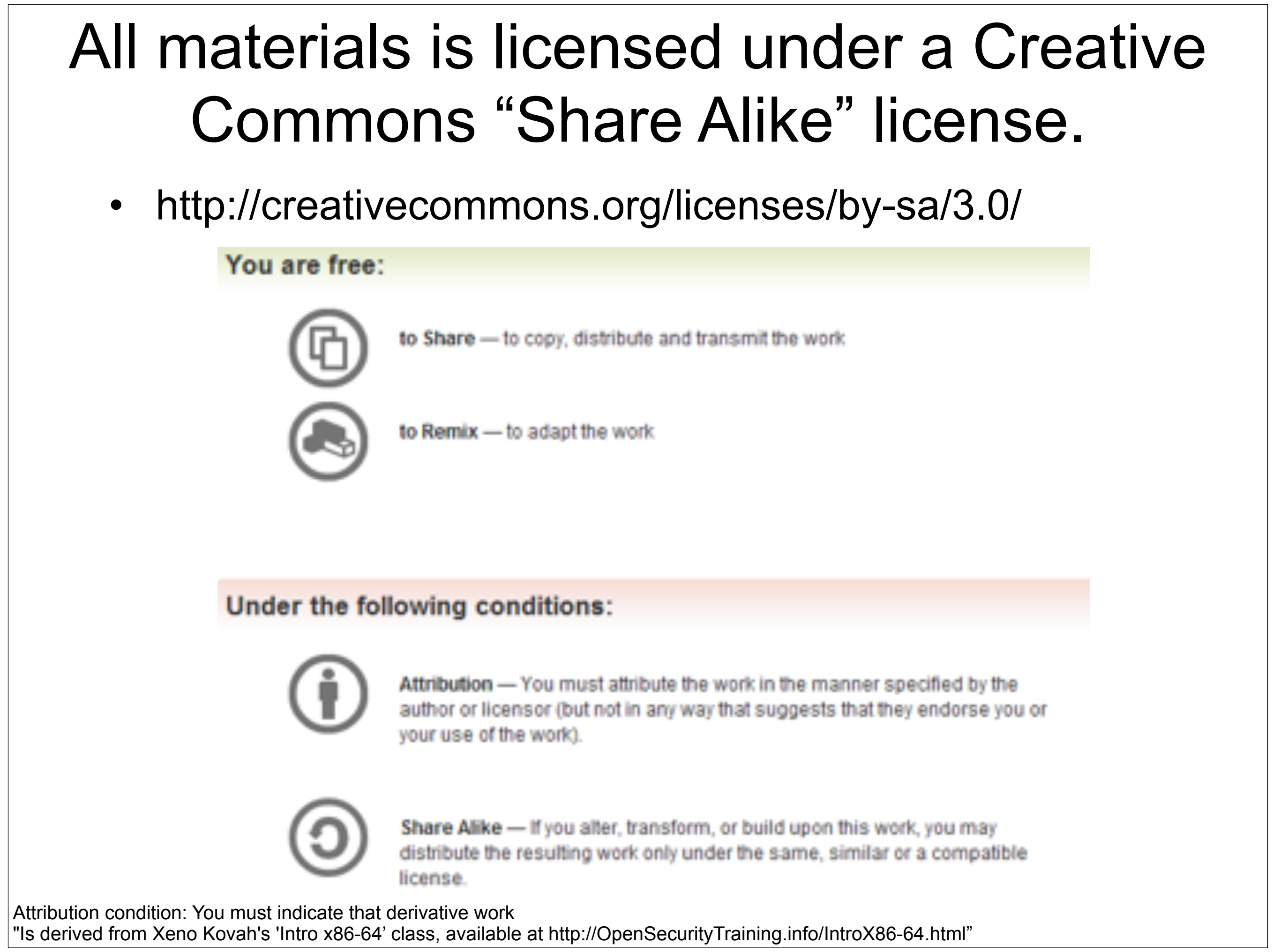
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

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/
You are free:
to Share — to copy, distribute and transmit the work
to Remix — to adapt the work
Under the following conditions:
Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
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"

