BÁO CÁO THỰC HÀNH

**Môn học: Cơ chế hoạt động của mã độc**

**Lab 4: Simple Worm**

*GVHD: Nguyễn Hữu Quyền*

1. **THÔNG TIN CHUNG:**

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: NT230.N21.ATCL

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Hoàng Văn Anh Đức | 20520890 | 20520890@gm.uit.edu.vn |
| 2 | Nguyễn Mạnh Cường | 20520421 | 20520421@gm.uit.edu.vn |
| 3 | Lê Quang Minh | 20520245 | 20520245@gm.uit.edu.vn |

1. **NỘI DUNG THỰC HIỆN:[[1]](#footnote-1)**

|  |  |  |
| --- | --- | --- |
| **STT** | **Công việc** | **Kết quả tự đánh giá** |
| 1 | C1 | 100% |
| 2 | C2 | 100% |
| 3 | C3 | 70% |

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Công việc** | **Thành viên thực hiện** | **Kết quả tự đánh giá** |
| 1 | C1 | Đức+Cường | 100% |
| 2 | C2 | Đức+Cường | 100% |
| 3 | C3 | Đức+Cường | 70% |

Text

Description automatically generated

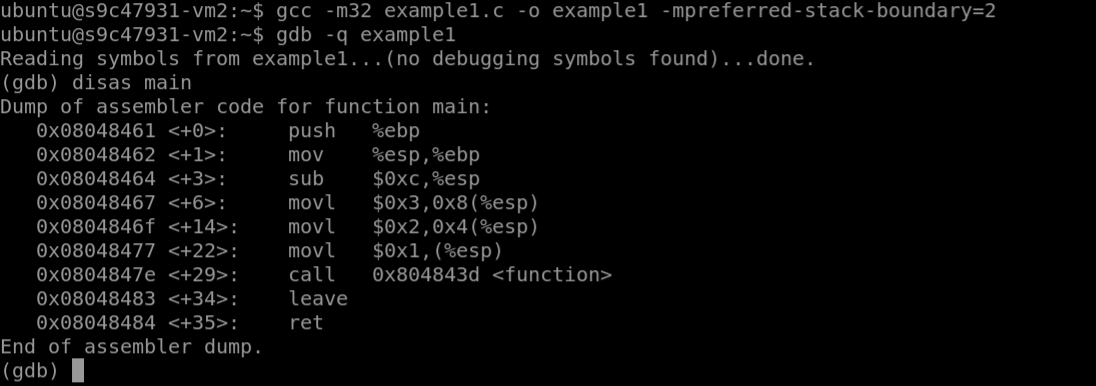
Biên dịch chương trình và sử dụng GDB để disassemble và quan sát các câu lệnh:

*(Option -mpreferred-stack-boundary yêu cầu trình biên dịch đặt stack trên địa chỉ chia hết cho 2)*

* *tối ưu hóa kích thước của stack và giảm tải cho bộ nhớ*

***Vì sao ?***

* Khi một hàm được **gọi** trong **chương trình**, nó thường **tạo ra một số biến cục bộ và đưa chúng lên stack để lưu trữ dữ liệu**. Kích thước của stack sẽ phụ thuộc vào số lượng biến cục bộ được sử dụng trong hàm và kích thước của mỗi biến đó.
* Nếu chúng ta sử dụng tùy chọn -mpreferred-stack-boundary để yêu cầu trình biên dịch đặt stack trên địa chỉ chia hết cho **4 bytes thay vì 16 bytes, kích thước của stack** sẽ được **giảm** đi và **các biến trên stack sẽ sử dụng ít bộ nhớ hơn**. Điều này đặc biệt hữu ích cho các chương trình có tính linh hoạt cao, có thể yêu cầu sử dụng nhiều biến cục bộ và tiêu tốn nhiều bộ nhớ trên stack



* Sau các câu lệnh tạo stack (push ebp; mov ebp, esp), vùng nhớ được giảm đi 0xC (12) để cấp phát cho 3 biến số nguyên được sử dụng như là đối số của hàm function (mỗi số nguyên có kích thước 4 bytes).
* Các đối số được đưa vào stack thông qua lệnh mov đến địa chỉ được lưu trong thanh ghi esp. Thứ tự các đối số được đưa vào stack theo chiều ngược lại:
  + Đầu tiên là giá trị 0x3 được lưu tại (ebp - 0x4),
  + Tiếp theo là giá trị 0x2 được lưu tại (ebp - 0x8),
  + Và cuối cùng là giá trị 0x1 được lưu tại esp (ebp - 12).
* Lệnh call được sử dụng để lưu địa chỉ của lệnh tiếp theo trong hàm main bằng cách push địa chỉ đó vào stack, để khi thoát khỏi hàm function, chương trình có thể trở lại địa chỉ đã được lưu trước đó và tiếp tục thực thi
* Ta thấy esp được trừ đi 0x10 (16) để chừa vùng nhớ cho mảng buffer1 và buffer2

