# Exploits: Buffer Overflows, Pathname Attacks, and SQL Injections

**Created by: Peter A. H. Peterson and Dr. Peter Reiher, UCLA {pahp, reiher}@ucla.edu**

**Contents**



## Overview

The purpose of this exercise is to introduce you to three kinds of common software vulnerabilities and give you a first-hand opportunity to see them in source code, exploit them, and patch them. After successfully completing this exercise, you will be able to:

1. Accurately identify and describe:
    1. Buffer Overflows
    2. Pathname attacks
    3. SQL-Injection attacks

2. Identify the above bugs in preexisting code including:
    1. A small webserver written in C
    2. A CGI script written in Perl
    3. A MySQL-based application written in PHP
3. Understand how vulnerabilities lead to:
    1. Software crashes
    2. Remote execution exploits
    3. Unauthorized access to private data
4. Repair simple examples of these kinds of security holes in the above languages
5. Author memos describing in detail your findings and code changes.

You should be familiar with the Unix command line, POSIX permissions, and basic programming. The exercise will use C, Perl, PHP, HTML, SQL, and HTTP, but at introductory levels.

# Required Reading

## Buffer Overflows

Buffer overflows are a unique kind of occurrence enabled by poor programming in certain languages (for example C, C++, and assembly code) that allow the use of fixed memory buffers for storing data and do not include automatic bounds checking. A buffer is a bounded region of memory into which data can be stored. Buffer overflows result when a buffer is assigned more data than it can hold. The buffer "overflows" into the next available memory space, overwriting the data. Strictly speaking, this is not necessarily an error. However, it has historically been the cause of many bugs and security flaws because so much commonly used code is written in these languages, including compilers, interpreters, and operating systems.

Buffers are different from traditional variables in strongly-typed languages because each genuine type is of a fixed, predetermined size. For example, on most computers an 'int' in C is a 32-bit signed integer. By using the keyword 'int', the programmer has declared to the compiler that this variable and associated memory will never need to exceed 32 bits of storage space, and in fact the compiler ensures that this is not possible. (This is why partly why integers "wrap" when they "overflow" -- the other part is two's complement arithmetic, but that's a lecture for a different class.)

Unfortunately, sometimes strong types are inadequate precisely because you don't know exactly how much memory something is going to require. For example, a user entering their birthplace could be from "Orr, Minnesota" (14 bytes), or they could be from "Llanfairpwllgwyngyllgogerychwyrndrobwyll-llantysiliogogogoch, Wales" (67 bytes). You know it's character data, but how many bytes do you need to reserve to store it? Of course, because memory is finite, we are forced to set some upper bound on the size that we hope will be sufficient.

This is where buffers come in. A buffer is allocated with a specified amount of space (the upper bound) and a type (usually char for character data) and are accessed via a pointer to the first byte of the buffer. Furthermore, allocated buffers will have some positional relationship in physical memory depending on how the compiler chooses to optimize the code. In our place name example, if we allocated a buffer of 60 bytes for "city" and then allocated a buffer of 20 bytes for "country" it is possible that they will be adjacent to each other (or perhaps some other buffers or variables). Unfortunately, programmers often neglect to make sure that input from the outside world (or even

from the program itself) will fit into the declared size of the buffer. Instead, they often think, "There's **no way** that X will *ever* be longer than Y bytes!!!" Of course, perhaps X is *usually* shorter than Y, except when there is an error -- or when someone is intentionally trying to disrupt or compromise your system.

In the strictest sense, a "buffer overflow" is when a buffer of size b is assigned data of size c where c > b. Languages like C and C++ in practice will blithely assign the data c to the memory location b, and this means that whatever memory addresses were after b will now be replaced by the overflow of c. If done unintentionally, this will typically cause bizarre data corruption at the very least and very probably segmentation faults if the overflowed memory is read or dereferenced. This can be used to create denial-of-service attacks by crashing applications remotely with bogus input (e.g., the "Ping of Death").

However, with some careful planning, source code inspection and/or experimentation it is often possible to overwrite the function pointer on the stack of the application, in effect controlling the next "step" the program will take after it finishes reading the new input. This can be used to make the program loop back on itself, or to crash the program. Even more insidiously, it is often possible to dynamically rewrite the running program by including new function code in the data payload you use to overflow the buffer and **then** change the function pointer to point back to the function *you just stuffed into memory*. The program will finish reading the data, consult the newly assigned function pointer, and jump to the code you just provided. This allows you to remotely execute code with the permissions of the running software. Typical exploit payloads will further compromise the system by creating a new user, changing a privileged password, or performing some other action that makes additional compromise easier. This often leads to a root terminal session on the server, after which point the server must be considered completely compromised.

