Writing Behind a Buffer

Angelo P. E. Rosiello

http://www.rosiello.org

Outline

- Introduction.
- Processor Technology.
- Description of Process memory organization: the main five segments.
- *) Description of adjacent memory overwrite attack: pre-conditions and effects.
- Adjacent memory overwrite attack: analysis of a practical example.
- Exploiting the vulnerable code.
- What do we mean with "writing behind a buffer"?
- Showing the new vulnerable code.
- Security Analysis of the introduced example.
- Did we find something new? Explaining reasons of our analysis.
- Conclusions.
- Questions and Answers.

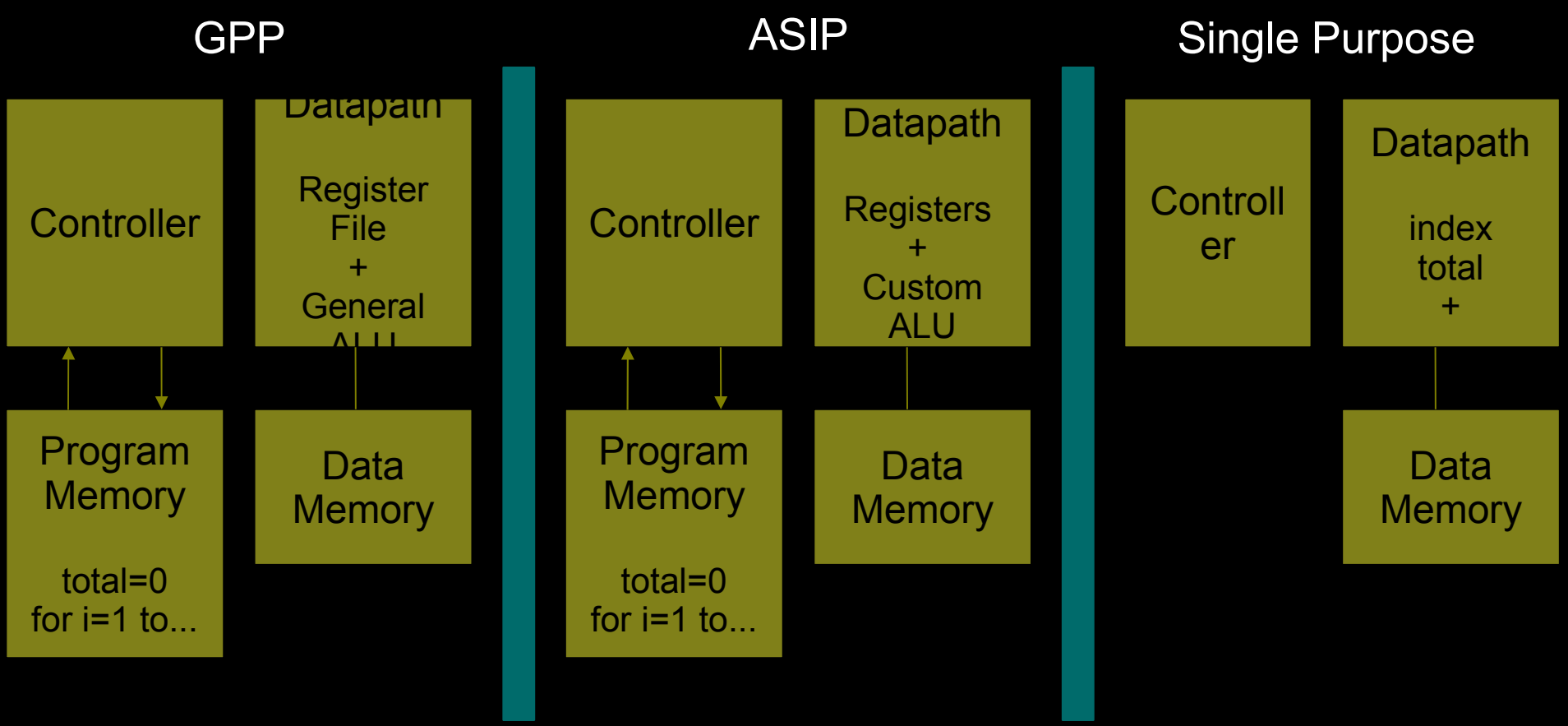
Introduction

- Adjacent memory attacks are known in the literature but poor documented.
- Why?
  - Statistically not numerous.
- Is it exploitable?
  - Yes.
- Example of exploitable context: adjacent locations of memory that can be concatenated.

