

Computer Security

04. Command Injection Attacks & Pathname Parsing

Paul Krzyzanowski

Rutgers University

Spring 2019

Last week, we looked at ...

Attacks

- Buffer overflows
 - Stack overflow & return address override
 - Off-by-one overflow & frame pointer override
 - Heap overflow & data or function pointer corruption
- printf attacks
 - If you have the ability to set the format string

Last week, we looked at ...

Defenses

- **Programming languages with bounds checks & strong typing**
 - Use "safe" functions in C/C++
 - Java, C# – Python is vulnerable in some areas
 - But native methods might be vulnerable
- **Data execution protection (DEP)**
no-execute memory pages for stack & heap
 - Attacks: return-to-libc or Return-Oriented-Programming attacks
- **Address Space Layout Randomization (ASLR)**
 - Attacks:
 - not all programs or libraries use ASLR
 - **NOP sled** – create a huge block of NOPs to increase chance of jumping to exploit
 - Try and try again if there isn't much entropy in the randomization
- **Stack canaries**
 - Attack: if canary is modified, the compiler causes an exception. If you can modify the exception handler, it can point to your code: **Structured Exception Handling** (SEH) exploit.

Security-Sensitive Programs

- Control hijacking isn't interesting for regular programs on your system
 - You might as well run commands from the shell
- It is interesting if the program
 - Has escalated privileges (*setuid*), especially root
 - Runs on a system you don't have access to (most servers)

Privileged programs are more sensitive & more useful targets

Injection attacks

- Injection is rated as the #1 software vulnerability in 2017 by the Open Web Application Security Project (OWASP)
- Allows an attacker to inject code into a program or query to
 - Execute commands
 - Modify a database
 - Change data on a website
- We looked at buffer overflows and format strings
... but there are other forms too

https://www.owasp.org/index.php/Top_10-2017_Top_10

Bad Input: SQL Injection

- Let's create an SQL query in our program

```
sprintf(buf,  
        "SELECT * WHERE user='%s' AND query='%s';",  
        uname, query);
```

- You're careful to limit your queries to a specific user
- But suppose *query* comes from user input and is:

```
foo' OR user='root
```

- The command we create is:

```
SELECT * WHERE user='paul' AND query='foo' OR user='root';
```

What's wrong?

- We should have used *snprintf* to avoid buffer overflow (but that's not the problem here)
- We didn't validate our input
 - And ended up creating a query that we did not intend to create!

Another example: password validation

- Suppose we're validating a user's password:

```
sprintf(buf,  
"SELECT * from logininfo WHERE username = '%s' AND password = '%s';",  
uname, passwd);
```

- But suppose the user entered this for a password:

' OR 1=1 --

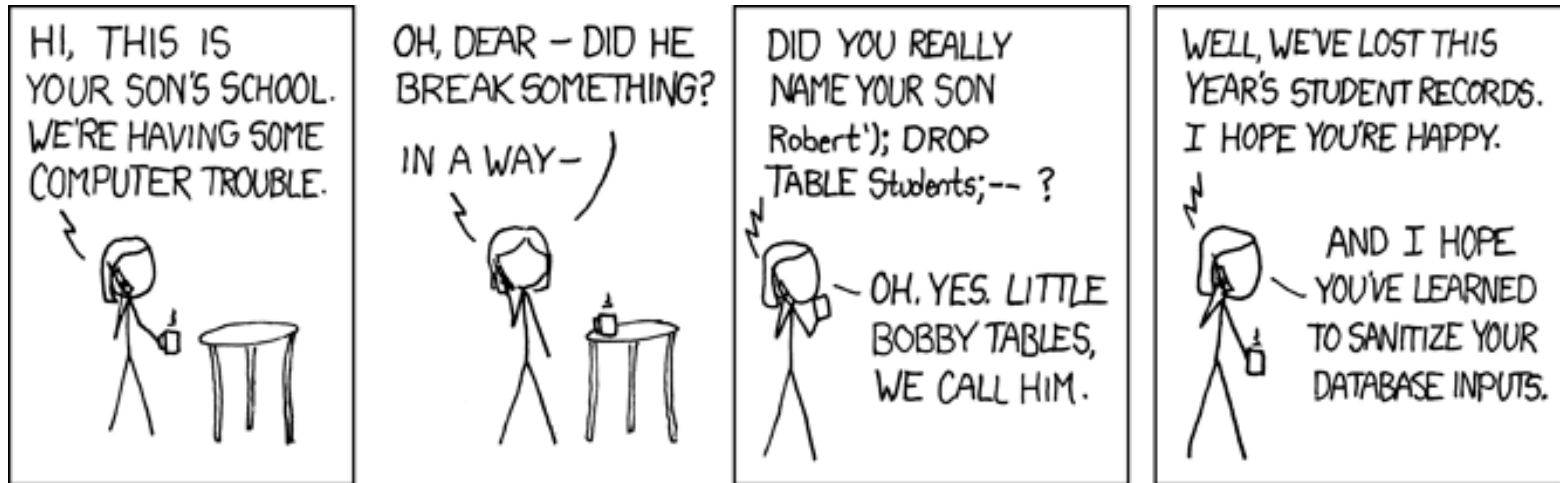
The -- is a comment that blocks the rest of the query (if there was more)

- The command we create is:

```
SELECT * from logininfo WHERE username = paul AND  
password = ' ' OR 1=1 -- ;
```

1=1 is always true!

Opportunities for destructive operations



<https://xkcd.com/327/>

Most databases support a batched SQL statement: multiple statements separated by a semicolon

```
SELECT * FROM students WHERE name = 'Robert';DROP TABLE Students; --
```

Protection from SQL Injection

- SQL injection attacks are incredibly common because most web services are front ends to database systems
 - Input from web forms becomes part of the command
- Type checking is difficult
 - SQL contains too many words and symbols that may be legitimate in other contexts
 - Use **escaping** for special characters
 - Replace single quotes with two single quotes
 - Prepend backslashes for embedded potentially dangerous characters (newlines, returns, nuls)
 - Escaping is error-prone
 - Rules differ for different databases (MySQL, PostgreSQL, dashDB, SQL Server, ...)

Don't create commands with user substrings added into them

Protection from SQL Injection

- Use parameterized SQL queries or stored procedures
 - Keeps query consistent: parameter data never becomes part of the query string

```
uname = getResourceString("username");  
passwd = getResourceString("password");  
query = "SELECT * FROM users WHERE username = @0 AND password = @1";  
db.Execute(query, uname, passwd);
```

General Rule

- If you invoke any external program, know its parsing rules
- Converting data to statements that get executed is common in some interpreted languages
 - Shell, Perl, PHP, Python

IFS

Shell variable IFS (Internal Field Separator) defines delimiters used in parsing arguments

- If you can change IFS, you may change how the shell parses data
- The default is space, tab, newline

try1.sh

```
#!/bin/bash
while read name password; do
    echo name=\"$name\", password=\"$password\"
done
```

names

```
james password
mary 123456
john qwerty
patricia letmein
robert shadow
jennifer harley
```

output

```
$ ./try1.sh <names
name="james", password="password"
name="mary", password="123456"
name="john", password="qwerty"
name="patricia", password="letmein"
name="robert", password="shadow"
name="jennifer", password="harley"
```

IFS

One small change: **IFS=+**

try1.sh

```
#!/bin/bash
IFS=+
while read name password; do
    echo name=\"$name\", password=\"$password\"
done
```

names

```
james password
mary 123456
john qwerty
patricia letmein
robert shadow
jennifer harley
```

output

```
$ ./try1.sh <names
name="james password", password=""
name="mary 123456", password=""
name="john qwerty", password=""
name="patricia letmein", password=""
name="robert shadow", password=""
name="jennifer harley", password=""
```

IFS

It gets tricky for output

`try.sh`

```
#!/bin/bash
```

```
IFS='+'
```

```
echo "$@" expansion'  
echo "$@"
```

```
echo "$*" expansion'  
echo "$*"
```

```
$ ./try.sh sleepy sneezy grumpy dopey doc  
"$@" expansion  
sleepy sneezy grumpy dopey doc  
"$*" expansion  
sleepy+sneezy+grumpy+dopey+doc
```

You really have to know what you're dealing with!