Buffer overflows are an excellent example of why **input validation** is absolutely critical when writing any software. Input validation is the process of programmatically ensuring that all accepted input fits within the logical constraints of the application. This can be as simple as making sure that a social security number has no alphabetic letters (e.g., a fill-in form), or as complex as parsing the input for syntax (e.g., a software compiler).

Ultimately, input validation is a part of the "principle of least privilege." We like to think that our programs can only do what we thought about while we designed them. But in fact, our programs can do (are granted the privilege to do) *whatever their code and environment allow*. Unvalidated input can often modify the behavior of an application, which directly modifies what the program is given the privilege to do. Thus, good programs always validate their input.

#### Additional Reading on Buffer Overflows

For more information, including detailed technical explanations, see:

- Smashing the Stack for Fun and Profit, by AlephOne. This is the canonical HOWTO for buffer overflows.
- Buffer Overflow at Wikipedia
- ...or many other online resources about this very popular attack.

## Pathname Attacks

Filesystem permissions and pathnames are critical entities for any computer system that places a

value on security. If data is to remain secure, filesystem permissions must be properly set and universally obeyed. Perhaps the simplest example of what file permissions are meant to enable is the protection of one user's data from another user. Pathnames are used to address resources on a filesystem, and are usually relative either to the application using the pathname or to the "top level" of the filesystem hierarchy. When combined with system login authentication, filesystem permissions and pathnames create a simple framework for allowing and denying access to various resources.

Unprivileged attackers must typically subvert this system, either by breaking it outright, or by circumventing it. Thanks to the hard work of operating systems designers, It is usually very difficult to successfully *break* the permissions system of modern operatings systems and do something that is explicitly prohibited. Unfortunately, it is often frighteningly easy to violate the *intent* of the system permissions while still making *permissible requests*.

This is loosely analogous to violating our legal "spirit of the law" while still obeying the "letter of the law". While the spirit of the law is infinite, omniscient, and intends to "do the right thing" in every circumstance -- the letter of the law must be finite and pragmatic. As a result, sometimes what might be a crime against the spirit of the law is permissible because of "loopholes" in the letter of the law as it is embodied in the application (e.g., buffer overflows).

A common approach is to use the permissions of an application to take an action on behalf of the attacker. One issue is that while programmers and administrators usually consider `user` permissions, they often overlook `group` & `other` permissions. If a program has been run with certain group permissions, it may have permission to access files completely unrelated to the task at hand. (For example, an application running with `wheel` group access probably has permission to do lots of unrelated things.)

Because of this, some applications run as a special "non privileged" user as a security measure with the express intention of *limiting* its privileges, but it is often the case that in reality the "non privileged" user account still has access to private resources, which is especially bad if the application has a public interface. For example, the Apache web server often runs as the user `www-data` -- which is meant to be a non-login, limited user. However, just like any `other`-class user, the `www-data` user has permission to access any files that are world-readable, such as `/etc/passwd`, many system configuration files, and quite often user home directories! While they may be "world-readable" on the local system, most such files are hardly meant to be seen by the web surfing public.

Also, since the webserver runs as `www-data`, any files or directories the webserver uses or creates must be readable or writeable by the `www-data` user. This also means that all web applications on a server run as the same user! One web application (e.g. a webmail program) may be vulnerable to an attack from a different web application on the same server (e.g, an image gallery program) because they run with the same permissions. The webmail program might be well written without security holes, but a hole in the image gallery might allow a user to see data from the webmail program because they are both `rwx` for the `www-data` user. Things get worse if the image gallery can be used to modify the webmail program.

An even more critical fact that programmers repeatedly overlook is that sometimes applications run *not* with the permissions of the *calling process*, but instead are set to run with the permissions of the *program file's user id* (the file's owner). This is called "setuid" or SUID, and is usually used if the program must do something with root user privileges such as changing a user password. From the perspective of the operating system, a "setuid root" program has the *permission* to do *anything*, and attackers are often able to leverage this permission in unexpected ways.

Pathnames are are also vulnerable to subversion by knowledgeable attackers. While a healthy filesystem hierarchy always has a consistent, canonical representation, there are often infinite ways to express a valid pathname identifying the same resource. Thus, when programs perform **input validation**, they must make sure that they are capable of properly validating all potential pathnames they *could* receive, rather than assuming any arbitrary limit on the form or content of pathnames -- otherwise an attacker only needs to find the one pathname attack that isn't defended.

Ultimately, these vulnerabilities illuminate the fact that a program running on a system is a direct extension of the entity with whose permissions the program is running. Because of this, your mail client is an extension of you and your authentication level -- this is why it can display *your* mail and attach *your* files to messages *you* send. This is both a blessing and a curse; the permissions that allow the mail client to do its job are the same permissions that make it a desirable target for attack.

