汇编语言

https://cloud.tencent.com/developer/article/1680490

内联汇编

类似于：内联函数

关键字inline

GCC (GNU Compiler for Linux) 使用AT&T/UNIX汇编语法。

GCC

64bit参数

a->rdi, b->rsi, c->rdx, d->r10, e-r8, f-r9

不用rcx，用r10

pwntools:asm

payload=asm(code,arch='amd64',os='linux')

1|add

主要是对pwntools的理解吧。

题目要求向程序输入，你的输入按照汇编代码被解析之后得到

```
add 0x331337,rdi
```

参考链接：

> https://docs.pwntools.com/en/stable/asm.html

很简单的程序：

```
In [15]: code = asm('mov rdi,0x1337',arch = 'amd64',os = 'linux')
[DEBUG] cpp -C -nostdinc -undef -P -I/usr/local/lib/python3.8/dist-packages/pwnlib/data/includes /dev/stdin
[DEBUG] Assembling
    .section .shellcode,"awx"
    .global _start
```

```
    .global __start
    .p2align 2
    _start:
    __start:
    .intel_syntax noprefix
    mov rdi,0x1337
[DEBUG] /usr/bin/x86_64-linux-gnu-as -64 -o
/tmp/pwn-asm-e0ebigf6/step2 /tmp/pwn-asm-
e0ebigf6/step1
[DEBUG] /usr/bin/x86_64-linux-gnu-objcopy -j
.shellcode -Obinary /tmp/pwn-asm-
e0ebigf6/step3 /tmp/pwn-asm-e0ebigf6/step4

In [16]: print(type(code))
<class 'bytes'>

In [17]: p=process('./embryoasm_level1')
[x] Starting local process
'./embryoasm_level1' argv=
[b'./embryoasm_level1']
[+] Starting local process
'./embryoasm_level1' argv=
[b'./embryoasm_level1'] : pid 2615

In [18]: p.send(code)
[DEBUG] Sent 0x7 bytes:
    00000000  48 c7 c7 37  13 00 00
                    |H··7|···|
    00000007
```

