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

Xeno Kovah – 2014-2015
xeno@legbacore.com
All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/

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:
int func(int a){
    int i = a;
    return i;
}
int main(){
    return func(0x11);
}
The stack looks like this at line 00000014000100F in func():

```
0x28 bytes

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000`0012FEB8</td>
<td>return address = 00000001400012FD</td>
</tr>
<tr>
<td>00000000`0012FEB0</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FEA8</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FEA0</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FE98</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FE90</td>
<td>arg1 = ecx = 0x11</td>
</tr>
</tbody>
</table>

0x18 bytes

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000`0012FE88</td>
<td>return address = 000000014000102E</td>
</tr>
<tr>
<td>...</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FE78</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FE70</td>
<td>undef 0x00000011</td>
</tr>
</tbody>
</table>
```

Huh? func() wrote the register-passed argument above the return address?

Because the asm only wrote a “dword ptr” (4 bytes) worth of memory at this location, so the top 4 bytes are undefined.
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?

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

int main(){
    return func(0x11);
}
```

```
func:
0000001400010000  mov     dword ptr [rsp+8],ecx
0000001400010004  sub     rsp,18h
0000001400010008  mov     eax,dword ptr [rsp+20h]
000000140001000c  mov     dword ptr [rsp],eax
000000140001000f  mov     eax,dword ptr [rsp]
0000001400010012  add     rsp,18h
0000001400010016  ret

main:
0000001400010020  sub     rsp,28h
0000001400010024  mov     ecx,11h
0000001400010029  call    func (0140001000h)
000000140001002e  add     rsp,28h
0000001400010032  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);
}

http://www.youtube.com/watch?v=Nr8r09c8ogg
The stack looks like this at line 0000000140001031 in func():

```
00000000`0012FEB8 return address = 000000014000132D
  ... undef
00000000`0012FEA8 undef
00000000`0012FEA0 arg5 = 0x55
00000000`0012FE98 arg4 = r9 = 0x44
00000000`0012FE90 arg3 = r8 = 0x33
00000000`0012FE88 arg2 = edx = 0x22
00000000`0012FE80 arg1 = ecx = 0x11
00000000`0012FE78 return address = 0000000140001067
  ... undef
00000000`0012FE78 undef
00000000`0012FE70 undef `fffff
```

A pattern emerges! Say hello to the Microsoft stack “Shadow Space”!

Because the asm only wrote a “dword ptr” (4 bytes) worth of memory at this location, so the top 4 bytes are undefined.
Who knows what the first 4 parameters passed in registers were when you’re trying to backtrace the stack calls?


TooManyParameters.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);
}

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)...
#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;
}

With command line arguments, 2 function parameters, and special maths known to generate particular asm as a teachable moment :)
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
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
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)
Instructions we now know (12)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB
- IMUL
- MOVZX/MOVVSX
- 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)