History shows us that it is often possible for an unauthorized user to subvert a program running with some desired permissions and encourage it to read and present data that would otherwise be inaccessible to the attacker. This is an access control violation in spirit even though the system has performed correctly and thus can lead to privileged information being divulged which can be costly in many different ways.

Therefore, because it is impossible to absolutely trust the underlying system permissions, it is critical in software design to make sure that a program is only able to access exactly what it needs to function properly and nothing more. This is another aspect of the "**principle of least privilege**". However, computers and software systems are typically granted access to anything *not specifically prohibited*, and this usually means that secure software must include code dedicated to enforcing *additional* limitations on the software that the underlying permission system does not impose. (This is like police officers wearing bullet-proof vests because their underlying bodies do not impose restrictions on bullets.)

These additional limitations (such as **input validation**) are especially critical for security if software is able to receive arbitrary input, and often have the beneficial side-effect of making software more robust.

#### Additional Reading

- Directory Traversal at Wikipedia
- Canonicalization at Wikipedia
- Old GNU tar directory traversal vulnerability report
- A nice article on hand-crafting HTTP requests and CGI parameters

## SQL Injection Description

SQL -- or the Structured Query Language -- "is a computer language designed for the retrieval and management of data in relational database management systems, database schema creation and modification, and database object access control management"[2]. Put simply, it's a database query language. SQL (pronounced "sequel" or S-Q-L) has been around since the 1970s, and was standardized by ANSI in 1986. SQL is ubiquitous -- practically all current relational databases in use today speak some vendor-specific variant of SQL, and most enterprise web applications use a relational database on the back end. Furthermore, scripting languages such as Python, Perl, Ruby, and PHP are often used for web development and all have robust, easy to use SQL modules. In fact, the combination of **L**inux servers, the **A**pache http daemon, the **M**ySQL relational database, and the

**P**HP scripting language are so popular for web development today that they have their own acronym -- **LAMP**.

SQL is valuable because it frees programmers from the tedious job of creating a customized data storage system (which is very likely to have its own bugs and shortcomings). Even more valuable is that SQL allows programmers to ignore the particulars of the database internals. Instead, they can use a (mostly) standardized query language to access the data they need, remaining largely independent of the database application, database memory management, and physical storage. This modularity makes maintenance and portability easier.

For example, an online retailer might have a database table for all their inventory with the columns **product number, name, price,** and number in **stock**. The programmer can craft a database query requesting all products whose **name**s' second letter is an "x", filtering out the 10 with the cheapest **price**, and alphabetizing the result. This result will typically be returned from SQL to the programming language as a list (array) or similar data structure. The programmer can then work with the list in her favorite language, make further queries, or return the data to some display function. In executing queries like this, the SQL database, not the application, performs the selection, filtering, and alphabetization.

Big database applications like MySQL are a win, because the performance critical code (the database) is written in a fast language like C/C++, while the interface can be created with a slower, user-friendly scripting language. This is especially important when databases contain millions of entries because the slower scripting languages quickly become prohibitively expensive if they are fully responsible for database mechanics. A related benefit of centralizing databases in this way is that SQL servers can be installed on dedicated hardware and accessed via the network, increasing efficiency and adding other benefits (like easier backups).

As we have already seen in this exercise, simple programming mistakes and omissions often result in unexpected negative security effects. Unfortunately, SQL-based applications are vulnerable to some of these kinds of problems. While generally immune to buffer overflows in user software, PHP is every bit as vulnerable to filesystem and directory traversal exploits as is Perl or Python because these vulnerabilities are a product of the underlying execution environment, not the language being used. Furthermore, these kinds of violations (and other unintended consequences such as crashes) often occur because of a lack of **input validation**. Rather than attempting to verify its validity, input is trusted to be valid and not malicious. Additionally, sometimes valid input still represents a security violation because it **exercises more privilege than the application requires** to properly operate.

Poorly written applications that interface with SQL are no different. A common class of attacks are called "SQL injection attacks" which -- like directory traversal and buffer overflow vulnerabilities -- are the result of trusting non-validated input and implicitly granting applications privilege they do not require. In this case, the non-validated input actually contains SQL statements and relies on the application to naively insert the user input into the application's own request.

For example, imagine a fatally flawed web interface for the Social Security Administration where you enter your Social Security number (123006789) and the system displays a summary of your account and personal information. Using MySQL and PHP, the application might include code like this:

```
$ssn = $_POST['ssn'];                                    // get ssn
from web form POST data
$query = "SELECT * FROM personal WHERE ssn LIKE '$ssn'";   // construct
query (notice embedded $ssn variable)
```

```
$result = mysql_query($query);                                    // execute
query
echo "$result\n";                                                 // output
result
```

After the application constructs the query, the executed SQL statement might look something like this:

```
SELECT * FROM personal WHERE ssn LIKE '123006789'
```

The query is executed, and the result is your personal data.

