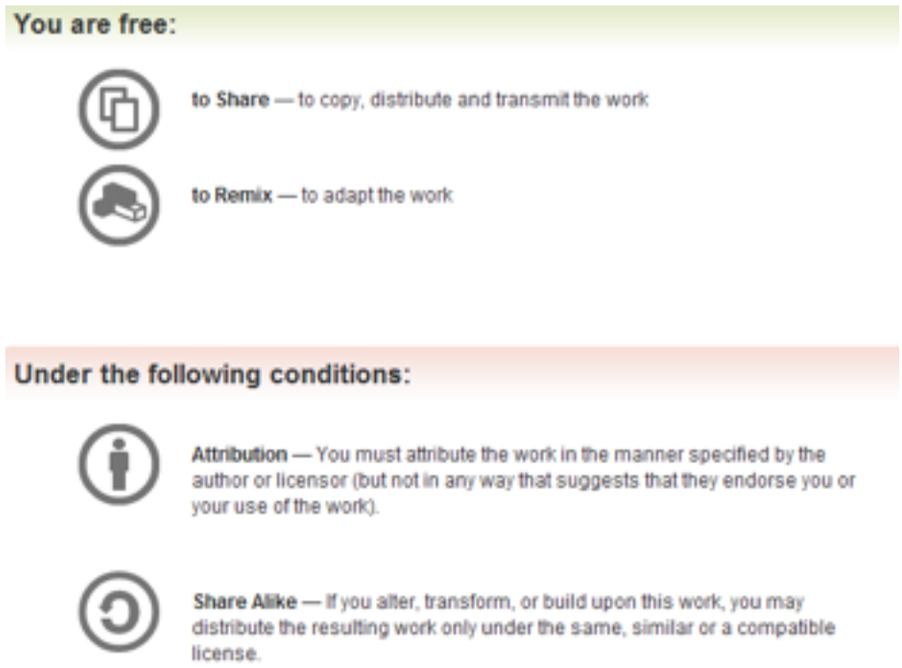
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

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"

Discussion: variable-length opcodes

- Any given sequence of bytes can be interpreted in different ways, depending on where the CPU starts executing it from
- This has many subtle implications, but it seems to get abused the most in the security domain
- Examples: inability to validate intended instructions, return-oriented-programming, code obfuscation and polymorphic/self-modifying code
- In comparison, RISC architectures typically have fixed instruction sizes, which must be on aligned boundaries, and thus makes disassembly much simpler

Variable-length opcode decoding example

```
(gdb) x/10i $rip
   0x4004ed <main>:    push   %rbp
   0x4004ee <main+1>:   mov    %rsp,%rbp
   0x4004f1 <main+4>:   movl   $0xdeadbeef,-0x4(%rbp)
   0x4004f8 <main+11>:  mov    -0x4(%rbp),%eax
   0x4004fb <main+14>:  mov    %eax,%eax
   0x4004fd <main+16>:  mov    %eax,%eax
   0x4004ff <main+18>:  mov    %eax,-0x4(%rbp)
   0x400502 <main+21>:  pop    %rbp
   0x400503 <main+22>:  retq
```

```
(gdb) x/10i $rip+9
   0x4004f6 <main+9>:   lods   %ds:(%rsi),%eax
   0x4004f7 <main+10>:  fimul  -0x3f7603bb(%rbx)
   0x4004fd <main+16>:  mov    %eax,%eax
   0x4004ff <main+18>:  mov    %eax,-0x4(%rbp)
   0x400502 <main+21>:  pop    %rbp
   0x400503 <main+22>:  retq
```

```
(gdb) x/10i $rip+3
   0x4004f0 <main+3>:   in     $0xc7,%eax
   0x4004f2 <main+5>:   rex.RB cld
   0x4004f4 <main+7>:   out    %eax,(%dx)
   0x4004f5 <main+8>:   mov    $0x458bdead,%esi
   0x4004fa <main+13>:  cld
   0x4004fb <main+14>:  mov    %eax,%eax
   0x4004fd <main+16>:  mov    %eax,%eax
   0x4004ff <main+18>:  mov    %eax,-0x4(%rbp)
   0x400502 <main+21>:  pop    %rbp
   0x400503 <main+22>:  retq
```

```
(gdb) x/10i $rip+15
   0x4004fc <main+15>:  rorb
$0x5d,-0x3ba7640(%rcx)
   0x400503 <main+22>:  retq
```

x86 has been called “self-synchronizing” because it does eventually seem to get back to the correct asm. That’s not a useful property for execution, only for disassemblers trying to speculate on a correct disassembly

Discussion: variable-length opcodes

- An interesting property of x86 is that even if you pick a wrong offset to start disassembling from, very frequently the disassembly will re-synchronize with the original, intended, instruction sequence
- In the preceding examples you can see that when disassembly is started at +3 bytes in, it re-synchs by +14 bytes. When started at +9, it re-synchs by +16, etc.
- This was noted also in “Obfuscation of Executable Code to Improve Resistance to Static Disassembly” by Linn & Debray
  - http://www.cs.arizona.edu/solar/papers/CCS2003.pdf