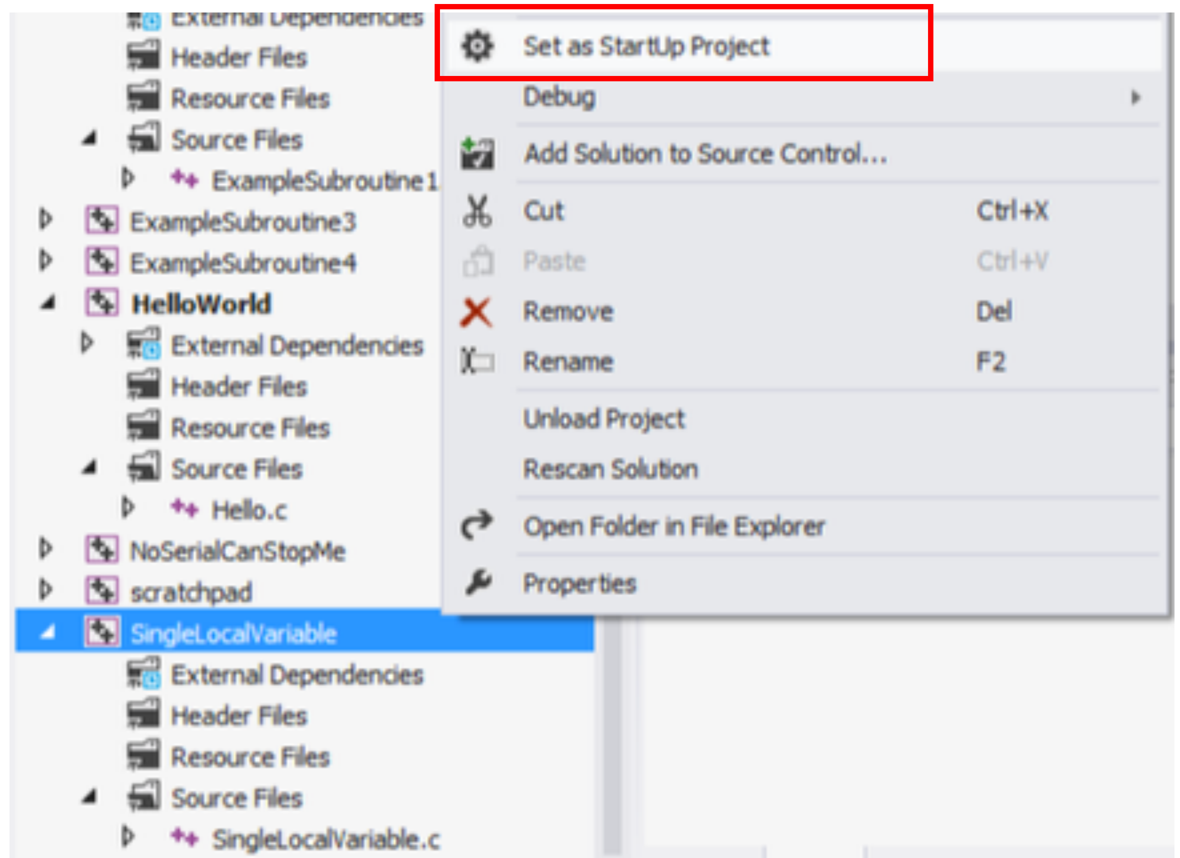
SingleLocalVariable.c

Adding a single local variable

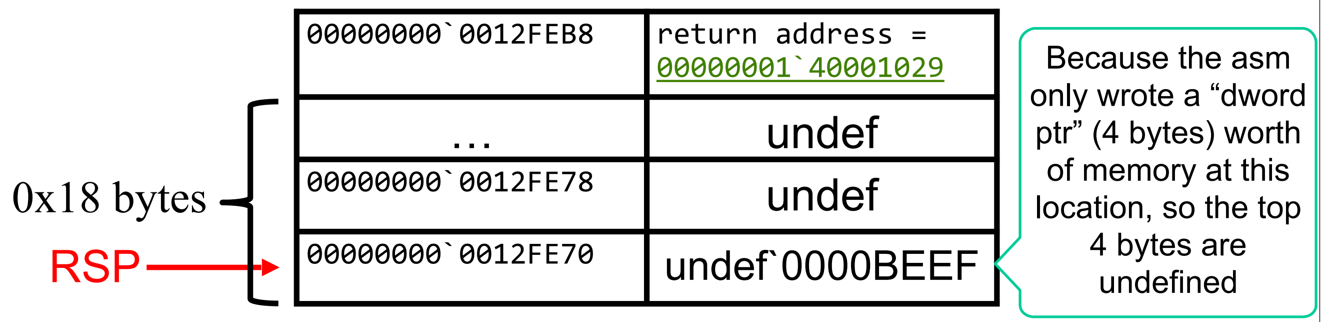
```
//SingleLocalVariable.c:
int func(){
    int i = 0xbeef;
    return i;
}
int main(){
    return func();
}
```

```
func:
0000000140001000  sub     rsp,18h
0000000140001004  mov     dword ptr [rsp],0BEEFh
000000014000100B  mov     eax,dword ptr [rsp]
000000014000100E  add     rsp,18h
0000000140001012  ret
main:
0000000140001020  sub     rsp,28h
0000000140001024  call    func (0140001000h)
0000000140001029  add     rsp,28h
000000014000102D  ret
```

Don’t forget to change the next project to the StartUp project as you’re moving between labs!