Now imagine that instead of entering *your* Social Security number, you enter an **SQL wildcard**. The application foolishly takes your input as valid and blithely plugs it in to the SQL statement:

```
SELECT * FROM personal WHERE ssn LIKE '%'
```

This would return the personal information of everyone in the table with a Social Security number!

In reality, this is a dangerously negligent application and a terrible SQL statement for at least two reasons. One reason is an example of **unnecessary privilege** -- since everyone has a unique Social Security number, there is **no chance** that a glob match (enabled by LIKE in SQL) will be required and in fact LIKE allows the wildcard operator to function. At the very least, the above code should have used the exact match test = (equals sign) instead of LIKE, which would have returned an error when it was given a wildcard to match. A more fundamental reason this application is flawed is the total lack of **input validation** -- since all Social Security numbers are 9 digits, there is **no chance** that a legitimate user would ever need to enter anything but 9 digits into the SSN field; punctuation, letters, and everything else should have been immediately flagged as an error before the SQL statement was even assembled.

Unfortunately, strict matching with = is not enough to stop knowledgeable attackers, because there are other ways to fool applications, including escaping SQL characters, using comments, cleverly extending SQL queries, using non-ASCII character sets like Unicode and UTF-8, and many others. The *only* response to this kind of vulnerability is to write code with the least privilege and comprehensive and correct validation of all input.

The above example exploit is extremely simple. The FCCU code and exploit will require multiple steps and a more creative approach.

**Additional Reading on SQL Injection**

For more information, see these articles:

- SQL on Wikipedia
- SQL Injection on Wikipedia
- SQL Injection article by Chris Shiflett

... for additional information, search online -- numerous resources and SQL tutorials exist.

## Software Tools

This section will describe some tools you may need to complete this exercise.

### telnet: cleartext remote shell

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

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

Telnetting to a suspected open port is still one of the fastest ways to see if a service is available or reachable. In this exercise, we'll use telnet to explore the vulnerable HTTP server in Part 1. Here's an example of how to use telnet to access a web page:

```
$ telnet yahoo.com 80

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

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

Connection closed by foreign host.
```

### netcat: a network swiss army knife

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

In this exercise, we will use netcat as a means of sending an attack payload to a vulnerable TCP server. To send a file to the host `receiver` at port `10000`, do something like this:

```
$ cat attackscript.txt | netcat receiver 10000
```

This uses `cat` to read the file `attackscript.txt`, and pipe the data through netcat to the host `receiver` at port 10000.

### CRLF vs. LF newlines: line endings and network protocols

This section describes some important information you will need for creating your exploits.

A file is really just a stream of characters. But humans read "lines" of text. How does the computer know where one line ends and another begins? The answer is that at the end of every line of text is an invisible newline character sequence. Unfortunately, there are two common newline character sequences, and most systems expect one or the other for normal text files. In particular, Windows/DOS tend to use CRLF (carriage return followed by line-feed), while Unix uses LF (line-feed). Some editors or commands create CRLFs at the end of a line, while others just create LFs. Likewise, some network servers expect CRLF, and others expect just LF (and these differences are not Windows or Linux specific). Handling this is easy if you follow these two steps:

1. Figure out what format you need to use for a particular task.
2. Figure out what format your editor creates, and learn how to convert it.

For example, Notepad in Windows adds CRLF to the end of each line. However, Wordpad in Windows only uses LF, and can save in either Unix or DOS format. `vim` on Unix uses LF (Unix-style) line endings by default, but will use CRLF (DOS-style) endings if the file is in DOS mode. You can also force `vim` to use DOS mode by typing:

```
:set fileformat=dos
```

This will use CRLF as the line termination, rather than just LF (the Unix standard). This is important if you are scripting commands for certain protocols, such as HTTP.

#### Scripting Exploits

 HTTP protocol expects CRLF line endings, as opposed to LF newlines.

The "example payload" files on your experimental node are already in DOS mode, so you shouldn't have to do anything to them. However, if you're going to create a new exploit file for this exercise or some other project, you should edit it in DOS mode or otherwise make sure that you are using CRLF as line endings instead of just LF.

### diff and patch: see differences and create source patches

In this exercise, you'll be fixing security vulnerabilities in a few simple programs. However, instead of your whole program, we only want the *differences* between your new, fixed, program, and the original. A file which contains only the *changes* between two revisions of a program is called a "patch." Fortunately, creating patch files for single-file source programs is easy.

To see the differences between two files on Unix, you use the `diff` utility:

```
$ diff one.txt two.txt
```

Another useful tool is called `patch`. `patch` takes properly-formatted `diff` output, and applies the changes to the original file. `diff` can generate this output with a few options:

```
$ diff -Naur oldcode.c newcode.c > fixed.patch
```

`diff` has many options to modify its behavior (see `man diff` for more information).