Suppose a program wants to send mail. It might call:

```
FILE *fp = popen("/usr/bin/mail -s subject user", "w")
```

If **IFS** is set to " / " then the shell will try to execute **usr bin mail...**

An attacker needs to plant a program named "usr" anywhere in the search path

system() and popen()

- These library functions make it easy to execute programs
 - *system*: execute a shell command
 - *popen*: execute a shell command and get a file descriptor to send output to the command or read input from the command
- These both run `sh -c command`
- Vulnerabilities include
 - Altering the search path if the full path is not specified
 - Changing IFS to change the definition of separators
 - Using user input as part of the command

```
snprintf(cmd, "/usr/bin/mail -s alert %s", bsize, user);  
f = popen(cmd, "w");
```

What if `user = "paul;rm -fr /home/*"`

```
sh -c "/usr/bin/mail -s alert paul; rm -fr /home/*"
```


Other environment variables

- **PATH**: search path for commands
 - If untrusted directories are in the search path before trusted ones (`/bin`, `/usr/bin`), you might execute a command there.
 - Users sometimes place the current directory (`.`) at the start of their search path
 - What if the command is a booby-trap?
 - If shell scripts use commands, they're vulnerable to the user's path settings
 - Use absolute paths in commands or set `PATH` explicitly in a script
- **ENV, BASH_ENV**
 - Set to a file name that some shells execute when a shell starts

Other environment variables

LD_LIBRARY_PATH

- Search path for shared libraries
- If you change this, you can replace parts of the C library by custom versions
 - Redefine system calls, *printf*, whatever...

LD_PRELOAD

- Forces a list of libraries to be loaded for a program, even if the program does not ask for them
- If we preload our libraries, they get used instead of standard ones

You won't get root access with this but you can change the behavior of programs

- Change random numbers, key generation, time-related functions in games
- List files or network connections that a program does
- Modify features or behavior of a program

Example of LD_PRELOAD

random.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
    int i;

    srand(time(NULL));
    for (i=0; i < 10; i++)
        printf("%d\n", rand()%100);
    return 0;
}
```

```
$ gcc -o random random.c
$ ./random
9
57
13
1
83
86
45
63
51
5
```

Let's create a replacement for rand()

rand.c

```
int rand() {  
    return 42;  
}
```

```
$ gcc -shared -fPIC rand.c -o newrandom.so # compile  
$ export LD_PRELOAD=$PWD/newrandom.so # preload  
$ ./random
```

42
42
42
42
42
42
42
42
42
42

We didn't have to recompile *random*!

Function interposition

interpose

(ĭn'tər-pōz')

1. Verb (transitive)

to put someone or something in a position
between two other people or things

*He swiftly interposed himself between his visitor
and the door.*

2. To say something that interrupts a conversation

- Change the way library functions work without recompiling programs
- Create wrappers for existing functions

File Descriptors

- On POSIX systems
 - File descriptor 0 = standard input (*stdin*)
 - File descriptor 1 = standard output (*stdout*)
 - File descriptor 2 = standard error (*stderr*)
- *open()* returns the first available file descriptor

Vulnerability

- Suppose you close file descriptor 1
- Invoke a *setuid* root program that will open some sensitive file for output
- Anything the program prints to *stdout* (e.g., via *printf*) will write into that file, corrupting it

File Descriptors - example

files.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
    int fd = open("secretfile", O_WRONLY|O_CREAT, 0600);

    fprintf(stderr, "fd = %d\n", fd);
    printf("hello!\n");
    fflush(stdout); close(fd);
    return 0;
}
```

```
$ ./files
fd = 3
hello!
$ ./files >&-
fd = 1
```

