Writing Behind a Buffer

Angelo Rosiello[*]

©Rosiello Security

http://www.rosiello.org

18/12/05

* Rosiello Security e-mail:angelo@rosiello.org

Abstract

In this paper we are going to describe a kind of vulnerability that is known in the literature but also poor documented. In fact, the problem that is going to be analyzed can be reduced to a memory adjacent overwriting attack but usually it is obtained exploiting the last null byte of a buffer, hence we are going to show that the same result is still possible writing behind a buffer, under certain conditions. To fully understand the subject of this article it's necessary to describe the memory organization[1] of running processes, then the memory adjacent overwrite attack, concluding with our analysis.

Memory Organization

A process can be defined as a running program, thus the operating system has loaded its instructions into memory and has allocated different areas of memory to manage its execution. The address space of a running process can be divided into five segments[1,2]:

- Code Segment: this segment contains the executable code of the program.
- Data and BSS Segment: both sectors are dedicated to the global variables and are allocated during the compile time. To be clear, the sector BSS contains not initialized data while data segment is reserved for static data.
- Stack Segment: local variables are allocated in this segment. It is particular useful for storing cotext and for function parameters. The stack memory grows downward.
- Heap Segment: this segment represents all the rest of memory of the process. The heap memory grows upward and is allocated dynamically.

In figure 1 we can observe all the memory segments described above.

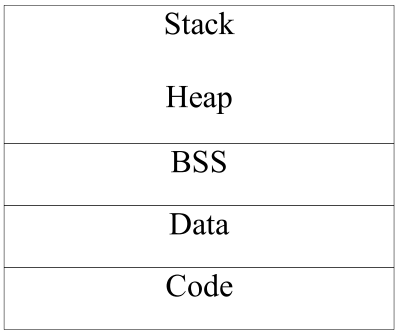

Figure 1

The memory adjacent overwrite attack, exploits the memory allocated into the stack for automatic variables to produce a buffer overflow[6] and to gain the control of the process execution flow.

Memory Adjacent Overwrite Attack

Last years were released some articles[4,5] about exploiting non-terminated adjacent memory space. The problem exists when the last null byte, terminating a buffer, is overwritten and another buffer precedes it.

1 The considered architecture is Intel[3] but the concepts can be extended to other architectures, too.

In fact, when a buffer is declared it is finished into the stack with a null byte to separate it from the rest of the stack. To stay clear let's bring an example written in C where we are going to use two buffers.

```
//Example 1
int main( ) {
  char buffer1[]="ab";
  char buffer2[]="cd";
  .................;
  return 0;
}
```

Exploring the stack runtime we will notice that buffer2 is near buffer1 and separated from it thanks its last null byte.

```
          Stack Memory
               [c]
               [d]
   (X)        [0x0]
               [a]
               [b]
              [0x0]
```

Thus, overwriting the null byte indicated with (X), buffer2 will be concatenated to buffer1 containing the whole string "cdab".

The above scenario doesn't represent a security problem yet, but if buffer2 is copied into some other buffer, it could lead to a stack overflow. Let's consider the following example:

```
//Example 2
void function( char buffer2[32] ) {
  char buffer3[32];
  strcpy( buffer3, buffer2 );
}
int main( ) {
  char buffer1[32]; //suppose buffer1 filled of chars
  char buffer2[32]; //suppose buffer2 filled of chars
  function( buffer2 );
  return 0;
}
```

Example 2 is not vulnerable but if 'buffer2[32]' is set to something different from the null byte then an overflow will occur overwriting the instruction pointer and giving the attacker the chance to gain the process execution flow control.

Behind a Buffer

Memory adjacent overwrite attack showed us the possibility to exploit stack memory organization to concatenate two regions of memory. Recently, we could notice the existance of a vulnerable scenario that is specular to the one introduced in the previous paragraph. Let's consider the following piece of code:

```
//Example 3
int main( ) {
  char buffer1[2];
  char buffer2[2];
  /* some code here that fills buffer1 and buffer2 and
returning an integer value i */
  buffer1[i]='X';
  ................;
  return 0;
}
```

Key security of this piece of code is the value of the variable 'i', in fact, if for some reason 'i' assumes the value '-1', the null byte of the buffer2 will be overwritten by 'X', exactly as it happened in example2.

In this case we worked from behind of buffer1, instead of proceeding over buffer2, but obviously the result is the same.

Exactly as in example2, in order to gain the control of the instruction pointer, there must be in the code some other vulnerable instruction, like strcpy() into function().

Conclusions

Both described techniques to exploit memory adjacent areas must be kept in consideration when coding an application. In fact, this security problem was at first described as consequence of an unsafe use of some standard C functions[7] (e.g. strncpy(), strncat(), etc.) that do not terminate buffers with a null byte, but it's reductive and we showed that the problem still remains also when those sensitive functions aren't used at all.

Fortunately these kind of bugs are statistically not numerous and with enough attention and a minimum knowledge they can be completely avoided.

References

[1] *Modern Operating Systems* by Andrew Tanenbaum. Prentice Hall; 2nd edition (February 28, 2001)

[2] *Operating Systems: Internals and Design Principles* by William Stallings. Prentice Hall; 4th edition (December 15, 2000)

[3]*http://developer.intel.com/design/pentium/manuals*

[4] *Adjacent Overwrite BUG* by Daniel Hodson. Info Security Writers (January 20, 2004).

[5] *Taking Advantageof non-terminated adjacent memory space* by twitch. Phrack 56 *(January 05, 200)*

[6] *Stack Overflow & SIMPLESEM* by Angelo Rosiello. Rosiello Security (September 09, 2003)

[7] *C Standard Library Functions*