This above options for `diff` will create a patch with the filenames and all necessary information that the `patch` program requires. This makes patching as simple as executing:

```
$ patch < fixed.patch
```

... and this will create a patched version of the program that you can test.

When submitting a patch file, it is **highly recommended** that you create the patch and then **test it** before submitting it to make sure that it works. You will not get any points for code that does not execute or compile in the exercise environment.

If you're having permissions problems, consider switching to root by executing `sudo su -` or change the permissions of the source directory in question.

**mysql: command line mysql client**

When attackers try to create SQL injection attacks, they often know very little about the the database schema. In our case, we have hands-on access to the database, so this should make the job of developing injection attacks easier. This is where the MySQL command-line client comes in.

To use the MySQL command line on the server, run a command like this:

```
$ mysql -uroot -pzubaz99
```

... `root` is the user and `zubaz99` is the password -- and the lack of spaces is important.

This is essentially an SQL "shell" and gives you root access to the entire database. Once you get logged in to the database, you need to select the database to use. Then, you can make selections from the database:

```
mysql> use fccu; # selects fccu database
mysql> select * from accounts; # print all accounts information
```

If you scroll up, or limit the query, you'll see that `mysql` very nicely adds a title to each column -- this is the column name. So you could create a query like this to display all account information for accounts with an id greater than 50:

```
mysql> select * from accounts where id > 50;
```

There are many online SQL tutorials -- including some on SQL Injection, so we won't cover more here.

When trying to develop an SQL injection attack, you must consider the queries hard-coded into the script -- they are your starting point. You need to find a way to bend the query to your will -- you can't just start over with a query you like better. A good idea is to copy the SQL query from the script (or create one like it) into the mysql command line client and play around with changing it in controlled ways until you can get the result you want.

**Restarting servers: bring a server back to life**

Sometimes, poking and prodding programs makes them die. In particular, your goal in part of this exercise is to cause a process to crash. But how can you know if your method is repeatable if you can't try more than once?

When you kill a server, you can restart it using this command:

```
$ sudo /etc/init.d/servername restart
```

In particular, you will probably need to restart `fhttpd` to verify that your exploit works reliably. You can do this by executing:

```
$ sudo /etc/init.d/fhttpd.init restart
```

... which restarts FrobozzCo's in-house web server. Remember that you can always use `ps`, `grep`, `top`, and other tools to see what processes are running on the system, and you can use the command `kill` to selectively kill processes. (If you're not familiar with those tools, read some basic shell tutorials.)

Finally, there are some cases where a process will die, but its socket will still be tied down in the kernel -- restarting the process may not bring it back immediately -- just wait a few minutes for the kernel to relinquish the port and try again. A second option is to run the server *manually*, on a different port. This actually provides useful debugging information.

#### Running fhttpd Manually

Running a server manually is often a good approach for debugging, because it typically prints logging information directly to the console as it happens, and you will immediately see if the process crashes.

You can start `fhttpd` manually by first opening a new SSH session and logging in to your experimental node. (You need a dedicated window because you won't be able to do any work in the one running the server.) Once you're logged in, change directories into the `/usr/local/fhttpd/` directory and run:

```
$ sudo ./fhttpd portnumber
```

... where `portnumber` is some port number that *isn't* already in use. (I usually use 8081, but it doesn't matter.)

The server will start up, and the output will be directed to your terminal. This is handy, since incoming requests will be displayed to your screen.

You can then use telnet to `telnet localhost portnumber` and attempt issuing HTTP commands from the telnet terminal. In this way, if the server crashes, you will know immediately and you can restart it quickly, choosing a different port if your new port is now "in use".

**HTTP and CGI Protocols: sending commands to servers**

In part of this exercise, you will be attacking a web server. A web server is an application that takes input over TCP port 80, and responds to the client based on the outcome of the request. *The language that web servers speak is known as HTTP -- the Hyper Text Tranfer Protocol.* If a request is successful, the server will provide the requested information along with a success message as defined in the protocol. If the request fails, the server is supposed to respond with an error message detailing why it failed. When trying to see where a bug or vulnerability is in a server application, it is often important to understand the protocol the server is supposed to understand.

RFC 1945 defines a version of HTTP, and is a good (although dense) resource for understanding the HTTP protocol. You can also look at the above section on `telnet`, which provides an example of a very simple web request. If you search online, you should be able to find many other resources describing HTTP.

Another protocol that web servers use is the Common Gateway Interface, or CGI. CGI defines the protocol for passing arguments to and from web applications (that's why many web apps end with the extension "cgi." For example, in Part 2 of this exercise, you will select a memo to view. CGI is used to tell the memo application which memo to open and display. There are many resources online explaining how CGI works and how to hand-craft HTTP requests and CGI parameters -- read some of them to help you experiment with `memo.cgi`.