 16 > 15

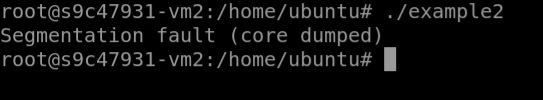
* Ở kiến trúc IA32, vùng nhớ được chia thành các word 4 bytes. Nên khi cấp phát vùng nhớ, trình biên dịch thường sẽ căn chỉnh theo các word. Đó là lí do vì sao ta được cấp phát 16 bytes vùng nhớ (4 words) thay vì chỉ 15 bytes kích thước mảng.

## Buffer Overflow

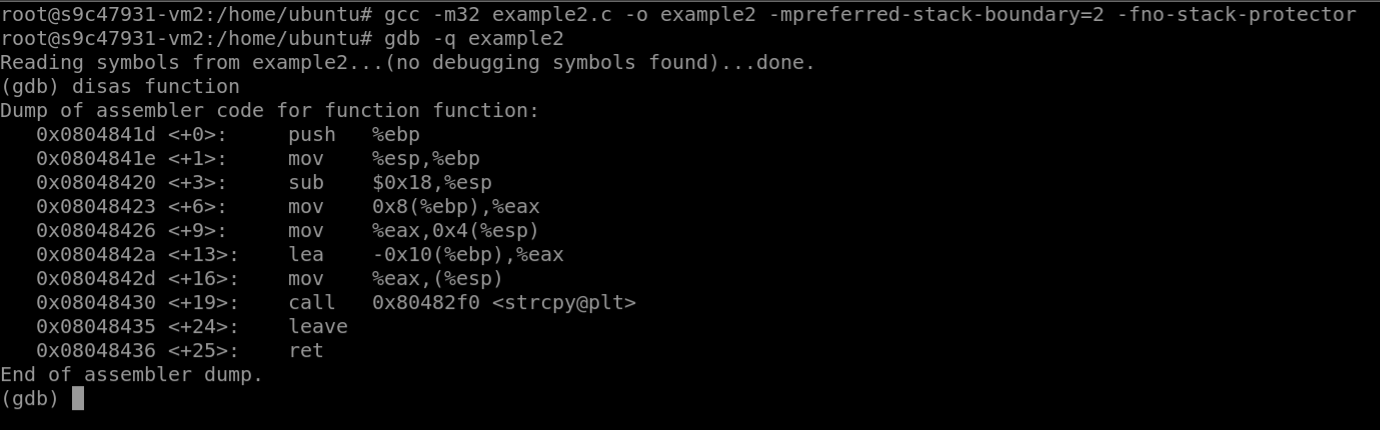
### example2.c

Graphical user interface, text, application

Description automatically generated

* Sử dụng hàm strcpy để sao chép một chuỗi kí tự đã gây ra lỗi buffer overflow trong chương trình. Vấn đề là hàm strcpy không kiểm tra độ dài của chuỗi đầu vào, vì vậy nếu vùng nhớ của chuỗi đích không đủ để chứa toàn bộ chuỗi nguồn, nó sẽ bị tràn.
* Trong đoạn mã được trình bày, chương trình cố gắng sao chép một chuỗi có độ dài 255 kí tự vào một vùng nhớ chỉ có kích thước 16 byte, dẫn đến tràn bộ đệm và gây ra lỗi Segmentation fault khi biên dịch và chạy chương trình.

Quan sát mã hợp ngữ ta thấy được rằng:



Địa chỉ [ebp - 0x10] (16) được gán cho thanh ghi eax và sau đó được đưa vào stack, đây là vùng nhớ đích có kích thước 16 byte.



Khi thực hiện strcpy, các kí tự 'A' thừa sẽ tràn sang các vùng nhớ khác, ghi đè lên ebp và return address của hàm function mà không được kiểm soát.

* Khiến cho giá trị của return address bị thay đổi
* Dựa vào cái này nếu chúng ta chèn payload vào ct và gây ra buffer overflow thì ta có thể chèn địa chỉ bắt đầu payload vào return address

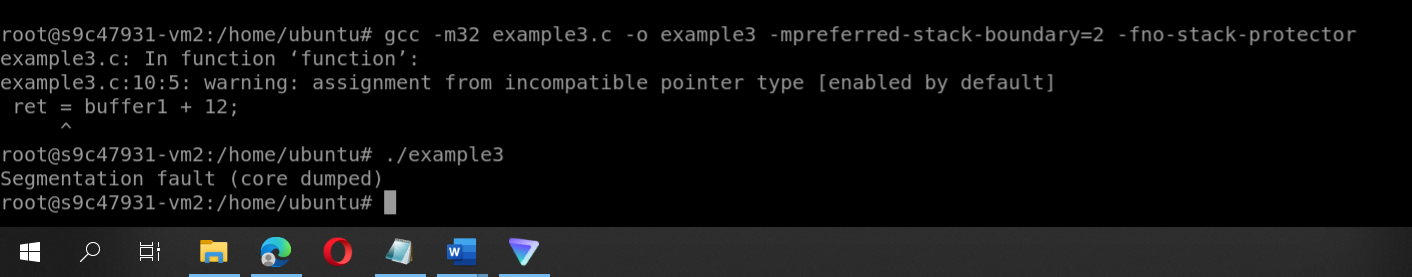
### example3.c

Graphical user interface, text, application

Description automatically generated with medium confidence

* Đoạn code này là một ví dụ của lỗ hổng tràn bộ đệm (buffer overflow) trong ngôn ngữ lập trình C. Khi hàm function() được gọi với các đối số a=1, b=2, và c=3, hai biến đệm buffer1 và buffer2 được tạo ra trên ngăn xếp (stack) của chương trình với kích thước lần lượt là 5 và 10 byte.
* Sau đó, con trỏ ret được khởi tạo để trỏ tới địa chỉ bên trong buffer1 với khoảng cách là 12 byte từ đầu buffer1. Tiếp theo, giá trị tại địa chỉ được trỏ tới bởi ret được tăng lên 8. Việc này dẫn đến việc ghi vào vùng nhớ không được cấp phát cho biến ret, gây ra tràn bộ đệm và có thể làm thay đổi các giá trị khác trong bộ nhớ của chương trình.
* Sau khi hàm function() thực thi xong, biến x được gán giá trị là 1 và in ra màn hình bằng lệnh printf(). Kết quả in ra sẽ là 1. Tuy nhiên, do lỗ hổng tràn bộ đệm, hành vi của chương trình có thể không dự đoán được và có thể dẫn đến các hành vi không mong muốn.

Compile và chạy chương trình trên ta gặp lỗi Segmentation fault:

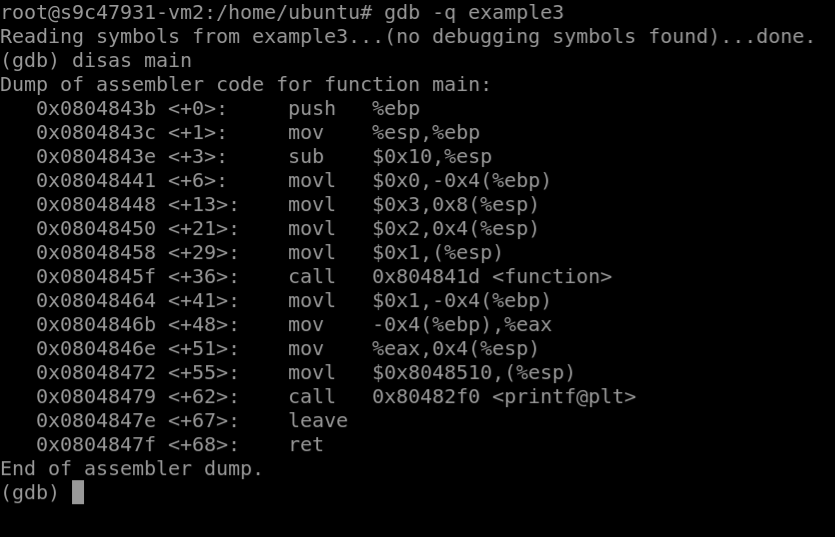


Quan sát mã hợp ngữ của chương trình khi thực thi ta thấy được:

Text

Description automatically generated with medium confidence

* 0x0804841d <+0>: push %ebp: đưa giá trị của thanh ghi ebp vào đỉnh của stack
* 0x0804841e <+1>: mov %esp,%ebp: sao chép giá trị của thanh ghi esp vào thanh ghi ebp
* 0x08048420 <+3>: sub $0x14,%esp: giảm giá trị của esp để cấp phát không gian cho buffer1 và buffer2
* 0x08048423 <+6>: lea -0x9(%ebp),%eax: tính toán địa chỉ của buffer1 và lưu trữ vào thanh ghi eax
* 0x08048426 <+9>: add $0xc,%eax: cộng giá trị 12 vào địa chỉ của buffer1 (tương đương với buffer1 + 12) và lưu trữ kết quả vào thanh ghi eax
* 0x08048429 <+12>: mov %eax,-0x4(%ebp): lưu trữ giá trị của thanh ghi eax vào địa chỉ trỏ đến bởi ebp - 4
* 0x0804842c <+15>: mov -0x4(%ebp),%eax: lấy giá trị từ địa chỉ trỏ đến bởi ebp - 4 và lưu trữ vào thanh ghi eax
* 0x0804842f <+18>: mov (%eax),%eax: lấy giá trị từ địa chỉ lưu trữ bởi eax và lưu trữ vào chính thanh ghi eax
* 0x08048431 <+20>: lea 0x8(%eax),%edx: tính toán địa chỉ của ô nhớ kế tiếp của thanh ghi eax và lưu trữ vào thanh ghi edx
* 0x08048434 <+23>: mov -0x4(%ebp),%eax: lấy giá trị từ địa chỉ trỏ đến bởi ebp - 4 và lưu trữ vào thanh ghi eax
* 0x08048437 <+26>: mov %edx,(%eax): ghi giá trị của thanh ghi edx vào địa chỉ trỏ đến bởi eax
* 0x08048439 <+28>: leave: khôi phục giá trị của esp và ebp để trả về khỏi hàm 0x0804843a <+29>: ret: trả về khỏi hàm.



0x0804843b <+0>: push %ebp

0x0804843c <+1>: mov %esp,%ebp

Hai lệnh này cùng thực hiện việc thiết lập stack frame để chương trình có thể truy cập đến các biến cục bộ và các tham số hàm. %ebp được sử dụng để lưu trữ địa chỉ cơ sở của stack frame, còn %esp lưu trữ địa chỉ của đỉnh stack.

0x0804843e <+3>: sub $0x10,%esp

Lệnh này dịch chuyển đỉnh stack đi 16 byte, tương đương với việc cấp phát một khoảng nhớ mới trên stack cho các biến cục bộ của hàm.

0x08048441 <+6>: movl $0x0,-0x4(%ebp)

Lệnh này khởi tạo giá trị ban đầu cho biến x.

0x08048448 <+13>: movl $0x3,0x8(%esp)

0x08048450 <+21>: movl $0x2,0x4(%esp)

0x08048458 <+29>: movl $0x1,(%esp)

Đây là lệnh chuẩn bị tham số cho hàm function. Cụ thể, giá trị 1, 2, 3 được đặt lần lượt vào các vị trí 0x0c(%esp), 0x08(%esp), và 0x04(%esp) của stack.

0x0804845f <+36>: call 0x804841d <function>

Lệnh này gọi hàm function. Sau khi thực hiện xong, điều khiển quay trở lại địa chỉ tiếp theo.

0x08048464 <+41>: movl $0x1,-0x4(%ebp)

Lệnh này gán giá trị 1 vào biến x

0x0804846b <+48>: mov -0x4(%ebp),%eax

0x0804846e <+51>: mov %eax,0x4(%esp)

0x08048472 <+55>: movl $0x8048510,(%esp)

0x08048479 <+62>: call 0x80482f0 <printf@plt>

Lệnh này in giá trị của biến x ra màn hình.

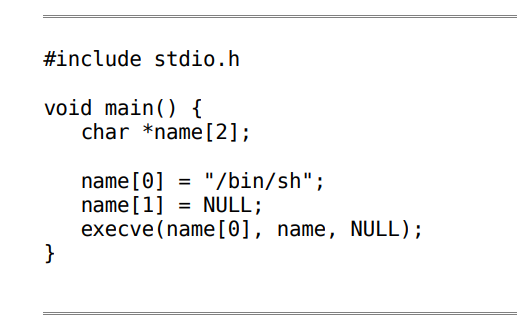
0x0804847e <+67>: leave

0x0804847f <+68>: ret

Hai lệnh này dùng để giải phóng stack frame và trả lại địa chỉ của lời gọi hàm.

## 3.Shellcode

### shellcode.c



* Đây là một chương trình C sử dụng hàm execve để thực thi một chương trình khác.
* Chương trình này sẽ thực hiện lệnh "/bin/sh" để mở một shell trên hệ thống.
* Chương trình sẽ tạo ra một con trỏ đến một mảng hai phần tử kiểu char, trong đó phần tử đầu tiên là "/bin/sh" và phần tử thứ hai là NULL.
* Sau đó, chương trình gọi hàm execve, đưa đường dẫn đến lệnh cần thực thi, tên chương trình và đối số cho chương trình.
* Trong trường hợp này, đối số cuối cùng là NULL, để chỉ ra không có thêm đối số nào được truyền cho chương trình.
* Khi được gọi, hàm execve sẽ thay thế tiến trình hiện tại bằng tiến trình mới được chạy, trong trường hợp này là một shell được mở.

Compile với flag -static để code của hàm execve được bao gồm vào file thực thi. Disassemble và quan sát hàm main:

Text

Description automatically generated

* Dòng 1-2: Khởi tạo frame cho hàm main.
* Dòng 3: Cấp phát 20 byte cho frame (0x14 = 20).
* Dòng 4: Đặt giá trị "/bin/sh" vào địa chỉ ô nhớ -0x8(%ebp).
* Dòng 5: Đặt giá trị 0 vào địa chỉ ô nhớ -0x4(%ebp).
* Dòng 6-7: Đưa địa chỉ của "/bin/sh" vào thanh ghi eax.
* Dòng 8-10: Chuẩn bị tham số cho hàm execve. Đưa địa chỉ của chuỗi NULL vào địa chỉ ô nhớ 0x8(%esp), đưa địa chỉ của mảng name vào địa chỉ ô nhớ 0x4(%esp), và đưa địa chỉ của "/bin/sh" vào địa chỉ ô nhớ (%esp).
* Dòng 11-12: Kết thúc hàm main và trả về giá trị 0.

A picture containing text

Description automatically generated

Execuve là hàm trong thư viện hệ thống (system library) của Linux và được sử dụng để thực thi một chương trình mới:

0x0806c540 <+0>:push %ebx: lưu giá trị của thanh ghi ebx vào stack.

0x0806c541 <+1>:mov 0x10(%esp),%edx: di chuyển giá trị từ vị trí stack offset 0x10 đến thanh ghi edx.

0x0806c545 <+5>:mov 0xc(%esp),%ecx: di chuyển giá trị từ vị trí stack offset 0xc đến thanh ghi ecx.

0x0806c549 <+9>:mov 0x8(%esp),%ebx: di chuyển giá trị từ vị trí stack offset 0x8 đến thanh ghi ebx.

0x0806c54d <+13>:mov $0xb,%eax: di chuyển giá trị 0xb vào thanh ghi eax. Đây là mã hệ thống (system call) tương ứng với execve trong Linux.

0x0806c552 <+18>:call \*0x80ea9f0: gọi hàm system call execve để thực thi chương trình mới.

0x0806c558 <+24>:cmp $0xfffff000,%eax: so sánh giá trị trả về của execve với giá trị tối đa được phép trả về.

0x0806c55d <+29>:ja 0x806c561 <execve+33>: nếu giá trị trả về lớn hơn giá trị tối đa được phép trả về thì nhảy tới địa chỉ 0x806c561 và thực hiện các lệnh tiếp theo.

0x0806c55f <+31>:pop %ebx: lấy giá trị của thanh ghi ebx từ stack. 0x0806c560 <+32>:ret: thoát khỏi hàm execve.

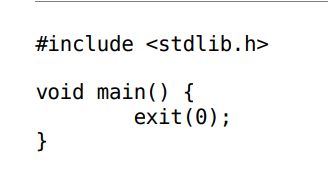
0x0806c561 <+33>:mov $0xffffffe8,%edx: di chuyển giá trị 0xffffffe8 vào thanh ghi edx.

0x0806c567 <+39>:neg %eax: đảo dấu giá trị của thanh ghi eax.

0x0806c569 <+41>:mov %gs:0x0,%ecx: di chuyển giá trị từ vị trí offset 0x0 của thanh ghi gs đến thanh ghi ecx.

0x0806c570 <+48>:mov %eax,(%ecx,%edx,1): lưu giá trị của thanh ghi eax vào vị trí được tính bằng cách thêm giá trị của thanh ghi edx lần kích thước của một

### exit.c



Text

Description automatically generated

0x08048e44 <+0>:push %ebp: Đưa giá trị của thanh ghi ebp lên đỉnh stack

0x08048e45 <+1>:mov %esp,%ebp: Sao chép giá trị của thanh ghi esp vào ebp 0x08048e47 <+3>:and $0xfffffff0,%esp: Thiết lập thanh ghi esp để nó có giá trị bằng với một bội số của 16

0x08048e4a <+6>:sub $0x10,%esp: Cấp phát 16 byte cho stack bằng cách giảm giá trị của esp đi 16

0x08048e4d <+9>:movl $0x0,(%esp): Đưa giá trị số 0 lên đỉnh stack

0x08048e54 <+16>:call 0x804e750 <exit>: Gọi hàm exit với đối số là 0, kết thúc chương trình.

Đoạn code này sử dụng thư viện stdlib.h để sử dụng hàm exit() để kết thúc chương trình với mã trả về là 0. Nói cách khác, khi chương trình thực thi đến đoạn code này, nó sẽ dừng lại và trả về kết quả là 0 cho hệ điều hành.

Quá trình thực hiện của syscall "exit" và cách thực hiện syscall "execve" để thực thi lệnh "/bin/sh". Các bước thực hiện được tóm tắt như sau:

1. Có một chuỗi kết thúc bằng null ,"/bin/sh" trong bộ nhớ
2. Có địa chỉ của chuỗi "/bin/sh" và một null word liền sau nó trong bộ nhớ
3. Copy giá trị 0xb vào thanh ghi EAX
4. Copy địa chỉ của địa chỉ của chuỗi "/bin/sh" vào thanh ghi EBX
5. Copy địa chỉ của chuỗi "/bin/sh" vào thanh ghi ECX
6. Copy địa chỉ của null word vào thanh ghi EDX
7. Thực hiện lệnh syscall "execve" bằng cách sử dụng lệnh "int $0x80"
8. Copy giá trị 0x1 vào thanh ghi EAX Copy giá trị 0x0 vào thanh ghi EBX
9. Thực hiện lệnh syscall "exit" bằng cách sử dụng lệnh "int $0x80"

Đặt chuỗi “/bin/sh” ở cuối, shellcode sẽ có dạng như sau:

Graphical user interface, text

Description automatically generated

Chương trình thực thi có cơ chế PIE làm cho việc xác định địa chỉ của shellcode trở nên khó khăn vì địa chỉ stack sẽ được cộng với một địa chỉ base ngẫu nhiên. Để giải quyết vấn đề này, ta có thể sử dụng kết hợp một lệnh JMP và một lệnh CALL. Hai lệnh này sử dụng địa chỉ tương đối từ con trỏ lệnh, cho phép nhảy đến một offset từ con trỏ lệnh mà không cần biết chính xác địa chỉ nơi cần nhảy đến. Nếu ta đặt lệnh CALL ngay trước chuỗi "/bin/sh", khi lệnh CALL được thực thi, địa chỉ chuỗi "/bin/sh" sẽ được push vào stack làm địa chỉ trả về của lệnh CALL. Sau đó, ta có thể pop địa chỉ này vào một thanh ghi. Shellcode sẽ có dạng như sau:

Graphical user interface, text, table

Description automatically generated

Chúng ta cần tính toán các offset sau đây:

Offset từ lệnh JMP đến lệnh CALL.

Offset từ lệnh CALL đến lệnh POP.

Offset từ địa chỉ của chuỗi "/bin/sh" đến mảng chứa chuỗi.

Offset từ địa chỉ của chuỗi "/bin/sh" đến null word.Table

Description automatically generated

Một số hệ điều hành đánh dấu vùng .text là chỉ đọc. Do đó, chúng ta cần đưa shellcode vào vùng nhớ stack hoặc vùng .data. Vì vậy, chúng ta sẽ đưa code của mình vào một mảng toàn cục trong vùng .data..

**shellcodeasm.c**

Table

Description automatically generated

Đoạn code này là một đoạn assembly được viết trong ngôn ngữ C. Nó sử dụng các lệnh assembly để tạo ra một shellcode, một đoạn mã máy tính được thiết kế để thực thi một shell. Khi chạy, shellcode sẽ thực hiện một số thao tác bao gồm lấy địa chỉ của chuỗi "/bin/sh" trong bộ nhớ và sử dụng nó để gọi hàm execve để thực thi một shell. Shellcode này có thể được sử dụng để thực hiện các cuộc tấn công trên hệ thống mà không cần quyền truy cập đầy đủ.

Text

Description automatically generated

Đưa địa chỉ của ebp vào stack (push %ebp) và đưa địa chỉ của stack pointer vào ebp (mov %esp, %ebp), chuyển sang frame mới của hàm.

Nhảy đến địa chỉ 0x2a. Pop giá trị từ đỉnh ngăn xếp và lưu trữ vào thanh ghi esi (pop %esi). Lưu trữ giá trị của esi vào địa chỉ 0x8(%esi) (mov %esi, 0x8(%esi)).

Thiết lập byte thứ 7 của thanh ghi esi thành 0 (movb $0x0,0x7(%esi)). Thiết lập giá trị của thanh ghi esi+12 thành 0 (movl $0x0,0xc(%esi)).

Thiết lập giá trị của thanh ghi eax thành 0xb (mov $0xb, %eax). Sao chép giá trị của thanh ghi esi vào thanh ghi ebx (mov %esi, %ebx).

Lấy địa chỉ của thanh ghi esi+8 và đưa vào thanh ghi ecx (lea 0x8(%esi), %ecx).

Lấy địa chỉ của thanh ghi esi+12 và đưa vào thanh ghi edx (lea 0xc(%esi), %edx).

Gọi lệnh hệ thống với giá trị 0x80 (int $0x80), với các giá trị trong thanh ghi được sử dụng để thực hiện lệnh.

Thiết lập giá trị của thanh ghi eax thành 1 (mov $0x1, %eax).

Thiết lập giá trị của thanh ghi ebx thành 0 (mov $0x0, %ebx).

Gọi lệnh hệ thống với giá trị 0x80 (int $0x80), để kết thúc tiến trình.

Gọi hàm tại địa chỉ -0x2f (call -0x2f), dự kiến là địa chỉ của hàm dự phòng, tuy nhiên đây là một địa chỉ không hợp lệ. Thực hiện chuỗi ký tự "/bin/sh" (".string "/bin/sh"").

### testc.c

Text

Description automatically generated

Compile và thực thi shellcode trên

*(cần thêm option -z execstack để shellcode ở .data section có thể thực thi được)*

Graphical user interface, text

Description automatically generated

Chương trình đã được thực thi thành công. Tuy nhiên, khi làm việc với character buffer, các kí tự null trong shellcode sẽ được coi là kí tự kết thúc chuỗi, do đó việc sao chép chuỗi sẽ kết thúc khi gặp kí tự này. Vì vậy, để loại bỏ các byte này trong shellcode của chúng ta, ta sẽ phải chỉnh sửa mã.

Table

Description automatically generated with medium confidence

Code assembly lúc này trở thành:

### shellcodeasm2.c

Table

Description automatically generated

Và testsc program trở thành:

### testsc2.c

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

## 4.Writing an Exploit

### overflow1.c

Graphical user interface, text, application, email

Description automatically generated

Ban đầu, chương trình sẽ tạo chuỗi large\_string bằng cách lưu địa chỉ của mảng buffer vào trong chuỗi này. Sau đó, shellcode sẽ được sao chép vào đầu chuỗi large\_string. Hàm strcpy sẽ được sử dụng để sao chép chuỗi large\_string vào mảng buffer mà không thực hiện boundary check. Khi thoát khỏi hàm main, địa chỉ trả về sẽ được ghi đè bằng địa chỉ của biến buffer. Vì vậy, khi đến lệnh ret, chương trình sẽ nhảy đến shellcode được lưu trong biến buffer.

Text

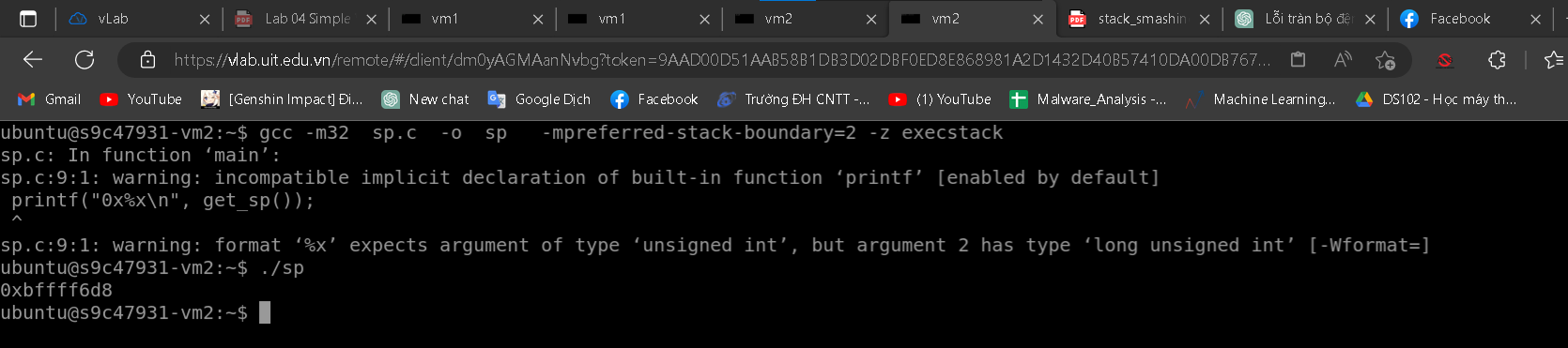
Description automatically generated

Tuy nhiên, khi khai thác một chương trình khác, việc xác định chính xác địa chỉ của vùng nhớ chứa shellcode là khó khăn. Tuy nhiên, đối với stack, khoảng cách giữa các ô nhớ luôn được giữ nguyên. Nếu biết được địa chỉ bắt đầu của stack, ta có thể ước lượng được vị trí lưu trữ shellcode. Dưới đây là chương trình in ra con trỏ của stack:

### sp.c

A picture containing application

Description automatically generated



Chúng ta sẽ thực hiện tấn công overflow trên chương trình sau đây:

### vulnerable.c

Background pattern

Description automatically generated with low confidence

"Chúng ta có thể viết một chương trình nhận đầu vào là buffer size và offset từ stack pointer (của vùng nhớ mắc lỗi buffer overflow), sau đó tạo payload và đưa vào biến môi trường để thuận tiện cho việc khai thác:"

### exploit2.c

Text

Description automatically generated

Text

Description automatically generated

Dễ thấy rằng việc đoán chính xác vị trí của shellcode là khó khăn, thậm chí là không thể, dù cho ta đã biết địa chỉ của stack pointer. Chỉ cần lệch 1 byte, shellcode sẽ không thể thực thi được.

