must not call other subroutines that have not been shared with the compartment, and must not dynamically compile code (using string `eval`) which contains opcodes not allowed within that compartment.

These are good defaults and help to limit what compartments can do in most cases. However, occasionally we'd prefer to call a subroutine defined outside of the compartment as if our code calling it was also outside of the compartment. To achieve this we can use the `Safe::Hole` class. For more information about this class, refer to CPAN.

Executing code and retrieving the return values

To compile and execute code inside our compartment we call either the `reval` or `rdo` methods. `reval` takes Perl code stored in a string and uses it, whereas the `rdo` method takes a filename with its absolute path and uses the code therein.

The code provided is compiled and any attempt to use an opcode which is not permitted will cause an error. For example, if we were to try to use the `open` opcode without explicitly allowing calls to `open` we might get:

```
'open' trapped by operation mask at (eval 2) line 2.
```

This error will not cause your program to halt, but will be stored in the variable `$@` just like when `eval` blocks fail. As this error is generated at the time that the code is being compiled no code in the compartment will be run.

If there is no error generating the compiled code, then the code is executed and the result of the last expression is returned. You can also use `return` as you would in a subroutine.

```
# Run my test code
my $return_value = $compartment->rdo( "/home/test/bin/test.pl" );

die "Failed to compile /home/test/bin/test.pl: $@" if $@;
```

When using `reval` you can return a list from your compartment. However as `rdo` is built on `do` you can only return a scalar from your compartment. This scalar can be a reference to an array or hash.

Running code in a Safe compartment can be difficult when you're getting compilation errors. For this reason we highly recommend breaking your compartment codes into separate files and using `rdo`. Compilation of these files can then be tested with **perl -c *`filename`***.

If you use the strict pragma in your Safe compartments, remember to allow your compartments to use the `require` opcode. Also, if you share variables with your compartment remember that they're provided as *package variables*. You can fully qualify them by adding `::` to the front of the variable name, this will prevent strict from complaining about them during compilation testing.

```
$::package_variable;    # doesn't need to be declared, so strict will be happy
```

A practical application

`Safe` provides us with a way to compartmentalise our programs into sections which only have the privileges they need. This means that we can ensure that only this part of the code can open files, while that only that other part of the code can write to the file system.

In the following example we use `Safe` to reduce the work required when auditing our code for security. By compartmentalising our sections it's easier to determine which pieces of code need special scrutiny and for what kind of errors.

```
#!/usr/bin/perl -wT
use strict;
use Safe;
use CGI;

use vars qw/$cgi %cgi_values/;
$cgi = CGI->new();

##########################################################
# Get my CGI variables out and do some validation.

my $cgi_comp = Safe->new( );
                # This compartment will need to do looping stuff
$cgi_comp->permit(qw/:base_loop require/);
$cgi_comp->share('$cgi');

%cgi_values = $cgi_comp->reval(
        q!
                use strict;
                my %values;
                my @keys = qw/card_type card_no card_expm card_expy/;
                @values{@keys} = map { $cgi->param($_) || undef } @keys;

                # do some sort of user validation and untainting
                ($values{card_type}) = ( $values{card_type} =~ /^(\w+)$/ );
                ($values{card_no})   = ( $values{card_no} =~ /^(\d+)$/ );
                ($values{card_expm}) = ( $values{card_expm} =~ /^(\d{2})$/ );
                ($values{card_expy}) = ( $values{card_expy} =~ /^(\d{2})$/ );

                foreach my $val (@keys) {
                        unless( $values{$val} ) {
                                die "Missing or invalid value for $val";
                        }
                }

                # Check that the credit card is valid
                unless( is_valid(($values{card_type}, $values{card_no},
                                $values{card_expm, $values{card_expy}) ) {
                        die "Invalid credit card values.";
                }

                $values{amount} = order_amount();
                return %values;

                sub is_valid {
                        # perform some credit card checking algorithm
                        return 1;
                }

                sub order_amount {
                        return '100.00';
                }
        !
);
```

```
exit if test_failure($@);

# At this point I don’t need the compartment or cgi object anymore.
undef($cgi_comp);
undef($cgi);

##########################################################
# Process the credit card
$ENV{PATH} = "";

my $ccard_comp = Safe->new( );
                # I’m going to use system commands
$ccard_comp->permit( qw/:subprocess require/);
$ccard_comp->share(’%cgi_values’);

my $result = $ccard_comp->reval(
        q!
                use strict;
                my $command = "/home/process/bin/process $cgi_values{card_no} ".
                              "$cgi_values{card_expm}-$cgi_values{card_expy} " .
                              "$cgi_values{card_type} $cgi_values{amount}";
                my $result = qx/ $command /;

                # Delete credit card values now that we no longer need them
                delete @cgi_values{qw/card_no, card_expm, card_expy/};

                # Success...
                if($result =~ /success/) {
                        return $result;
                }
                else {
                        die "Failed to process credit card: $result";
                }
        !
);

exit if test_failure($@);
undef($ccard_comp);

##########################################################
# Add order to database, etc....
# ...
# ...

##########################################################
# Error handling
sub test_failure {
        my $err = shift;

        if ($err) {       # Either I died or I didn’t compile
                if($err =~ /trapped by operation mask/) {
                        # I didn’t compile, big problem here.
                        print STDERR "Problem: $err";
                }
                else {
                        # Print pretty CGI error page and ask the user
                        # to try again
                        print STDERR "CGI error: $err";
                }
                return 1;
        }
        return 0;       # no failure
}
```

Common pitfalls

Running untrusted code

In the vast majority of cases `Safe` is used to run code in a pseudo-sand-boxed environment. This code may exist because you allow clients to create their own templates, or you allow people to upload certain types of code for execution, or because auditing the unknown code for security flaws is too hard, too time consuming or too boring.

While this is the intended use of `Safe`, it's also a really bad idea. Running un-audited third-party code will eventually cause you grief. Even in a `Safe` sand-box, you cannot be 100% certain that the code will be entirely harmless. As the `Safe` documentation says; `Safe` cannot always protect you from attacks which consume all of your systems memory, use up your CPU with infinite loops, copy (and possibly send outwards) private information from your system, cause signals such as `SIGALARM` which might upset your process or change the seed on your random number generator.