# Introduction

You are the security administrator for FrobozzCo, a large corporation with a great many secrets. You have just come back from a much-needed four week vacation in West Shanbar, only to find that FrobozzCo has been having some *serious* security issues with 3 of its most important servers! In order to do everything you need, you've prepared a test environment on DETER with the software installed.

# Assignment Instructions

## Setup

This exercise consists of three separate exercises, with three separate submissions. However, everything you need is in one DETER experiment.

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

Make sure that you save your work as you go. See the instructions in the submission section of this exercise for information about save and restore scripts. Save any changes you make to the sourcecode, your patches, memos, etc. in your home directory.

You will probably want to set up port forwarding for tunelling HTTP over ssh so you can test the web applications with a browser on your own desktop.

## Tasks

### Part 1: Buffer Overflows -- The Webserver

FrobozzCo has a longstanding tradition of *reinventing the wheel whenever possible*. As the old saying goes, "Why use something great that someone else made when you can use a mediocre thing you made yourself?" Additionally, the prevailing belief in management is that in-house software is more secure than third-party software since FrobozzCo alone has access to the sourcecode. To that end, when the Great Web Revolution hit and statistics relating to frobnick production were needed by remote facilities, the higher-ups at FrobozzCo insisted that their engineers write a webserver daemon, and it has been dutifully (if unspectacularly) serving web pages for many years.

Unfortunately, it is clear that someone has "rooted" (i.e., gained unauthorized superuser access to) the server; a number of root access files were copied out over the Internet and then the server started sending tons of spam. Fortunately, no data was lost, but the intruder had full control of the server and it is still unknown how they got in. You are convinced there is an exploitable buffer overflow bug in the web server software, but your boss, William H. Flathead III laughed off your suspicions saying, "I wrote the web server software -- and I'd never have made that mistake!"

Nevertheless, he suggests you investigate the possibility of a buffer overflow, "just to be sure."

He asks you to produce a one page memo with an attached **working demo** that targets a specific buffer overflow (should one exist) causing the server to crash. This should be an intentionally exploitable hole in the code and not simply a robustness issue. If you find a vulnerability, your boss wants a patch to fix it. Finally, he wants to know how to clean up this mess -- how severe is this *specific* compromise? How can we restore the system to a safe state?

**Buffer Overflow Tasks**

1. Load your Exploits experiment in DETER.
2. Find the buffer overflow in the `fhttpd` webserver code.
    - The sourcecode is in the directory /usr/src/fhttpd.
3. Exploit the overflow, causing the software to crash.
    - **Please note: you may be able to crash the software in other ways -- we are only specifically interested in a buffer overflow caused by input that is not properly bound-checked.**
    - See RFC 1945, which covers the HTTP protocol. In this document, you can learn about what the HTTP API looks like and how to send commands to web servers manually.
4. Create an executable program (script or binary with sourcecode), demonstrating your exploit. We've created a skeleton exploit in `/root/exploit1.sh` and `/root/payload1`. Edit these with a text editor.
5. Fix the buffer overflow bug in the fhttpd sourcecode and create a patch against the original.
    - To compile the code:
        - `cd /usr/src/fhttpd`
        - `sudo make`
    - To install and run the code:
        - `sudo make install` (this copies in your new binary and restarts the default server on port 8080)
        - See the tools section of this document for instructions on `diff`,`patch`, and other utilities.
        - See also the section on restarting servers.
6. Write a ~1 page memo:
    1. Describe the security flaw you found, how you fixed it, and how your demo exploit works. (The memo itself should quote as little sourcecode as possible; for longer sections, refer to filenames and line numbers in the original or your attached patch.)
    2. Considering fhttpd alone, include in your memo:
        1. An evaluation of the seriousness of the breach
        2. A recovery plan for the server. (Is it enough to fix the flaw? Why or why not?)
        3. Any other observations or thoughts you might have.
7. Put the following files into in `/root/part1`:
    1. your memo
    2. your working demo with instructions
    3. your patch
8. Use the scripts described in the section for creating a submission tarball.

## Part 2: Common Filesystem Exploits -- The Memo Software

Something else is rotten in the state of FrobozzCo, and this time it *is* an inside job. FrobozzCo has an internal server that it uses for disseminating official company memoranda. All employees have access to this server via a web interface, but it is not reachable from the Internet without going through a firewall, and the only accesses in the logs have come from internal addresses. Somehow, for the last several weeks, user accounts on the server have been getting hijacked one by one. It seems clear: someone obtained and is "brute forcing" the password file and is using the newfound access to read private data. How could this be happening?

