Function กับ Stack

หลังจากผ่านเรื่อง Assembly ที่น่าเบื่อ คราวนี้ผมจะพูดถึงการ call (เรียก) function หนึ่ง จะมีผลอย่างไรกับ memory ในส่วน stack และจะมีคำสั่ง Assembly อะไรบ้างที่เกี่ยวข้อง

ปกติโปรแกรมจะทำงานเป็นลำดับ โดยมี EIP ชี้ไปคำสั่งที่จะถูกประมวลผล แต่เมื่อมีการ call function หนึ่ง EIP จะกระโดดไปทำงานใน function ใหม่ และเมื่อทำงานใน function ใหม่จบแล้ว EIP จะต้องกระโดดกลับมาทำงานที่ function เดิมต่อ แล้วโปรแกรมรู้ได้อย่างไรละว่า EIP ต้องกระโดดกลับไปที่ไหน?

เพื่อให้ EIP ชี้ไปที่คำสั่งถัดไปหลังจากกลับมาจาก function ที่เรียก คำสั่งที่ใช้ในการ call function คือ "call" (เช่น "call printf") จะทำการ "push eip" ลงไปใน stack ก่อน แล้วค่อยกระโดดไปทำงานที่ function ใหม่ และที่จบ function ก็จะมีคำสั่ง "ret" ซึ่งจะเท่ากับ "pop eip" ทำให้โปรแกรมสามารถกลับไปทำงานตามปกติได้ เช่น สมมติว่าโปรแกรมเรา EIP ชี้ไปที่ address 0x08112200 และคำสั่งที่ call some\_fn ที่ address 0x08112100 และคำสั่งถัดไปคือ address 0x08112205 ผลของการ call ก็จะได้ตามรูปข้างล่าง

Note: ไม่มีคำสั่ง Assembly ที่แก้ไขค่า หรืออ่านค่า EIP โดยตรง ที่เห็นผมเขียน "push eip" และ "pop eip" นั้น เพื่อใช้ในการอธิบายเท่านั้น

แล้วถ้า function มีการส่ง arguments ละ จะเป็นอย่างไร?

วิธีการส่ง arguments จริงๆแล้ว แล้วแต่ compiler ว่าจะใช้ call convention (รูปแบบการเรียก)ไหน แต่ที่ใช้โดยทั่วไป arguments ของแต่ละ function นั้นจะถูก push ลงไปใน stack จาก arguments ตัวหลังสุดไล่ไปยังตัวหน้าสุด ก่อนจะมีการ call function เช่น ใน C เราเขียน some\_args(1, 2, 3) เราจะได้ assembly code เป็น (เพื่อเป็นการประหยัดพื้นที่ ดูรูปแีรกในห้วข้อ Stack Frame ข้างล่างนะครับ)

push $3

push $2

push $1

call some\_args

Note: วิธีส่ง arguments ไม่จำเป็นต้อง push นะครับ แค่ทำให้เหมือนกันก็พอ ซึ่งจะได้เห็นในหัวข้อถัดไป

ส่วนการส่งค่ากลับของ function คนเขียนโปรแกรมคงคุ้นเคยกันอยู่แล้วว่า function ในจะส่งค่ากลับได้เพียงแค่ค่าเดียว ซึ่ง compiler โดยทั่วไปจะส่งกลับผ่านทาง register EAX โดยการ set ค่าที่ EAX แล้วค่อย ret

Function Call Convention

ในที่นี้ ผมจะพูดถึงแค่ 3 แบบเท่านั้นนะครับ โดยทั้ง 3 แบบ arguments จะถูก push จากตัวหลังสุดไปหน้าสุด