Using `Safe` can provide an artificial sense of security if you blindly use it in this manner. `Safe` should be considered another tool in which to assist you to make your code more secure while being aware that there is no substitute to a proper security audit by a experienced Perl programmer.

Poor access restrictions

Using `Safe` doesn't prevent the code in the `Safe` compartment from being able to access things it shouldn't. For example if you give a compartment permission to open files, `Safe` will be just as happy to let the code open `/etc/passwd` as it will be for any other files. Likewise if you allow your compartment to execute one system command, you've given it permission to execute all of them.

This lack of access restrictions means that it's your responsibility to ensure that you give out the minimum opcodes required to make sure the code works. You must also ensure that where you allow opcodes out of the `:default` optag that you carefully consider what those opcodes allow and how to avoid disasters.

Not using other safety mechanisms

It is unfortunately common that people learn one new skill only to stop using the skills they using were previously. In Perl programming this often means that programmers stop worrying about taint checking when they learn about the three argument version of `open` and the multi-argument versions of `system` and `exec`.

Using `Safe` does not remove your need to untaint your variables and use the these safer functions. Likewise using taint and the three argument version of `open` shouldn't prevent you from trying to restrict your allowed opcodes as much as possible. Using `Safe` certainly shouldn't mean that you don't get someone else to review your code and try to spot problems with it.

Using `Safe` should be a valuable aid in securing your code and increasing auditability. It can also be used with caution to test third party code. It does not replace the need for code review, security audits and other good programming practices.

chroot

Unix systems provide a technique wherein a process is permanently restricted to an isolated subset of the file system. This technique is called `chroot` (*changes root (directory)*) and, as its name suggests, it changes the process's root directory (usually represented by `/`) to be something other than the true filesystem root.

By changing the root directory, the `chroot` call will allow you to 'jail' your process into a restricted part of the file system. This gives you an effective way to shield the other parts of your file system from your process.

Upon using `chroot`, it is good practice to explicitly `chdir` into your new environment using an absolute path. The reason for this is that on many operating systems the `chroot` call does not change the current working directory, and the process may be left with a working directory outside of the 'chroot jail'. It is also a good idea to then drop privileges immediately.

```
use Proc::UID ':func';
# do stuff that needs to be done as root

chroot("/path/to/chroot/jail") or die "Cannot chroot";

# Make sure we're in the jail.
chdir("/") or die "Failed to change into jail: $!";

# Drop privileges
drop_priv_perm($harmless_uid);

# do stuff as $harmless_user
```

It should be noted that filehandles will remain open across a call to `chroot`, and using these a process may still have access to resources outside of the jail.

The chroot jail

Once your process is jailed inside the chroot jail, it can no longer see the system binaries or anything other files outside the jail. This means that if your code needs to be able to access the file system you may need to provide file system binaries and a file system tree appropriately.

The construction and maintenance of an effective chroot jail is a complex topic, and beyond the scope of this course. However, a process that should be running without file system access can be safely `chroot`ed to an empty directory and then privileges dropped so it does not hold permissions to write to the directory.

An example of setting up a chroot jail can be found in the Chroot-BIND HOWTO (http://www.linuxsecurity.com/docs/LDP/Chroot-BIND-HOWTO.html). This describes how to install the BIND 9 name server inside a chroot jail to run as a non-root user.

chroot limitations

`chroot` is far from perfect, here is a non-exhaustive list of some of the limitations upon `chroot`.

- `chroot` is normally restricted to the superuser.

- `chroot` can offer a false sense of security, as it does nothing to stop your process from interacting with other process, making use of already open file handles, or consuming disk, memory, CPU and other resources.
- If the `chroot` call is not followed by a `chdir` call, the process can potentially escape the jail by walking up the directory tree (`chdir("..")`).
- If privileges aren't dropped after the `chroot` call then the process can often escape from the jail. The 'classic' way of doing this involves utilising open file handles and a second call to `chroot`.
- Non-root processes in a chroot jail can escape from that jail by 'attaching' themselves (in a similar mechanism to how debuggers work) to another process running outside the jail which is running with the same privileges. Once the process has attached itself to this other process it can control it remotely.

An alternative - use a virtual machine

Using a virtual machine is a sensible alternative to `chroot`. A virtual machine is often easier to separate from the 'real' machine, as well as being easier to understand and maintain. Furthermore, some virtual machines can even be run by a non-privileged user, rather than requiring administrative access to the machine.

Virtual machines are useful for a great number of reasons. By their very nature, it is easy to 'roll-back' or duplicate a virtual machine, making them excellent testing environments. Using a number of virtual machines and strict traffic-filtering rules between is also arguably better than running a great many services on a single machine. Should one of the virtual machines be compromised, an attacker will be more restricted in available resources than if they had succeeded in accessing a more monolithic system.

User Mode Linux

An example virtual machine is User Mode Linux (UML). This virtual machine allows users to run multiple, separate and isolated instances of Linux on a single Linux box. UML runs entirely in user space and each virtual machine has its own kernel and file system which are entirely configurable. UML does not limit the user to using the same distribution as the original box, in fact each instance of UML can run a different distribution if desired.

For further information about User Mode Linux visit the User Mode Linux Website (http://user-mode-linux.sourceforge.net/)

Chapter summary

- Unix systems have three user ID and group ID values per process.
- Using `Proc::UID` these values can be properly manipulated to drop privileges temporarily or permanently.

- Separating the parts of code which needs to work with elevated privileges from those parts which do not is a sound security practice.
- The `ops` pragma allows us to specify which opcodes we intend our program to need. If we then attempt to use other opcodes then compilation will fail.
- `Safe` compartments allow us to create compartments which can perform certain operations and run code within them. If the code attempts to use operations which are not permitted compilation fails.
- Using `Safe` allows us to increase the auditability of our code.
- `chroot` allows us to change the apparent root directory for a process.
- Using virtual machines provide a better option for security and usefulness than `chroot`.

Chapter 8. Database Security

In this chapter...

Databases are a common part of many modern applications. The most common way of integrating with a database in Perl is using the `DBI` module. This chapter covers a number of commonly seen programming mistakes with the `DBI` module.

SQL injection attacks

SQL injection attacks refers to any condition where an end-user may be able execute SQL commands without authority. Often these attacks are easy to execute, but luckily for us they're also easy to avoid.

The effects of a successful SQL injection attack can be revealing of data, corruption or destruction of data, or denial of service. In some rare circumstances an attacker may be able to even execute commands or code, depending upon the database in question and the privileges of the account being used.

The most common form of SQL injection attack involves feeding *SQL meta-characters* to a program. These meta-characters change the interpretation of the SQL being executed, in the same way that shell-meta-characters can change the execution of a shell command.

Most commonly the SQL quote character (*single-tick (')* on most systems) is used to try and prematurely escape from a string. The current command is then either altered or aborted, and a new command (or sequence of commands) are executed in its place.

The code below is an example of code that is vulnerable to an SQL injection attack:

```
my $dbh = DBI->connect($dsn,$user,$pass) or die "Cannot connect to DB\n";

my $username = <STDIN>;   # Could just as easily be from CGI
my $password = <STDIN>;
chomp($username,$password);

# Authenticate the user before allowing them access to their account.
# Verify that username and password match what is in the database.

# (DON'T DO THIS!)
my $result = $dbh->selectrow_hashref("
        SELECT account_details
        FROM   accounts
        WHERE  user = '$username' AND
               password = '$password'
");

# Work with $result...
```

The mistake that has been made is that both `$username` and `$password` may contain SQL meta-characters as these have not been escaped or cleaned in any way. Use of any variable interpolation inside an SQL statement is a strong warning sign that an SQL injection attack may be possible.

While many database drivers will only allow a single statement to be executed each request, this is not guaranteed to be the case. A simple `SELECT` statement can be manipulated by a clever attacker.

Using the example above, in some cases an attacker could gain access to any account (in this example we’ll use ’buffy’) by supplying a password of `' OR user = 'buffy' AND ' ' = '` . This results in the following statement being executed:

```
        SELECT account_details
        FROM   accounts
        WHERE  user = 'buffy' AND
               password = '' OR user = 'buffy' AND ' ' = ' '
```

The condition `' ' = ' '` is always true, and the injection attack succeeds, side-stepping password authentication. It’s possible for successful attacks to be made against such SQL even when no knowledge of that underlying SQL is available.

A commonly seen, but less than ideal way of avoiding this problem is to use a regular expression to escape the meta-characters first, like so:

```
$username =~ s/'/\\'/g;
```

Unfortunately, this is also error-prone. Different databases will use different methods of quoting. The SQL standard uses a double apostrophe, so `don't` is quoted as `'don''t'`. Some databases require the use of a backslash instead. Some will accept either.

Aside from the inconsistency, this method of quoting places the onus upon the programmer to do the right thing. Mistakes do happen, and sometimes data ends up not being escaped, or sometimes it’s escaped too many times, resulting in ’over-quoting’ and data corruption.

None of this even begins to touch upon what occurs if we’re dealing with binary data, which may not only require special quoting or handling, but may also contain characters (such as the null byte) which cannot be passed directly to many databases. If you’re starting to think this is all too hard, then you’d be right.

Fortunately, there is a better way to do things. We can use placeholders.

The `DBI` module provides the concept of place holders, which stand in the place of data that would otherwise need quoting. Here’s an example:

```
# Authenticate the user before allowing them access to their account.
# Verify that username and password match what is in the database.

my $result = $dbh->selectrow_hashref(q{
        SELECT account_details
        FROM   accounts
        WHERE  user = ? AND
               password = ?
}, undef, $username, $password);
```

Note we use a question-mark (`?`) as a *place holder* of where our data will appear. We do not use any quotes around the place holder, nor do we escape characters in the data that we are passing to the query. `DBI` handles these requirements for us, and does so in a way that is appropriate to the database that we are connected to.

Using place holders has another advantage as well, many databases are able to optimise repeated queries to run more quickly when place holders are used.

```
# We can prepare our statement once...
my $sth = $dbh->prepare("
        UPDATE accounts SET balance = 0 WHERE username = ?"
);
```

```
# ...and then use it many times.
foreach my $user (@users) {
        $sth->execute($user);
}
```

DBI and taint

Unless your Perl program has taint mode turned on, the taint features in `DBI` have no effect.

The `DBI` module is taint-aware, and from version 1.31 has supported the following attributes, which only have an effect when Perl is running in taint mode.

- `TaintIn`, when set, causes the arguments to most `DBI` method calls to be checked for taintedness. This means that you cannot insert tainted data into your database, or used tainted data as part of a `connect` call.
- `TaintOut`, when set, causes any data from fetch operations to be marked as tainted. In future versions, the results of other `DBI` calls may also return tainted data.
- `Taint` is simply a shortcut which allows the setting of both TaintIn and TaintOut at the same time.

Clearly, using `DBI`'s taint features can be a good idea. Just because data has been fetched from the database does not guarantee it's clean enough to swing past the shell on most systems. Likewise, insisting that you perform basic checks upon your data before inserting it into the database helps to ensure that no bad records are created.

```
#!/usr/bin/perl -wT
use strict;
use DBI;

my $dbh = DBI->connect($dsn, $user, $passwd, {
                AutoCommit => 1,
                RaiseError => 1,
                Taint      => 1
});

my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName
        FROM StaffAddress
        WHERE StaffID = ?
});

print "Which staff id? ";
my $staffid = <STDIN>;
chomp $staffid;

$sth->execute($staffid);          # Dies with an error
```

In order to use data in an SQL query we have to untaint it. We do this by capturing it out of a regular expression.

```
my $dbh = DBI->connect($dsn, $user, $passwd, {
                AutoCommit => 1,
                RaiseError => 1,
                Taint      => 1,
});
```

```
my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName
        FROM StaffAddress
        WHERE StaffID = ?
});

print "Which staff id? ";

my $staffid = <STDIN>;
chomp $staffid;

# Check that the staff id contains only digits
my ($safe_staffid) = ($staffid =~ /^(\d+)$/);

# If $safe_staffid is empty, the regex didn’t succeed
if(not $safe_staffid) {
        die "staff id contains invalid characters!";
}

# No error
$sth->execute($staffid);
```

Likewise, data coming out of the database must be untainted before passing it to any function that cares about tainted data. These include `open` when opening files for writing, `system`, `exec` and many more:

```
my $results = $dbh->selectrow_hashref(q{
                SELECT FirstName
                FROM StaffAddress
                WHERE StaffID = ?
        },
        undef, 12345,
);

# Dies with an error
system("mkdir /tmp/$results->{FirstName}");
```

Temporarily disabling Taint

`DBI`’s taint features are very flexible; it’s possible to turn `TaintIn` and `TaintOut` on and off for particular statement handles. For example, you may disable `TaintOut` for some select statements that you consider trustworthy.

```
#!/usr/bin/perl -wT
use strict;
use DBI;

$ENV{PATH} = ’/bin:/usr/bin’;

my $dbh = DBI->connect($dsn, $user, $passwd, {
                AutoCommit => 1,
                RaiseError => 1,
                Taint      => 1,
});

my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName
        FROM StaffAddress
        WHERE StaffID = ?
});
```

```
print "Input staff id: ";
my $staffid = <STDIN>;
chomp $staffid;
unless(($staffid) = ($staffid =~ /^(\d+)$/)) {
        die "staff id contains invalid characters!";
}

$sth->{TaintOut} = 0;      # Data from _this_ statement handle is safe.
$sth->execute($staffid);   # We still need to pass _IN_ untainted data

my @row = $sth->fetchrow_array();       # @row is not tainted
system ( "echo @row" );                 # no error
```

Chapter summary

- SQL injection attacks refer to any condition where an end-user may be able execute SQL other than that intended.
- Using place holders in our SQL allows us to prevent SQL injection attacks.
- If we’re using taint checking in our program then we can also ensure that tainted data is not added to the database and that data from the database is considered tainted. This allows us to increase the security of our program.

Chapter 9. Tricks and traps

In this chapter...

In this chapter we’ll cover a number of useful tricks and a few things to watch out for. We’ll also discuss some of Perl’s more interesting security issues in the past (now fixed) and discuss the issues between compiled and interpreted scripts with elevated privileges.

Tricks

Allowing relative paths

Occasionally we may want to allow users to specify relative paths to our programs. What we don’t want is allow them to abuse this, backtrack up our directory tree, and get to things they’re not supposed to.

An obvious solution would be to ensure that the string "`../`" doesn’t appear in the path. However this only solves the problem for operating systems that use ".." to indicate the parent directory, resulting in code that is not portable across operating systems. It can also cause problems if we have oddly named files that contain ".." in their filenames.

Fortunately the Perl module `File::Spec` provides us a platform independent way to determine if we are dealing with a relative or absolute path, and if a relative path is trying to access a parent directory. `File::Spec` comes with the Perl core distribution.

```
#!/usr/bin/perl -wT
use strict;
use File::Spec;

my $filename = <STDIN>;

# Filenames may contain word chars, dots, forward slashes and minuses
my ( $clean_filename ) = ( $filename =~ m!^([\w./-]+)$! );

unless( $clean_filename ) {
       die "Filename contains illegal characters.\n";
}

# Make sure it is not an absolute filename (don’t want /etc/passwd)
if( File::Spec->file_name_is_absolute( $clean_filename ) ) {
        die "Absolute paths in filenames are not allowed";
}

# Make sure that we’re not trying to walk up the file tree
my $updir = File::Spec->updir();
if( grep {$_ eq $updir} File::Spec->splitdir($clean_filename) ) {
        die "Attempts to backtrack are not allowed";
}

# Work with file
```

Even when we disallow attempts to walk up the directory tree, it’s still possible to navigate out of a directory tree if symlinks are present.

Restricting information give-away

One matter that should be given careful consideration on any system that is exposed to the public is that of information give-away. Put simply, a system should be careful not to provide more information than is required to a user, particularly an un-authenticated one.

The classic example of information give-away is the response to a failed login. A well-designed system will respond with a message such as "login failed". This clearly informs the user that their login was unsuccessful, but doesn't tell them if that's because the account does not exist, the password is wrong, or another reason (such as the account being locked).

Compare this to a message such as "Incorrect Password", or "No such username". These messages clearly allow a person accessing the system to tell whether or not a given username exists. The problem of this approach is that it gives a potential attacker this information as well. If an account is known to exist, the attacker can concentrate their attempts on compromising that specific account. These may include social attacks -- having found an account that does exist, a skilled attacker may be able to contact tech-support and claim they have forgotten their password, and ask for it to be reset.

Diagnostics and web applications

When developing web applications, allowing debugging information to go to the browser is an almost essential tool in the development and testing process. However once an application has gone into production, these features should be disabled.

Allowing a user to see warnings and errors is great if that user is a developer, but it can also reveal a great deal of information about your system to a potential attacker. This could include the location of files, configuration information, or even passwords (particularly from stack back-traces).

Most web frameworks have standard ways to report diagnostics, and standard configuration options to control where they are displayed. You should investigate and use these if they are available to you.

For a flexible and extensible logging mechanism, the `Log::Log4perl` module is a good choice. Used sensibly, `Log::Log4perl` or other logging mechanisms can mean the only thing you need to change between testing and production is a configuration file.

Traps

The diamond (<>) construct

We all know that the three-argument version of `open` is much safer than the two-argument version. We also know that if we do use the two-argument version then Perl will *assume* we intending to open the file for reading if we don't say otherwise.

Unfortunately, the commonly used diamond construct uses the two-argument version of `open` internally, and worse still, it doesn't specify in what mode the file should be opened in. That means if we have a filename starting with > we're going to clobber and write to a file. If we have a filename starting or ending with a pipe (`|`), we will end up executing code. Oh dear!

Put simply, if an attacker is able to able to define a filename that will be opened through use of the <> construct, then they can execute code with the privileges of the process performing the open. Combined with symbolic links, practically any file or command on the system can be referenced.

This is of particular concern in commands invoked from `cron`, or any command that may read from a directory not exclusively controlled by the command's owner.

Luckily, if we're programming using taint checks, then an attempt by <> to open a file for any operation except reading will result in an exception being thrown. This is not a perfect solution, since there are still some undesirable operations (like stream duplication) that Perl won't catch, but it's considerably better than allowing arbitary code to be executed using funny filenames.

