CS52500 Project 1 report:
Name: Hrishikesh Deshpande (hdeshpa@purdue.edu)

Analysis of file vulnerable1.c

Code:

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

#define BUFFER_SIZE 200

void launch(char * user_argument)
{
      char buffer[BUFFER_SIZE];
      strcpy(buffer, user_argument);
}

int main(int argc, char * argv[])
{
      if (argc != 2)
      {
            fprintf(stderr, "Usage: %s ARGUMENT\n", argv[0]);
            exit(EXIT_FAILURE);
      }

      launch(argv[1]);
      return(0);
}
```

1. Briefly describe the behavior of the program:

This program accepts one input parameter as a command line argument which will be stored in argv[1]. The programs errors out with failure in case no argument was specified during program execution.

The accepted argument in argv[1] is then passed from main() to another function launch(char *). The launch() function declares a local variable “buffer” which is a character buffer of fixed size BUFFER_SIZE. The BUFFER_SIZE constant is defined to be equal to 200 using a #define macro.

The launch function perform a simple string copy. It copies the string pointed to by the char* parameter “user_argument into the buffer using the “strcpy” function and returns back to main(). main() then executes a return(0) and the program terminates.

2. Identify and describe the vulnerability as well as its implications:
The vulnerability in the program is due to usage of the unsafe “strcpy” library call in the launch() function as shown below:
strcpy (buffer, user_argument);

The issue is that strcpy is an unsafe function that does not perform any bounds checking on the size of data being copied into the target buffer array. Since the local buffer is of fixed size BUFFER_SIZE (200), its possible to overflow this buffer by supplying a user argument which is a string of length greater than or equal to BUFFER_SIZE (excluding null character in the length).

If a string of length greater than BUFFER_SIZE is supplied as a user argument, then strcpy continues to copy into the char buffer beyond its actual size until the terminating NULL character is encountered. This overflows the buffer. Since the buffer is declared as a local variable on the stack, the extra overflow characters can overwrite the return address on the stack.

We can thus supply a shell code (or any other malicious code) as the input command line argument to vulnerable1. This code gets copied into the buffer using strcpy. We can add sufficient no of NOP instructions in the malicious code such that the return address gets overwritten to point back into the buffer where our malicious code resides. Now, when launch() returns, the instruction pointer will have the overwritten written address which is the address of our malicious code. So the instruction pointer will point to malicious code in the stack and execute the malicious code. Since the stack is executable, this can compromise the system (ex: if a shell code is supplied, it can open a shell with root access).

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
Since we need to overwrite the return address in launch() function’s stack frame, we note the address of the local variable “buffer”. This is the buffer that will hold our shellcode. So we need to overwrite the return address in launch’s stack frame with the starting address of “buffer”.

The exploit code basically basically populates an environment variable $RET. The variable contains the “buffer” address (which will be used to overwrite the return address), shellcode and a large no of NOP instructions. The rationale behind using NOP’s is that even if we are off by a few bytes in the return address that is overwritten, we can be sure that the upon executing “ret” instruction, the instruction pointer will at least point to one of these NOP’s. NOP instructions only occupy a single byte and are supported by x86 instruction set. This reduces the chance of error. The contents of the environment variable are then supplied to the vulnerable1 executable as:
/tmp/vulnerable1 $RET
This will overflow the buffer and overwrite the return address to point back into the buffer that has our malicious shellcode. We need to ensure that we change the permissions on vulnerable1 (using setuid and chmod) so that the opened shell has root access to the system.

4. Provide your attack as a self-contained program written in C, perl, or python.
Check file exploit1.c for the attack program

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

The attack rests on the fact that strcpy being an unsafe function does not perform bounds checking which allows us to overflow a local buffer and overwrite the return address on the stack. The fix for this vulnerability is to use safe string copy functions such as **strncpy** which enforces bounds checking by writes only a fixed no of chars to the target buffer and thus prevents buffer overflows.

Making the stack non-executable and providing a stack canary are other defense mechanisms against such vulnerabilities.

Analysis of file vulnerable2.c:

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

#define MAXIMUM_TWEETS 528

struct live_feed {
      double tweets;
      double retweets;
      int update;
};

void launch(char * cursor, int feed_count)
{
      struct live_feed buffer[MAXIMUM_TWEETS];

      if (feed_count < MAXIMUM_TWEETS)
      {
            memcpy(buffer, cursor, feed_count * sizeof(struct live_feed));
      }
}

int main(int argc, char * argv[])
{
      int feed_count;
      char * cursor;
```

```
	if (argc != 2)
	{
		fprintf(stderr, "Usage: %s [number of tweets],[data]\n",
argv[0]);
		exit(EXIT_FAILURE);
	}

	feed_count = strtoul(argv[1], &cursor, 10);

	if ((*cursor != ',') || (strlen(cursor + 1) < sizeof(struct
live_feed) * feed_count))
	{
		fprintf(stderr, "Usage: %s [number of tweets],[data]\n",
argv[0]);
		exit(EXIT_FAILURE);
	}

	cursor = cursor + 1;
	launch(cursor, feed_count);
	return(0);
}
```

