[REVS] Buffer Overflow Exploitation and Prevention

Source: http://www.derkeiler.com/Mailing–Lists/Securiteam/2005–04/0116.html

---

From: SecuriTeam (*support_at_securiteam.com*)
Date: 04/21/05

```
To: list@securiteam.com
Date: 21 Apr 2005 15:06:00 +0200
```

The following security advisory is sent to the securiteam mailing list, and can be found at the SecuriTeam web site: http://www.securiteam.com
– – promotion

The SecuriTeam alerts list – Free, Accurate, Independent.

Get your security news from a reliable source.
http://www.securiteam.com/mailinglist.html

– – – – – – – – –

Buffer Overflow Exploitation and Prevention
——————————————————————————————————————

SUMMARY

This paper will attempt to explain usual problems encountered when trying
to exploit buffer overflows in particular contexts of the stack, and
propose a method to easily solve those problems.

What do you have to know before reading?
You have to know assembly language, C language and Linux/Unix. Of course,
you have to know what a buffer overflow is (we highly recommend reading
[1]).

DETAILS

Starting with a sample problem:
Let's have a look at the following short program:

Code Snipps:
```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
void copy(char *str)
{
	char buf[256];
	int i;

	memset(buf, 0, 256);
	for (i = 0; str[i]; i++) {
		buf[i] = str[i];
	}
}

int main(int ac, char **av)
{
	if (ac < 2) {
		fprintf(stderr, "I take at least one argument.\n");
		return (0);
	}
	copy(av[1]);
	return (0);
}
```

This program does nothing of interest by itself – copy() copies its only argument to a fixed size (256 bytes) buffer. Notice that there is no added null byte at end of copy.

It is obvious that the vulnerability lies in copy(): when argument pointed to by `str' is too long, we smash, outside of the buffer, EBP (not always stored), then EIP.

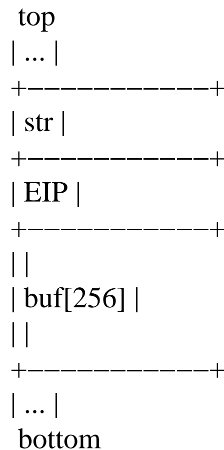
Let's suppose we are prevented by the hosting operating system from using brute force to change the return address on a shellcode that we would have passed in parameter. The goal is now to use a method where address prediction would help, like in a return–into–libc ([2]). Indeed using a return–into–libc implies that you guess the addresses you need, so it's far less random.

We'll have to appropriately prepare the stack in order for our return–into–libc trick to succeed. Let's take a closer look to the copy() function. The pointer to the string to copy is the one and only parameter passed to the function, it is located just after EIP on the stack.

This fact is important, it is our problem:
If the argument which potentially creates the overflow – a string too long is one of the first passed to the function, it is impossible for us to set in the stack arbitrary values we need to perform a common return–into–libc: smashing the pointer used for the copy is going to change the source of the copy to another location and fill the pointer with random data! For each byte copied onto the pointer, the reference to your data will change, and the copy will end on a null byte or a SIGSEGV. An usual return–into–libc will never succeed here.

Solution: ret–onto–ret:
Let's look at what the stack looks like before we call copy():

Diagram:



If we smash EIP with the address of a RET instruction, another return will be performed. This second return will use the address in ESP (str), and put it into EIP. The execution of the program goes on, using the content of our string, which could contain a shellcode.

Because the program execution continues at the beginning of our string, we can use the 256 bytes in the buffer for our shellcode, without having to use dummy instructions such as NOP, but we can only do that if the stack is executable.

We only need one RET here because our argument is the first on the stack. If there are several arguments, we can write several RET instruction's addresses in a row on the stack in order to equalize EIP and str !

The main difficulty is to find the address of a RET instruction, which is not that much of a challenge, given the number of functions in the libc. We'll apply classic prediction methods if we can't use the local system's procfs...

Kernel developers have been kind enough to add support for virtual system calls in Linux 2.6. That means that userland routines now have the ability to directly use code mapped into kernel space, without the expense of using a system call (and the associated interrupt and CPU cycles). For instance gettimeofday(2) now has its vsyscall equivalent.

Vsyscalls codes are written in assembler and are stored in arch/i386/kernel (look at arch/x86_64/kernel for x86 64bits processors). As code is written in assembler, it gives a real low probability for code to vary from a system to another. Code is loaded into memory as a library (shared object). At start of vsyscalls pages you can find the ELF signature and other ELF segments are loaded into the address space. Here is a memory map file from a cat process :

```
−−> cat /proc/self/maps
08048000−0804c000 r−xp 00000000 03:01 589888 /bin/cat
0804c000−0804d000 rw−p 00003000 03:01 589888 /bin/cat
0804d000−0806e000 rw−p 0804d000 00:00 0
40000000−40016000 r−xp 00000000 03:01 671776 /lib/ld−2.3.2.so
40016000−40017000 rw−p 00015000 03:01 671776 /lib/ld−2.3.2.so
40017000−40018000 rw−p 40017000 00:00 0
40022000−4014b000 r−xp 00000000 03:01 2048049 /lib/tls/libc−2.3.2.so
4014b000−40153000 rw−p 00129000 03:01 2048049 /lib/tls/libc−2.3.2.so
40153000−40157000 rw−p 40153000 00:00 0
bffff000−c0000000 rw−p bffff000 00:00 0
ffffe000−fffff000 −−−p 00000000 00:00 0
```

The vsyscalls (and a potential RET instruction) we're looking for are mapped at 0xffffe000. If we attach gdb to the running process we can disassemble the memory region:

```
(gdb) x/i 0xffffe413
0xffffe413 <__kernel_vsyscall+19>: ret
```

We've tested this on numerous 2.6.x kernel versions and found that there is a RET at this address every time. An exploit could be a lot more effective using such kind of trick. This is why we are about to study...