Using the `-n` flag with Perl causes Perl to assume that your program is enclosed in the following loop:

```
while (<>) {
        # your code ends up here
}
```

and using the `-p` flag causes Perl to assume this loop structure:

```
while (<>) {
        # your code ends up here
} continue {
        print or die "-p destination: $!\n";
}
```

As a result these flags are subject to the diamond construct's weakness and special care should be taken when using them.

To be absolutely safe, it's recommended to avoid the diamond construct, or at the least least to rigously sanitize `@ARGV` before invoking it.

The poison null-byte

Many of Perl's built-in functions are written in C. To terminate an array of characters, in C, a null-byte is placed after the final character. This allows C to represent variable length "strings" to some extent, and saves C programmers from having to clear all of an array before writing a smaller string to it. For example:

```
char a[55];     /* create an array called a with 55 characters */
int i = 0;
                               /* a can contain any old junk */
strcpy(a, "this is a string"); /* a now contains "this is a string\0" */
for(i = 0; i < 5; i++) {
        a[i] = ’a’;
}                              /* a now contains "aaaaais a string\0" */
a[i] = ’\0’;                   /* a now only contains "aaaaa\0" */
```

Perl itself treats null-bytes as merely another string character, rather than anything special, which makes Perl excellent at handling binary data. Unfortunately Perl will happily pass data containing null-bytes to C code and that C code will assume that the data ends at the null-byte.

The rest of this section will discuss some cases in which the presence of null-bytes can cause behaviour unanticipated by the programmer.

Opening the wrong file

Imagine we have code which gets a filename from the user and opens that file for reading or writing, but we’re not using taint mode. The following code uses one example of getting a filename from outside our program, there are many other ways as well.

```
#!/usr/bin/perl -w
# File: script.pl
#
# This program opens a file whose name is provided by the user.  The
# contents of the file are then displayed to the user.  We make sure that
# the user is not providing absolute paths or walk out of our directory.
use strict;
use Fcntl;
use CGI;
use File::Spec;
use autodie qw(sysopen);

my $cgi = CGI->new();
print $cgi->header;

my $file = $cgi->param(’file’);

# Check we were given a filename
unless( $file ) {
        die "You forgot to give me the filename!";
}

# Make sure it is not an absolute filename (don’t want /etc/passwd)
if( File::Spec->file_name_is_absolute( $file) ) {
        die "Absolute paths in filenames are not allowed";
}

# Make sure that we’re not trying to walk up the file tree
my $updir = File::Spec->updir();
if( grep {$_ eq $updir} File::Spec->splitdir($file) ) {
        die "Attempts to backtrack are not allowed";
}

# We’re expecting the filename to be something like "foo" meaning we
# actually want to open "foo.html".  We use sysopen with O_RDONLY so that
# we ensure the file exists.

sysopen(my $fh, "$file.html", O_RDONLY);

print <$fh>;
```

It appears that we’re almost doing everything right in this code. We check we have a value, we check it’s not using a path attack, we insist the file can only be opened if it exists and we append `.html` to the filename in order to avoid non-html files being opened.

But we haven’t thought about the null-byte.

When a user of this program provides the filename `script.pl%00` we run into trouble. (`%00` is the html representation of the null character). This filename is not empty and it doesn’t look like a path attack. When `sysopen` checks that the file already exists, it checks for `script.pl` not `script.pl\0.html`! This is because `sysopen` is implemented in C, and C interprets the null byte as the end of the filename. So the source of `script.pl` will be displayed to the user.

Passing the wrong value

It's probably not a surprise that many system utilities also do not consider null-bytes as valid string characters and will truncate input at the first occurrence. Consider the case where we allow users to see what processes any user is current running, except `root`.

```
#!/usr/bin/perl -w
# Displays user information for given user, excepting root
use strict;
use autodie qw(system);

my $user = <STDIN>;
chomp $user;
die "Need a user name!" unless $user;

if($user ne "root") {
        system "finger", $user;
}
else {
        print "Naughty, naughty\n";
}
```

If we provide Perl with the string `root\0`, for `$user`, the first test will fail. However, when we pass this value to the shell it sees the string `root`.

Taint to the rescue

Had we untainted the user input in the previous values with a sensible regular expression - such as one which specifies the characters we want to keep rather than those we don't want - the null bytes should have caused the user strings to be discarded as invalid. Validating your data careful is the best defense against null bytes, and having taint *force* you to validate your data is better yet.

```
#!/usr/bin/perl -wT
# Displays user information for given user, excepting root
use strict;
use autodie qw(system);

my $user = <STDIN>;
chomp $user;
die "Need a user name!" unless $user;

# No null bytes will get past this untainting
unless( ($user) = ($user =~ /^(\w{1,8})$/) ) {
        die "Invalid username!";
}

if($user ne "root") {
        system "finger", $user;
}
else {
        print "Naughty, naughty\n";
}
```

Past issues with Perl

glob

In versions of Perl before 5.6.0, the `glob` function (and <*`WILDCARD`*>) ran an external program to get the list of filenames. This program was often the *csh* shell. As a result, glob could be subject to a buffer overflow attack and would return truncated strings (in particular without the last path component) if the full result didn't fit into the *csh* shell's buffer.

As of Perl version 5.6.0, both `glob` and <*`WILDCARD`*> are implemented with `File::Glob`, which is not vulernable.

Algorithmic complexity attacks

Certain of Perl's internal algorithms can be attacked by using carefully crafted input designed to consume large amounts of time, space or both. This abuse can lead into denial of service attacks. We cover a few of these here.

Perl's hash function

Perl's hashing algorithm has been designed to be as fast as possible. It wasn't designed to take security into account until Perl version 5.8.1. Prior to this version it was possible to generate data which would hash to the same bucket causing the internal structure of the hash to degenerate and therefore consuming large amounts of time.

Since Perl 5.8.1 a pseudo-random seed has been added to the hash function increasing the complexity of generating such data.

A side effect of this change is that hash keys no longer have the same ordering in different program invocations. This may confuse some applications (like `Data::Dumper`) which relied on keys having a consistent ordering. It's important to note that Perl has never guaranteed any ordering of the hash keys. The effects of this change can be turned off by setting the environment variable `PERL_HASH_SEED` to zero. We don't recommend that, however.