1. Briefly describe the behavior of the program.
main() function:
This program accepts a single command line argument which is a string of the form [no_of_tweets],[data].
No_of_tweets is expected to be a numeric value. This is followed by a comma and then character data (string).

The program uses strtoul on the string input which converts the string to an unsigned long. This indicates the no_of_tweets supplied by the user. The function terminates when it encounters the first character that is not a number (except that we can have a leading +/- for the number).

This length is then stored in an integer feed_count. The program checks the rest of the string to confirm that the length of remaining data is less feed_count*struct(live_feed). This is meant to verify there are feed_count no of feeds and that there's no extra data in the input.

The statement cursor = cursor + 1 advances the char pointer beyond the "comma" to point to the actual feed data.

The main() function then passes on the feed_count and the actual data feed pointed to by cursor (which is char *) to the launch() function

launch() function:
The launch function declares a buffer of fixed length MAXIMUM_TWEETS (defined as 528). The buffer is really an array of structures where each element of the array is of type struct live_feed. The function checks whether feed count is less than MAXIMUM_TWEETS. If feed_count < MAXIMUM_TWEETS returns true, then it uses the memcpy function to copy the data into the live_feed buffer. After this, launch returns back to the main function.

2. Identify and describe the vulnerability as well as its implications:
This program has several issues.
Firstly, the program does not perform any validation checks on the input data (apart from checking its length).
struct live_feed structure has 2 doubles and an int as its fields. On the other hand, the input parameter “cursor” (char *) is string data. The function blindly performs a memcpy of this data into live_feed buffer w/o performing any validation on the data.
It’s possible to pass invalid characters which are not necessarily numeric and corrupt the live_feed structure with invalid ascii codes.

The next and most important flaw in the program that leads to a vulnerability is the possibility of integer overflow.
Let us consider this line:
feed_count = strtoul(argv[1], &cursor, 10);

Now strtoul converts a string to unsigned long while feed_count is an integer. This can create a situation of an integer overflow. If the count (which is an unsigned long) does not fit into an integer, then feed_count overflows. Specifically, feed_count can be negative if the value returned by strtoul is greater than INT_MAX (max possible +ve integer). In this case, feed_count has an integer overflow resulting in wrapping around of values and a negative value getting stored in feed_count.

Next we can consider this statement from the program:
```
if ((*cursor != ',') || (strlen(cursor + 1) < sizeof(struct live_feed) * feed_count))
```
Since feed_count has an integer overflow (-ve value) , this results in the check of strlen(cursor+1) < sizeof(struct live_feed)*feed_count returning false. The reason is that C’s promotion rules say that when a signed integer (feed_count) is multiplied with an unsigned long (returned by size), the signed integer is interpreted as unsigned. So sizeof(struct live_feed)*feed_count is actually returns a very large no greater than strlen(cursor + 1) and the check strlen(cursor+1) < sizeof(struct live_feed)*feed_count returns false. By bypassing this check, we can sneak in more data than what the destination buffer can hold.