C Calling Convention (cdecl) - เป็นแบบที่ compiler ปัจจุบันใช้ โดย function ที่เรียกจะทำหน้าที่ clear stack เช่นการเรียก some\_args(1, 2, 3) จะได้ Assembly เป็น

push $3

push $2

push $1

call some\_args

add $12,%esp

Standard Convention (stdcall) - การเรียกแบบนี้ Microsoft คิด และใช้ใน dll ของ Microsoft เอง ถ้าใครเคยเขียนโปรแกรมโดยใช้ WIN32 API คงจะเคยเห็น WINAPI หน้า function ซึ่งถ้าไล่ดูใน header file ของ WIN32 API ก็จะเห็นว่า define เป็น \_stdcall การเรียกแบบนี้ต่างจากแบบแรกคือ function ที่ถูกเรียกจะทำหน้าที่ clear stack โดยใช้คำัสั่ง Assembly "RET n" เช่นการเรียก some\_args(1, 2, 3) จะได้ Assembly ของ function ที่เรียกคือ

push $3

push $2

push $1

call some\_args

และใน function ที่ถูกเรียกจะจบด้วย Assembly

ret $12

Fastcall Convention (fastcall) - แบบนี้จะคล้ายแบบ "Standard Convention" ต่างกันตรงที่ argument ตัวแรกจะเก็บไว้ใน ECX และตัวที่สองเก็บไว้ใน EDX ส่วนที่เหลือ push ลง stack เหมือนเดิม การเรียกแบบนี้ ผมไม่ยกตัวอย่าง Assembly นะครับ เพราะใช้น้อยมาก เมื่อเทียบกับ 2 แบบแรก

Note: หลังจากนี้ ถ้าผมพูดถึงการ call function โดยไม่บอกรูปแบบก็ถือว่าเป็น C Calling Convention

Local Variables

ในแต่ละ function จะมี local variables ที่ใช้ภายใน function เท่านั้น และเมื่อจบ function พวก local variables จะถูกทำลายอัตโนมัติ

local variables นั้น จะถูกเก็บไว้ใน stack วิธีการจองคือ compiler ทำการคำนวณขนาดของ local variables ทั้งหมด แล้วเพิ่มคำสั่งลบ ESP ไว้ที่ตอนเริ่มของ function (อย่าลืมนะครับว่า stack ใน x86 ขยายจาก High Address ไป Low Address) เช่นใน function มีการประกาศตัวแปร "int i; char buf[16];" ได้ขนาด local variables เป็น 20 bytes ซึ่ง compiler จะเพิ่มคำสั่ง

sub $20, %esp

Stack Frame

ในหัวข้อ Assembly ผมได้พูดถึง ESP กับ EBP สั้นๆ ในหัวข้อนี้จะได้เห็นว่า register 2 ตัวนี้ถูกใช้งานอย่างไรใน stack

โดยปกติ EBP ชี้ไปยัง address ของ stack ข้างบน EIP ที่ถูก push ลงไปใน stack ก่อนมีการ call function โดยหน้าที่หลักคือ ใช้อ้างอิง function arguments และ local variables ซึ่งเมื่อใช้ EBP การอ้างอิงทั้ง arguments และ local variables นั้นจะไม่มีการเปลี่ยนแปลงตามรูปข้างล่าง โดยสมมติว่ามีการ call function ที่มี 3 arguments และใน function นั้นมีการประกาศตัวแปรไว้เป็น "int i; char buf[8];" (เหมือนตัวอย่างในหัวข้อ "Buffer Overflow คืออะไร" ใน function main แต่ใช้ i แทน magic)

จากรูปข้างบน อาจจะเรียกทั้งหมดว่า "Stack Frame" โดยถ้าเราต้องการจะอ้างถึงตัวแปร buf ก็ใช้ EBP-12 ส่วนถ้าต้องการอ้างถึง argument ตัวที่ 1 ก็ใช้ EBP+8 และตัวอื่นๆ ตามรูป

ส่วน "saved EIP" ในรูปนั้น ผมได้อธิบายไปในตอนต้นแล้ว มันคือ EIP ของคำสั่งที่จะถูกทำงานหลังจากจบ function ที่เรียก