A new problem:
As seen previously in our example, our code doesn't add any null byte at the end of the copy. In the case it does, we would hardly be able to make a quick common return−into−stack : it would take us a lot of tries to succeed. We could use vsyscalls to decrease drastically the number of failures.

As things may become a bit confusing now we will slowly go deeper into the problem. In order to understand let's look at a common stack frame again.

Diagram:

```
 high addresses
| ... |
+−−−−−−−−−−−−+
| str |
+−−−−−−−−−−−−+
| EIP |
+−−−−−−−−−−−−+
| |
| buf[256] |
| |
+−−−−−−−−−−−−+
| ... |
 low addresses
```

Remember how it goes on. First there is the copy of str to buf. Now we consider that a null byte write occurs at end of copy: how could an

effective attack occur ? With an usual brute force ?

Solution: ret−onto−jmp:
We know that any function, especially our vulnerable function, is first called by a calling function (e.g. main()). So our stack frame looks like this:

```
 high addresses
| ... |
+−−−−−−−−−−−−+
| EIP |
+−−−−−−−−−−−−+
| EBP | <−− Here is EBP from calling function
+−−−−−−−−−−−−+ 0xbfffeed0
| |
| ... |
| |
+−−−−−−−−−−−−+
| EIP |
+−−−−−−−−−−−−+
| EBP | <−− Here is EBP from our vulnerable function (= 0xbfffeed0)
+−−−−−−−−−−−−+ 0xbfffee90
| |
| buf[256] |
| |
+−−−−−−−−−−−−+
| ... |
 low addresses
```

EBP from our vulnerable function is a stack address. This address is pointing to the base of the calling function's stack−frame (it points on EBP in that stack−frame). That means, as the stack is growing to low addresses, that this pointer to previous stack frame has a value which is greater than the address of buf[256].

In other words if you overwrite the stored EBP value in vulnerable function stack−frame with a null byte you would decrease it and then it may probably point into the buffer.

Indeed, in our example, 0xbfffeed0 is the saved EBP value in our vulnerable function stack−frame. As the copy will overwrite the address with a null byte, and considering we are on little−endian hardware, saved EBP becomes 0xbfffee00.

Now look at a function's common epilog:

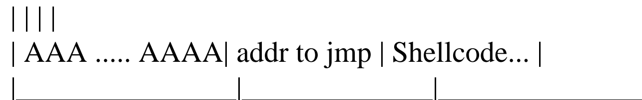
```
    mov %ebp, %esp
    pop %ebp
    ret
```

When it will be first executed in our vulnerable function only EBP register will be affected by our poisoning. At the next epilog (return

from calling function) ESP will be equal to the erroneous EBP register. In
other words the EBP value we modified in stack has been transfered, due to
two successive epilogs, in the register ESP. ESP now points to somewhere
on our buffer... Finally a POP occurs, followed by a RET.

We could have a linear representation of our buf[256] example:

0 128 132 256



AAA ..... AAAA	addr to jmp	Shellcode...
________________	______________	_______________

256 bytes

As our buffer is large enough, we are sure EBP is now pointing to it, but
we don't know exactly where. Anyway what we must do is to have four dummy
bytes followed by an address pointing to a... 'jmp *%esp' instruction !
Take a look at:

(gdb) x/i 0xffffe6cb
0xffffe6cb: jmp *%esp

Considering the linear schema of our buffer, if we nullify the first byte
of EBP and if then it precisely points to offset 124 of our buffer, so on
another successive epilog it will take the four dummy bytes "AAAA", will
store them in EBP, and will return with our address to 'jmp *%esp'. Then
execution should continue on our shellcode and the game is over.

The weakness of such a method is that it needs a few tries to obtain a
buffer which matches the stack requirements. But each cloud has a silver
lining: you can try to inject dummy bytes four by four in your buffer.
Indeed first byte of register EBP is overwritten with a null byte, and
buffer's address should be four bytes aligned. Solution arises in a few
cycles.

Conclusion:
These short examples prove that Linux 2.6 vsyscalls may be used as
powerful attack vectors in buffer overflows exploitations.

In our second example 'jmp *%esp' is not a real instruction written by
kernel developers. In fact we just use dummy bytes in section's data which
are interpreted by the processor as 'jmp *%esp'. This one was found from a
standard disassembly of vsyscalls pages with gdb. Other bytes in vsyscalls
may be considered as interesting exploitable instructions... There may be
a lot of various ways to do the job.

ADDITIONAL INFORMATION

The information has been provided by <mailto:thadeum@gmail.com> Clad
Strife .

The original article can be found at:
<http://www.lse.epita.fr/publications/2005/ret–onto–vsyscall.txt>
http://www.lse.epita.fr/publications/2005/ret–onto–vsyscall.txt

==

This bulletin is sent to members of the SecuriTeam mailing list.
To unsubscribe from the list, send mail with an empty subject line and body to:
list–unsubscribe@securiteam.com
In order to subscribe to the mailing list, simply forward this email to: list–subscribe@securiteam.com

====================
====================

DISCLAIMER:
The information in this bulletin is provided "AS IS" without warranty of any kind.
In no event shall we be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages.