**Intro to x86 64**

Máy tính thực thi mã máy, được mã hóa dưới dạng byte, để thực hiện các tác vụ trên máy tính. Vì các máy tính khác nhau có bộ xử lý khác nhau, mã máy được thực hiện trên các máy tính này dành riêng cho bộ xử lý. Trong trường hợp này, chúng ta sẽ xem xét kiến trúc bộ hướng dẫn Intel x86-64 được tìm thấy phổ biến nhất hiện nay. Mã máy thường được đại diện bởi một hình thức dễ đọc hơn của mã được gọi là mã assembly. Máy này là mã thường được sản xuất bởi một trình biên dịch, lấy mã nguồn của tệp và sau khi trải qua một số giai đoạn trung gian, tạo ra mã máy có thể được thực thi bởi máy tính. Không đi sâu vào chi tiết, Intel lần đầu tiên bắt đầu bằng cách xây dựng bộ hướng dẫn 16 bit, tiếp theo là 32 bit, sau đó cuối cùng họ đã tạo ra 64 bit. Tất cả các bộ hướng dẫn này đã được tạo ra để tương thích reverse, vì vậy mã được biên dịch cho kiến trúc 32 bit sẽ chạy trên các máy 64 bit. Như đã đề cập trước đó, trước khi một tệp thực thi được tạo ra, mã nguồn đầu tiên được biên dịch thành các tệp assembly(.s), sau đó bộ lắp ráp chuyển đổi nó thành một chương trình đối tượng (tệp.o) và các hoạt động với một liên kết cuối cùng làm cho nó có thể thực thi được. Cách tốt nhất để thực sự bắt đầu giải thích assembly là bằng cách lặn vào. Chúng tôi sẽ sử dụng radare2 để làm điều này - radare2 là một khuôn khổ cho kỹ thuật đảo ngược và phân tích nhị phân. Nó có thể được sử dụng để tháo rời nhị phân (dịch mã máy sang lắp ráp, thực sự có thể đọc được) và gỡ lỗi các nhị phân nói trên (bằng cách cho phép người dùng bước qua việc thực hiện và xem trạng thái của chương trình).

The first step is to execute the program intro by running ./intro

Which then just shows the following output  
https://lh5.googleusercontent.com/JjT_G7sF5ScGMJWTisYH3N49djt64Dx2_6CkOtXBSezoheO0uo7wlu0FQBLBLTyjA_PsRDHrRYTYrvqtA0NVFG0Kt2EGosxx7QvBf32cEjSMSYEOh85uRFJFKy2AxLhsovfUTT9O

Từ việc thực hiện, có thể thấy rằng chương trình đang tạo ra hai biến và chuyển đổi giá trị của chúng. Thời gian để xem những gì nó thực sự làm dưới mui xe!

Go to the introduction folder on the virtual machine and run the command:

r2 –d intro

Điều này sẽ mở binary trong chế độ gỡ lỗi. Khi binary được mở, một trong những điều đầu tiên cần làm là yêu cầu r2 phân tích chương trình và điều này có thể được thực hiện bằng cách nhập vào: aa

Đó là lệnh phân tích phổ biến nhất. Nó phân tích tất cả các ký hiệu và điểm vào trong thực thi. The run e asm.syntax=att to set the disassembly syntax to AT&T.

Phân tích trong trường hợp này liên quan đến việc trích xuất tên chức năng, thông tin kiểm soát dòng chảy và nhiều hơn nữa! hướng dẫn r2 thường dựa trên một ký tự duy nhất, vì vậy thật dễ dàng để có thêm thông tin về các lệnh. Để được trợ giúp chung, hãy chạy:

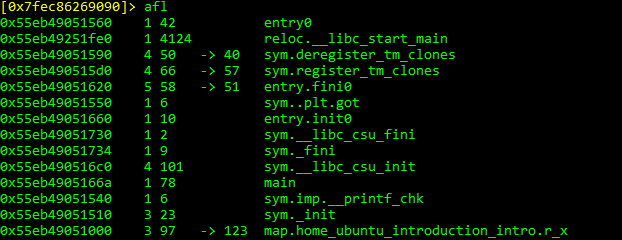
?

Để biết thêm thông tin cụ thể, ví dụ, về analysis , chạy

a?

Khi phân tích hoàn tất, bạn sẽ muốn biết bắt đầu phân tích từ đâu - hầu hết các chương trình đều có điểm vào được xác định là chính. Để tìm danh sách các hàm đang chạy:

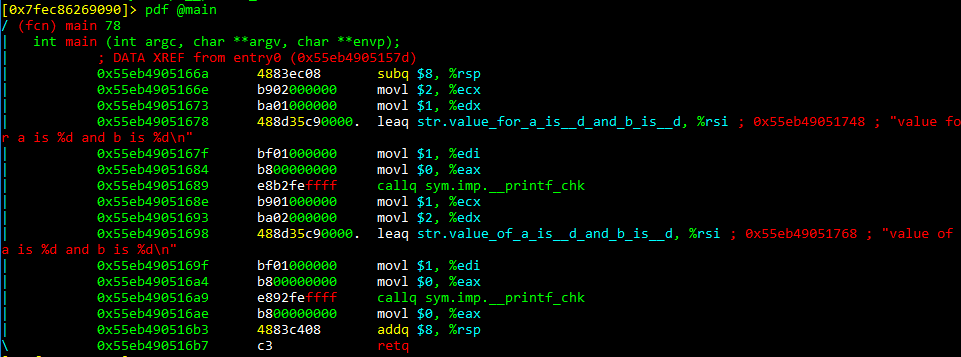
afl



Như đã thấy ở đây, thực sự có một chức năng tại Main. Hãy xem xét mã assembly tại Main bằng cách chạy lệnh

pdf @main

Trong đó pdf có nghĩa là in chức năng tháo rời. Làm như vậy sẽ cho chúng ta following view sau đây.



Như chúng ta có thể thấy từ trên, các giá trị trên cột bên trái hoàn chỉnh là địa chỉ bộ nhớ của các hướng dẫn và chúng thường được lưu trữ trong một cấu trúc gọi là stack (mà chúng ta sẽ nói về sau). Cột giữa chứa các hướng dẫn được mã hóa trong byte (thường là mã máy) và cột cuối cùng thực sự chứa các hướng dẫn có thể đọc được của con người.

Cốt lõi của assembly language liên quan đến việc sử dụng các thanh ghi để thực hiện những việc sau:

* Truyền dữ liệu giữa bộ nhớ và thanh ghi, và ngược lại
* Thực hiện các phép toán số học trên sổ đăng ký và dữ liệu
* Chuyển quyền kiểm soát sang các phần khác của chương trình

Vì kiến trúc là x86-64, các thanh ghi là 64 bit và Intel có danh sách 16 registers:

|  |  |
| --- | --- |
| **64 bit** | **32 bit** |
| %rax | %eax |
| %rbx | %ebx |
| %rcx | %ecx |
| %rdx | %edx |
| %rsi | %esi |
| %rdi | %edi |
| %rsp | %esp |
| %rbp | %ebp |
| %r8 | %r8d |
| %r9 | %r9d |
| %r10 | %r10d |
| %r11 | %r11d |
| %r12 | %r12d |
| %r13 | %r13d |
| %r14 | %r14d |
| %r15 | %r15d |

Mặc dù các sổ register là 64 bit, có nghĩa là chúng có thể chứa tới 64 bit dữ liệu, các phần khác của sổ đăng ký cũng có thể được tham chiếu. Trong trường hợp này, sổ đăng ký cũng có thể được tham chiếu dưới dạng giá trị 32 bit như được hiển thị. Những gì không được hiển thị là các register có thể được tham chiếu là 16 bit và 8 bit (cao hơn 4 bit và thấp hơn 4 bit).

6 registers đầu tiên được gọi là general purpose registers (thanh ghi mục đích chung). %rsp là con trỏ ngăn xếp và nó trỏ đến đầu ngăn xếp có chứa địa chỉ bộ nhớ gần đây nhất. Ngăn xếp là một cấu trúc dữ liệu quản lý bộ nhớ cho các chương trình. %rbp là một con trỏ khung và trỏ đến khung của hàm hiện đang được thực thi - mọi hàm được thực thi trong một khung mới. Để di chuyển dữ liệu bằng sổ đăng ký, hướng dẫn sau đây được sử dụng:

Movq source, destination

Điều này liên quan đến:

* Chuyển các hằng số (được đặt trước bằng toán tử $), ví dụ: movq $ 3 rax sẽ di chuyển hằng số 3 vào register
* Chuyển các giá trị từ một register, ví dụ: movq% rax% rbx liên quan đến việc chuyển giá trị từ rax sang rbx
* Truyền các giá trị từ bộ nhớ được hiển thị bằng cách đặt các thanh ghi bên trong dấu ngoặc, ví dụ: movq% rax (% rbx) có nghĩa là di chuyển giá trị được lưu trữ trong% rax đến vị trí bộ nhớ được đại diện bởi% rbx.

|  |  |  |
| --- | --- | --- |
| Intel Data Type | Hậu tố | Kích thước |
| Byte | b | 1 |
| Word | w | 2 |
| Double Word | l | 4 |
| Quad word | q | 8 |
| Single Precision | s | 4 |
| Double Precision | l | 8 |

Khi xử lý thao tác bộ nhớ bằng cách sử dụng register, có những trường hợp khác cần được xem xét:

* (Rb, Ri) = Vị trí bộ nhớ [Rb + Ri]
* D (Rb, Ri) = Vị trí bộ nhớ [Rb + Ri + D]
* (Rb, Ri, S) = Bộ nhớ Local (Rb + S \* Ri]
* D (Rb, Ri, S) = Vị trí bộ nhớ [Rb + S \* Ri + D]

Một số intruction quan trọng khác sẽ là:

* Leaq source, destination: lệnh này đặt điểm đến thành địa chỉ được biểu thị bằng biểu thức trong nguồn
* Addq source, destination: destination = destination + source
* Subq source, destination: destination = destination - source
* Imulq source, destination: destination = destination \* source
* Salq source, destination: destination = destination << source where << là toán tử dịch chuyển bit sang trái
* Sarq source, destination: destination = destination >> source where >> là toán tử dịch chuyển bit sang phải
* Xorq source, destination: destination = destination XOR source
* Andq source, destination: destination = destination & source
* Orq source, destination: destination = destination | source

Trước khi hiểu cách các chương trình hoạt động, điều quan trọng là phải hiểu register, thao tác bộ nhớ và một số hướng dẫn cơ bản. Các phần tiếp theo sẽ có nhiều hands hơn về việc sử dụng radare2.

**If Statements**

The general format of an if statement is

if(condition){

  do-stuff-here

}else if(condition) //this is an optional condition {

  do-stuff-here

}else {

  do-stuff-here

}

If statements use 3 important instructions in assembly:

* cmpq source2, source1*:* it is like computing a-b without setting destination
* *testq source2, source1*: it is like computing a&b without setting destination

Jump instructions được sử dụng để chuyển điều khiển sang các hướng dẫn khác nhau và có nhiều jumps khác nhau:

|  |  |
| --- | --- |
| Jump Type | Description |
| jmp | Unconditional |
| je | Equal/Zero |
| jne | Not Equal/Not zero |
| js | Phủ định |
| jns | Khẳng định |
| jg | Greater |
| jge | Greater or Equal |
| jl | Less |
| jle | Less or Equal |
| ja | Above (không dấu) |
| jb | Below (không dấu) |

Hai giá trị cuối cùng của bảng đề cập đến số nguyên không dấu. Số nguyên không dấu không thể âm trong khi số nguyên có dấu đại diện cho cả giá trị tích cực và tiêu cực. Vì máy tính cần phân biệt giữa chúng, nó sử dụng các phương pháp khác nhau để giải thích các giá trị này. Đối với các số nguyên đã ký, nó sử dụng một thứ gọi là đại diện bổ sung của hai và đối với các số nguyên không được ký tên, nó sử dụng các tính toán nhị phân bình thường.

**If Statements Continued**

Đi tới thư mục if-statement và Bắt đầu r2 vớir2 -d if1

And run the following commands:

aaa

afl

pdf @main

Điều này phân tích chương trình, liệt kê các chức năng và tháo rời chức năng chính.

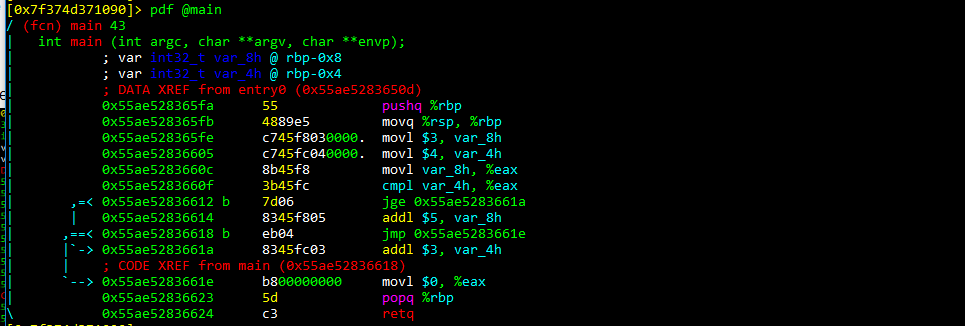


Sau đó, chúng ta sẽ bắt đầu bằng cách thiết lập break point trên **jge** và **jmp** instruction bằng cách sử dụng lệnh:

db 0x55ae52836612 (là địa chỉ hex của jge instruction)

db 0x55ae52836618 (là địa chỉ hex của lệnh jmp instruction)

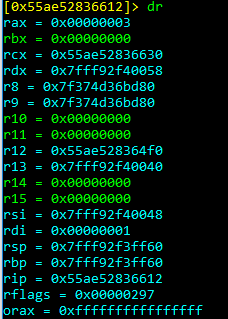
Chúng tôi đã thêm các breakpoint để ngăn chặn việc thực hiện chương trình tại những điểm đó để chúng tôi có thể thấy trạng thái của chương trình. Làm như vậy sẽ hiển thị những điều sau đây:



Bây giờ chúng ta chạy dc để chương trình bắt đầu thực thi và chương trình sẽ bắt đầu thực thi và dừng lại tại break point. Hãy kiểm tra những gì đã xảy ra trước khi đạt đến breakpoint:

* 2 dòng đầu tiên là về việc đẩy con trỏ khung lên ngăn xếp và lưu nó (đây là về cách các hàm được gọi và sẽ được xem xét sau)
* 3 dòng tiếp theo là về việc gán giá trị 3 và 4 cho các đối số / biến cục bộ var\_8h và var\_4h. Sau đó, nó lưu trữ giá trị trong var\_8h trong thanh ghi %eax.
* cmpl instruction so sánh giá trị của eax với giá trị của đối số var\_8h

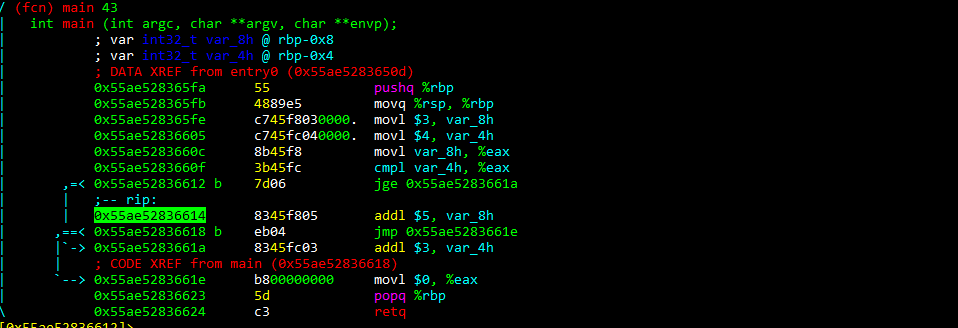
To view the value of the registers, type in: dr



Chúng ta có thể thấy rằng giá trị của rax, là phiên bản 64 bit của eax chứa 3. Chúng ta thấy rằng lệnh jge đang nhảy dựa trên việc giá trị của eax có lớn hơn var\_4h hay không. Để xem có gì trong var\_4h, chúng ta có thể thấy rằng ở đầu hàm chính, nó cho chúng ta biết vị trí của var\_4h. Chạy lệnh: px @ rbp-0x4

Và điều đó cho thấy giá trị của 4.

Chúng ta biết rằng eax chứa 3 và 3 không lớn hơn 4, vì vậy bước nhảy sẽ không thực hiện. Thay vào đó, nó sẽ chuyển sang hướng dẫn tiếp theo. Để kiểm tra điều này, hãy chạy lệnh ds tìm kiếm/di chuyển sang intructions tiếp theo.

 Rip (là con trỏ hướng dẫn hiện tại) cho thấy rằng nó chuyển sang intruction tiếp theo - cho thấy chúng ta đúng. Intruction hiện tại add 5 vào var\_8h đó là local argument. Để thấy rằng điều này thực sự xảy ra, trước tiên hãy kiểm tra giá trị của var\_8h, chạy ds và kiểm tra lại giá trị. Điều này sẽ cho thấy nó tăng lên 5.



Lưu ý rằng vì chúng tôi đang kiểm tra địa chỉ chính xác, chúng tôi chỉ cần kiểm tra để bù trừ 0. Giá trị được lưu trữ trong bộ nhớ được lưu trữ dưới dạng hex. Hướng dẫn tiếp theo là một bước nhảy vô điều kiện và nó chỉ nhảy để xóa eax register. Popq instruction liên quan đến việc bật ds hướng dẫn hiện tại. Trong trường hợp này, nó cho thấy việc thực hiện chương trình đã được hoàn thành. Để hiểu rõ hơn về cách thức hoạt động của câu lệnh if, bạn có thể kiểm tra tệp C tương ứng trong cùng một thư mục

Loops

Có 2 vòng lặp thường được sử dụng: vòng lặp for và vòng lặp while.   
Định dạng chung của vòng lặp while là:

while(condition){

 Do-stuff-here

 Change value used in condition

}

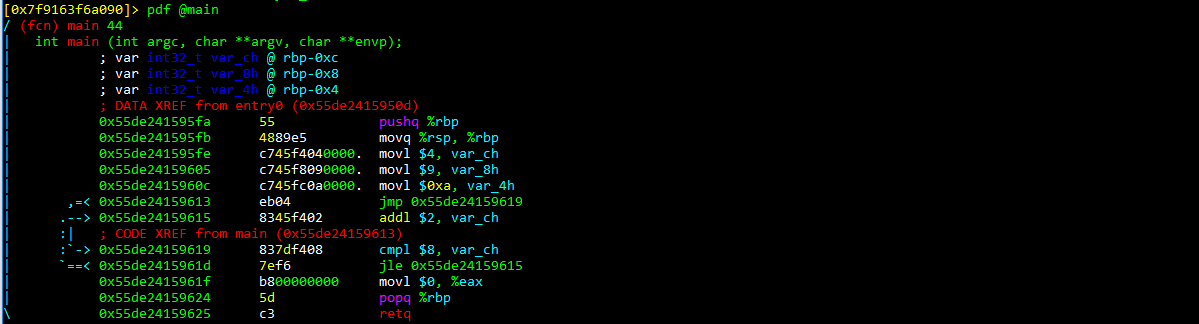
Định dạng chung của vòng lặp for là:

for(initialise value: condition; change value used in condition){

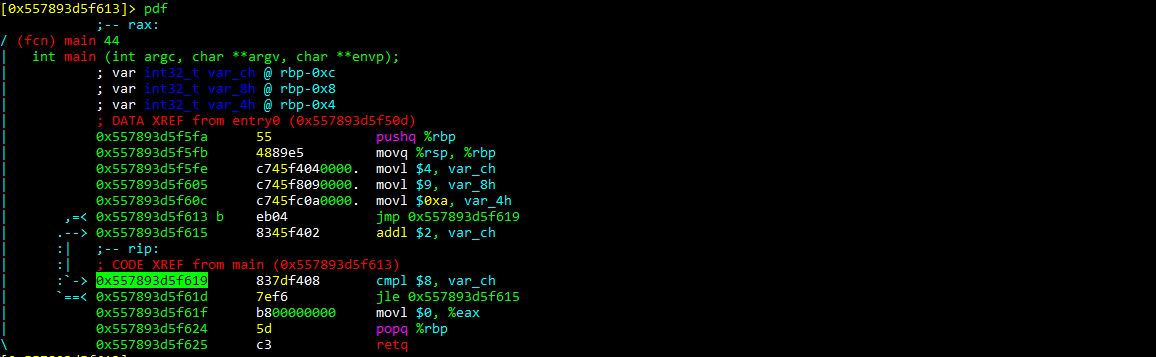
 do-stuff-here

}

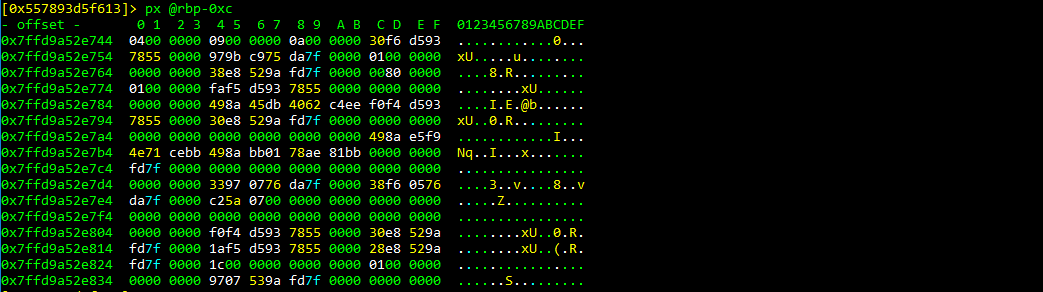
Hãy bắt đầu tra cứu các vòng lặp bằng cách nhập thư mục vòng lặp, chạy r2 với loop1 file. Sau đó, phân tích mọi thứ, liệt kê các chức năng và tháo rời chức năng chính.



Hãy bắt đầu bằng cách đặt một break point tại jmp instruction bằng cách sử dụng lệnh: db address-of-instruction

Làm điều này cho phép chúng tôi bỏ qua một vài dòng instruction đầu tiên, như chúng ta đã thấy bằng cách sử dụng if statements, nó chỉ truyền các giá trị cho các local arguments (lưu ý rằng hằng số được hiển thị bởi $0xa đại diện cho giá trị đó là 10 trong hex). Khi execution đạt đến break point tại jmp intruction, chạy ds để di chuyển đến instruction tiếp theo. Vì đây là một bước nhảy vô điều kiện, nó sẽ chuyển sang cmpl intruction.  


Ở đây cmpl instruction đang cố gắng so sánh những gì trong local argument var\_ch với giá trị 8. Để xem có gì trong var\_ch, hãy kiểm tra sự bắt đầu của disassembled function và kiểm tra bộ nhớ. Trong trường hợp này, nó là rbp-0xc



Và cho thấy rằng nó chứa 4. intruction tiếp theo là một jle sẽ kiểm tra giá trị là var-ch nhỏ hơn hoặc bằng 8. Vì 4 là ít hơn 8, nó sẽ nhảy đến addl instruction.



Addl instruction sẽ thêm 2 vào giá trị của var-ch và tiếp tục đi đến cmpl instruction. Vì 2 đã được thêm vào var\_ch, var\_ch bây giờ sẽ chứa 6 vẫn còn ít hơn 8, và nó sẽ nhảy trở lại addl instruction. Điều này có thể được nhìn thấy bằng cách tiếp tục thực hiện, sử dụng ds statement. Chúng tôi biết đây là một vòng lặp vì addl instruction đang được thực hiện nhiều hơn một lần và điều này kết hợp với việc so sánh giá trị của var\_ch với 8. Vì vậy, chúng ta có thể suy ra cấu trúc của vòng lặp là

while(var\_ch < 8){

 var\_ch = var\_ch + 2

}

Một cách nhanh hơn để kiểm tra vòng lặp sẽ là thêm một break point vào cmpl instruction và chạy dc. Vì đây là một vòng lặp, chương trình sẽ luôn bị break ở cmpl instruction (vì instruction này kiểm tra điều kiện trước khi thực hiện những gì bên trong vòng lặp). Bạn có thể kiểm tra tệp loop1.c để xem cấu trúc của vòng lặp!