Traditional UNIX systems kept their hashed passwords in `/etc/passwd` where they were encrypted (but world-readable). This made it trivial for an unprivileged user to feed the password file into a password cracker (sometimes on the same machine!) and obtain other authentication -- sometimes root or other privileged accounts. Modern unices combat this vulnerability by keeping the password portion in a file called `/etc/shadow`, where only root has access to it. Your system uses an `/etc/shadow` file, so you know that regular user accounts do not have access to password information. Furthermore, there is only one piece of in-house software on this computer, and it is not written in a language like C that is susceptible to buffer overflows; it is written in a scripting language with built-in memory management that is not vulnerable to that kind of attack. Yet, somehow someone has been able to read private data files like `/etc/shadow` using the memorandum software. This is bad.

You are sure it must be somehow related to the memo viewing application, because all third-party software is up to date. The memo system works like this: users can log into the system via ssh or access the system via NFS and can publish memos by writing files into the `memo` directory in their home directories. Then, the memo-reading CGI (which is setuid root ostensibly so it can read multiple users' `memo` directories) searches the `memo` directories and publishes a list of memos available for reading. A user can then use a web interface to select one of the memos from a list and read it.

Your boss, William H. Flathead III is skeptical that the memo reader is the problem, because he wrote the memo reading program when he was a summer intern 15 years ago. Still, just in case, he asks you to produce a 1 page memo, **working demo** of the exploit, and a patch for the memo reading software, should a vulnerability exist. Either way, he also wants to know how to clean up this mess -- how severe is the compromise? How can we restore the system to a safe state?

### Pathname Attack Tasks

1. Load your Exploits image in DETER. (You don't need to reload it if it is already active.). Note: this attack needs to be launched against Apache webserver since fhttpd cannot serve binary content.
2. Find the directory traversal vulnerability in the memo.cgi code.
   - The sourcecode is located at /usr/lib/cgi-bin/memo.cgi
   - If you have set up ssh tunneling via port **80** (a good idea), the memo application can be accessed at http://localhost/cgi-bin/memo.cgi
3. Exercise a remote vulnerability in memo.cgi to read the private file /etc/shadow
4. Create an executable program demonstrating your exploit. Your program should display or save /etc/shadow. We've created a skeleton exploit script in `/root/exploit2.sh`. Edit it with your favorite text editor.

5. Fix the flaw and create a patch of your new code against the original. Your fix should add input validation and make `memo.cgi` non SUID-root.
    - This will force you to redesign the application's back end, such as moving directories, changing permissions, creating groups, possibly changing how or where users create/edit memos, etc., but should not change the appearance or general function of the application.
    - Users should not be able to delete each others' memos, but the memos themselves are not private.
6. Write a ~1 page memo:
    1. describe the security flaw you found, how you fixed it, and how your demo exploit works. The memo itself should quote as little sourcecode as possible; for longer sections, refer to filenames and line numbers in the original or your attached source.
    2. Considering memo.cgi alone, include in your memo:
        1. A recovery plan for the server, answering: How serious was this breach? What should be done with the server in order to secure it?
        2. An explanation of why it **is not** sufficient to simply make sure that the pathnames start with `/home/username/memo/` or `/root/memo/`
        3. Any other observations or thoughts you might have.
7. Store the following files in `/root/part2`:
    1. your memo
    2. your working demo with instructions
    3. your patch
8. Use the scripts described in the section for creating a submission tarball.

**Part 3: SQL Injection -- FrobozzCo Credit Union**

FrobozzCo has its own internal company credit union, FrobozzCo Community Credit Union (FCCU). FCCU has an Internet-accessible web-based application that allows employees to access their paychecks and pay bills via a money wiring system. There are very few bank employees, and they use a a special administrative interface that runs on a different system that is not network accessible. In true FrobozzCo fashion, the public banking software was written in house by the CTO's nephew (who is a nice kid but not the brightest candle on the cake).

As it turns out, a *lot* of money has been disappearing from the credit union while you've been gone. It looks like someone has figured out how to force other accounts to wire money... to an anonymous bank account in the Cayman Islands! Worse yet, several employees have had serious identity theft problems of late -- clearly someone has access to personal information and you have a hunch it's all coming from this server. To top it all off, the company itself is showing a deficit of $32,767 and it looks like it was somehow drawn through FCCU.

In a surprising display of foresight, your predecessor installed a network monitor watching the FCCU server. However, you are shocked to find out (from the network monitor and server logs) that *nobody* has logged into the server directly -- in fact, the only interaction that anyone has had with the server has come through the Internet facing web interface. It looks like insecure software is to blame, again.