Processor Technology (1/2)

- We can distinguish three kind of processor types:
  - Single Purpose: digital circuit designed to execute exactly one program which is hard-wired.
  - Application Specific: instruction-set processor with a custom ALU that can be programmed by writing software (e.g. DSPs, microcontrollers).
  - General Purpose: instruction-set processor with a general ALU that can be programmed by writing software.

Processor Technology (2/2)



Which Technology is Vulnerable?

* For practical reasons for the examples we will consider only GPP and in particular Intel's architecture*

Knowledge Requirements

- To fully understand the problem we need to analyze:
  - Memory organization of running processes.
  - Memory adjacent overwriting attacks.
  - Trivial buffer overflow attacks.

Process Memory Organization

Processes & Memory Organization

- A process can be defined as a running program; a program is a passive entity, while a process is an active entity.
- An operating system provides the environment within which programs are executed. It loads run-time instructions of a process in the memory and allocates different memory sections for its execution.
- The address space of a process can be divided into five main sections...

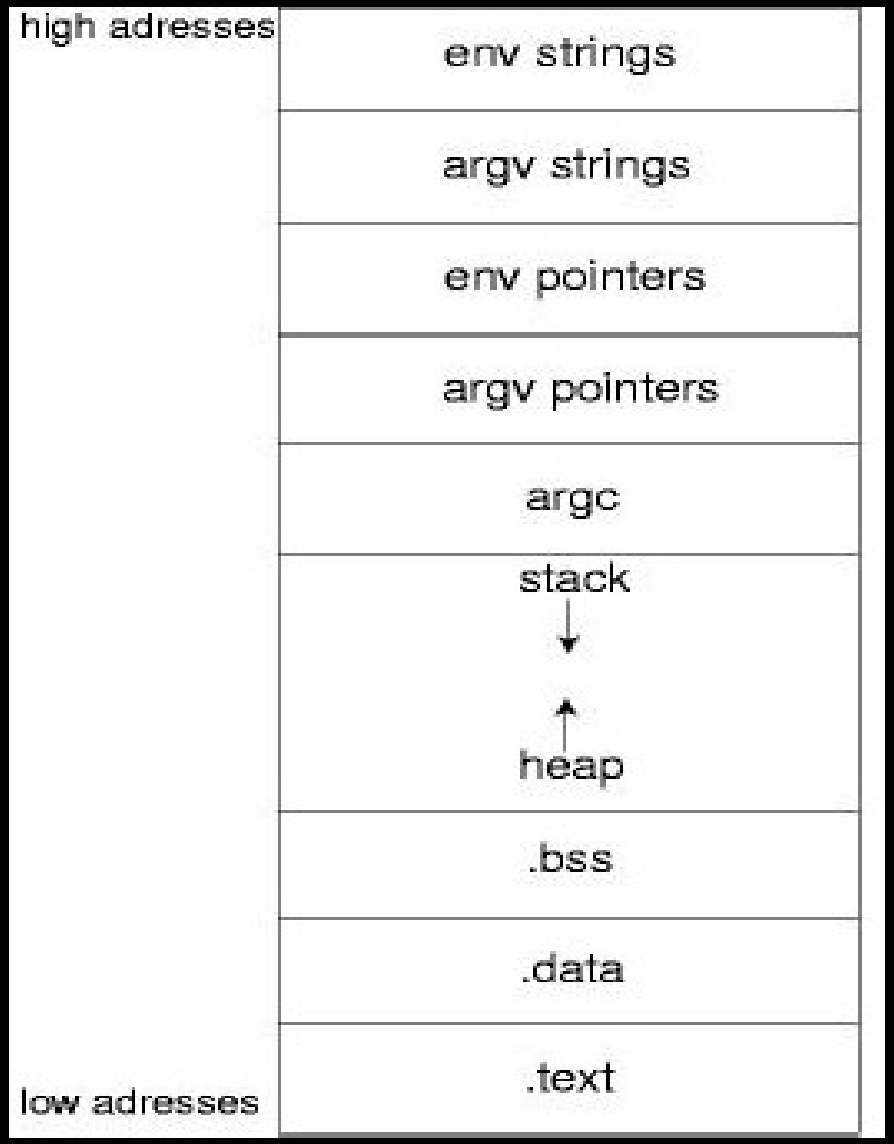
Five Segments (1/2)