At the time of writing, all current releases of Perl (5.8.8 and 5.10.0) will cough up the hash seed if the environment variable `PERL_HASH_SEED_DEBUG` is set, even if the code is running in taint mode.

For further information read the section on `PERL_HASH_SEED` in **perldoc perlrun**.

Scott A Crosby and Dan S Wallach cover some specific attacks against Perl's earlier hash implementations in their Usenix Security presentation (http://www.cs.rice.edu/~scrosby/hash/).

Sorting

Prior to Perl version 5.8.0, Perl's `sort` function used the *quicksort* algorithm. *Quicksort* is very fast at sorted randomly ordered lists, however it also very, very slow when sorting a sorted or almost sorted list.

Since Perl 5.8.0 `sort` uses the *mergesort* algorithm. *Mergesort* is insensitive to the ordering of input data.

Malloc wrapping

Prior to Perl version 5.8.4, attempts to assign pathologically large chunks of memory could suffer from integer wrap-around during size calculations. This would cause a misallocation and result in Perl crashing.

Since Perl 5.8.4, Perl can detect attempts to assign such large chunks of memory and avoids this issue.

suidperl

`suidperl`, which allows Perl programs to run with elevated privileges historically been implicated with security flaws in Perl. In April 1997 an exploitable buffer overrun was discovered in Perl 5.003 (AusCERT, AA-97.13), and in August 2000 an exploit was discovered allowing privilege escalation caused between `suidperl` and `/bin/mail`. These were subsequently fixed, however further security vulnerabilities are always possible.

`suidperl` is neither built or nor installed by default, and may be removed from later version of Perl. It is recommended that you use dedicated, single-purpose tools such as **sudo** instead of `suidperl` where possible.

`suidperl` is discussed more fully in the next section.

Issues with scripts vs compiled code

There is a long standing issue with setuid interpreted scripts on Unix systems. When a setuid script is starting, the operating system sets the appropriate privileges and then looks at the program to be executed. Upon finding the `#!` line, it starts up the interpreter, and provides it with the name of the file to interpret.

Unfortunately, this set of events can result in a race condition. In between the privileged interpreter starting, and it opening and reading the script, the script may have changed. This could particularly be the case if the script was a *symbolic link* to a setuid script. The link, being owned by the attacker, can be easily removed and swapped to a different file. In the case where this is successful, the attacker's own code will be executed with raised privileges.

This race condition represented an easy way of an attacker to gain privileges, and has been responded to by different operating systems in different ways. Some operating systems disallow setuid scripts altogether, and drop privileges as soon as they see the `#!` line. Others pass the script using a special `/dev/fd/X` filename, thereby avoiding the problem by effectively passing in an open file descriptor. Another set don't fix the problem at all, but these are often older systems that are no longer supported.

Those systems which pass in an open file descriptor will have setuid scripts work correctly, regardless of the language used. Those systems that ignore setuid on scripts would appear to represent a problem should you desire to write setuid scripts of any sort. However, Perl comes with a solution to this, called `suidperl`.

suidperl

When Perl starts, it examines the file containing the code it is to execute. If it finds that the code is setuid, it starts `suidperl`, which is a special Perl that runs setuid. `suidperl` will perform a number of checks to ensure that it's not being deceived, set its permissions to the appropriate values, and the execute the required script. This allows setuid scripts to still work, even on systems where the setuid bit on scripts is ignored, such as Linux.

`suidperl` is a user-space solution to what is fundamentally a kernel problem, and it requires a number of non-trivial checks to ensure that it's operating correctly. Bugs have been found in older versions of `suidperl`, ranging from the relatively minor (such as revealing the presence of a file in a directory that an unprivileged user does not have mission to read), to major (ability to elevate to root access with very old suidperls on some systems).

An alternative to `suidperl` does exist, and that is the idea of having a compiled, setuid wrapper program. The purpose of this program is simply to launch your Perl script. In this scenario your Perl program would not be marked setuid, however your wrapper would. It would then proceed to execute your Perl code with its privileges already set.

```
#define REAL_PATH "/path/to/script"

main(int argc, char **argv) {
        execv(REAL_PATH, argv);
}
```

As the wrapper program is only ever executing a file in a secure, known location, there's no possibility that it will follow a user-created symlink in between start-up and script execution. It also provides a very concise and easy-to-understand mechanism for setuid code to be executed.

A problem does exist with the use of wrappers, however it is social, not technical. Because the wrapper and the script are separate, one may write a script assuming that it cannot be run setuid, not realising that a setuid wrapper for it exists.

The use of `suidperl` is not recommended if you can obtain the same functionality through dedicated security tools such as sudo (http://www.sudo.ws/). If you must use `suidperl`, then it is recommended that you use one from Perl version 5.8.7 or above, as these versions do not contain any known vulnerabilities (as of June 2008).

Chapter summary

- The `File::Spec` module allows us to test for path attacks in a portable fashion.
- An important security consideration is restricting information give-away.
- The diamond construct (<>) uses the two argument version of `open` with an implicit open. This can be a trap for the unwary.
- Perl is happy to accept null bytes as string characters, however this can affect the behaviour of the underlying C functions. This is another good reason why you should use taint checks.
- Previous versions of Perl suffered from a myriad of security problems ranging from the arcane to the dangerous. As these are found, Perl improves.
- Due to the problems with setuid scripts, `suidperl` was created. However this has also had its problems and has now been deprecated in favour of `sudo` and similar solutions.

Chapter 10. Random numbers

In this chapter...

This chapter will contain a very brief overview of random numbers in Perl and a selection of Perl's cryptography modules.

Random Numbers

Generating a random number means that before it was produced all possible numbers in the set were equally likely to have been generated. Thus knowledge of all the earlier numbers so far generated should provide no extra information about which number will be generated next.

This statistical independence is utilised in many fields of programming. For example random numbers are used in shuffling algorithms for card games, noise resonance studies in physics, simulations, cryptography and the selection of electronic lottery numbers. If the generated numbers were not random then cryptography keys could be broken, simulations would provide misleading results, and people would be able to predict lottery numbers or cards.

