**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**HỌC VIỆN KỸ THUẬT MẬT MÃ**



**BÁO CÁO BÀI TẬP LỚN**

Sinh viên thực hiện :

Đào Huy Hà – AT160122

Giảng viên hướng dẫn:

**TS. Nguyễn Tuấn Anh**

Khoa An toàn thông tin – Học viện Kỹ thuật mật mã

Hà Nội 2023

[MỤC LỤC i](#_Toc136798865)

[GIỚI THIỆU MÔI TRƯỜNG THỰC HÀNH 1](#_Toc136798866)

[Level01 2](#_Toc136798867)

[Nhiệm vụ 2](#_Toc136798868)

[Bước 1: Phân tích mã nguồn 2](#_Toc136798869)

[Bước 2: Lấy flag 2](#_Toc136798870)

[Level02 3](#_Toc136798871)

[Nhiệm vụ 3](#_Toc136798872)

[Bước 1: Phân tích mã nguồn 3](#_Toc136798873)

[Bước 2: Lấy flag 3](#_Toc136798874)

[Bufferoverflow-overwrite-command 4](#_Toc136798875)

[Mô tả nhiệu vụ 4](#_Toc136798876)

[Bước 1: Phân tích mã nguồn 4](#_Toc136798877)

[Bước 2 : Biên dịch mã nguồn để tạo file thực thi 4](#_Toc136798878)

[Bước 4: Lấy flag 5](#_Toc136798879)

[Bufferoverflow-homemade-cookie-v1 6](#_Toc136798880)

[Bước 1: Phân tích mã nguồn 6](#_Toc136798881)

[Bước 2: Tạo kịch bản khai thác lỗ hổng: 7](#_Toc136798882)

[Bước 3: Tạo payload: 7](#_Toc136798883)

[Formatstring-leak-flag-in-mem-stack 9](#_Toc136798884)

[Mô tả nhiệm vụ 9](#_Toc136798885)

[Bước 1: Phân tích mã nguồn 9](#_Toc136798886)

[Bước 2: Chạy file thực thi ‘Formatstring-leak-flag-in-mem-stack’ 10](#_Toc136798887)

[Bước 3: Lấy flag 10](#_Toc136798888)

[Integer-overflow-1 13](#_Toc136798889)

[Mô tả nhiệm vụ 13](#_Toc136798890)

[Bước 1: Phân tích mã nguồn 13](#_Toc136798891)

[Bước 2: Lấy flag 14](#_Toc136798892)

[Integer-overflow-2 15](#_Toc136798893)

[Mô tả nhiệm vụ 15](#_Toc136798894)

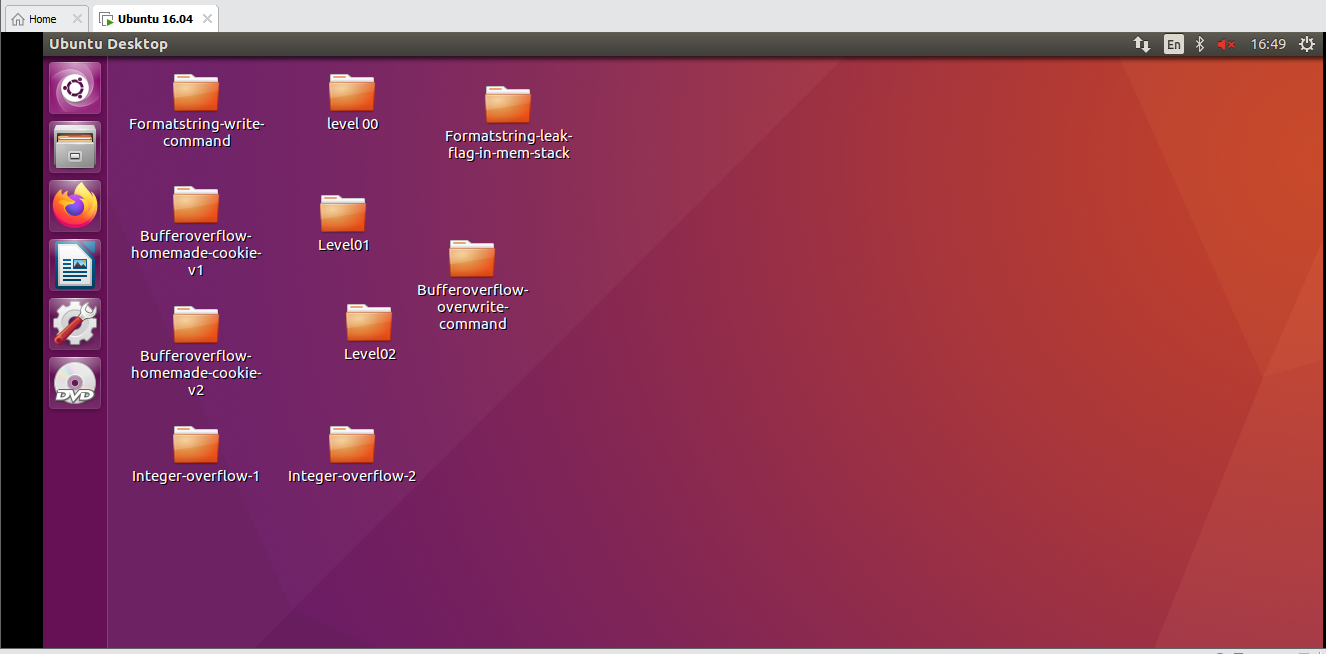
[Bước 1: Phân tích mã nguồn 15](#_Toc136798895)

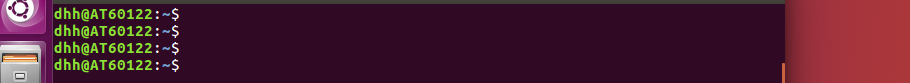
[Bước 2: Lấy flag 16](#_Toc136798896)

[TỰ CHẤM ĐIỂM 18](#_Toc136798897)

# GIỚI THIỆU MÔI TRƯỜNG THỰC HÀNH

Sử dụng Kali Linux làm môi trường thực hành. Cài đặt máy ảo Ubuntu 16.04 trên công vụ VMware Workstation và đặt hostname là mã sinh viên, username là viết tắt các chữ cái đầu tiên trong họ và tên của sinh viên.





Sử dụng công cụ decode online theo đường dẫn :

<https://www.rapidtables.com/convert/number/hex-to-ascii.html>

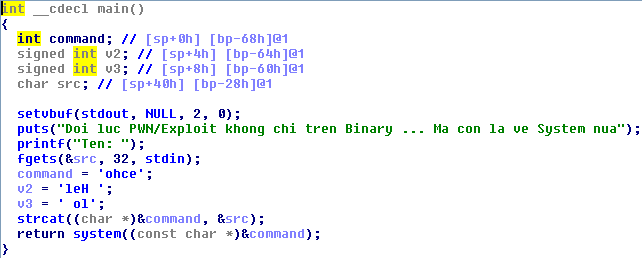
<https://www.scadacore.com/tools/programming-calculators/online-hex-converter/>

# Level01

## Nhiệm vụ

Lấy flag bằng cách kết thúc câu lệnh.

## Bước 1: Phân tích mã nguồn

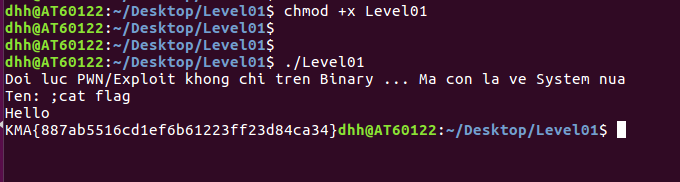


Biến ‘scr’ có độ lớn là 40 byte nhưng đọc vào 32 ký tự. Như vậy không thể khai thác lỗi tràn bộ đệm. Chuỗi command = “echo Hello ”, sau đó sẽ được nối với chuỗi scr trên để tạo thành câu lệnh “echo Hello <src>”.

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 2: Lấy flag

Sử dụng dấu ‘;’ sau đó dùng lệnh ‘ cat flag’ để lấy flag.



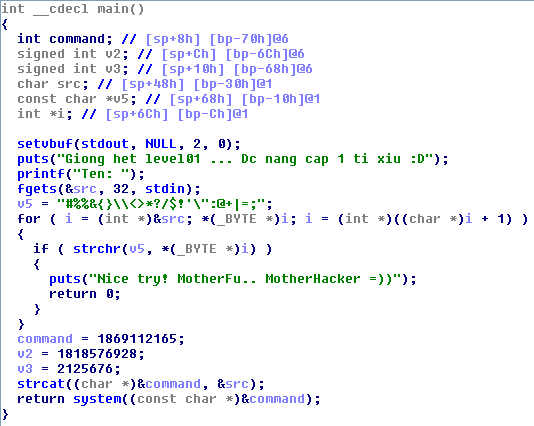
Flag : KMA{887ab5516cd1ef6b61223ff23d84ca34}

# Level02

## Nhiệm vụ

Lấy flag bằng cách sử dụng backtick

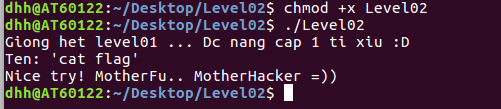
## Bước 1: Phân tích mã nguồn



Giống với Level01 nhưng chỉ cho phép thực thi một câu lệnh là “echo Hello <scr>”.

Sử dụng backstick thay thế lệnh bên trong nó thành output. Sử dung dấu ‘``’để lấy flag qua câu lệnh ‘cat flag’.

## Bước 2: Lấy flag



Như vậy, flag là ‘KMA{213979182bc06fef699425b09bbad7cc}’

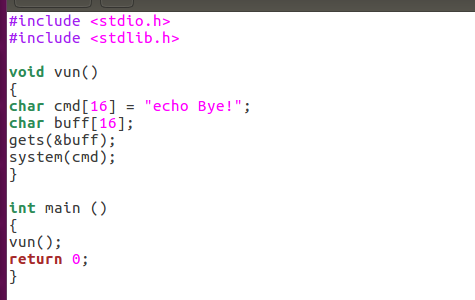
# Bufferoverflow-overwrite-command

## Mô tả nhiệu vụ

Thực hiện khai thác tràn bộ đệm để lấy được flag.

## Bước 1: Phân tích mã nguồn

Mã nguồn :

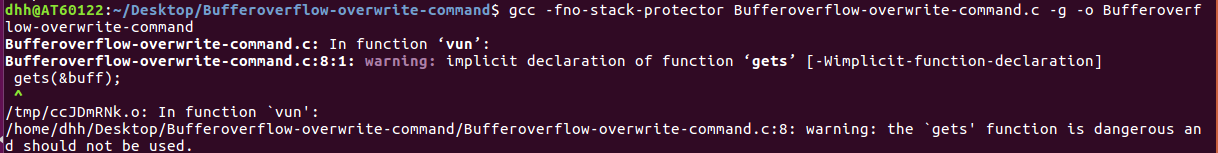


Biến ‘buff’ được khai báo với kích thước 16 byte. Khi nhập vào một chuỗi dài hơn 16 byte, hàm ‘gets()’ sẽ ghi đè lên không gian bộ nhớ vượt quá kích thước được cấp phát cho ‘buff’.

Cần biện dịch file mã nguồn thành file thực thi để có thế chạy trên hệ thống.

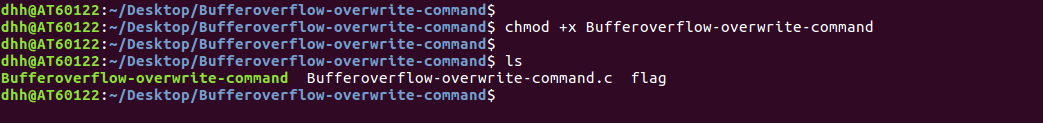
## Bước 2 : Biên dịch mã nguồn để tạo file thực thi

Tiếp theo đó chúng ta sẽ biên dịch file bằng câu lệnh gcc, tham số ” -fno-stack-protector ” cho phép chạy file mà không kiểm tra lỗi tràn bộ đệm, tiếp theo là đến file nguồn Bufferoverflow-overwrite-command.c : -g cho phép chúng ta “Debug” file: -o là file đầu ra



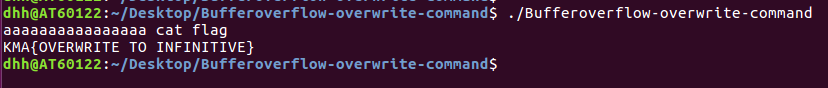
Cấp quyền thực thi cho file Bufferoverflow-overwrite-command mới tạo.

Sử dụng câu lệnh : ‘chmod +x Bufferoverflow-overwrite-command’ để cấp quyền thực thi cho file Bufferoverflow-overwrite-command.



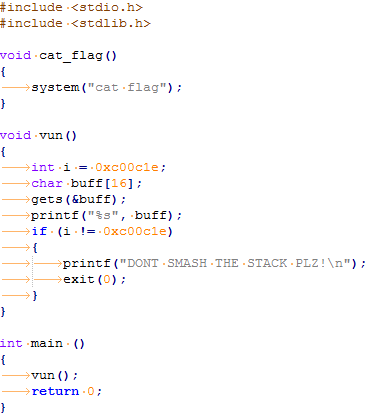
## Bước 4: Lấy flag

Chạy file thực thi ‘overflow’, cần nhập 16 kí tự rác sau đó nhập câu lệnh ‘cat flag’ để lấy flag.



# Bufferoverflow-homemade-cookie-v1

## Bước 1: Phân tích mã nguồn



Hình 2.28 Mã nguồn Bufferoverflow-homemade-cookie-v1

Trong đoạn mã trên, ta thực hiện một hàm vun() để tạo một lỗ hổng buffer overflow. Ta sẽ phân tích chi tiết cách khai thác lỗ hổng buffer overflow trong trường hợp này.

Hàm vun():

Biến i được khởi tạo với giá trị 0xc00c1e.

Mảng buff có độ dài là 16.

Hàm gets() được sử dụng để đọc một chuỗi từ người dùng và ghi vào mảng buff.

Tiếp theo, chuỗi buff được in ra bằng hàm printf().

Sau đó, kiểm tra xem giá trị của biến i có bằng giá trị ban đầu 0xc00c1e hay không. Nếu không, in ra thông báo "DONT SMASH THE STACK PLZ!" và kết thúc chương trình.

## Bước 2: Tạo kịch bản khai thác lỗ hổng:

Với độ dài của mảng buff là 16, ta có thể nhập vào một chuỗi dài hơn để gây tràn bộ đệm và ghi đè lên biến i.

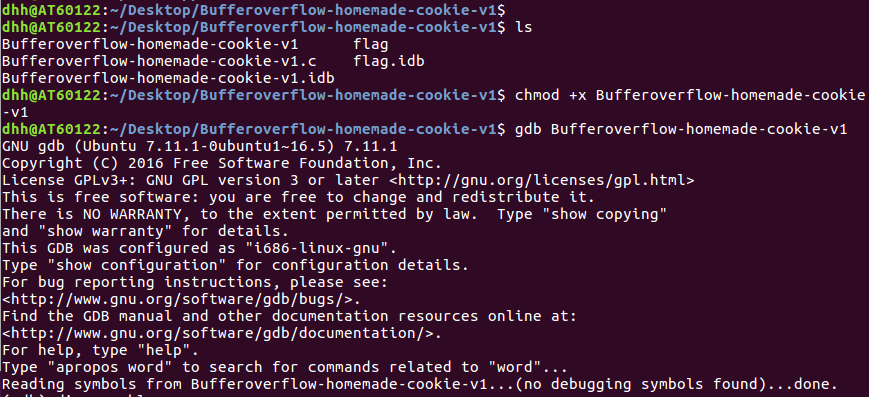
Địa chỉ của biến i nằm sau mảng buff trong bộ nhớ.

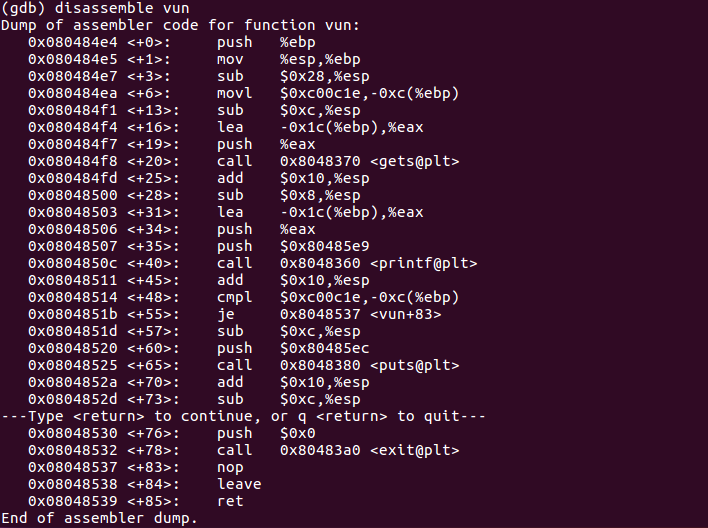
Nếu ta nhập một chuỗi đủ dài, nó sẽ tràn qua mảng buff và ghi đè lên biến i. Điều này làm thay đổi giá trị của biến i.

Khi chương trình quay lại hàm main(), nó không thể xác định xem biến i có bị thay đổi hay không và sẽ tiếp tục thực hiện các lệnh tiếp theo.

## Bước 3: Tạo payload:

Để tận dụng lỗ hổng buffer overflow, ta có thể tạo một payload bằng cách nhập đủ ký tự để gây tràn qua mảng buff và ghi đè lên biến i bằng giá trị khác 0xc00c1e.

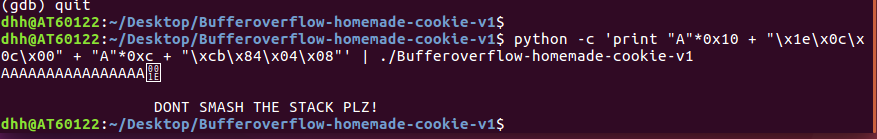
Sau đó, ta có thể ghi đè lên ô nhớ chứa địa chỉ trả về của hàm main() bằng địa chỉ của hàm cat\_flag().



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"'



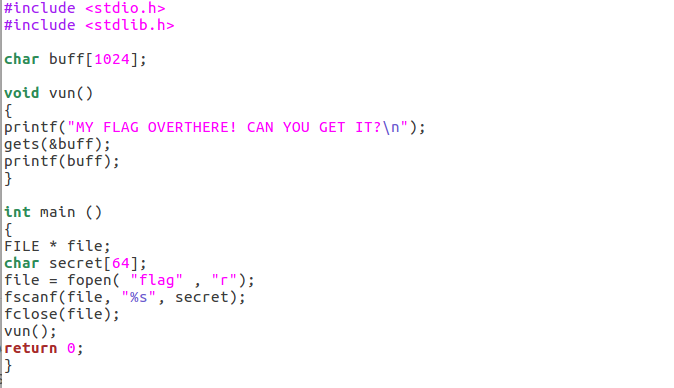
# Formatstring-leak-flag-in-mem-stack

## Mô tả nhiệm vụ

Đọc bộ nhớ stack này để lấy FLAG trong đoạn mã.

## Bước 1: Phân tích mã nguồn

Mã nguồn:

 Trong đoạn code trên, hàm vun() sử dụng hàm gets() để đọc một chuỗi từ người dùng và sau đó in chuỗi đó ra màn hình bằng hàm printf(). Tuy nhiên, việc sử dụng hàm gets() là một lỗ hổng bảo mật nghiêm trọng.

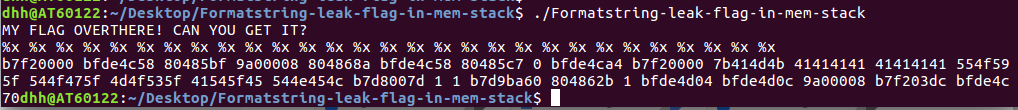
Lỗ hổng format string có thể được khai thác thông qua việc sử dụng các chuỗi format trong hàm printf(). Nếu người dùng cung cấp một chuỗi format đặc biệt, nó có thể dẫn đến việc đọc hoặc ghi các giá trị từ bộ nhớ. Trong trường hợp này, người dùng có thể sử dụng chuỗi format để đọc giá trị của biến secret từ bộ nhớ.

Ví dụ, người dùng có thể nhập chuỗi format %s để yêu cầu hàm printf() in ra giá trị của biến secret. Điều này sẽ dẫn đến việc đọc dữ liệu từ bộ nhớ và hiển thị giá trị của secret. Tuy nhiên, để thành công khai thác lỗ hổng này, người dùng cần biết định dạng của chuỗi format và vị trí của biến secret trong bộ nhớ.

Lưu ý rằng việc sử dụng hàm gets() cũng là một lỗ hổng bảo mật nghiêm trọng vì nó không kiểm tra độ dài của chuỗi nhập vào và có thể dẫn đến tràn bộ nhớ. Để khắc phục lỗ hổng này, nên sử dụng hàm an toàn hơn như fgets() và kiểm tra độ dài của chuỗi nhập vào trước khi sử dụng.

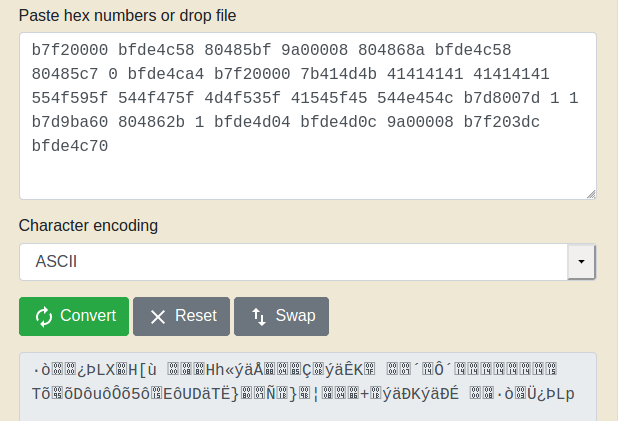
## Bước 2: Chạy file thực thi ‘Formatstring-leak-flag-in-mem-stack’

Chạy file thực thi và nhập ‘%x’ với số lượng tùy ý để lấy flag dưới dạng mã hex.

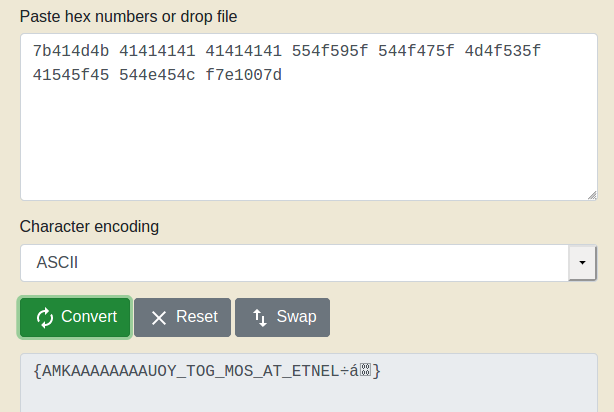


Chúng ta cần decode hex đoạn mã trên để lấy flag.

## Bước 3: Lấy flag



Ta thấy đoạn ‘7b414d4b 41414141 41414141 554f595f 544f475f 4d4f535f 41545f45 544e454c f7e1007d’ có flag là:



Cần phải sắp xếp lại đoạn mã ‘7b414d4b 41414141 41414141 554f595f 544f475f 4d4f535f 41545f45 544e454c f7e1007d’ theo kiểu sắp xếp little endian để lấy được flag chính xác. Sau khi sắp xếp đoạn mã có kết quả là :

4B 4D 41 7B

41 41 41 41

41 41 41 41

5F 59 4F 55

5F 47 4F 54

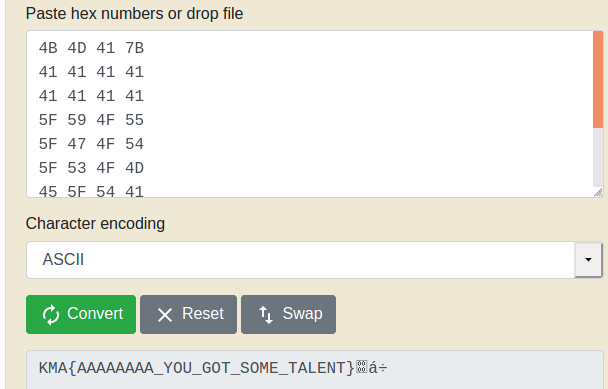
5F 53 4F 4D

45 5F 54 41

4C 45 4E 54

7D 00 E1 F7

Decode đoạn mã vừa sắp xếp ta được kết quả là :



* Flag là: KMA{AAAAAAAA\_YOU\_GOT\_SOME\_TALENT}

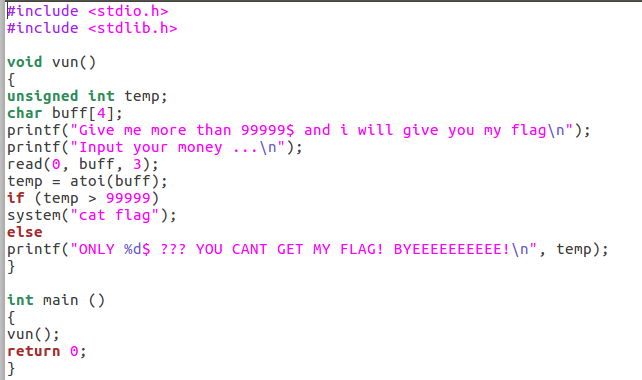
# Integer-overflow-1

## Mô tả nhiệm vụ

Khai thác lỗ hổng tràn số nguyên trong đoạn code với yêu cầu được nhập 3 kí tự giá trị số lớn 5 chữ số.

## Bước 1: Phân tích mã nguồn

Mã nguồn:

Trong hàm vun(), chúng ta có biến temp kiểu unsigned int và mảng buff kiểu char có kích thước 4. Sau đó, chương trình sử dụng hàm read để đọc đầu vào từ người dùng với giới hạn là 3 kí tự. Điều này có thể dẫn đến tràn bộ đệm (buffer overflow), vì buff chỉ có độ dài 4, nhưng hàm read được sử dụng để đọc 3 kí tự.

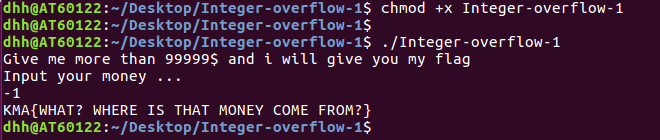
Sau khi đọc chuỗi vào biến buff, chương trình sử dụng atoi(buff) để chuyển đổi chuỗi thành một số nguyên. Tuy nhiên, nếu chuỗi nhập vào có giá trị quá lớn, lớn hơn giới hạn của kiểu unsigned int, atoi sẽ trả về một giá trị không chính xác, gây ra lỗi integer overflow.

Trong kiến trúc 32-bit, kiểu unsigned int có giới hạn giá trị từ 0 đến 4294967295 (2^32 - 1). Khi giá trị nhập vào vượt quá giới hạn này, giá trị của temp sẽ không lưu trữ được giá trị chính xác và việc so sánh temp > 99999 có thể không đúng. Điều này đồng nghĩa với việc người dùng có thể vượt qua điều kiện kiểm tra và hiển thị nội dung của file "flag".

Để khai thác lỗ hổng này, người dùng có thể nhập một chuỗi đủ lớn để vượt quá giới hạn của kiểu unsigned int, và atoi sẽ trả về một giá trị vượt quá giới hạn cho phép. Vì vậy, giá trị của temp sẽ không chứa giá trị chính xác và điều kiện temp > 99999 có thể trở thành đúng, cho phép người dùng hiển thị nội dung của file "flag".

## Bước 2: Lấy flag

Chạy file thực thi và nhập giá trị -1 để lấy flag



Thu được:

‘KMA{WHAT? WHERE IS THAT MONEY COME FROM?}’

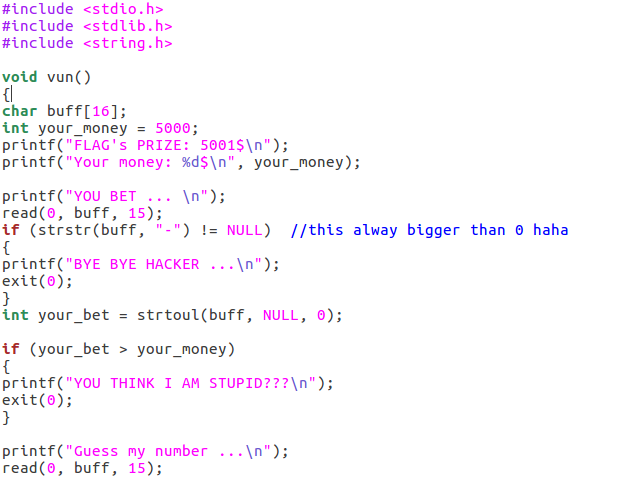
# Integer-overflow-2

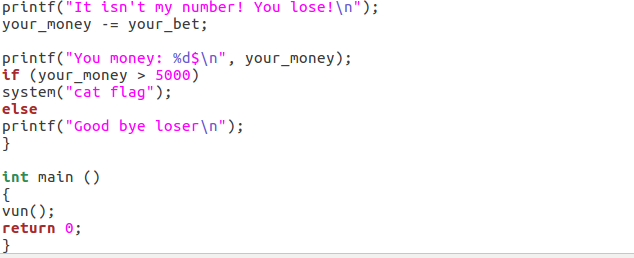
## Mô tả nhiệm vụ

Khai thác lỗi tràn số nguyên giá trị MAX = -1 và lấy flag.

## Bước 1: Phân tích mã nguồn

Mã nguồn :



Trong đoạn code trên, hàm strtoul được sử dụng để chuyển đổi một chuỗi thành một giá trị số nguyên dương kiểu unsigned long int. Tuy nhiên, nếu chuỗi đầu vào là một số âm và bắt đầu bằng dấu trừ ("-"), hàm strtoul sẽ trả về một giá trị lớn hơn giới hạn của biến your\_bet, dẫn đến lỗi integer overflow.

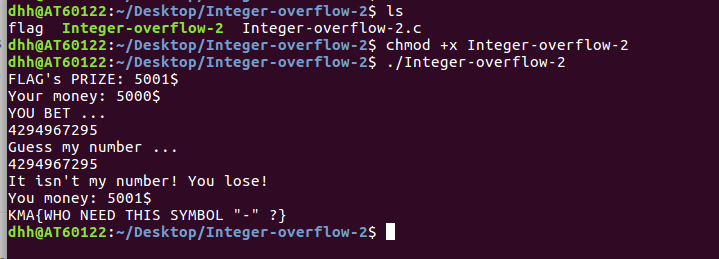
Trong hệ thống 32-bit, kiểu unsigned long int có giới hạn giá trị từ 0 đến 4294967295 (2^32 - 1). Khi ta cố gắng chuyển đổi chuỗi "-1" thành kiểu unsigned long int, hàm strtoul sẽ trả về giá trị 4294967295. Điều này xảy ra vì hàm strtoul không xử lý số âm, mà chỉ xử lý các giá trị không dấu.

Vì vậy, trong trường hợp này, nếu người dùng nhập vào chuỗi "-1", hàm strtoul sẽ trả về giá trị 4294967295, và sau đó giá trị này được gán cho biến your\_bet kiểu int. Khi giá trị này được gán vào một biến kiểu int, xảy ra lỗi integer overflow và biến your\_bet sẽ có giá trị -1.

Lỗi integer overflow này có thể được khai thác để kiểm soát luồng thực thi và gây ra các hành vi không mong muốn trong chương trình.

## Bước 2: Lấy flag

Chạy file thực thi và điền input 4294967295



Flag : KMA{WHO NEED THIS SYMBOL "-" ?}

# TỰ CHẤM ĐIỂM

Điểm tự đánh giá: …7.0 (điểm)

Diễn giải điểm tự chấm:

|  |  |
| --- | --- |
| **Tiêu chí** | **Diễn giải** |
| Số lượng bài làm được: ≥ 7 | Đã làm được 7 bài |
| Có ít nhất một bài về format string | Đã làm bài Formatstring-leak-flag-in-mem-stack |
| Có ít nhất một bài về buffer overflow | Đã làm bài Bufferoverflow-overwrite-command, Bufferoverflow-homemade-cookie-v1 |
| Có ít nhất một bài về integer overflow | Đã làm bài Integer-overflow-1, Integer-overflow-2 |