Bash command to close a file descriptor
We close the standard output

Obscurity

Windows CreateProcess function

```
BOOL WINAPI CreateProcess(  
    _In_opt_      LPCTSTR          lpApplicationName,  
    _Inout_opt_  LPTSTR           lpCommandLine,  
    _In_opt_      LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    _In_opt_      LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    _In_          BOOL              bInheritHandles,  
    _In_          DWORD              dwCreationFlags,  
    _In_opt_      LPVOID            lpEnvironment,  
    _In_opt_      LPCTSTR           lpCurrentDirectory,  
    _In_          LPSTARTUPINFO      lpStartupInfo,  
    _Out_         LPPROCESS_INFORMATION lpProcessInformation);
```

- 10 parameters that define window creation, security attributes, file inheritance, and others...
- It gives you a lot of control but do most programmers know what they're doing?

Pathname parsing

App-level access control: filenames

- If we allow users to supply filenames, we need to check them
- App admin may specify acceptable pathnames & directories
- Parsing is tricky
 - Particularly if wildcards are permitted (*, ?)
 - And if subdirectories are permitted

Parsing directories

- Suppose you want to restrict access outside a specified directory
 - Example, ensure a web server stays within `/home/httpd/html`
- Attackers might want to get other files
 - They'll put `..` in the pathnaame
 - `..` is a link to the parent directory

For example:

`http://pk.org/../../../../etc/passwd`

- The `..` does not have to be at the start of the name – could be anywhere

`http://pk.org/419/notes/../../../../416/../../../../etc/passwd`

- But you can't just search for `..` because an embedded `..` is valid

`http://pk.org/419/notes/some..junk..goes..here/`

- Also, extra slashes are fine

`http://pk.org/419////notes///some..junk..goes..here///`

Basically, it's easy to make mistakes!

Application-Specific Syntax: Unicode

Here's what Microsoft IIS did

- Checked URLs to make sure the request did not use `../` to get outside the *inetpub* web folder

Prevents attempts such as

`http://www.pk.org/scripts/../../../../winnt/system32/cmd.exe`

- Then it passed the URL through a decode routine to decode extended Unicode characters
- Then it processed the web request

What went wrong?

Application-Specific Syntax: Unicode

- What's the problem?
 - / could be encoded as unicode `%c0%af`
- UTF-8
 - If the first bit is a 0, we have a one-byte ASCII character
 - Range 0..127
 - `/ = 47 = 0x2f = 0010 0111`
 - If the first bit is 1, we have a multi-byte character
 - If the leading bits are 110, we have a 2-byte character
 - If the leading bits are 1110, we have a 3-byte character, and so on...
 - 2-byte Unicode is in the form `110a bcde 10fg hijk`
 - 11 bits for the character # (codepoint), range 0 .. 2047
 - C0 = 1100 0000, AF = 1010 1111 which represents 0x2f = 47
 - Technically, two-byte characters should not process # < 128
 - ... but programmers are sloppy ... and we want the code to be fast

Application-Specific Syntax: Unicode

- Parsing ignored `%c0%af` as `/` because it shouldn't have been one
- So intruders could use IIS to access *ANY* file in the system
- IIS ran under an IUSR account
 - Anonymous account used by IIS to access the system
 - IUSER is a member of *Everyone* and *Users* groups
 - Has access to execute most system files, including `cmd.exe` and `command.com`
- A malicious user had the ability to execute any commands on the web server
 - Delete files, create new network connections

Parsing escaped characters

Even after Microsoft fixed the Unicode bug, another problem came up

- If you encoded the backslash (`\`) character
(Microsoft uses backslashes for filenames & accepts either in URLs
... and then encoded the encoded version of the `\`, you could bypass the security check