You assume that there must be one or more vulnerabilities in the code that interfaces with the SQL database -- in the FCCU software, the directory, or both -- and that somehow a malicious

user is able to make the system do something it's not supposed to, like write checks. Worse yet, it seems like the attacker has managed to view private information like social security numbers, birthdates, and so on. You've heard about a class of attacks called "SQL Injection," and it seems likely that this is the kind of exploit being used.

Surprisingly, your boss agrees with you and instructs you to produce a one page memo, a **detailed transcript** demonstrating the exploit, and a patch for the software. Additionally, he also wants to know how to clean up this mess -- how severe is the compromise? How can we restore the system to a safe state?

**SQL Injection Tasks**

1. Load your Exploits image in DETER. (You don't need to reload it if it is already active.) Note: this attack needs to be launched against Apache webserver since fhttpd cannot serve binary content.
    - The sourcecode is located at /usr/lib/cgi-bin/FCCU.php
    - If you have set up ssh tunnelling via port **80** (a good idea), the memo application can be accessed at http://localhost/cgi-bin/FCCU.php
2. Exercise a remote SQL-Injection vulnerability to perform these unauthorized tasks on the SQL server:
    1. Show how you can log into a single account without knowing any id numbers ahead of time.
    2. Show how you can log into **any** account you like (without knowing any id numbers ahead of time).
    3. Make some account (your choice) wire its total balance to the bank with routing number: **314159265** and account number: **271828182845**
    4. Explain why you can't create a new account or arbitrarily update account balances (or show that you can).
3. Create an exploit transcript in the file `/root/exploit3.txt`, including your answers and how you completed the above attacks.
    - document all SQL Injection strings you used, including which pages and in what order.
    - This is **not** an *executable* script, but should be a step-by-step "walkthrough" of the attack that a colleague could follow without assistance.
4. Fix the vulnerability in the FCCU application by adding input validation and either character escaping or prepared statements.
5. Create a patch against the original source.
6. Quoting as little sourcecode as possible, write a ~1 page memo, including:
    1. A description of the security flaw in the FCCU application
    2. A description of how you fixed the flaw. How does your fix solve the problem?
    3. Considering the FCCU application alone, describe a recovery plan for the server, answering:
        1. How serious was this breach? Could attackers gain root access through this vulnerability?
        2. What should be done with the server in order to secure it?
        3. Include any other observations or thoughts you might have.
7. Store the following files in `/root/part3`:

    1. your memo
    2. your exploit walkthrough
    3. your patch
8. Use the scripts described in the section for creating a submission tarball.

### What can go wrong

Some helpful advice:

- For Part 1, make sure you are attacking `fhttpd` on port 8080 (or whatever port you are manually running it on) and **not** Apache, which is running on port 80. (Apache's vulnerabilities are much harder to find.)
- For Part 2 and Part 3 make sure you are launching attacks against Apache webserver. The vulnerabilities we target are not in the Apache code.

# Extra Credit

### Remote Execution

For Part 1, the assigned task is to simply crash `fhttpd` with an attack payload. However, it is possible to inject code via the payload and not crash `fhttpd` but take over the application. Since `fhttpd` runs with root privileges, you can execute anything on the system if you manage to take over the program. For extra credit, do #1 and/or #2 below, and then #3:

1. Create and successfully execute a remote execution exploit using the vulnerability in `fhttpd`. This could be as simple as writing a file to the system.
2. 0wn the box: Create a remote shell you can send commands to, add a login user with root access, or some other reliable remote access facility.
3. Draft a short writeup describing the steps you took to create the exploit and what you learned in the process.

# Submission Instructions

For this exercise, you will submit **3 tarballs**; one for the buffer overflow, another for the pathname attack, and the third for the SQL injection. Separating the work into three tarballs makes it easier to deal with the swapin/out nature of DETER.

Accordingly, there are 6 scripts, submit[1-3].sh and restore[1-3].sh in /root/ on the `exploits` host for creating and restoring those tarballs. In contrast to the permissions exercise, the restore scripts do not have any options -- they will automatically overwrite the particular files they are meant to restore. So make sure that you really want to restore your files before you do it!

# submit1.sh and restore1.sh

`submit1.sh` will back up:

- the sourcecode directory `/usr/src/fhttpd/`

- `/root/exploit1.sh`
- `/root/payload1`
- the `/root/part1` directory (where you should put your memo for part 1)

`restore1.sh` will restore those files to their original locations, automatically overwriting whatever is there.

The scripts for parts 2 and 3 work like submit1.sh and restore1.sh, with the following exceptions:

- They back up/restore `/usr/lib/cgi-bin/memo.cgi` or `/usr/lib/cgi-bin/fccu.php` instead of the fhttpd sourcecode directory.
- They back up/restore the appropriate `partN` directory
- part 2 does not require a payload
- for part 3, instead of exploit3.sh and payload3 there is a transcript exploit3.txt for you to fill in.

Submit all three files to your instructor.