Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015
xeno@legbacore.com

Attribution condition: You must indicate that derivative work

"Is derived from Xeno Kovah's ‘Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

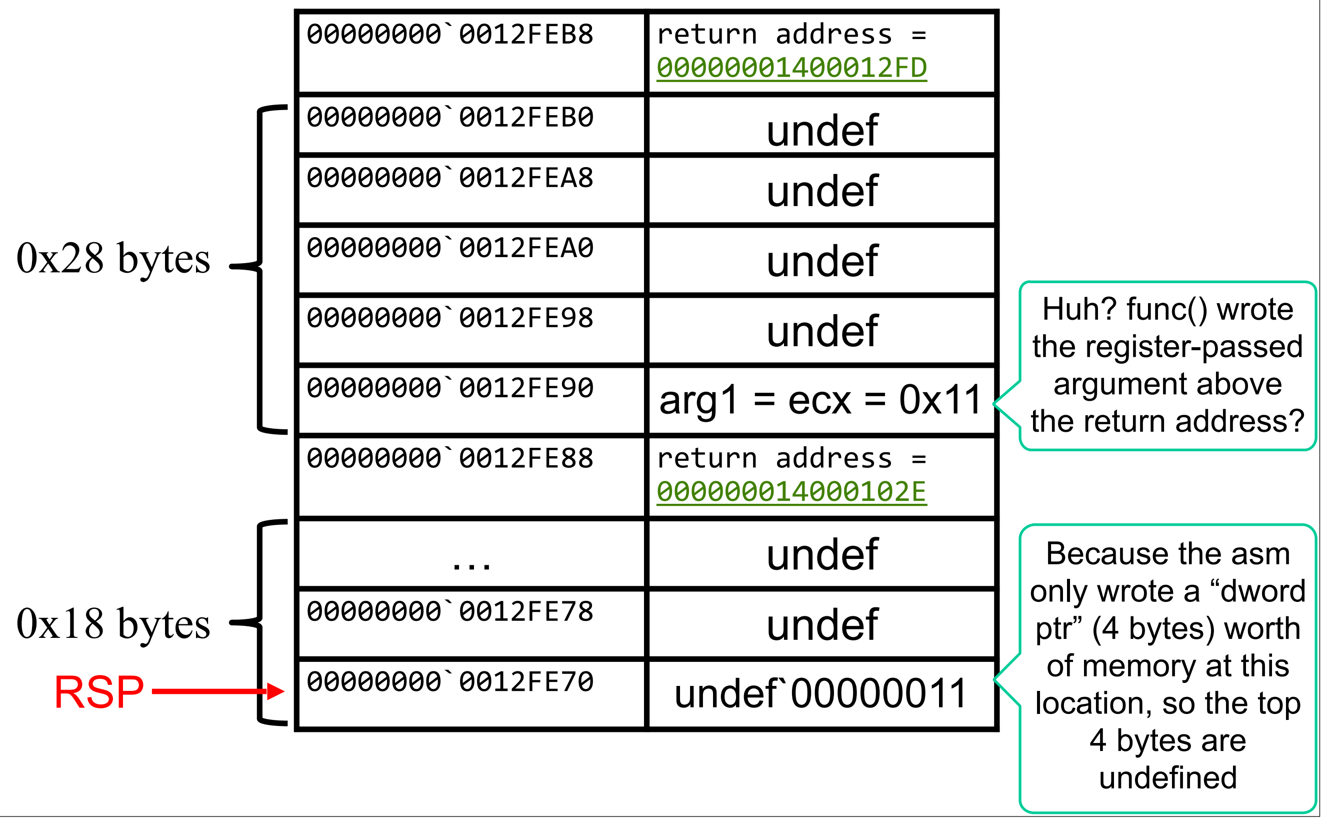
Pass1Parameter.c

Adding a single argument

```c
//Pass1Parameter.c:
int func(int a){
    int i = a;
    return i;
}
int main(){
    return func(0x11);
}
```

```
func:
0000000140001000  mov        dword ptr [rsp+8],ecx
0000000140001004  sub        rsp,18h
0000000140001008  mov        eax,dword ptr [rsp+20h]
000000014000100C  mov        dword ptr [rsp],eax
000000014000100F  mov        eax,dword ptr [rsp]
0000000140001012  add        rsp,18h
0000000140001016  ret
main:
0000000140001020  sub        rsp,28h
0000000140001024  mov        ecx,11h
0000000140001029  call       func (0140001000h)
000000014000102E  add        rsp,28h
0000000140001032  ret
```