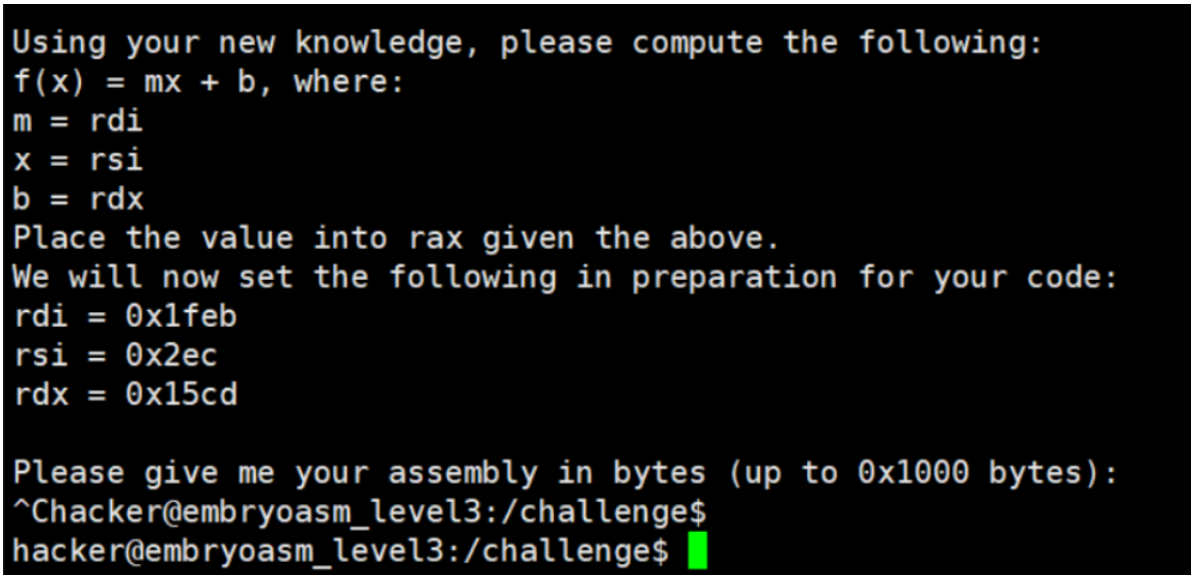
```
In [19]: p.interactive()
```

2

和1是一样的。

3|mul/imul

截图如下：



刚开始做不出来，后来发现就是一个***send***和***sendline***的区别。只要汇编代码没问题就可以。

注意mul和imul的区别。

mul指令会影响rdx的取值，并且刚开始rax的值你并不知道，所以要处理这几个问题。

关于mul：

mul register

默认rax*register

关于imul:

```
mov rax,1
mov r8,rdx
mul rdi
mul rsi
add rax,r8
```

可以使用reppel来进行汇编代码的学习和测试。

反汇编代码如下，做不出来，因为不理解题意：

```
#我给删除了，没用
```

4|div和/

---

div指令:

```
div register1
```

注意，这里的register1被当做被除数，rax是除数。

div register1等于是rax/register1.

得到的结果，商放在rax里面，余数放在rdx里面。

5|%

exp:

```
mov rax,rdi
div rsi
mov rax,rdx
```

解析的结果:

```
---------------- CODE ----------------
0x400000:    mov     rax, rdi
0x400003:    div     rsi
0x400006:    mov     rax, rdx
--------------------------------------
```

6|傻逼mov

7|shl,shr

```
In [2]: code='''
   ...: shr rdi,32
   ...: mov rbx,rdi
   ...: mov al,bl
   ...: '''
```

shr，向右位移，前边补充0（重点）

shl，向左位移，后面补充0（重点）

8|and

9|or,xor,and,not

```
or rdi,0xfffffffffffffffe
and rdi,0x0000000000000001
xor rax,rax
and rax,rdi
```

not register

对寄存器每个bit取反。

如果

rax=00000000

那么not rax之后

rax=11111111

10|取地址方式

https://gowa.club/Asm/X86%E6%B1%87%E7%BC%96%E6%93%8D%E4%BD%9C%E6%95%B0%E5%AF%BB%E5%9D%80.html

也叫做寻址方式

我的输入：

```
mov     rax, 0x404000
mov     rbx, [rax]
mov     rcx, [rax]
add     rbx, 0x1337
mov     [rax], rbx
mov     rax, rcx
```

解析后的输入：

```
---------------- CODE ----------------
0x400000:   mov     rax, 0x404000
0x400007:   mov     rbx, qword ptr [rax]
0x40000a:   mov     rcx, qword ptr [rax]
0x40000d:   add     rbx, 0x1337
0x400014:   mov     qword ptr [rax], rbx
0x400017:   mov     rax, rcx
--------------------------------------
```

11|qword ptr,dword ptr,word ptr,byte ptr

> - Quad Word = 8 Bytes = 64 bits
> - qword ptr
> - Double Word = 4 bytes = 32 bits
> - dword ptr
> - Word = 2 bytes = 16 bits
> - word ptr
> - Byte = 1 byte = 8 bits
> - byte ptr

```
mov al, [address]        <=>

moves the least significant
byte from address to rax

mov ax, [address]        <=>

moves the least significant
word from address to rax

mov eax, [address]        <=>

moves the least significant
double word from address to
rax

mov rax, [address]        <=>

moves the full quad word from
address to rax
```

输入：

```
In [7]: code='''
   ...: mov rdi,0x404000
   ...: mov ax,[rdi]
   ...: mov rbx,rax
   ...: mov eax ,[rdi]
   ...: mov rcx,rax
   ...: mov rdx,[rdi]
   ...: xor rax,rax
   ...: mov al,[rdi]
   ...: '''
```

解析后的输入：

```
---------------- CODE ----------------
0x400000:   mov     rdi, 0x404000
0x400007:   mov     ax, word ptr [rdi]
0x40000a:   mov     rbx, rax
0x40000d:   mov     eax, dword ptr [rdi]
0x40000f:   mov     rcx, rax
0x400012:   mov     rdx, qword ptr [rdi]
0x400015:   xor     rax, rax
0x400018:   mov     al, byte ptr [rdi]
--------------------------------------
```

12|Little endian

0x12345678

1234是高16bit，5678是低16bit。

4004是高地址，存放高位数据。

小端是：

```
                    <---0x12345678
                   ↓


-----------------------------------
4001: 78
-----------------------------------
4002: 56
-----------------------------------
4003: 34
-----------------------------------
4004: 12
-----------------------------------
```

