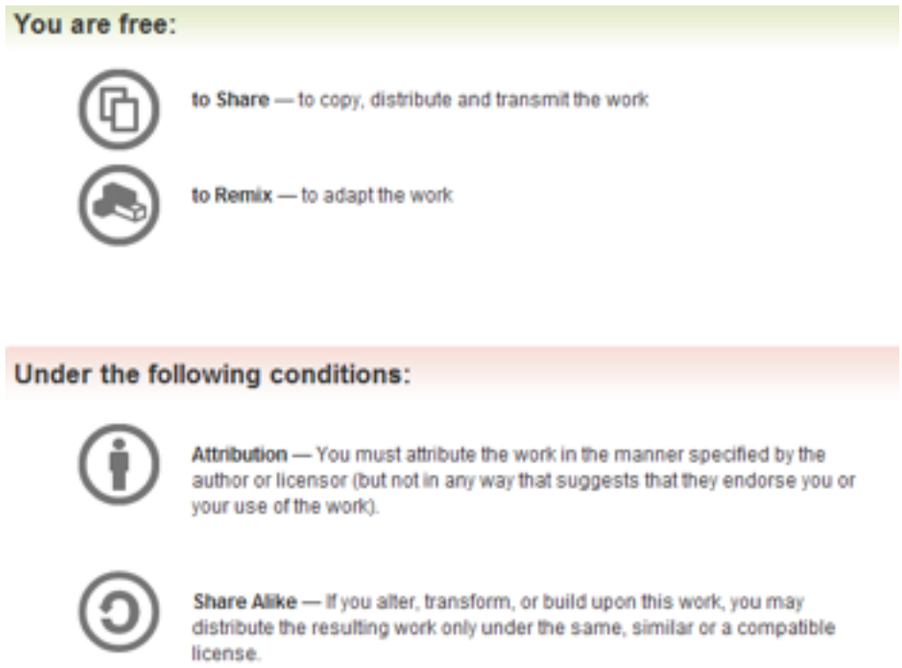
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

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"

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"

Inline assembly

- Inline assembly is a way to include assembly directly in a C/C++ file. However, the syntax will differ between compilers and assemblers.
- There are times when you actually have to code asm in order to do something from a C/C++ file.
  - Very common in OS or driver design, because there are many aspects of hardware which can only be accessed with special instructions
  - In crypto you might want access to the “rol/ror - rotate left/right” instructions which don’t have corresponding C syntax like shifts do
- Or maybe you just want full control over the code being generated for optimization purposes
  - Keep in mind the compiler may still optimize your inline asm
- Also it’s a great way to simply experiment with instructions
  - Though getting the syntax right for the desired instructions is sometimes annoying

GCC inline assembly

- GCC syntax - AT&T syntax
- asm(“instructions separated by \n”);
  - **DO** need a semicolon after close parentheses

```
int myVar = 0xdeadbeef;
asm("movl -0x4(%rbp), %eax\n"
    "cmp $0xdeadbeef,%eax\n"
    "je myLabel\n"
    "xor %eax, %eax\n"
    "myLabel: movw $0xFFFF, %cx\n"
    "and %ecx, %eax”);
```

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

http://www.cs.virginia.edu/~clc5q/gcc-inline-asm.pdf

GCC inline assembly 2

- Syntax using C variables (aka “extended asm”):

```
asm ( assembler template
: output operands                   /* optional */
: input operands                    /* optional */
: list of clobbered registers       /* optional */
);
```

```
int myVar;
//value into C variable from register
asm ("movl %%eax, %0" : "=r" (myVar) );
//value into register from C variable
asm ("movl %0, %%eax" : : "r" (myVar) );
```

.byte

- Once you learn about opcodes later on, you can even specify exactly the instructions you want to use by using the “.byte” keyword, to place specific bytes into the code.
- Those bytes can then be interpreted as instructions or data
- This is sometimes useful if you can’t figure out the inline asm syntax for the instruction you want to use, but you know its opcodes (either from seeing them elsewhere, or by reading the manual)
- Examples:
  - asm(“.byte 0x55”); is “push %rbp”
  - asm(".byte 0x48 ; .byte 0x89 ; .byte 0xE5"); is “mov %rsp, %rbp”

_emit syntax: http://msdn.microsoft.com/en-us/library/1b80826t(VS.80).aspx

VisualStudio assembly

- Visual Studio does not support inline assembly for x64 code
- If you must run hand-written assembly, you must write write assembly functions in a separate asm file, use an assembler to assemble it, and then link it with your C code
- Will not go into detail here, but instructions can be found here:
- http://www.codeproject.com/Articles/271627/Assembly-Programming-with-Visual-Studio

VisualStudio assembly 2

- Certain assembly instructions can be inserted into C code by using VS compiler intrinsics
- These look like C functions calls, but the compiler substitutes them with literal assembly instructions
- Examples:
  - __writeeflags
  - __stosX (rep stos)
  - __movsX (rep movs)
  - __cpuid
  - _rotrX (ror)
- Many, many more:
- http://msdn.microsoft.com/en-us/library/hh977022.aspx

Bonus Slides

Visual Studio Inline Assembly for
32-bit code

VisualStudio inline assembly

- VisualStudio syntax - intel-syntax
- __asm{ instructions separated by \n};
  - That's two underscores at the beginning
  - Don't even need a semicolon after it, but I put them there since it makes the auto-indent work correctly

```
__asm{          mov eax, [esp+0x4]
                cmp eax, 0xdeadbeef
                je myLabel
                xor eax, eax
myLabel:        mov bl, al
};
```

VisualStudio inline assembly 2

- Syntax using C variables is the same, just put the variable in place of a register name for instance. (The assembler will substitute the correct address for the variable.)
- http://msdn.microsoft.com/en-us/library/4ks26t93(VS.80).aspx

```
int myVar;
//value into C variable from register
__asm {mov myVar, eax};
//value into register from C variable
__asm {mov eax, myVar};
```

_emit and .byte

- Once you learn about opcodes later on, you can even specify exactly the instructions you want to use by using the “_emit” or “.byte” keywords, to place specific bytes into the code.
- Those bytes can then be interpreted as instructions or data
- This is sometimes useful if you can’t figure out the inline asm syntax for the instruction you want to use, but you know its opcodes (either from seeing them elsewhere, or by reading the manual)
- Examples:
  - __asm{_emit 0x55} is __asm{push ebp}
  - __asm{_emit 0x89};__asm{_emit 0xE5} is __asm{mov ebp, esp}
  - asm(“.byte 0x55”); is asm(“push %ebp”);
  - asm(".byte 0x89 ; .byte 0xE5"); is asm(“mov %esp, %ebp”);

_emit syntax: http://msdn.microsoft.com/en-us/library/1b80826t(VS.80).aspx