Finally, we have this check in the launch() function:
if (feed_count < MAXIMUM_TWEETS)

Again due to the integer overflow of feed_count (-ve value of feed_count), we can make this check return true (but in reality the user input is going to be much larger such that the buffer overflows).

Memcpy does not perform any bounds check on whether the target buffer can actually hold the said amount of data. This results in a buffer overflow of the target buffer. Using this overflow vulnerability we can overwrite the return address to make it point back into the buffer which has our shellcode. This results in a system compromise.

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
The exploit code takes in two arguments one a buffer that will be filled with shell code and another the offset of starting address of the buffer in launch() w.r.t the current stack pointer.

We then fill in a local buffer with the address of the “launch buffer”. This will serve as the address which will overwrite the return address in launch. Next, we supply the shellcode and sufficient no of NOP’s at the beginning of the buffer. NOP instructions reduce the chance of error in case we are off the exact return address of the shellcode by a few bytes (in that case NOP’s are executed and then we execute the actual shellcode).

We prepend the buffer contents with “RET=<integer>,” The integer (this is passed on as the no_of_tweets parameter to vulnerable 2) and comma are meant to ensure that we are compliant with the input format of vulnerable. We set the no_of_tweets input to be more than INT_MAX. In this case, we can set it to 2147484180. This overflows, feed_count to some negative value. sizeof(struct live_feed)*feed_count evaluates to 10640. The maximum size of the local buffer is MAXIMUM_TWEETS*sizeof(struct live_feed) = 528 * 20 = 10560.

So our integer value is sufficient to overflow the launch buffer. Since feed_count is set to negative, the check feed_count < MAXIMUM_TWEETS evaluates to true. And sizeof(struct live_feed)*feed_count evaluates to a value > 10560 (since feed_count is -ve, when it’s multiplied by unsigned long size, the result is a positive value). This allows memcpy to perform a copy for > 10560 bytes and overflows the launch buffer. The return address is overwritten to be the start address of the buffer and thus when launch returns, the instruction pointer points back into the buffer which contains our shellcode. Since /tmp/vulnerable2 has setuid and modified permissions, the executed shellcode has root access to our system.

4. Provide your attack as a self-contained program written in C, perl, or python:
File exploit2.c contains the attack code for this vulnerability.

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?
The first important thing is to perform validation checks on the input data especially if the data (which is accepted as a string) is to be converted to numeric types like int, float, double etc.

The next issue is to check for the return values of functions such as strtoul or the sizeof() operator which return an unsigned long. We need to store the return value in a matching variable of the same data type to avoid truncation/wrapping around. Assigning an unsigned value to a signed integer or assigning an integer to a short variable are examples of bad code that might result in integer overflows.

Finally, we should avoid using unsafe functions such as memcpy which do not perform bounds check on the maximum size of the destination. We can use a safe function like memcpy_s which is being used at Microsoft as a replacement to memcpy.

Using stack canaries, making the stack non-executable and using Address Space Randomization are other generic mitigation mechanisms that apply to all kinds of buffer overflows.

Analysis for vulnerable3.c

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

#define BUFFER_SIZE 192

void strcpyn(char * destination, unsigned int destination_length, char *
source, unsigned int source_length)
{
      unsigned int i;

      for (i = 0; i <= destination_length && i <= source_length; i++)
            destination[i] = source[i];
}

void copy_user_argument(char * user_argument, unsigned int
argument_length)
{
      char buffer[BUFFER_SIZE];
      strcpyn(buffer, sizeof(buffer), user_argument, argument_length);
}

void launch(char * user_argument)
{
      copy_user_argument(user_argument, strlen(user_argument));
}