一个Byte一个Byte（字节）的存储，

一个地址指向一个Byte，8bit

---

exp:（超长版）

一个Byte一个Byte

```
mov al,0x37
mov [rdi],al
inc rdi
mov al,0x13
mov [rdi],al
inc rdi
mov al,0x00
mov [rdi],al
inc rdi
mov al,0x00
mov [rdi],al
inc rdi
mov al,0xef
mov [rdi],al
inc rdi
mov al,0xbe
mov [rdi],al
inc rdi
mov al,0xad
mov [rdi],al
inc rdi
mov al,0xde
mov [rdi],al

mov al,0x0
mov [rsi],al
inc rsi
mov al,0x0
mov [rsi],al
inc rsi
```

```
mov al,0xee
mov [rsi],al
inc rsi
mov al,0xff
mov [rsi],al
inc rsi
mov al,0xc0
mov [rsi],al
inc rsi
mov al,0x0
mov [rsi],al
inc rsi
mov al,0x0
mov [rsi],al
inc rsi
mov al,0x0
mov [rsi],al
```

缩短版：

一个Word一个Word

```
mov ax,0x1337
mov [rdi],ax
add rdi,2
mov ax,0x0
mov [rdi],ax
add rdi,2
mov ax,0xbeef
mov [rdi],ax
```

```
add rdi,2
mov ax,0xdead
mov [rdi],ax

mov ax,0x0
mov [rsi],ax
add rsi,2
mov ax,0xffee
mov [rsi],ax
add rsi,2
mov ax,0x00c0
mov [rsi],ax
add rsi,2
mov ax,0x0
mov [rsi],ax
```

再缩短：

一个Dword一个Dword

```
mov eax,0x00001337
mov [rdi],eax
mov eax,0xdeadbeef
mov [rdi+4],eax
mov eax,0xffee0000
mov [rsi],eax
mov eax,0x000000c0
mov [rsi+4],eax
```

应该不能再缩短了，对于没有引用的寄存器，只能放入32bit的立即数？

对于直接存入64bit的立即数，mov指令会被转变为movabs指令。

13|[register+x]

eax是4Byte，rax是8Byte。

eax是32bit，rax是64bit

[rdi+1]意思是，下一个字节。

所以eax的话间隔就是+4。

```
xor rax,rax
xor rbx,rbx

mov eax,[rdi]
add rbx,rax

mov eax,[rdi+4]
add rbx,rax

mov eax,[rdi+8]
add rbx,rax
```

```
mov eax,[rdi+12]
add rbx,rax

mov [rsi],rbx
```

下面的是按照qword来做：

```
mov rax,[rdi]
mov rbx,[rdi+8]
add rax,rbx
mov [rsi],rax
```

14|pop,push,stack

注意pwntools:asm，

```
payload=asm(code,arch='amd64',os='linux')
#arch是机器版本，不同机器版本汇编语言不一样
#i386,amd64
```

exp:

64bit(只要这个了)

```
mov rax,[0x7fffff1ffff8]
sub rax,rdi
pop rdi/随便一个register都可以
push rax
```

总结一下：

1. 不同位数的机器，汇编语言不一样，比如32bit就无法识别rax，rbx
2. 由于本题中，如果栈区的数值很大，32bit的寄存器就不能实现加减，所以使用eax什么的比较麻烦，就不研究了

15|only use pop,push

```
push rdi
push rsi
pop rdi
pop rsi
```

16|rsp

so easy!

```
add rax,[rsp]
add rax,[rsp+8]
add rax,[rsp+16]
add rax,[rsp+24]
mov rbx,4
div rbx
mov [rsp-8],rax
```