`\` = `%5c`

- `%` = `%25`
- `5` = `%35`
- `c` = `%63`

For example, we can also write:

- `%%35c` \Rightarrow `%5c` \Rightarrow `\`
- `%25%35%63` \Rightarrow `%5c` \Rightarrow `\`
- `%255c` \Rightarrow `%5c` \Rightarrow `\`

Yuck!

http://help.sap.com/SAPHELP_NWPI71/helpdata/en/df/c36a376a3a43ceaaa879ab726f0ec8/content.htm

These are application problems

- The OS uses whatever path the application gives it
 - It traverses the directory tree and checks access rights as it goes along
 - “x” (search) permissions in directories
 - Read or write permissions for the file
- The application is trying to parse a pathname and map it onto a subtree
- Many other characters also have multiple representations
 - á = U+00C1 = U+0041,U+0301

Comparison rules have to be handled by applications and be application dependent

More Unicode issues

Unicode represents virtually all the worlds glyphs

- Some symbols look the same (or similar) but have different values

Potential for deception

They're totally different to software but look the same to humans

/ = solidus (slash) = U+002F

/= fraction slash = U+2044

/ = division slash = U+2215

/ = combining short solidus overlay = U+0337

/ = combining long solidus overlay = U+0338

/ = fullwidth solidus = U+FF0F

Yuck!

Access check attacks

Setuid file access

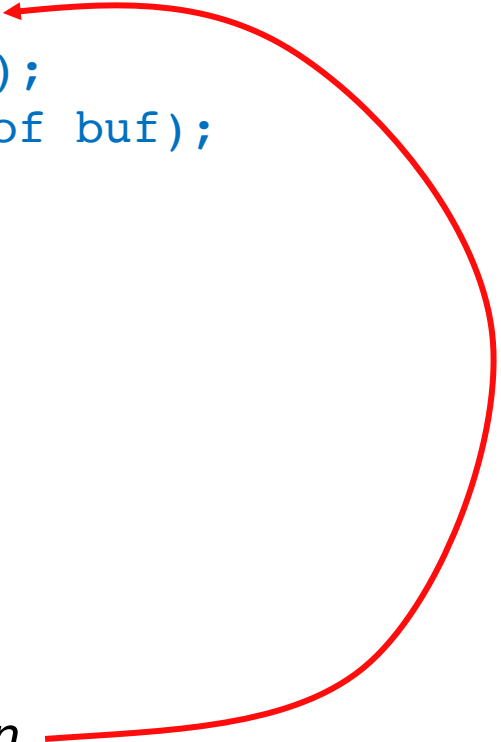
Some commands may need to write to restricted directories or files but also access user's files

- Example: some versions of *lpr* (print spooler)
 - Read users' files and write them to the spool directory
- Let's run the program as *setuid* to *root*
But we will check file permissions first to make sure the user has read access

```
if (access(file, R_OK) == 0) {  
    fd = open(file, O_RDONLY);  
    ret = read(fd, buf, sizeof buf);  
    ...  
}  
else {  
    perror(file);  
    return -1;  
}
```

Problem: TOCTTOU

```
if (access(file, R_OK) == 0) {  
    fd = open(file, O_RDONLY);  
    ret = read(fd, buf, sizeof buf);  
    ...  
}  
else {  
    perror(file);  
    return -1;  
}
```




- Race condition:
TOCTTOU: Time of Check to Time of Use
- Window of time between *access* check & *open*
 - Attacker can create a link to a readable file
 - Run *lpr* in the background
 - Remove the link and replace it with a link to the protected file
 - The protected file will get printed

mktemp is also affected by this race condition

Create a temporary file to store received data

```
if (tmpnam_r(filename)) {  
    FILE* tmp = fopen(filename, "wb+");  
    while((recv(sock, recvbuf, DATA_SIZE, 0) > 0) && (amt != 0))  
        amt = fwrite(recvbuf, 1, DATA_SIZE, tmp);  
}
```



