CS2107 Tutorial 8 (Software Security)

School of Computing, NUS

25 – 29 October 2021

1. *(Buffer overflow vulnerability)*: Try out this `badprogram.c` C program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char text[16];

    strcpy(text, argv[1]); /* copy the 1st argument into array "text" */
    printf("The supplied first argument is :\n");
    printf("%s", text);
    printf("\n");

    return 0;
}
```

Compile the program above (e.g. `gcc -o badprogram badprogram.c`), and execute it. Notice that the program takes in an argument. By executing, for instance, `./badprogram 'hello world'`, the program `badprogram` will take in the string `'hello world'` as an argument, store the argument into the array `text`, and then print it.

Try running it with different arguments. What would happen if the argument is:

(a) `two words`

(b) `'two words'`

(c) a very long string (say, longer than 32 characters).

Solution

You can just try running the executable with the listed arguments.
The output lines are as follows:

(a) `two`

(b) `two words`

(c) ⟨*the_entered_long_string*⟩ `/ *** stack smashing detected ***:`
`./badprogram terminated Aborted (core dumped)`

2. *(UNIX commands)*: Familiarize yourself with UNIX/Linux commands like `ls`, `cat`, `sh`, `echo`.

Solution

Please refer to your UNIX/Linux references, and do your own practice. A free and good resource to learning Linux commands is: William Shotts, "The Linux Command Line", `http://linuxcommand.org`.

3. *(Safe/unsafe functions)*: Find out more about the following C library functions. Which usages should be avoided, and why?

   (a) `strcat (dest, source);`

   (b) `strncat (dest, source, n);`

   (c) `memcpy (dest, source, n);`

   (d) `strncpy (dest, source, strlen(source));`

   (e) `sprintf (str, f);`

   (f) `printf ("Please key in your name:  "); gets (str);`

   (g) `scanf ("%s", str);`

   (h) `scanf ("%20s", str);`

Solution

(a) Unsafe. The buffer `dest` can get overflowed since there is no limit to the number of characters concatenated into it.

(b) Safe if `n` < the remaining characters (bytes) available on the `dest` buffer. Note that the `dest` buffer must be large enough to contain the concatenated resulting string and also *an additional null character*. See `https://linux.die.net/man/3/strncat` for details.

(c) Safe if `n` $\leq$ the sizes of the `dest` and `src` buffers. Note that `memcpy` performs a binary-copying operation. See `http://www.cplusplus.com/reference/cstring/memcpy/` for details.

(d) Unsafe. The buffer `dest` can get overflowed since `strlen(source)` can be greater than `dest`'s length.

(e) Unsafe. The buffer `str` can get overflowed since the formatted string output can be longer than `str`'s length.

(f) Unsafe. The buffer `str` can get overflowed since there is no limit to the number of characters read and stored into it.

(g) Unsafe. The buffer `str` can get overflowed since there is no limit to the number of characters read and stored into it.

(h) Safe if the size of `str` $>$ 20 since at most 20 characters are stored by `scanf()` into `str`. Note that a *terminating null character* is added at the end of `str`. The specified maximum input length (i.e. 20) does not include this additional terminator character. As such, `str` must be at least one character longer than the specified maximum input length. See also `https://en.wikipedia.org/wiki/Scanf_format_string`.

4. *(Memory initialization)*: Consider the following C program.

```
#include <stdio.h>

int main()
{
  unsigned char a[10000];

  for (int i=0; i<10000; i++)
      printf ("%c", a[i]);

  return 0;
}
```

(a) What would be the output? What is its implication to secure programming?

(b) A possible preventive measure is to always initialize the array. What is the disadvantage of doing that?

Solution

(a) In C, the value of an uninitialized local variable is *indeterminate/undefined*, which basically can be anything. Accessing such an uninitialized variable leads to an "*undefined behavior*". The program above, which prints out the uninitialized array `a`, poses a security risk since the printed data could be sensitive. This is the case since memory chunks in a running process' memory are basically "recycled", both between different process executions and during the process execution. If the array `a` happens to contain a sensitive piece of information, the information will then get leaked out to the external user.

(b) Extra processing time is required.

5. *(Integer overflow vulnerability)*: Consider the following C program.

```
#include <stdio.h>
#include <string.h>

int main()
{
   unsigned char a, total, secret; // Each of them is a 8-bit unsigned integer
   unsigned char str[256];       //  str is an array of size 256
   a = 40;
   total = 0;
   secret = 11;

   printf ("Enter your name: ");
   scanf ("%255s", str);         // Read in a string of at most 255 characters

   total = a + strlen(str);

   if (total < 40) printf ("This is what the attacker wants to see: %d\n", secret);
   if (total >= 40) printf ("The attacker doesn't want to see this line.\n");
}
```

If the user follows the instruction and enters his/her name honestly, he/she will be unable to see the secret. Suppose you are the attacker, how would you cause the secret number to be displayed?

Solution

Notice that `total` is a 8-bit unsigned integer. Hence, its value ranges from 0 to 255. Addition operations done on `total` are thus actually carried out under module 256.
To display the secret number `secret`, we need to overflow `total`, yet make its overflowed value less than 40, i.e. between 0 and 39 (inclusive). To this end, we need to supply a string with length between $256-40 = 216$ and 255 (inclusive).

Notice, however, that the secret number still be displayed if we enter a string longer than 255. *Why?* (Hint: pay attention to the format specifier supplied to the `scanf()` invocation in the code.)

6. *Terminology*: CVE, Black list, White list, Black hat, White hat, Spamhaus, CERT, SingCert, SOC (Security Operation Center), SIEM.

Solution

Please google/wiki the terms.

— End of Tutorial —