Một cách để tăng khả năng nhảy đúng đến shellcode là pad thêm các byte NOP '\x90'. NOP (no-operation) là các byte không thực hiện bất kỳ tác vụ gì, chỉ đơn giản là nhảy sang lệnh tiếp theo. Bằng cách pad thêm một số lượng lớn byte NOP, ta có thể khiến chương trình "trượt" đến shellcode của mình thay vì phải đúng chính xác địa chỉ của shellcode. Ta sẽ đặt shellcode ở giữa payload và pad thêm các byte NOP ở đầu payload. Nếu return address rơi vào giữa các byte NOP, chương trình sẽ tiếp tục thực thi cho đến khi đến shellcode.

Vì vậy, chương trình khai thác lúc này sẽ được sửa đổi như sau:

### exploit3.c

A picture containing text

Description automatically generated

Có thể chọn một buffer size lớn hơn khoảng 100 byte so với kích thước của buffer cần khai thác. Việc này sẽ đặt shellcode ở phía cuối của buffer, tạo thêm chỗ trống để thêm các byte NOP và vẫn có thể ghi đè được địa chỉ trả về của hàm. Ví dụ, nếu kích thước của buffer cần khai thác là 512 byte, ta có thể chọn kích thước buffer là 800. Sau đó, biên dịch và chạy chương trình khai thác mới, và ta sẽ có được quyền điều khiển shell.

Text

Description automatically generated

Text

Description automatically generated

**C.2 Remote Buffer Overflow**

1.Thực thi vul\_server.c để máy chủ mở cổng 5000

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Trên máy client thao tác

Text

Description automatically generated

Thấy client kết nối đến

Text

Description automatically generated with medium confidence

=====

**Sau khi hoàn thành đoạn những thao tác trên. Chúng ta chuyển sang trường hợp phức tạp hơn.**

**Lắng nghe trên port 4444 để nhận connect từ server**

Graphical user interface, text, application

Description automatically generated

**Trên server chạy ct vul\_server để client exploit trên port 5000**



**Telnet đến Server để xem địa chỉ của buffer**

Text

Description automatically generated

Sau “our” là 1 phần string của hàm không đc in ra , chính chỗ này gây ra stack overflow



Ta có thể thấy address trả về là nơi con trỏ của lệnh EIP (PC) sẽ trỏ vào khi kết thúc hàm

Vì remoteexploit.c của chúng ta là chèn các lệnh nop và sau các lệnh nop là shellcode nên ta phải chỉnh giá trị của RET đến các lệnh nop đó vì khi thực hiện xong các lệnh nop nó sẽ thực hiện shellcode

**Config lại file remoteexploit.c để Server connect back lại client**

Ở đây ta không sửa chính xác thành Retrurn address (0xff9e5464) vì tại vị trí đó không chứa các lệnh nop mà ta chèn vào

(do phần string vượt quá buffer “có thể in ra ngoài màn hình” nên chèn lên sau return address nên ta phải sửa RET để nó lưu địa chỉ của câu lệnh nop)

Khi dùng debug thì ta thấy các lệnh nop ở địa chỉ không phải address trả về mà ở sau đó , ta có thể sửa thành bất cứ địa chỉ vào có chứa lệnh nop

Câu lệnh để debug vul\_ser trên máy vimtim :

Gdb -q ./vul\_server

x/10i 0xff945464

x là lệnh examine memory để xem nội dung của vùng nhớ

. /10i là định dạng xuất ra 10 instruction tiếp theo từ địa chỉ cần xem.

0xff945464 là địa chỉ của vùng nhớ cần xem.

Ta sẽ thấy đc lệnh nop ở đâu sau địa chỉ 0xff945464 và thay RET = địa chỉ của lệnh nop

Text

Description automatically generated

Shellcode[33]-> Shellcode[36] : là ip của attacker (client)

Exploit server

Text

Description automatically generated

Server lắng nghe được kết nối đến và connect back lại attacker (client)

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Tạo 1 chương trình mới “simpleworm” dựa trên remoteexploit.c để thực hiện khai thác cho server connect lại client qua port 4444 và mở port 10k trên server ở client truyền ct “simpleworm” và thực thi nó tự động trên server với mục đích lây nhiễm server khác

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Mở port trên client cho server connect



Biên dịch simpleworm



Trên máy server trc khi truyền simpleworm

Text

Description automatically generated

A picture containing text

Description automatically generated

Mở port 5000 trên server



Tiến hành chạy simpleworm







Trên máy Server

Text

Description automatically generated

Website

Description automatically generated with medium confidence

1. Ghi nội dung công việc, các kịch bản trong bài Thực hành [↑](#footnote-ref-1)