jmp的类型

按照是否有条件：

- 无条件转移

jum

- 条件转移

ja，jg，je等各种变形

按照寻址方式：

- Relative jumps
- Absolute jumps
- Indirect jumps

17|unconditional jmp

题目描述：

```
;formulaic operation
;公式化操作

;In this level you will be working with
control flow manipulation. This involves
using instructions to both indirectly and
directly control the special register `rip`,
the instruction pointer.You will use
instructions like: jmp, call, cmp, and the
like to implement requests behavior.

;Earlier, you learned how to manipulate data
in a pseudo-control way, but x86 gives us
actual instructions to manipulate control
flow directly. There are two major ways to
manipulate control flow:
;1. through a jump;
;2. through a call. In this level, you will
work with jumps.

;There are two types of jumps:
1. Unconditional jumps
2. Conditional jumps
```

```
;Unconditional jumps always trigger and are
not based on the results of earlier
instructions.

;As you know, memory locations can store data
and instructions. You code will be stored at
0x4000b9 (this will change each run).
For all jumps, there are three types:
1. Relative jumps
2. Absolute jumps
3. Indirect jumps
In this level we will ask you to do both a
relative jump and an absolute jump. You will
do a relative
jump first, then an absolute one. You will
need to fill space in your code with
something to make this
relative jump possible. We suggest using the
`nop` instruction. It's 1 byte and very
predictable.
Useful instructions for this level is:
jmp (reg1 | addr | offset) ; nop
Hint: for the relative jump, lookup how to
use `labels` in x86.

Using the above knowledge, perform the
following:
Create a two jump trampoline:
1. Make the first instruction in your code a
jmp
```

```
2. Make that jmp a relative jump to 0x51
bytes from its current position
3. At 0x51 write the following code:
4. Place the top value on the stack into
register rdi
5. jmp to the absolute address 0x403000

We will now set the following in preparation
for your code:
- Loading your given gode at: 0x4000b9
- (stack) [0x7fffff1ffff8] = 0xb9

Please give me your assembly in bytes (up to
0x1000 bytes):
```

怎么说呢，还算是蛮简单的，对于汇编语言的理解。主要是相对地址跳转，是相对于什么+offset？

注意，是相对于当前指令的下一条指令+offset。

ip里面存放的是当前正在执行的指令的下一条指令

也就是ip+offset

exp:

```
jmp mycode
;padding begin
mov rax,0xdeadbeef
mov rax,0xdeadbeef
```

```
mov rax,0xdeadbeef
mov rax,0xdeadbeef
mov rax,0xdeadbeef
mov rax,0xdeadbeef
mov rax,0xdeadbeef
mov rax,0xdeadbeef
nop
;padding end
;size is 0x51(81)

mycode:
mov rdi,[rsp]
mov rax,0x403000
jmp rax
```

18|conditional jmp

题目描述：

```
;In this level you will be working with
control flow manipulation. This involves
using instructions to both indirectly and
directly control the special register `rip`,
the instruction pointer.
```

```
;You will use instructions like: jmp, call,
cmp, and the like to implement requests
behavior.

;We will be testing your code multiple times
in this level with dynamic values! This means
we will be running your code in a variety of
random ways to verify that the logic is
robust enough to survive normal use. You can
consider this as normal dynamic value se

;We will now introduce you to conditional
jumps--one of the most valuable instructions
in x86.In higher level programming languages,
an if-else structure exists to do things
like:
if x is even:
   is_even = 1
else:
   is_even = 0
;This should look familiar, since its
implementable in only bit-logic. In these
structures, we can control the programs
control flow based on dynamic values provided
to the program. Implementing the above logic
with jmps can be done like so:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
; assume rdi = x, rax is output
```

```
; rdx = rdi mod 2
mov rax, rdi
mov rsi, 2
div rsi
; remainder is 0 if even
cmp rdx, 0
; jump to not_even code is its not 0
jne not_even
; fall through to even code
mov rbx, 1
jmp done
; jump to this only when not_even
not_even:
mov rbx, 0
done:
mov rax, rbx
; more instructions here
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
```

Often though, you want more than just a single 'if-else'. Sometimes you want two if checks, followed by an else. To do this, you need to make sure that you have control flow that 'falls-through' to the next `if` after it fails. All must jump to the same `done` after execution to avoid the else.There are many jump types in x86, it will help to lea2rn how they can be used. Nearly all of them rely on something called the ZF, the Zero Flag. The ZF is set to 1 when a cmp is equal. 0 otherwise.

;Using the above knowledge, implement the following:

```
if [x] is 0x7f454c46:
   y = [x+4] + [x+8] + [x+12]
else if [x] is 0x00005A4D:
   y = [x+4] - [x+8] - [x+12]
else:
   y = [x+4] * [x+8] * [x+12]
```

where:

x = rdi, y = rax.