แล้ว "saved EBP" ละมีไว้ทำอะไร เนื่องด้วยเราได้ใช้ EBP เป็น strack frame pointer เพื่อที่จะได้อ้างอิง local variables และ function arguments ได้สะดวก ดังนั้นเมื่อมีการ call function หนึ่ง EBP จะต้องถูกเลื่อนไปที่ stack frame ของ function ที่ถูกเรียก ดังนั้นเราต้องทำเหมือนกับ EIP คือเก็บไว้ใน stack เพื่อจะได้เอา (restore) EBP ของ function เดิมกลับมาได้ (ถ้างง ให้อ่านไปก่อนนะครับ จะมีตัวอย่างอีกอัน)

จะเห็นว่าก่อนจะเริ่มทำงานใน function แต่ละครั้งนั้น จะมีการเก็บค่า EBP, ย้าย EBP และจองเนื้อที่สำหรับ local variables (ไม่มีการเก็บค่า EIP นะครับ อันนี้ถูกรวมอยู่ในคำสั่ง call) และเมื่อจบ function ก็จะมีการ clear stack ที่จองไว้สำหรับ local variables และ restore EBP ก่อนที่จะเรียกคำสั่ง ret

สิ่งที่ต้องทำก่อนเริ่มทำงานใน function จะเรียกว่า function prologue ซึ่งถ้านำตัวอย่างข้างบนมาเขียนเป็น assembly จะเป็น

push %ebp # เก็บค่า EBP ที่ใช้ใน function ก่อนหน้าไว้ใน stack

mov %esp, %ebp # เลื่อนค่า EBP มาที่ ESP (top of stack)

sub $12, %esp # เลื่อน ESP เพื่อจอง memory ให้ local variables

และสิ่งที่ต้องทำก่อนจบ function จะเรียกว่า function epilogue ซึ่งเขียนเป็น assembly ได้เป็น

mov %ebp, %esp # clear memory สำหรับ local variables โดยการย้าย ESP มาที่ EBP

pop %ebp # restore EBP จากค่าที่เก็บไว้ใน stack

ret

เนื่องจาก ใน x86 มีคำสั่งสำหรับทำ function epilogue คือ leave ซึ่งเท่ากับ mov %ebp,%esp และ pop %ebp ทำให้โดยปกติ เราจะเห็น function epilogue เมื่อเรา disassembly เป็น

leave

ret

บางคนอาจจะสังเกตเห็น ESP ในรูป แล้วสงสัยว่าทำไมถึงไม่ใช้ ESP ในการอ้างอิง local variables และ function arguments ละ ในเมื่อ ESP ชี้ไปที่ top of stack เสมออยู่แล้ว และก็อยู่ใกล้ local variables กับ function arguments เหตุผลก็คือ

1. การถ้าใช้ ESP ต้องมีการคำนวณทุกครั้ง ที่มีการ push หรือ pop ว่า local variables และ function arguments ห่างจาก ESP เท่าไร แต่การใช้ EBP ทำให้การอ้างอิงค่าแต่ละตัวเหมือนเดิมตลอดๆ ไม่ว่า stack จะเปลี่ยนแปลงอย่างไร ทำให้ง่ายต่อการ debug

2. เนื่องด้วยต้องคำนวณระยะห่างของ ESP ที่กล่าวไปในข้อ 1 ทำให้ compiler ทำงานช้าลง

จริงๆ แล้ว compiler เกือบทุกตัว มี option ให้ใช้แต่ ESP แล้วเก็บ EBP ไว้ใช้เหมือน register ตัวอื่นๆ เช่นใน gcc จะใช้ -fomit-frame-pointer ส่วนเหตุผลว่า บางครั้งทำไมต้องใช้แบบนี้ ไม่ขอกล่าวในนี้ เดี๋ยวจะยาวเกิน