The stack looks like this at line 000000014000100F in func():



Pass1Parameter.c takeaways

- Something very interesting is going on with the stack!
- The value which is passed in a register is then still being stored on the stack…doesn't that kind of defeat the speed benefit of passing in registers?

```
//Pass1Parameter.c:
int func(int a){
    int i = a;
    return i;
}
int main(){
    return func(0x11);
}
```

```
func:
0000000140001000  mov       dword ptr [rsp+8],ecx
0000000140001004  sub       rsp,18h
0000000140001008  mov       eax,dword ptr [rsp+20h]
000000014000100C  mov       dword ptr [rsp],eax
000000014000100F  mov       eax,dword ptr [rsp]
0000000140001012  add       rsp,18h
0000000140001016  ret
main:
0000000140001020  sub       rsp,28h
0000000140001024  mov       ecx,11h
0000000140001029  call      func (0140001000h)
000000014000102E  add       rsp,28h
0000000140001032  ret
```

TooManyParameters.c

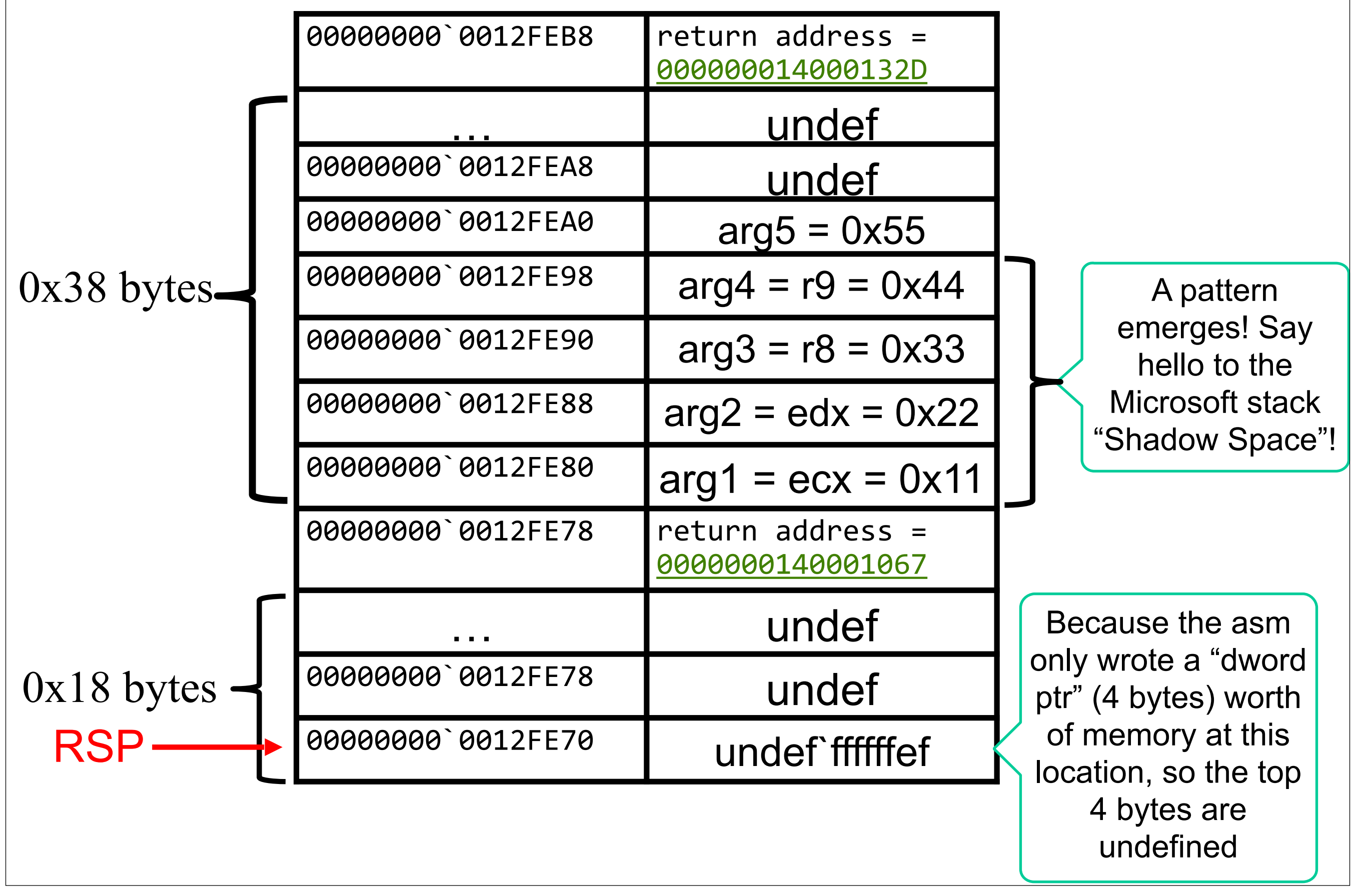
Using more than 4 arguments, to force it to pass the extra parameters via the stack

```
//TooManyParameters:
int func(int a, int b, int c, int d, int e){
  int i = a+b-c+d-e;
  return i;
}
int main(){
  return func(0x11,0x22,0x33,0x44,0x55);
}
```

```
func:
0000000140001000  mov  dword ptr [rsp+20h],r9d
0000000140001005  mov  dword ptr [rsp+18h],r8d
000000014000100A  mov  dword ptr [rsp+10h],edx
000000014000100E  mov  dword ptr [rsp+8],ecx
0000000140001012  sub  rsp,18h
0000000140001016  mov  eax,dword ptr [rsp+28h]
000000014000101A  mov  ecx,dword ptr [rsp+20h]
000000014000101E  add  ecx,eax
0000000140001020  mov  eax,ecx
0000000140001022  sub  eax,dword ptr [rsp+30h]
0000000140001026  add  eax,dword ptr [rsp+38h]
000000014000102A  sub  eax,dword ptr [rsp+40h]
000000014000102E  mov  dword ptr [rsp],eax
0000000140001031  mov  eax,dword ptr [rsp]
0000000140001034  add  rsp,18h
0000000140001038  ret
main:
0000000140001040  sub  rsp,38h
0000000140001044  mov  dword ptr [rsp+20h],55h
000000014000104C  mov  r9d,44h
0000000140001052  mov  r8d,33h
0000000140001058  mov  edx,22h
000000014000105D  mov  ecx,11h
0000000140001062  call  0000000140001000
0000000140001067  add  rsp,38h
000000014000106B  ret
```

