CSCI 3403 **Brennon Lee**
Homework 7 **Fall 2018, CU-Boulder**

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

   The three distinct types of locations are the stack, heap, and data section of a process.

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

   Consequences include: Corruption of program data, unexpected transfer of control, memory access violations, and code executed from an attacker.

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

   A NOP sled is a way for an attacker to find the starting address of the code that they want to execute. Code is normally smaller than the space made available by the buffer so an attacker can exploit this. By padding the code they want to execute with NOP's, the attacker can replace the return address to be somewhere in this range of NOP's where if hit, execution of code will slide down these no operation instructions right into the attackers code.

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

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

  next_tag(str1);
  fgets(str2, 8, stdin);
  if(strncmp(str1, str2, 8) == 0)
      valid = TRUE;
  printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

5. **Problem 10.3** Rewrite the program shown in Figure 10.5a so it is no longer vulnerable to a stack buffer overflow.

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

  printf(”Enter value for %s: ”, tag);
  fgets(inp, 16, stdin);
  printf(”Hello your %s is %s\n”, tag, inp);
}
```

6. **Problem 10.4** Rewrite the program shown in Figure 10.7a so it is no longer vulnerable to a stack buffer overflow.

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

void display(char *val)
{
  char tmp[16];
  snprintf(tmp, 16, ”read val: %s\n”, val);
  puts(tmp);
}

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

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

User keyboard entries, mouse entires, files, network connections, data supplied in the execution env, values of any configuration and values supplied by the OS.

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

   Cross-site scripting involves the inclusion of script code in HTML content of a webpage displaed by a user's browser. The script code can be JavaScript, ActiveX, VBScript, Flash, or another client side scripting language. To support some categories of web applciations, script code may be needed to access data associated with other pages currently displayed to the user. So security measures only allow pages originating from the same site to have this kind of data access. Cross-site scriping exploits this assupmtion and attempts to bypass that security check to gain access privileges to sensitive data. An example is the XSS reflection vulnerability. This attack includes a malicious script content in data supplied to a site. This data could be displayed to other users and they may click on it which will execute the malicious script.