ก่อนจะจบ ผมขอยกตัวอย่าง ที่มาจากการ compile จริงๆ และจะได้ฝึก Assembly ไปด้วย โดยมีโปรแกรมที่เขียนด้วยภาษา C ดังนี้ (ex\_04\_1.c)

int fn\_second(int n1, int n2, char \*s)

{

char bb[16];

return 1;

}

void fn\_first(int num)

{

int i;

char buf[8];

fn\_second(i, num, buf);

}

int main()

{

fn\_first(5);

return 0;

}

แล้ว compile ด้วย gcc ตามนี้ (ครั้งนี้ ผม compile ให้ใช้วิธี push argument แล้ว call function เพื่อให้เข้าใจง่าย แต่ในหัวข้อถัดไป ผมจะให้ดูอีกรูปแบบหนึ่ง)

$ gcc -march=i586 -fno-pie -fno-stack-protector -z norelro -z execstack -mpreferred-stack-boundary=2 -o call\_stack call\_stack.c

เมื่อผมทำการ disassembly ออกมาจะได้ (ผมเอามาแสดงแค่ 3 function ที่มีใน C code นะครับ และเป็น address จริงๆ ในเครื่องของผม)

<fn\_second>:

# function prologue โดยจองเนื้อที่ขนาด 16 bytes สำหรับ local variable

0x08048394 <+0>: push %ebp

0x08048395 <+1>: mov %esp,%ebp

0x08048397 <+3>: sub $0x10,%esp

0x0804839a <+6>: mov $0x1,%eax # set ค่า 1 ที่จะ return ใน EAX

# function epilogue

0x0804839f <+11>: leave

0x080483a0 <+12>: ret

<fn\_first>:

# function prologue โดยจองเนื้อที่ขนาด 12 bytes สำหรับ local variable

0x080483a1 <+0>: push %ebp

0x080483a2 <+1>: mov %esp,%ebp

0x080483a4 <+3>: sub $0xc,%esp

0x080483a7 <+6>: lea -0xc(%ebp),%eax # load address ของ buf ไว้ที่ EAX

0x080483aa <+9>: push %eax # push address ของ buf (argument ตัวที่ 3)

0x080483ab <+10>: pushl 0x8(%ebp) # push ค่า num (argument ตัวที่ 2)

0x080483ae <+13>: pushl -0x4(%ebp) # push ค่า i (argument ตัวที่ 1)

0x080483b1 <+16>: call 0x8048394 <fn\_second>

0x080483b6 <+21>: add $0xc,%esp # clear arguments ที่ส่งผ่านใน stack

# function epilogue

0x080483b9 <+24>: leave

0x080483ba <+25>: ret

<main>:

# function prologue มี 2 คำสั่งเพราะ ไม่มี local variables

0x080483bb <+0>: push %ebp

0x080483bc <+1>: mov %esp,%ebp

0x080483be <+3>: push $0x5 # push argument ตัวที่ 1

0x080483c0 <+5>: call 0x80483a1 <fn\_first>

0x080483c5 <+10>: add $0x4,%esp # clear arguments ที่ส่งผ่านใน stack

0x080483c8 <+13>: mov $0x0,%eax # set ค่า 0 ที่จะ return ใน EAX

# function epilogue

0x080483cd <+18>: leave

0x080483ce <+19>: ret

ถ้าใครอ่าน Assembly code แล้วไม่เห็นภาพ ผมก็มีรูปให้ดู (หวังว่าคนที่ยังไม่เข้าใจ ดูแล้วจะเข้าใจ) โดยผมจะเริ่มคำสั่งจากใน main ที่ address 0x080483be และจบที่ address 0x080483c5 โดย EBP และ ESP ชี้ไปที่ address 0xbffff728 อยู่ (address จริงในเครื่องผม) และเนื่องด้วยถ้าทำเป็น step ทั้งหมดรูปจะใหญ่มาก ผมขอไม่เข้าไปใน "call fn\_second" และคำสั่งที่ address 0x080483c5 ผมไม่แสดง โดยผลลัพธ์จะเหมือนขั้นตอนแรก