int main(int argc, char * argv[])
```

```
{
	if (argc != 2)
	{
		fprintf(stderr, "Usage: %s ARGUMENT\n", argv[0]);
		exit(EXIT_FAILURE);
	}

	launch(argv[1]);
	return(0);
}
```

1. Briefly describe the behavior of the program.
Answer: The program accepts a single command line argument in argv[1]. If no arguments are specified then the program exits with an error.
The address of the string input is passed on to the launch function which then further invokes the copy_user_argument function passing on the user_argument string to the copy_user_argument function.

In the copy_user_argument function, we allocate a local variable buffer on the stack. The buffer is of type char and has a fixed size BUFFER_SIZE (currently set to 192).
It then invokes the strcpyn function with the destination buffer (and its length) and the source string (and its length) as arguments.

This function copies one character at a time from the source to the target buffer in a loop that goes over i = 0 to min(source_length, destination).

2. Identify and describe the vulnerability as well as its implications.
The vulnerability in the code arises from the faulty comparison in the for loop in the strncpy function.

Let's look at this line:
```
for (i = 0; i <= destination_length && i <= source_length; i++)
```

Here i incorrectly loops from 0 to min(destination_length, source_length).
So if source_length >= destination length meaning that the source string is longer or of same size as the destination buffer, then it opens up the scope for a "1 byte" buffer overflow.

This 1 byte buffer overflow can be exploited in the following way:
We know that the target buffer is a local variable stored on the stack frame of copy_user_argument function. The frame pointer EBP points to a location just beyond the last index location of the buffer. Since the code allows a 1 byte overflow, the last byte of the location pointed to by EBP can be tampered with.

Next, when copy_user_argument returns to launch, we execute these instructions:
mov %ebp,%esp and pop %ebp.
When ebp is popped from the stack, ebp now points to the tampered value. Next, the launch function needs to return.

While returning from the launch function we again execute the following:
mov %ebp,%esp
pop %ebp

Since ebp is pointing to a tampered location, esp also points to a tampered location. We can tamper the 1 byte so that esp points to the local buffer within copy_user_argument.
Next, we execute the “ret” instruction which fetches a return address from the top of the stack (esp). Since esp points to a location of the char buffer where the new return address is stored. The instruction pointer is loaded with this address and wll point to the start of the same buffer and are able to execute the shellcode.

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
The exploit code works very similar to the previous exploits. We then fill in a local buffer with the address of the “vulnerable buffer”. This will serve as the address which will overwrite the return address in launch. Next, we supply the shellcode and sufficient no of NOP’s at the beginning of the buffer. NOP instructions reduce the chance of error in case we are off the exact return address of the shellcode by a few bytes (in that case NOP’s are executed and then we execute the actual shellcode).

The contents of this buffer (ret addr, shellcode and NOP’s) are then pumped into an environment variable $RET which will be passed on as an argument to vulnerable3. The no of bytes in $RET is exactly equal to BUFFER_SIZE (192) in order expose a 1 byte buffer overflow which tampers the last byte of the location pointed to by EBP.

The tampered EBP value gets copied into ESP by the instruction mov %ebp,%esp. ESP now points to the a location within the vulnerable buffer. When the return address is popped from the stack top, the instruction pointer is loaded with this address. The instruction pointer now points to the start of the vulnerable buffer that has our shellcode. This executes the shellcode giving us root access.

4. Provide your attack as a self-contained program written in C, perl, or python.
The file exploit3.c contains the attack code for this vulnerability.

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

First off, we should ensure that we do a proper bounds check while copying the contents of one buffer to another. Characters in a buffer of length say “l” are always stored in indexes from 0 to n-1.

Programming errors like accidentally stored a character at buffer[length] or running a loop from 0 to length will result in 1 byte overflows which can be exploited to execute malicious code (like triggering a shell input).

So we need to do avoid programming error by performing proper bounds check for all buffer writes and ensure that all data written to the buffer is in the range 0 to length of buffer.

Using stack canaries, making the stack non-executable and using Address Space Randomization are other generic mitigation mechanisms that apply to all kinds of buffer overflows.

Analysis of vulnerable4.c

1. Briefly describe the behavior of the program.

This program takes in one command line argument which is a filename. If no arguments are specified, then it aborts with an error.

The said file is opened in read/write mode. If file open fails, then we abort with an error. In the get_file_size() function, we determine the size of the file. The file descriptor and file size are then passed to the launch() function. In the launch function, we declare a local “buffer” whose size is equal to the size of file which we accepted as program input.

The launch function performs a check on the file size in the line:
if (read(file_descriptor, file_buffer, file_size) != file_size)

This is meant to the ensure that that size of the allocated buffer is equal to the size of the input file. Launch then invokes the function user_interaction which allows users to supply commands.
A command ‘r’ can read data from the buffer and can be used like r,100 → read the value from an offset of 100 from the start of the buffer.
A command ‘w’ is used to write an integer into the buffer and can be used like: w,100,8 → write 8 at offset 100 from the start of the buffer.

The user_interaction function returns to launch when the user hits ‘q’ (quit) or save quit (‘s). If save quit option was selected, launch copies the updated buffer contents back into the file and returns to main(). If not, it simply closes the file and returns to main().

2. Identify and describe the vulnerability as well as its implications.
The vulnerability in the code arises from the fact that the user_interaction function allows us to write a value at any random offset from the file_buffer and does not perform a bounds check on whether the offset from buffer is actually within the bounds of the buffer.

The vulnerability occurs in the following statements:

```
if (request_buffer[0] == 'w')
{
        if (sscanf(request_buffer, "w,%lld,%d\n", &offset, &value) == 2)
            file_buffer[offset] = value;
 }
```

Since offset can be any value and not necessarily a valid offset that lies within the file_buffer bounds, we can exploit this vulnerability to overwrite the return address and make it point to malicious code. Now we need to note that since the stack is non-executable, we cannot add malicious shellcode within the file_buffer. However, we can exploit this vulnerability to launch a return to libc attack.

Specifically, we want to overwrite the return address with the address of the library function "**system"**. The arguments to system should be "/bin/sh" which is the shell that we want to open. This lets us obtain root shell access to the system resulting in system compromise.

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
The exploit code supplies a series of write commands via the user_interaction routine. The commands are of the form: w,1100,160.

Specifically, we want to overwrite the return address in the launch function's stack frame to be address of the library function "system". Here's a map of the stack frame of the launch function after the attack.

EBP + 12 0xbffffaA4 Address of environment variable EGG (which contains the parameter /bin/sh which will be supplied to the syscall "system")
EBP + 8 0xb7eb0a30 (Address of the "exit" library function)
EBP + 4 Return address (EIP) → 0xb7ebb7a0 (address of "system" library function)
EBP

The return address in launch's stack frame (which was originally supposed to point to an instruction in main) is overwritten with system syscall's address.
Now when the return address is popped from the stack, the instruction pointer (EIP) is loaded with the address of "system" syscall and control is transferred to system.
Inside system, we execute these instructions:
move %esp, %ebp

push %ebp
So, EBP now points to an address which is old_EBP + 4
Now, EBP + 8 (which is old_EBP + 12) gives us the parameter /bin/sh which is accepted as a parameter to “system” which opens up a shell with root access.

After we exit from the shell, system syscall executes the instruction: mov %ebp, %esp, pop %ebp.

Now EBP + 4 is the return address from system syscall (address of the exit system call → which is the return address in “system” syscall’s stack frame)

esp thus points to the exit syscall address which is popped from the stack and is given to the instruction pointer. The instruction pointer executes the exit code and the attack terminates gracefully.

The string “/bin/sh” which is used as a parameter to system, is stored in an environment variable EGG.

4. Provide your attack as a self-contained program written in C, perl, or python.
The files: exploit4.c, exploit4_input and exploit4_commands contain the exploit code.

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?
The fix for this vulnerability is to perform a bounds check on the offset from the file_buffer start address. We need to confirm that this offer lies within the file_buffer size bounds.

vulnerable4.c already uses a stack canary and provides a non-executable stack. So it’s not possible to store the attack code in a local buffer on the stack and make the return address point to the buffer.

But this file is susceptible to return to libc attacks. By using address space randomization, we can make the libc libraries to be loaded at arbitrary memory locations so that it is hard for the attacker to guess the exact address of libc functions and thus making it harder to launch return to libc style attacks.

1. What is the value of the stack canary? How did you determine this value?
The value of the stack canary is %gs:0x14 which in our case evaluates to 0xff0a0000. I determined this value using gdb. Basically, I disassembled the launch function and checked the following instructions:
0x08048852 <launch+21>: mov %gs:0x14,%eax
0x08048858 <launch+27>: mov %eax,-0x14(%ebp)

Now, gs is a segment register which indexes into a global descriptor table (GDT) and 0x14 is the offset from the base index. This evaluates a random address which will be used as the value of our stack canary.

2. Does the value change between executions? Does the value change after rebooting your virtual machine?

Since we do not use address space randomization in this project, the value of the canary does not change between executions or even after rebooting the virtual machine.

3. How does the stack canary contribute to the security of vulnerable4?

The stack canary adds on to the security of vulnerable4 in that the attacker can no longer randomly overflow the buffer since in that case the canary value would be overwritten resulting in a stack protection violation.

However, in our exploit we simply bypass the canary value and directly overwrite the return address so that it points to the libc function “system” and then pass on /bin/sh as an argument to system. So a stack canary mitigate some attacks but does not protect against return to libc style of attacks.

Analysis of mystery shellcode:

The mystery shell code can be disassembled to the following assembly code as shown below:
Please note that the direction of flow of data is from right to left.

mov <dest>, <source>

```
0:  31 c9                   xor    ecx,ecx                 # ecx = 0
2:  b9 0e 0f 10 21          mov    ecx,0x21100f0e          # ecx = 0x21100f0e
7:  81 f1 21 21 21 21       xor    ecx,0x21212121          # ecx = ecx xor 0x21212121 = 0x00312e2f
= /.1 →
d:  51                      push   ecx          # push value of ecx on to stack. ESP points to this value
e:  31 c9                   xor    ecx,ecx                 # ecx = 0
10: b9 0e 55 4c 51          mov    ecx,0x514c550e  # ecx = 0x514c550e
15: 81 f1 21 21 21 21       xor    ecx,0x21212121  # ecx = ecx xor 0x21212121 = 0x706d742f =
pmt/ → /tmp
1b: 51                      push   ecx  # push value of ecx on to stack. ESP points to this value
```

So we now have, /tmp /.1 in the stack

```
1c: 89 e3                   mov    ebx,esp  # ebx = esp [ebx stores the curr address of stack top].
1e: 31 c0                   xor    eax,eax   # eax = 0
20: 31 c9                   xor    ecx,ecx   # ecx = 0
22: 31 d2                   xor    edx,edx  # edx = 0
24: b0 05                   mov    al,0x5   # al = 0x5. So eax = 0x00000005 = code for open system
call.
26: b1 41                   mov    cl,0x41 # cl = 0x41 O_CREAT(40) | O_WRONLY (1) |
28: b6 01                   mov    dh,0x1 # dh = 0x1 So edx = 0x00000100
```

2a: b2 c0 mov dl,0xc0 # dl = 0xc0. So edx = 0x000001c0 = 1c0 = **S_IRUSR (100) | S_IWUSR(80) | S_IXUSR(40)**
2c: cd 80 int 0x80 # syscall **open** with parameters in ebx → file name /tmp/.1, ecx → , edx → contains the parameters for the open system call.
2e: 89 c3 mov ebx,eax # so now **ebx = file descriptor of the opened file.**
30: 31 c9 xor ecx,ecx # ecx = 0
32: b9 44 0f 01 2b mov ecx,0x2b010f44 #ecx = 0x2b010f44
37: 81 f1 21 21 21 21 xor ecx,0x21212121 #ecx = ecx xor 0x21212121 = 0x0a202e65 = \n <space> . e ---> This corresponds to the string "e. \n"
3d: 51 push ecx #ecx = **e. \n** This is now pushed onto the stack
3e: 31 c9 xor ecx,ecx # ecx = 0
40: b9 4c 52 49 4e mov ecx,0x4e49524c # ecx = 0x4e49524c
45: 81 f1 21 21 21 21 xor ecx,0x21212121 #ecx = ecx xor 0x4e49524c = 0x6f68736d = ohsm → this stores the **string "msho"**
4b: 51 push ecx # The string msho is stored onto the stack.
4c: 31 c9 xor ecx,ecx # ecx = 0
4e: b9 0d 01 46 54 mov ecx,0x5446010d # ecx = 0x5446010d
53: 81 f1 21 21 21 21 xor ecx,0x21212121 # ecx = ecx xor 0x21212121 = 0x7567202c = ug <space> , → this corresponds to the **string ", gu"**
59: 51 push ecx # the string ", gu" is stored on the stack.
5a: 31 c9 xor ecx,ecx # ecx = 0
5c: b9 01 4b 4e 43 mov ecx,0x434e4b01 #ecx = 0x434e4b01
61: 81 f1 21 21 21 21 xor ecx,0x21212121 # ecx = ecx xor 0x21212121 = 0x626f6a20 = boj<space> → this corresponds to the string **"job "**
67: 51 push ecx # the string **"job "** is now pushed onto the stack.
68: 31 c9 xor ecx,ecx # ecx = 0
6a: b9 4f 48 42 44 mov ecx,0x4442484f # ecx = 0x4442484f
6f: 81 f1 21 21 21 21 xor ecx,0x21212121 # ecx = ecx xor 0x21212121 = 0x6563696e = ecin → this corresponds to the string **"nice"**
75: 51 push ecx // the string "nice" is now pushed to the stack
76: 89 e1 mov ecx,esp // ecx = current value of esp which contains the string **nice job, gumshoe. \n**
78: 31 c0 xor eax,eax // eax = 0
7a: b0 04 mov al,0x4 // al = 0x4 → eax = 0x00000004 → this code corresponds to the **system call "write"**
7c: 31 d2 xor edx,edx // edx = 0
7e: b2 14 mov dl,0x14 // dl = 0x14 → edx = 0x00000014. This indicates the no of bytes to write to the file. 0x14 = **20 bytes are to be written to the file /tmp/.1**
80: cd 80 int 0x80 //invoke **syscall write** to perform the file write operation.
82: 31 c0 xor eax,eax //eax = 0
84: b0 06 mov al,0x6 //al = 0x6. This is the code for the **close system call.** But the file **does not get closed** since the instructions to set ebx = file_descriptor of the opened file and the interrupt int 80 are missing.

```
86: 31 db                xor    ebx,ebx // ebx = 0
88: 31 c0                xor    eax,eax //eax = 0
8a: b0 01                mov    al,0x1 //al = 0x1 → eax = 0x1
8c: cd 80                int    0x80 //invoke system call with code 0x1. This is the exit system
call.
                         add, %al, (%eax)
```

Brief description for the mystery shellcode:
The mystery shell code opens a file
fd = open("/tmp/.1", O_CREAT|O_WRONLY, S_IRUSR | S_IWUSR | S_IXUSR);

Next, the mystery code writes the string **"nice job gumshoe. \n"** into the file.
write(fd, "nice job, gumshoe. \n", 20);

The shellcode should have typically closed the file. It does have instructions to load the system call code for close (0x6) into eax but it does not invoke int80 to actually close the file. So the file remains open.

Next, it executes the **"exit"** system call to finally terminate the program.