- Code: executable code of the program.
- Data & BSS:
  - BSS: not initialized data.
  - Data: static data.
  - Both allocated at compile-time.

Five Segments

- Stack:
  - Local variables.
  - Particular useful for storing context and for function parameters.
  - It grows downward.
- Heap:
  - All the remaining memory of the process.
  - Allocated dynamically.
  - It grows upward.

Process Memory Dump



Adjacent Memory Overwrite Attack

This technique let an attacker exploit the memory allocated into the stack for strings to produce a buffer overflow and to gain the control of the process execution flow.

Adjacent Memory Overwrite Attack

- Last years some articles [Twitch, Hodson] came out about exploiting non-terminated strings adjacent memory spaces.
- Pre-conditions:
  - the last null-byte terminating a buffer 'X' is overwritten,
  - a buffer 'Y' preceedes 'X'.

Buffer Declaration

When a buffer is declared it is finished into the stack with a null-byte to separate it from the rest of the stack.

Buffer Declaration: an example

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

Stack Memory

[c]

[d]

(X) [0x0]

[a]

[b]

[0x0]

buffer2 is near buffer1 and separated from it thanks its last null-byte. Overwriting in some way the null-byte indicated with (X), buffer2 will be concatenated to buffer1 containing the whole string “cdab”

Dangerous Functions

- Many standard C functions that a programmer may take to be safe against buffer overflows, do not automatically terminate strings/buffers with a NULL byte.
- Let's consider the function strncpy():

char * strncpy(char *dst, const char *src, size_t len)

"The **strncpy**() function copies at most len characters from src into dst. If src is less than len characters long, the remainder of dst is filled with `\0' characters. Otherwise, dst is not terminated"

Strncpy(): Bad Scenario (1/2)

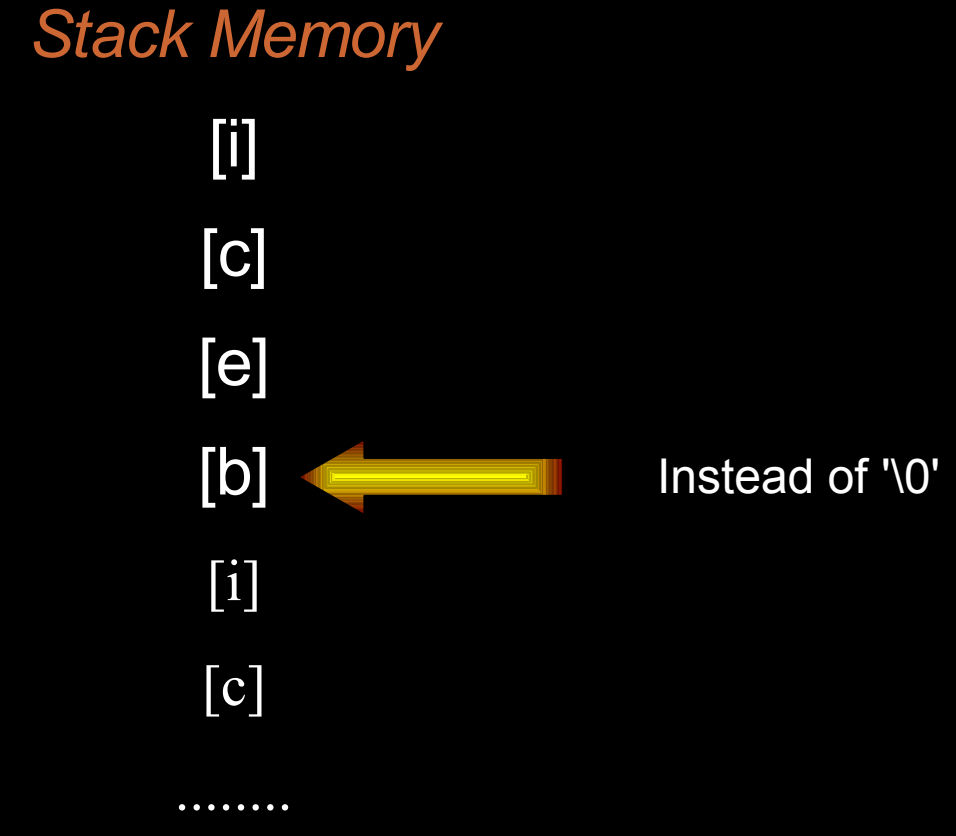
- Let's consider Twitch's example:

```
//Twitch's example
void func() {
char buf1[8];
char buf2[4];
fgets(buf1, 8, stdin);
strncpy(buf2, buf1, 4); }
```

- If the user entered the string “iceburn”, printing *buf2* we obtain: “icebiceburn”.

Strncpy(): Bad Scenario (2/2)

- Let's have a look at the stack:



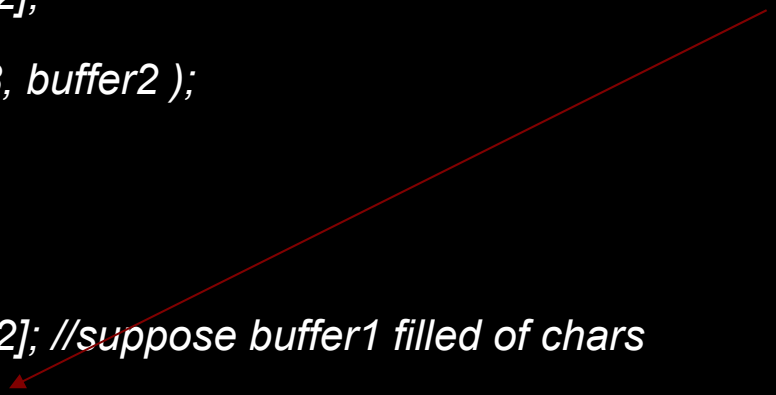
- The strings got concatenated.

Where is The Security Menace?

- Both “Example 1” and Twitch's example don't represent a vulnerable scenario but if the new concatenated string is copied into some other buffer, a buffer overflow is possible.
- To stay clear, let's consider another example...

Practical 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 the last null byte of buffer2 is overwritten, then an overflow will occur overwriting the instruction pointer and giving the attacker the chance to gain the control of the process' execution!

Behind a Buffer

What do We mean?

- Adjacent memory overwrite attacks showed us the possibility to exploit the stack memory organization concatenating two strings. This happened because some functions do not always terminate buffers with a NULL byte, such as *strncpy()*.
- Another vulnerable scenario exists and it is specular to the one introduced in the previous slides.

Another Vulnerable Scenario

```
//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;

}
```

Where is the problem here?

Security Analysis

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

- Key security of “Example 3” is the value of the variable 'i'.
- What happens if 'i=-1'?
- The null-byte of buffer2 will be overwritten by 'X', exactly as it happened in “Example 2”.

Something New?

- Adjacent memory overwrite attacks were at first described as direct consequence of an unsafe use of some standard C functions (e.g. strncpy(), strncat(), etc.) that do not terminate buffers with a null-byte but...

  This approach is quite reductive and we showed that the problem still remains also when those sensitive functions aren't used at all!

Conclusions

- During our discussion we considered memory adjacent overwrite attacks but “writing behind a buffer” could be extended to other vulnerable contexts...
- While programming keep always in consideration runtime memory accesses.
- Fortunately these kind of bugs are statistically not numerous and with enough attention and a good analysis-testing they can be completely avoided.

Questions & Answers

- Contact: angelo@rosiello.org
- Web site: http://www.rosiello.org