http://www.youtube.com/watch?v=Nr8r09c8ogg

The stack looks like this at line 0000000140001031 in func():



Who knows what the first 4 parameters passed in registers were when you're trying to backtrace the stack calls?

"Shadow space" reference http://msdn.microsoft.com/en-us/library/zthk2dkh.aspx

http://ifanboy.com/wp-content/uploads/2011/10/The-Shadow-Alex-Ross-Cover-1.jpg

TooMany Parameters.c takeaways

- Microsoft compiler specifically augments the calling convention by not only passing the first 4 arguments through registers, but also still reserving “shadow space” for them on the stack.
- “The callee has the responsibility of dumping the register parameters into their shadow space if needed.”
- Compiler reserves this space even if no function parameters are passed to another function

```
//ExampleSubroutine4:
int func(int a, int b, int c, int d, int e){
  int i = a+b-c+d-e;
  return i;
}
int main(){
  return func(0x11,0x22,0x33,0x44, 0x55);
}
```

```
func:
 mov   dword ptr [rsp+20h],r9d
 mov   dword ptr [rsp+18h],r8d
 mov   dword ptr [rsp+10h],edx
 mov   dword ptr [rsp+8],ecx
 sub   rsp,18h
 mov   eax,dword ptr [rsp+28h]
 mov   ecx,dword ptr [rsp+20h]
 add   ecx,eax
 mov   eax,ecx
 sub   eax,dword ptr [rsp+30h]
 add   eax,dword ptr [rsp+38h]
 sub   eax,dword ptr [rsp+40h]
 mov   dword ptr [rsp],eax
 mov   eax,dword ptr [rsp]
 add   rsp,18h
 ret
main:
 sub   rsp,38h
 mov   dword ptr [rsp+20h],55h
 mov   r9d,44h
 mov   r8d,33h
 mov   edx,22h
 mov   ecx,11h
 call  0000000140001000
 add   rsp,38h
 ret
```