There has been a number of well-documented cases where poor random number generators had been used, often with spectacular results. Examples include being able to break all communications encrypted with older Netscape web-browsers, and the ability to deduce the cards of other players at on-line poker rooms.

Uses of random numbers

In the above section we listed a small number of the many uses of random numbers. In this section we're going to focus on three common uses:

- Cryptography
- Session IDs
- Simulations

Cryptography

Proper random numbers are essential in cryptography. To use an given encryption algorithm effectively you must be able to select a suitable random number as the key. Sometimes there are further requirements on this number, such as it's size or primality, however it's most important to select it in such a way that no one else can easily guess (or find out by brute force) which number you selected.

Pseudo-random number generators should not be used for cryptographic key selection as they are subject to too many attacks. Furthermore, any published random sequence is a poor choice, as are sequential digits in irrational numbers (such as pi). In cryptography random bit streams need to be both random and secret. Using publicly observable phenomena such as stock results or news net posts increase the ease in which the key can be discovered.

Session IDs

Many websites support the idea of user sessions - each user who connects to the site is issued with a unique session ID which is used to identify all subsequent requests made by that user. The session ID allows the server to store data on a per user basis such as their individual preferences or the contents of a shopping cart.

Session IDs are preferable to identify a user when compared to sending a username and password with every request. As session IDs are often short lived (the duration of a single login), the disclosure of a session ID is much less serious than the disclosure of a username and password.

It's important that session IDs are not predictable, as this prevents potential attackers from being able to guess an existing session ID and impersonate the user associated with that session. This risk can be mitigated somewhat by associating a session ID with a particular IP address and/or browser string. However, this can negatively impact the experience of users who have a regularly changing IP address, including mobile users with laptops.

Since session IDs must be unique, it's tempting to base them combine a set of fields that guarantee uniqueness, such as the current time and process-ID for non-persistent programs. However these values are very predictable, and so should be avoided as the sole source of data for session IDs.

To ensure unpredictability we need to add a source of randomness. This randomness can be generated by a pseudo-random number generator, but that will result in a weak session key that's significantly easier to guess. Depending upon your application that may be acceptable, but for properly secure sessiosn you'll want to add a strong source of randomness.

Regardless of what you use to generate your session ID, you should always use hashing function like SHA1 to generate your final session key. This ensures that all randomness is evenly distributed throughout your key, and makes it very difficult for an observer to deduce how that key was originally formed.

Rather than implementing your own Session IDs you may want to consider the excellent `CGI::Session` and `Apache::Session` modules. While at the time of writing (June 2008) these modules do not provide strong session key generators, it is possible to supply new generators generators if required. The following provides an example of creating a new ID generator for `CGI::Session` called `CGI::Session::ID::stronger` which uses `Crypt::Random::Source` to generate random data.

```
# Inside CGI/Session/ID/stronger.pm

# Note that CGI::Session insists on ID generators having a lower-case
# name, hence 'stronger' rather than 'Stronger'.

package CGI::Session::ID::stronger;

use Crypt::Random::Source qw(get_weak);
use Digest::SHA1;

sub generate_id {
        my $sha1 = Digest::SHA1->new;

        # We don't want to block on getting our random values, so
        # we're using get_weak() rather than get_strong() (which can
        # block)

        $sha1->add( get_weak(16) );     # 16 bytes, adjust to taste

        return $sha1->hexdigest;
}

1;
```

```
# ----------------------------------------------------------------------

#!/usr/bin/perl -wT

# Inside our main program.

use CGI::Session;

CGI::Session::IP_MATCH = 1; # Throw exceptions on IP address changes.

# Create a new session, using our strong generator.

my $session = CGI::Session->new("id:stronger",undef);

my $session_id = $session->id;

# Store some data.

$session->param("username",$user);
$session->param("friends",\@posse);
```

Care should be taken whenever replacing a session generator to ensure that your new code is using it correctly.

Simulations

In many fields of engineering and science, computers are used to simulate systems. Examples of such computer experiments are simulation studies of physical processes like atomic collisions and weather, simulation of queueing models in system engineering, and sampling in applied statistics. Alternatively, we can simulate a mathematical model, which cannot be treated by analytical methods.

In all cases a simulation is a computer experiment to determine probabilities empirically. In these applications we require a good source of random numbers. If our numbers aren't random, then the predicted outcomes may be due to artefacts in our simulation, and not meaningful results.

Perl's random numbers (rand and srand)

The Perl `rand` function produces pseudo-random numbers based upon a seed generated by a call to `srand`. Although it is popularly believed that the Perl `rand` function always uses the C `rand` underneath this is actually incorrect. Perl's `rand` is configured when Perl is compiled and uses the first of `drand48()`, `random()` or `rand()` that can be found.

Most programs don't ever need to call `srand` directly. Instead `srand` is called implicitly at the first use of `rand`. `srand` generates its seed based on input from the `/dev/urandom` device (if available) or by combining information from the time of day, process ID and memory allocation.

It should be kept in mind that the only *true* randomness that's provided by Perl's `rand` function is that contained in the seed. You may be using `rand` to generate 128-bit keys, but if your seed only contains 32-bits of randomness, it's more efficient for an attacker to try and brute-force your seed, rather than your key.

If you need a stronger source of randomness, we generally need to look outside of Perl to find it. Many unix systems provide `/dev/random` for strong random bytes (and will block until they are available), and `/dev/urandom` for random bytes that will not result in blocking (and hence may not be as strong).

The `Crypt::Random` and `Crypt::Random::Source` modules can provide an easier and more portable way to generate random data.

Pseudo-random numbers

Any sequence of numbers generated by a deterministic machine, without reference to non-deterministic characteristics or devices, can be at best pseudo-random. Most "random" number generators fall into this category and are clever algorithms which can automatically create long runs of numbers with good random properties. However pseudo-random number generators will repeat the same sequence of numbers when given the same seed, and will also repeat the sequence exactly if sufficient numbers are requested.

The most common pseudo-random number generator (PRNG) is the linear congruential generator which uses the recurrence:

```
f(n+1) = (a * f(n) + c) mod m
```

where `f(n+1)` is our "random" number, `f(n)` is our previous number, and a, c, and m are constants. This generator can generate a maximum of `m` numbers, and is completely deterministic if you know the constants and the previous number generated.