Based on the asm, we can infer the stack looks like this at line 000000014000100B in func()



SingleLocalVariable.c takeaways

- Local variables lead to an allocation of space on the stack, within the function where the variable is scoped to
- In VS (when optimization is turned off), there is an over-allocation of space for local variables
    - 0x18 reserved for only 0x4 (int) worth of data

```c
//SingleLocalVariable.c:
int func(){
    int i = 0xbeef;
    return i;
}
int main(){
    return func();
}
```

```
func:
    sub     rsp,18h
    mov     dword ptr [rsp],0BEEFh
    mov     eax,dword ptr [rsp]
    add     rsp,18h
    ret
main:
    sub     rsp,28h
    call    func (0140001000h)
    add     rsp,28h
    ret
```

ArrayLocalVariable.c

Adding and accessing an array local variable

```c
//ArrayLocalVariable.c:
short main(){
    int a;
    short b[6];
    long long c;
    a = 0x100d;
    c = 0xd00d;
    b[1] = (short)a;
    b[4] = b[1] + (short)c;
    return b[4];
}
```

```
main:
0000000140001000  sub    rsp,28h
0000000140001004  mov    dword ptr [rsp],100Dh
000000014000100B  mov    qword ptr [rsp+8],0D00Dh
0000000140001014  mov    eax,2
0000000140001019  imul   rax,rax,1
000000014000101D  movzx  ecx,word ptr [rsp]
0000000140001021  mov    word ptr [rsp+rax+10h],cx
0000000140001026  mov    eax,2
000000014000102B  imul   rax,rax,1
000000014000102F  movsx  eax,word ptr [rsp+rax+10h]
0000000140001034  movsx  ecx,word ptr [rsp+8]
0000000140001039  add    eax,ecx
000000014000103B  mov    ecx,2
0000000140001040  imul   rcx,rcx,4
0000000140001044  mov    word ptr [rsp+rcx+10h],ax
0000000140001049  mov    eax,2
000000014000104E  imul   rax,rax,4
0000000140001052  movsx  eax,word ptr [rsp+rax+10h]
0000000140001057  add    rsp,28h
000000014000105B  ret
```

9

IMUL - Signed Multiply

- FYI, Visual Studio seems to have a predilection for imul over mul (unsigned multiply). You'll see it showing up in places you expect mul.
  - I haven't been able to get it to generate the latter for simple examples.
- Three forms. One, two, or three operands
  - imul r/mX — rdx:rax = rax * r/mX
  - imul reg, r/mX — reg = reg * r/mX
  - imul reg, r/mX, immediate — reg = r/mX * immediate
- **Three** operands? Possibly the only "basic" instruction (meaning non-added-on-instruction-set(MMX,AVX,AEX,VMX,etc)) of it's kind? (see link in notes)

Book page 65

http://www.microsoft.com/msj/0698/hood0698.aspx - "There's even a form of the IMUL instruction that takes three operands. To my knowledge, this is the only instruction in the Intel opcode set with this distinction."

I found that quote while trying to find a way to make visual studio naturally emit a MUL instruction. Also, while things containing an r/mX can encode a mnemonic which looks like is has more operands, the information is still contained in a the normal combo of one or two bytes

IMUL - examples

initial → operation → result

imul ecx

Initial:

edx	eax	r/mX(ecx)
0x0	0x44000000	0x4

Result:

edx	eax	r/mX(ecx)
0x1	0x10000000	0x4

imul rax, rcx

Initial:

rax	r/mX(rcx)
0x20	0x4

Result:

rax	r/mX(rcx)
0x80	0x4

imul rax, rcx, 0x6

Initial:

rax	r/mX(rcx)
0x20	0x4

Result:

rax	r/mX(rcx)
0x18	0x4

10

MOVZX - Move with zero extend

11

MOVSX - Move with sign extend

- Used to move small values (from smaller types) into larger registers (holding larger types)
- Support same r->r, r->m, m->r, i->m, i->r forms as normal MOV
- "Zero extend" means the CPU unconditionally fills the high order bits of the larger register with zeros
- "Sign extend" means the CPU fills the high order bits of the destination larger register with whatever the sign bit is set to on the small value

Book page 53

MOVZX/MOVSX - examples

- mov eax, 0xF00DFACE
- movzx rbx, eax

  now rbx = 0x00000000`F00DFACE
- movsx rbx, eax

  now rbx = 0xFFFFFFFF`F00DFACE,

because the sign bit (most significant bit) of 0xF00DFACE is 1

Based on the asm, we can infer the stack looks like this at line 0000000140001049 of main():

Address	Value
00000000`0012FEB8	return address = 000000014000131d
…	undef
00000000`0012FEAC	undef
00000000`0012FEAA	short b[5] = undef
00000000`0012FEA8	short b[4] = *maths!*
00000000`0012FEA6	short b[3] = undef
00000000`0012FEA4	short b[2] = undef
00000000`0012FEA2	short b[1] = *maths!*
00000000`0012FEA0	short b[0] = undef
00000000`0012FE98	long long c = 0xd00d
00000000`0012FE90	int a = 0x100d

0x28 bytes

I'm cheating here and not using only 0x8-sized entries, for clarity of array indices' address

RSP

Note how there's wasted space by storing a 4 byte value ("int a") in an 8 byte space

ArrayLocalVariable.c takeaways

- Local variables *need not be* stored on the stack in the same order they are defined in the high level language
- (In VS unoptimized code) Array access is typically done by multiplying the size of the array element (2 bytes for a short in this case), times the index that is desired to be access (indices 1 and 4 in this case)
- Moving a small value to a large register will result in a zero extend. Addition using signed values could result in a sign extend, if the arithmetic is done in a larger register

```
//ArrayLocalVariable.c:
short main(){
    int a;
    short b[6];
    long long c;
    a = 0x100d;
    c = 0xd00d;
    b[1] = (short)a;
    b[4] = b[1] + (short)c;
    return b[4];
}
```

```
main:
 sub        rsp,28h
 mov        dword ptr [rsp],100Dh
 mov        qword ptr [rsp+8],0D00Dh
 mov        eax,2
 imul       rax,rax,1
 movzx      ecx,word ptr [rsp]
 mov        word ptr [rsp+rax+10h],cx
 mov        eax,2
 imul       rax,rax,1
 movsx      eax,word ptr [rsp+rax+10h]
 movsx      ecx,word ptr [rsp+8]
 add        eax,ecx
 mov        ecx,2
 imul       rcx,rcx,4
 mov        word ptr [rsp+rcx+10h],ax
 mov        eax,2
 imul       rax,rax,4
 movsx      eax,word ptr [rsp+rax+10h]
 add        rsp,28h
 ret
```