You can confirm that the typical sub rsp, 0x28 that you see is due to shadow space reservation by just changing func() to only take 4 parameters instead of 5, and seeing that it only reserves 0x28 again instead of 0x38 like seen here. But if you then add a local variable, which should technically fit in the reserved stack space (just like this 5th parameter should have), it will again bump it up to 0x38. Still not sure what’s up with the over-allocation of stack space in general (possibly just 0x10 alignment)…

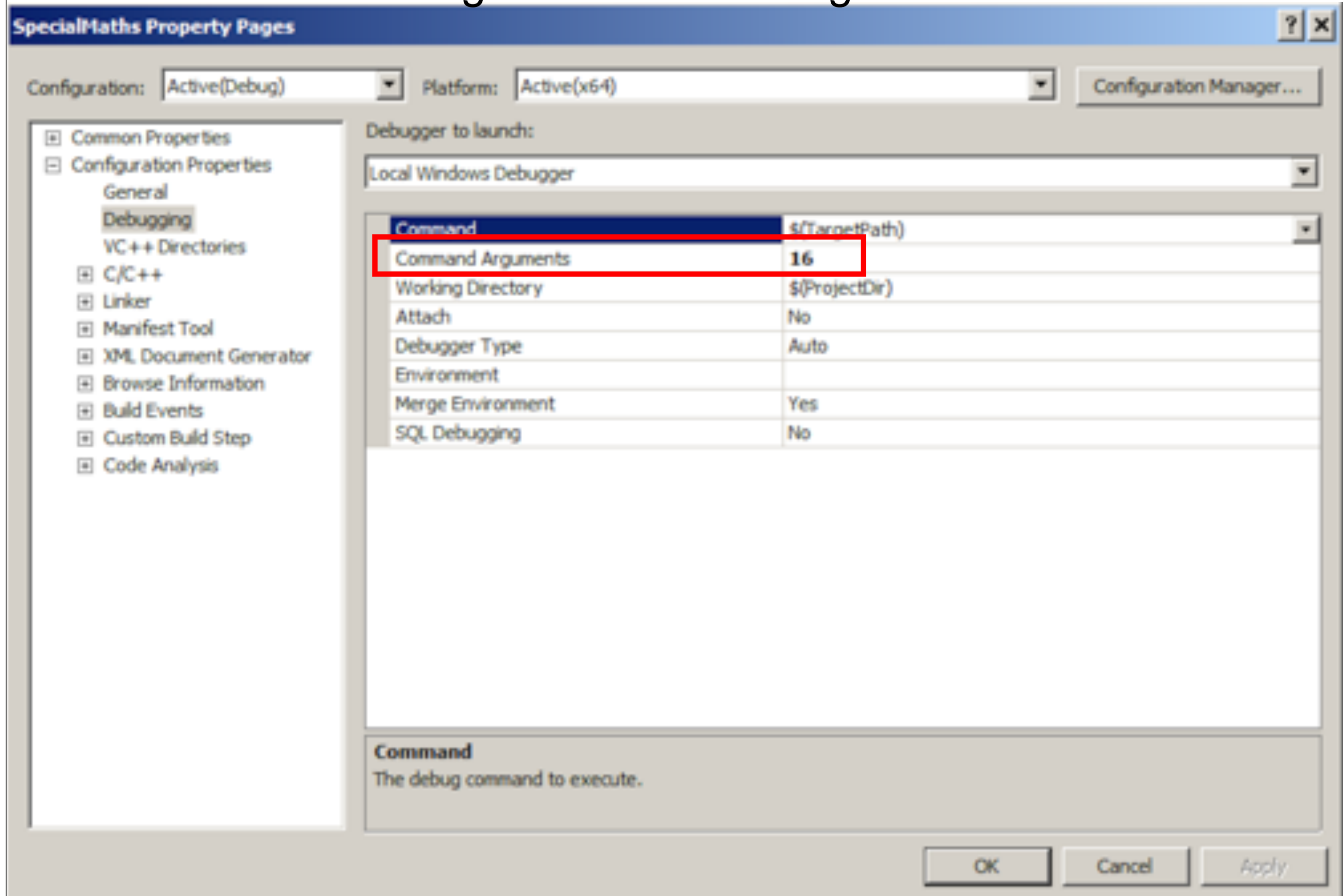
SpecialMaths.c

With command line arguments, 2 function parameters, and special maths known to generate particular asm as a teachable moment :)

```c
#include <stdlib.h>
int main(int argc, char ** argv){
	int a;
	//reminder: atoi() converts an ASCII string to an integer
	a = atoi(argv[1]);
	return 2*argc + a;
}
```

```
main:
0000000140001000  mov   qword ptr [rsp+10h],rdx
0000000140001005  mov   dword ptr [rsp+8],ecx
0000000140001009  sub   rsp,38h
000000014000100D  mov   eax,8
0000000140001012  imul  rax,rax,1
0000000140001016  mov   rcx,qword ptr [rsp+48h]
000000014000101B  mov   rcx,qword ptr [rcx+rax]
000000014000101F  call  qword ptr [400020F8h]
0000000140001025  mov   dword ptr [rsp+20h],eax
0000000140001029  mov   eax,dword ptr [rsp+20h]
000000014000102D  mov   ecx,dword ptr [rsp+40h]
0000000140001031  lea   eax,[rax+rcx*2]
0000000140001034  add   rsp,38h
0000000140001038  ret
```

Setting command line arguments



“r/mX” Addressing Forms Reminder

- Anywhere you see an r/mX it means it could be taking a value either from a register, or a memory address.
- I’m just calling these “r/mX forms” because anywhere you see “r/mX” in the manual, the instruction can be a variation of the below forms.
- In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
  - mov rax, rbx
  - mov rax, [rbx]
  - mov rax, [rbx+rcx*X] (X=1, 2, 4, 8)
  - mov rax, [rbx+rcx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
- Most complicated form is: `[base + index*scale + disp]`

More info: Intel v2a, Section 2.1.5 page 2-4
in particular Tables 2-2 and 2-3

12

LEA - Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- Uses the r/mX form but **is the exception to the rule** that the square brackets [ ] syntax means dereference ("value at")