Perl uses the `drand48` function on most systems, and the constants used in `drand48` are well known. As such, an attacker can predict all the future numbers from `rand()` having been given only one of them!

Good sources of random numbers

The generation of good random numbers should use entropy gained from non-deterministic events. Typically, in computers, these are found through monitoring information from hardware. For example, the timing of interrupts from keyboard presses or mouse movements or disk access, thermal noise from resistors and semiconductors and so forth.

The `/dev/random` and `/dev/urandom` devices collect this information and make it available to userspace programs. Not all systems have these devices available.

These devices hold random bytes generated by the kernel random number generator device. This device produces random number from data and devices available to the kernel and estimates the amount of randomness (or entropy) collected from these sources. The `/dev/random` device only returns random bytes when there is a sufficient amount of entropy collected. If the required amount is not available, it blocks until it can fulfil the request.

The bytes retrieved from `/dev/random` are the highest quality random numbers produced by the generator and are suitable for use in generating long term cryptographic keys.

The `/dev/urandom` device returns bytes regardless of the entropy available. Where more data is requested than entropy exists, this device will use a pseudo-random number generator algorithm to supply the required number of bytes. These returned values are theoretically vulnerable to a cryptographic attack and as such are considered to be a lower quality than the values returned by `/dev/random`.

The best way of using these in Perl is to use a module that can locate a good source of randomness for you, such as Vipul Ved Prakash's `Crypt::Random` module, which provides cryptograpically strong random numbers. This allows interfacing with `/dev/random`, `/dev/urandom` and the *entrophy gathering daemon* found on some systems.

```
use Crypt::Random qw(makerandom makerandom_itv);

# Generate a 512-bit strong key.
my $key512 = makerandom( Size => 512, Strength => 1);

# Generate a 128-bit key, using /dev/urandom (or other non-blocking sources).
my $key128 = makerandom( Size => 128, Strength => 0);

# Generate a strong random number between 1 and 6 inclusive.
my $d6 = makerandom_itv(Lower => 1, Upper => 6, Strength => 1);
```

An alternative to `Crypt::Random` is Yuval "nothingmuch" Kogman's `Crypt::Random::Source`, which is demonstrated below:

```
use Crypt::Random::Source qw(get_weak get_strong);

# Generate a 512-bit (64 byte) strong key:

my $key512 = get_strong(64);

# Generate a 128-bit (16 byte) key, using /dev/urandom
# (or another non-blocking source).

my $key128 = get_weak(16);
```

At the time of writing (June 2008), `Crypt::Random::Source` does not provide an in-built mechanism for generating random numbers within a given interval.

Chapter summary

- Random numbers are used in shuffling algorithms, noise resonance studies, simulations, cryptography and many other areas.
- Poor random number generators can often lead to spectacular results.
- Perl's random number generator (rand) isn't very good at producing random numbers.
- Good sources of random numbers uses entropy gained from non-deterministic events.
- The /dev/random/ and /dev/urandom devices provide some access to randomness, as do systems with entropy gathering daemons.
- The `Crypt::Random` and `Crypt::Random::Source` module provides cryptographically secure, true random numbers.

Chapter 11. Conclusion

What you've learnt

Now you've completed Perl Training Australia's Perl Security module, you should be confident in your knowledge of the following fields:

- What is computer security and why you want your programs to be secure.
- How to use taint and untaint your variables.
- What UNIX privileges are and how to drop them in Perl
- How to safely interact with the file system
- How taint checking can be used with DBI to ensure that tainted data either from the user or database isn't passed to the shell.
- How sometimes Perl's eagerness to make doing things easy can cause unexpected insecurities in your code

Where to now?

To further extend your knowledge of Perl, you may like to:

- Work through any material not included during the course
- Visit the websites in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/)
- Extend your knowledge with further Perl Training Australia courses such as:
  - CGI Programming with Perl
  - Database Programming with Perl
  - Object Oriented Perl

  Information about these courses can be found on Perl Training Australia's website (http://www.perltraining.com.au/).

Further reading

Books

- Simson Garfinkel, Gene Spafford and Alan Schwartz, *Practical Unix and Internet Security*, O'Reilly and Associates, 2003, ISBN 0-596-00323-4

- David A Wheeler, *Secure Programming for Linux and Unix HOWTO*, online (http://www.dwheeler.com/secure-programs/), 2003.

Online

- Linux Security Resources (http://www.linuxsecurity.com/docs/) - containing white papers and guides for increasing your system security and handling incidents.
- Linux Security HOWTO (http://www.linuxsecurity.com/docs/LDP/Security-HOWTO/)
- The Perl homepage (http://www.perl.com/)
- The Perl Review (http://www.theperlreview.com/)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup
- Comprehensive Perl Archive Network (http://www.cpan.org)

Chapter 12. Colophon

Cover Art

The `shark` code which appears on the cover of this book was written by Frank Booth. Shark was the US name for the "Triton" enigma machine used on U-Boats and this code implements that machine. As it is a symmetric algorithm it can be used for both encryption and decryption. Enter the input you wish to encrypt on the first run and the input you wish to decrypt on a subsequent run.

```
VUDOT AQ OFJX ONMP MLJ BKW KV KA XO.
EDTMB NM XPSB ISRX ISS OMT LJ UH XP.
ISTJP RZ BUVI KDCY GFM EBV BK FN TF.
JIPLM QP CSYT ONMO LYA HVF PH GA KU.
HAUVM ED LKEH GFEU YJS YBX SK PC XN.
```

can be decrypted to give:

```
THERE IS MORE THAN ONE WAY TO DO IT.
THERE IS MORE THAN ONE WAY TO DO IT.
THERE IS MORE THAN ONE WAY TO DO IT.
THERE IS MORE THAN ONE WAY TO DO IT.
THERE IS MORE THAN ONE WAY TO DO IT.
```

The `shark` code can be found in its native habitat on the PerlMonks website (http://perlmonks.org/index.pl?node_id=126571).

Frank Booth is the patron saint of red cabbage, and more information about him can be found on his homenode (http://perlmonks.org/index.pl?node=frankus).