;Assume each dereferenced value is a signed dword. This means the values can start as a negative value at each memory position.A valid solution will use the following at least once:jmp (any variant), cmp

```
We will now run multiple tests on your code,
here is an example run:
- (data) [0x404000] = {4 random dwords]}
- rdi = 0x404000

Please give me your assembly in bytes (up to
0x1000 bytes):
```

exp:

```
mov esi,[rdi]
mov edx,0x7f454c46
mov r10d,0x00005A4D

cmp esi,edx
je code1
cmp esi,r10d
je code2
xor rax,rax
mov rax,1
mov r8d,[rdi+4]
mul r8d
mov r8d,[rdi+8]
mul r8d
mov r8d,[rdi+12]
mul r8d
;任何一个判断都要jmp
jmp done

code1:
```

```
xor rax,rax
add eax,[rdi+4]
add eax,[rdi+8]
add eax,[rdi+12]
;任何一个判断都要jmp
jmp done

code2:
xor rax,rax
add eax,[rdi+4]
sub eax,[rdi+8]
sub eax,[rdi+12]
;任何一个判断都要jmp
jmp done

done:
nop
```

19|jump table

题目描述：

```
;The last set of jump types is the indirect
jump, which is often used for switch
statements in the real world. Switch
statements are a special case of if-
statements that use only numbers to determine
where the control flow will go. Here is an
example:
```

```
switch(number):
    0: jmp do_thing_0
    1: jmp do_thing_1
    2: jmp do_thing_2
    default: jmp do_default_thing

;The switch in this example is working on
`number`, which can either be 0, 1, or 2. In
the case that `number` is not one of those
numbers, default triggers. You can consider
this a reduced else-if type structure.

;In x86, you are already used to using
numbers, so it should be no suprise that you
can make if statements based on something
being an exact number. In addition, if you
know the range of the numbers,a switch
statement works very well. Take for instance
the existence of a jump table. A jump table
is a contiguous section of memory that holds
addresses of places to jump. In the above
example, thejump table could look like:

[0x1337] = address of do_thing_0
[0x1337+0x8] = address of do_thing_1
[0x1337+0x10] = address of do_thing_2
[0x1337+0x18] = address of do_default_thing
```

```
;Using the jump table, we can greatly reduce the amount of cmps we use. Now all we need to check is if `number` is greater than 2.
;If it is, always do:
jmp [0x1337+0x18]
Otherwise:
jmp [jump_table_address + number * 8]

;Using the above knowledge, implement the following logic:
if rdi is 0:
    jmp 0x403062
else if rdi is 1:
    jmp 0x4030db
else if rdi is 2:
    jmp 0x4031e9
else if rdi is 3:
    jmp 0x40325d
else:
    jmp 0x403340

Please do the above with the following constraints:
- assume rdi will NOT be negative
- use no more than 1 cmp instruction
- use no more than 3 jumps (of any variant)
- we will provide you with the number to 'switch' on in rdi.
- we will provide you with a jump table base address in rsi.
```

```
Here is an example table:
    [0x404180] = 0x403062 (addrs will change)
    [0x404188] = 0x4030db
    [0x404190] = 0x4031e9
    [0x404198] = 0x40325d
    [0x4041a0] = 0x403340

Please give me your assembly in bytes (up to
0x1000 bytes):
```

exp:

```
xor rax,rax
mov rbx,3
mov rax,[rsi+32]
cmp rdi,rbx
ja code

xor rax,rax
mov rax,rdi
mov rdx,8
mul rdx

mov r10,rsi
add r10,rax
mov r8,[r10]
jmp r8
code:
jmp rax
```

简化：

```
mov rbx,3
cmp rdi,rbx
ja code

mov rax,rdi
mov rbx,8
mul rbx

mov rdx,rsi
add rdx,rax
jmp [rdx]

code:
mov rax,[rsi+32]
jmp rax
```

注意地址和值的区别：

address	value	address	value
rsi	0x404180	0x404180	add1
		0x404188	add2
		0x404190	add3
		0x404198	add4
		0x4041a0	add5

重点：

不能这样使用`条件转移`

```
cmp rdi, 4
jg [rsi + 0x20]
jmp [rsi + rdi * 8]
```

可以这样：

As I found out, you can't do conditional jumps to `relative locations`.

```
cmp rdi, 4
jg foo

foo:
  jmp [rsi + 0x20]
```

20|for-loop

题目描述：

```

```

```
;In  a previous level you computed the
average of 4 integer quad words, which was a
fixed amount of things to compute, but how do
you work with sizes you get when the program
is running? In most programming languages a
structure exists called the for-loop, which
allows you to do a set of instructions for a
bounded amount of times.The bounded amount
can be either known before or during the
programs run, during meaning the value is
given to you dynamically.

As an example, a for-loop can be used to
compute the sum of the numbers 1 to n:

sum = 0
i = 1
for i <= n:
    sum += i
    i += 1

Please compute the average of n consecutive
quad words, where:
rdi = memory address of the 1st quad word
rsi = n (amount to loop for)
rax = average computed

We will now set the following in preparation
for your code:
- [0x404108:0x404350] = {n qwords]}
```

```
- rdi = 0x404108
- rsi = 73


Please give me your assembly in bytes (up to
0x1000 bytes):
```

exp:

```
xor rax,rax
mov r8,rdi
mov r9,rsi
loop1:
sub r9,1
mov r10,[r8]
add rax,r10
add r8,8
cmp r9,0
jne loop1
div rsi

---------------- CODE ----------------
0x400000:   xor     rax, rax
0x400003:   mov     r8, rdi
0x400006:   mov     r9, rsi
0x400009:   sub     r9, 1
0x40000d:   mov     r10, qword ptr [r8]
0x400010:   add     rax, r10
0x400013:   add     r8, 8
0x400017:   cmp     r9, 0
```

```
0x40001b:   jne     0x400009
0x40001d:   div     rsi
-------------------------------------
pwn.college{8CzCM3sZt-
dYKPkycfeRkQYjkti.dNTMywCO5IzW}
```

错误的exp：

```
xor rax,rax
mov rbx,rsi
mov rdx,rdi
loop1:
sub rbx,1
mov r8,[rdx]
add rax,r8
add rdx,8
cmp rbx,0
jne loop1
div rsi

---------------- CODE ----------------
0x400000:   xor     rax, rax
0x400003:   mov     rbx, rsi
0x400006:   mov     rdx, rdi
0x400009:   sub     rbx, 1
0x40000d:   mov     r8, qword ptr [rdx]
0x400010:   add     rax, r8
```

```
0x400013:   add     rdx, 8
0x400017:   cmp     rbx, 0
0x40001b:   jne     0x400009
0x40001d:   div     rsi
-------------------------------------
Sorry, no flag :(.
```

修改：

```
;rdi address rsi count
;rbx address rdx count
mov rbx,rdi
mov rdx,rsi
loop1:
sub rdx,1
mov r8,[rbx]
add rax,r8
add rbx,8
cmp rdx,0
jne loop1
div rsi
;成功
```

21|while-loop

题目描述：

```
```

```
;In previous levels you discovered the for-loop to iterate for a *number* of times, both dynamically and statically known, but what happens when you want to iterate until you meet a condition?
A second loop structure exists called the while-loop to fill this demand.
;In the while-loop you iterate until a condition is met. As an example, say we had a location in memory with adjacent numbers and we wanted to get the average of all the numbers until we find one bigger or equal to 0xff:
average = 0
i = 0
while x[i] < 0xff:
    average += x[i]
    i += 1
average /= i

Using the above knowledge, please perform the following:

Count the consecutive non-zero bytes in a contiguous region of memory, where:
rdi = memory address of the 1st byte
rax = number of consecutive non-zero bytes
Additionally, if rdi = 0, then set rax = 0 (we will check)!
```

```
An example test-case, let:
rdi = 0x1000
[0x1000] = 0x41
[0x1001] = 0x42
[0x1002] = 0x43
[0x1003] = 0x00
then: rax = 3 should be set

We will now run multiple tests on your code,
here is an example run:
- (data) [0x404000] = {10 random bytes},
- rdi = 0x404000

Please give me your assembly in bytes (up to
0x1000 bytes):
```

exp:

```
xor rax,rax
cmp rdi,0
je code2
code1:
mov bl,[rdi]
cmp bl,0
je code2
inc rax
inc rdi
jmp code1
code2:
nop
```

22 | ret

;In this level you will be working with functions!This will involve manipulating both ip control as well as doing harder tasks than normal. You may be asked to utilize the stack to save things and call other functions that we provide you.

utilize 利用
contiguous 连续的

;In previous levels you implemented a while loop to count the number of consecutive non-zero bytes in a contiguous region of memory. In this level you will be provided with a contiguous region of memory again and will loop over each performing a conditional operation till a zero byte is reached.

All of which will be contained in a function!

segment 段

A function is a callable segment of code that does not destory control flow.

Functions use the instructions "call" and "ret".

The "call" instruction pushes the memory address of the next instruction onto the stack and then jumps to the value stored in the first argument.

Let's use the following instructions as an example:

```
0x1021 mov rax, 0x400000
0x1028 call rax
0x102a mov [rsi], rax
```

1. call pushes 0x102a, the address of the next instruction, onto the stack.
2. call jumps to 0x400000, the value stored in rax.

The "ret" instruction is the opposite of "call". ret pops the top value off of the stack and jumps to it.

Let's use the following instructions and stack as an example:

```
                              Stack ADDR  VALUE
0x103f mov rax, rdx           RSP + 0x8
0xdeadbeef
0x1042 ret                    RSP + 0x0
0x0000102a
ret will jump to 0x102a
```

```
implement 实施

Please implement the following logic:
str_lower(src_addr):
    rax = 0
    if src_addr != 0:
        while [src_addr] != 0x0:
                if [src_addr] <= 90:
                [src_addr] = foo([src_addr])
                rax += 1
            src_addr += 1
foo is provided at 0x403000. foo takes a
single argument as a value

We will now run multiple tests on your code,
here is an example run:
- (data) [0x404000] = {10 random bytes},
- rdi = 0x404000

Please give me your assembly in bytes (up to
0x1000 bytes):
```

exp:

```
;first time judge
;rax as count
;rbx as address(rdi)
;r9 as the value of rbx
if1:
```

```
mov r11,0x403000
xor rax,rax
xor r12,r12
mov rbx,rdi
cmp rbx,0
je done

while:
mov r9b,[rbx]
cmp r9b,0
je done
call if2
inc rbx
jmp while

if2:
cmp r9b,90
ja big90

small90:
xor rdi,rdi
mov rdi,r9
mov r12,rax
call r11
mov [rbx],al
mov rax,r12
inc rax
ret

big90:
```

```
ret

done:
ret
```

23

题目描述：

```
In this level you will be working with
functions! This will involve manipulating
both ip control
as well as doing harder tasks than normal.
You may be asked to utilize the stack to save
things
and call other functions that we provide you.

In the previous level, you learned how to
make your first function and how to call
other functions. Now
we will work with functions that have a
function stack frame. A function stack frame
is a set of
pointers and values pushed onto the stack to
save things for later use and allocate space
on the stack
```

```
for function variables.
First, let's talk about the special register
rbp, the Stack Base Pointer. The rbp register
is used to tell
where our stack frame first started. As an
example, say we want to construct some list
(a contigous space
of memory) that is only used in our function.
The list is 5 elements long, each element is
a dword.
A list of 5 elements would already take 5
registers, so instead, we can make pace on
the stack! The
assembly would look like:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
; setup the base of the stack as the current
top
mov rbp, rsp
; move the stack 0x14 bytes (5 * 4) down
; acts as an allocation
sub rsp, 0x14
; assign list[2] = 1337
mov eax, 1337
mov [rbp-0x8], eax
; do more operations on the list ...
; restore the allocated space
mov rsp, rbp
ret
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
Notice how rbp is always used to restore the
stack to where it originally was. If we don't
restore
the stack after use, we will eventually run
out TM. In addition, notice how we subtracted
from rsp
since the stack grows down. To make it have
more space, we subtract the space we need.
The ret
and call still works the same. It is assumed
that you will never pass a stack address
across functions,
since, as you can see from the above use, the
stack can be overwritten by anyone at any
time.
Once, again, please make function(s) that
implements the following:
most_common_byte(src_addr, size):
    b = 0
    i = 0
    for i <= size-1:
        curr_byte = [src_addr + i]
        [stack_base - curr_byte] += 1
    b = 0

    max_freq = 0
    max_freq_byte = 0
    for b <= 0xff:
```

```
        if [stack_base - b] > max_freq:
            max_freq = [stack_base - b]
            max_freq_byte = b

    return max_freq_byte
Assumptions:
- There will never be more than 0xffff of any
byte
- The size will never be longer than 0xffff
- The list will have at least one element
Constraints:
- You must put the "counting list" on the
stack
- You must restore the stack like in a normal
function
- You cannot modify the data at src_addr

Please give me your assembly in bytes (up to
0x1000 bytes):
```