- Example: rbx = 0x2, rdx = 0x1000
  - lea rax, [rdx+rbx*8+5]
  - rax = 0x1015, not the value at 0x1015

Book page 112

SpecialMaths.c takeaways

- When a compiler sees “special math” that can be computed in the form “a + b*X + Y” (derived from the “r/mX” form, where X = {1, 2, 4, 8}, and Y = {0-2^32-1}), then it can compute the result faster if it uses the LEA instruction, rather than a IMUL instruction for instance.
- More evidence that pass-by-register function arguments are being stored onto the stack at some point (this time both ecx and rdx)

```
#include <stdlib.h>
int main(int argc, char ** argv){
	int a;
	a = atoi(argv[1]);
	return 2*argc + a;
}
```

```
main:
0000000140001000  mov         qword ptr [rsp+10h],rdx
0000000140001005  mov         dword ptr [rsp+8],ecx
0000000140001009  sub         rsp,38h
000000014000100D  mov         eax,8
0000000140001012  imul        rax,rax,1
0000000140001016  mov         rcx,qword ptr [rsp+48h]
000000014000101B  mov         rcx,qword ptr [rcx+rax]
000000014000101F  call        qword ptr [400020F8h]
0000000140001025  mov         dword ptr [rsp+20h],eax
0000000140001029  mov         eax,dword ptr [rsp+20h]
000000014000102D  mov         ecx,dword ptr [rsp+40h]
0000000140001031  lea         eax,[rax+rcx*2]
0000000140001034  add         rsp,38h
0000000140001038  ret
```

Instructions we now know (12)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB
- IMUL
- MOVZX/MOVSX
- LEA

Back to Hello World

```
.text:0000000140001000 main
.text:0000000140001000
.text:0000000140001000 sub     rsp, 28h
.text:0000000140001004 lea     rcx, Format     ; "Hello World!\n"
.text:000000014000100B call    cs:__imp_printf
.text:0000000140001011 mov     eax, 1234h
.text:0000000140001016 add     rsp, 28h
.text:000000014000101A retn
```

Are we all comfortable with this now?

(other than the fact that IDA hides the address of the string which is being calculated and replaces it with “Format” for the format string being passed to printf?)

Windows Visual C++ 2012, /GS (buffer overflow protection) option turned off
Optimize for minimum size (/O1) turned on
Disassembled with IDA Pro 6.6 (with some omissions for fitting on screen)