CSCI 3403 **Erika BAILON**
Homework 7 **Fall 2018, CU-Boulder**

1. **Review 10.2** List the three distinct types of locations in a process address space that buffer overflow attacks typically target

   The buffer could be located on the **1. stack** , in the **2. heap** , or in the **3. data section** of the process. These locations holds other program variables or parameters or program control flow data such as return addresses and pointers to previous stack frames. (page 321)

2. **Review 10.3** What are the possible consequences of a buffer overflow occurring?

   The consequences of this error include corruption of data used by the program, unexpected transfer of control in the program, possibly memory access violations, and very likely eventual program termination. (page 321)

3. **Review 10.9** Describe what a NOP sled is and how is it used in a buffer overflow attack.

   Is important to first know that NOP means "No - Operation". From this, we get that NOP sled are a series or sequence of 'no operation' instructions that are going to pad the space in the buffer to the end, so that the attacker can insert a new code to change the return address causing it to go to another executable code. This is, according to the book, dealing with the inability to precisely determine the starting address of this code, the attacker then can exploit the fact that the code is often much smaller than the space available in the buffer. Then padding till near the end of the buffer withNOP instructions, the attacker specifies the return address used to enter this malevolous code as a location somewhere in this run, and this is called NOP sled. (Page 338)

4. **Problem 10.2** Rewrite the program shown in Figure 10.1a so it is no longer vulnerable to a buffer overflow.

```
int main(int argc, char *argv[]) {
      int valid = FALSE;
      char str1[12];
      char str2[8];

      next_tag(str1);
      fgets(str2);     //fgets() guarantees to read no more characters than
                        the buffer size **
```

```
        if (strcomp(str1, str2, 8) == 0)
             valid = TRUE;
        printf("buffer1:  str%s), str2(%s), valid(%d)\n", str1, str2, valid)}
}
```

5. **Problem 10.3**. Rewrite the function shown in Figure 10.5a so it is not longer vulnerable to a stack buffer overflow.

```
void hello(char *tag)
{
     char inp[16];

     printf("Enter a value for %s: ", tag);
     fgets(inp);      //fgets() guarantees to read no more characters than
                        the buffer size **
     printf("Hello your %s is %s/n", tag, inp);
}
```

6. **Problem 10.4**. Rewrite the function shown in Figure 10.7a so it is not longer vulnerable to a stack buffer overflow.

```
void gctinp(ohar * inp, int siz)
{
   puts("Input value: ") ;
   fgets(inp, siz, stdin);
   printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
   char tmp[32];       //to handle 2-16 size strings
   snprintf(tmp, "read val: %s\n"’ val);   //snprintf( ) Writes the
                           results to a character string buffer.
                           at most buf_size - 1 characters are written
   puts(tmp);
}

int main(int argc, char *argv[ ] )
{
   char buf[16];
   getinp (buf, sizeof (buf));
   display(buf);
   printf("buffer3 done\n");
}
```

7. **Review 11.3** List some possible sources of program input.

   We can have data read into the program from:
   -the user keyboard or mouse entry
   -from files, or network connections
   Some possible sources are also:
   -data supplied in the execution environment
   -values of any configuration or other data read from files by the program
   -values supplied by the operating system to the program.
   (Page 362)

8. **Review 11.6**. Define a cross-site scripting attack. List an example of such an attack.

   This vulnerability involves the inclusion of script code in the HTML content of a webpage displayed by a user's browser. The script code could be in Javascript, ActiveX, Flash, or just about any client-side scripting language supported by user's browser. To support some categories of Web applications, script code may need to access data associated with other pages currently displayed by the user's browser.
   Cross-site scripting attacks attempt to exploit this assumption and attempt to bypass the browser's security checks to gain elevated access privileges to sensitive data belonging to another site. These data can include page contents, session cookies, and a variety of other objects. Attackers use a variety of mechanisms to inject malicious script content into pages returned to users by the targeted sites.
   The most common variant is the XSS reflection vulnerability. **For example** Lets say you have a website that is called *www.letsgiveanexample.com*, if you were to type *www.letsgiveanexample.com/page/whatsthis*, you would get an error message saying the resource *whatsthis* wasn't found. This means the parameter was sent back to you. So

then, instead of using that parameter, you could use a malicious one. This is, you could input something like ¡script¿alert(1)¡/script¿whatsthis and hit enter, or any constructed request because this code, if the page is vulnerable, then it can be injected and this will cause JavaScript code that is supplied by "'you" the attacker, and this coed will be executed within the user's browser in the context of that session and you can obtain information from that user if the user follows those links you're displaying so you can obtain something like the token or the login credentials.