- API functions to create a temporary filename
 - C library: *tmpnam*, *tempnam*, *mktemp*
 - C++: *_tmpnam*, *_tempnam*, *_mktemp*
 - Windows API: *GetTempFileName*
- They create a unique name when called
 - But no guarantee that an attacker doesn't create the same name before the filename is used
 - Name often isn't very random: high chance of attacker constructing it

From https://www.owasp.org/index.php/Insecure_Temporary_File

mktemp is also affected by this race condition

If an attacker creates that file first:

- Access permissions may remain unchanged for the attacker
 - Attacker may access the file later and read its contents
- Legitimate code may append content, leaving attacker's content in place
 - Which may be read later as legitimate content
- Attacker may create the file as a link to an important file
 - The application may end up corrupting that file
- The attacker may be smart and call *open* with `O_CREAT` | `O_EXCL`
 - Or, in Windows: `CreateFile` with the `CREATE_NEW` attribute
 - Create a new file with exclusive access
 - But if the attacker creates a file with that name, the *open* will fail
 - Now we have *denial of service* attack

From https://www.owasp.org/index.php/Insecure_Temporary_File

Defense against mktemp attacks

Use *mkstemp*

- It will attempt to create & open a unique file
- You supply a template
A name of your choosing with xxxxxx that will be replaced to make the name unique

```
mkstemp("/tmp/secretfileXXXXXX")
```
- File is opened with mode 0600: r-- --- ---
- If unable to create a file, it will fail and return -1
 - You should test for failure and be prepared to work around it.

The main problem: *interaction*

- To increase security, a program must minimize interactions with the outside
 - Users, files, sockets
- All interactions may be attack targets
- Must be controlled, inspected, monitored

Relative Attack Surface Quotient (RASQ)

- Microsoft metric of application vulnerability
 - Attempts to mathematically quantify the attackability of software
- Roughly, measures # of input channels
 - Some channels are easier to exploit
 - Some channels are more accessible to others
- Sum of “**effective attack surface values**” for all “**root attack vectors**”

Root attack vector	feature that can positively or negatively affect the security of a product
Attack bias	value representing risk of compromise for an attack <ul style="list-style-type: none">• Subjective measure: 0=no threat, 1=maximum threat
Attack surface	targets for an attacker - # of things that can be attacked <i>Sum of attack vectors</i>
Effective attack surface value	Product of the {# of attack surfaces within a root attack vector} and the {attack bias}

<https://www.microsoft.com/windowsserver2003/docs/AdvSec.pdf>

RASQ Sample root vectors & bias values

Root vector	Bias value	Comment
Open sockets	1.0	Every open & listening socket is a potential target
Open RPC endpoints	0.9	Like sockets but require more skill
Enabled accounts	0.7	Default accounts simplify brute-force password attacks
Enabled accounts in the Administrator group	0.9	Admin accounts are higher risk
Weak ACLs in file system	0.2	Most files in the system are targeted after a system is compromised
Weak ACLs on file shares	0.9	Default shares are commonly known and often targeted

<https://www.microsoft.com/windowsserver2003/docs/AdvSec.pdf>

Summary

- Better OSes, libraries, and strict access controls would help
 - A secure OS & secure system libraries will make it easier to write security-sensitive programs
 - Enforce principle of least privilege
 - Validate all user inputs ... and try to avoid using user input in commands
- Reduce chances of errors
 - Eliminate unnecessary interactions (files, users, network, devices)
 - Use per-process or per-user /tmp
 - Avoid error-prone system calls and libraries
 - Or study the detailed behavior and past exploits
 - Minimize comprehension mistakes
 - Specify the operating environment & all inputs
 - And validate or set them at runtime: PATH, LD_LIBRARY_PATH, user input, ...
 - Don't make user input a part of executed commands

The end