StructLocalVariable.c

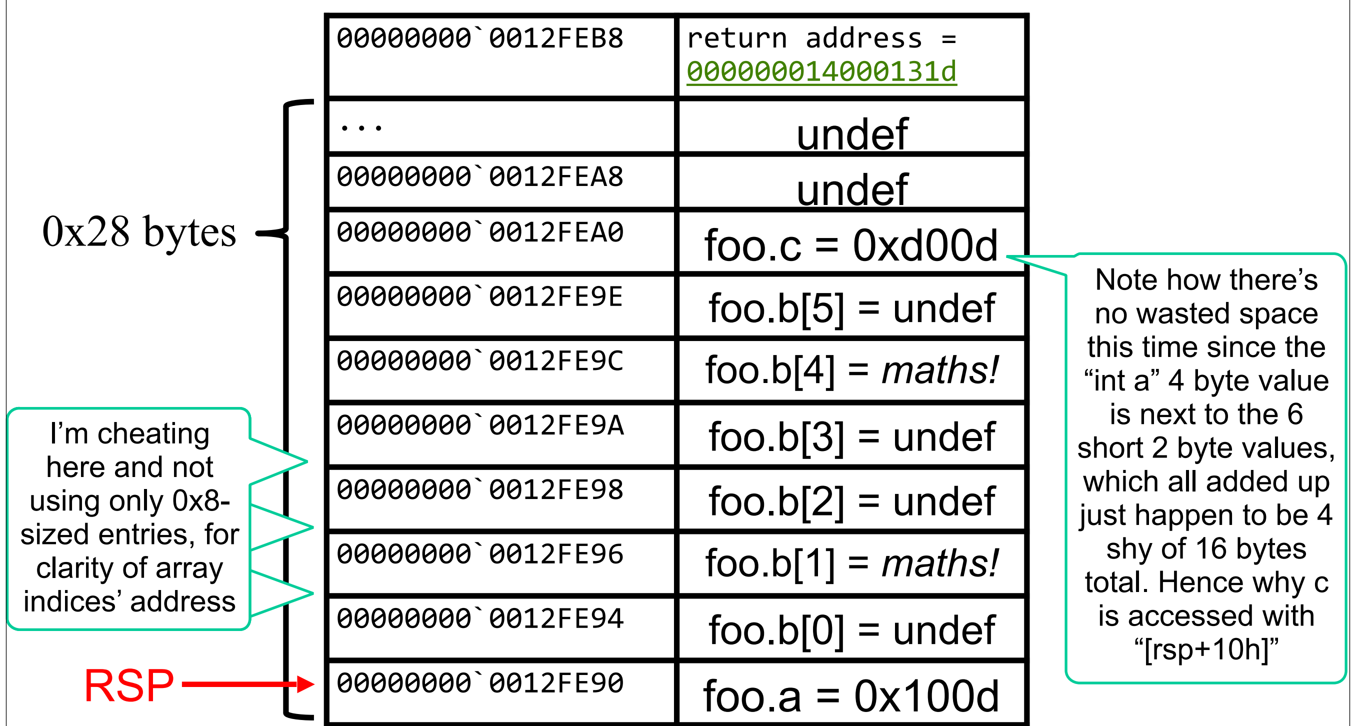
Accessing a struct local variable

```c
//StructLocalVariable.c:
typedef struct mystruct{
    int a;
    short b[6];
    long long c;
} mystruct_t;

short main(){
    mystruct_t foo;
    foo.a = 0x100d;
    foo.c = 0xd00d;
    foo.b[1] = foo.a;
    foo.b[4] = foo.b[1] + foo.c;
    return foo.b[4];
}
```

```
main:
0000000140001000  sub        rsp,28h
0000000140001004  mov        dword ptr [rsp],100Dh
000000014000100B  mov        qword ptr [rsp+10h],0D00Dh
0000000140001014  mov        eax,2
0000000140001019  imul       rax,rax,1
000000014000101D  movzx      ecx,word ptr [rsp]
0000000140001021  mov        word ptr [rsp+rax+4],cx
0000000140001026  mov        eax,2
000000014000102B  imul       rax,rax,1
000000014000102F  movsx      rax,word ptr [rsp+rax+4]
0000000140001035  add        rax,qword ptr [rsp+10h]
000000014000103A  mov        ecx,2
000000014000103F  imul       rcx,rcx,4
0000000140001043  mov        word ptr [rsp+rcx+4],ax
0000000140001048  mov        eax,2
000000014000104D  imul       rax,rax,4
0000000140001051  movzx      eax,word ptr [rsp+rax+4]
0000000140001056  add        rsp,28h
000000014000105A  ret
```

Based on the asm, we can infer the stack looks like this:



StructLocalVariable.c

- Fields in a struct *must be* stored in the same order they are defined in the high level language. And they will appear with the first field at the lowest address, and all subsequent fields higher.

```
//StructLocalVariable.c:
typedef struct mystruct{
    int a;
    short b[6];
    long long c;
} mystruct_t;

short main(){
    mystruct_t foo;
    foo.a = 0x100d;
    foo.c = 0xd00d;
    foo.b[1] = foo.a;
    foo.b[4] = foo.b[1] + foo.c;
    return foo.b[4];
}
```

```
main:
 sub      rsp,28h
 mov      dword ptr [rsp],100Dh
 mov      qword ptr [rsp+10h],0D00Dh
 mov      eax,2
 imul     rax,rax,1
 movzx    ecx,word ptr [rsp]
 mov      word ptr [rsp+rax+4],cx
 mov      eax,2
 imul     rax,rax,1
 movsx    rax,word ptr [rsp+rax+4]
 add      rax,qword ptr [rsp+10h]
 mov      ecx,2
 imul     rcx,rcx,4
 mov      word ptr [rsp+rcx+4],ax
 mov      eax,2
 imul     rax,rax,4
 movzx    eax,word ptr [rsp+rax+4]
 add      rsp,28h
 ret
```

Instructions we now know (11)

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