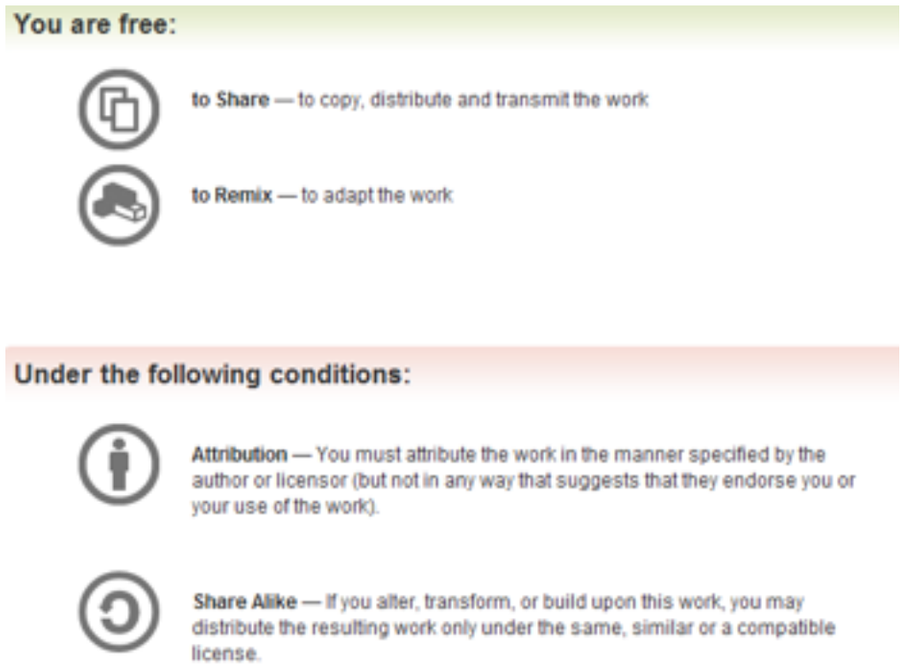
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

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"

Control Flow

- Two forms of control flow
  - Conditional - go somewhere if a condition is met. Think “if”s, switches, loops
  - Unconditional - go somewhere no matter what. Function calls, goto, exceptions, interrupts.
- We’ve already seen procedure calls manifest themselves as call/ret, let’s see how goto manifests itself in asm.

GotoExample.c

```
//Goto example
#include <stdio.h>
int main(){
        goto mylabel;
        printf("skipped\n");
mylabel:
        printf("goto ftw!\n");
        return 0xf00d;
}
```

```
main:
0000000140001010  sub     rsp,28h
0000000140001014  jmp     0000000140001023
0000000140001016  lea     rcx,[40006000h]
000000014000101D  call    qword ptr [40008368h]
$mylabel:
0000000140001023  lea     rcx,[40006010h]
000000014000102A  call    qword ptr [40008368h]
0000000140001030  mov     eax,0F00Dh
0000000140001035  add     rsp,28h
0000000140001039  ret
```

Visual Studio “RIP-relative” display errors.
Goto “RIPRelativeAddressing” slides

JMP - Jump

- Change rip to the given address
- Main forms of the address
  - Short relative (1 byte displacement from end of the instruction)
    - “jmp 0000000140001023” doesn’t have the number 0000000140001023 anywhere in it, it’s really “jmp 0x0E bytes forward”
    - Some disassemblers will indicate this with a mnemonic by writing it as “jmp short”
  - Near relative (4 byte displacement from current eip)
  - Absolute (hardcoded address in instruction)
  - Absolute Indirect (address calculated with r/m32)
    - TODO CONFIRM R/M64
- jmp -2 == infinite loop for short relative jmp :)

GotoExample.c takeaways

- goto == jmp in asm :)

```
//Goto example
#include <stdio.h>
int main(){
        goto mylabel;
        printf("skipped\n");
mylabel:
        printf("goto ftw!\n");
        return 0xf00d;
}
```

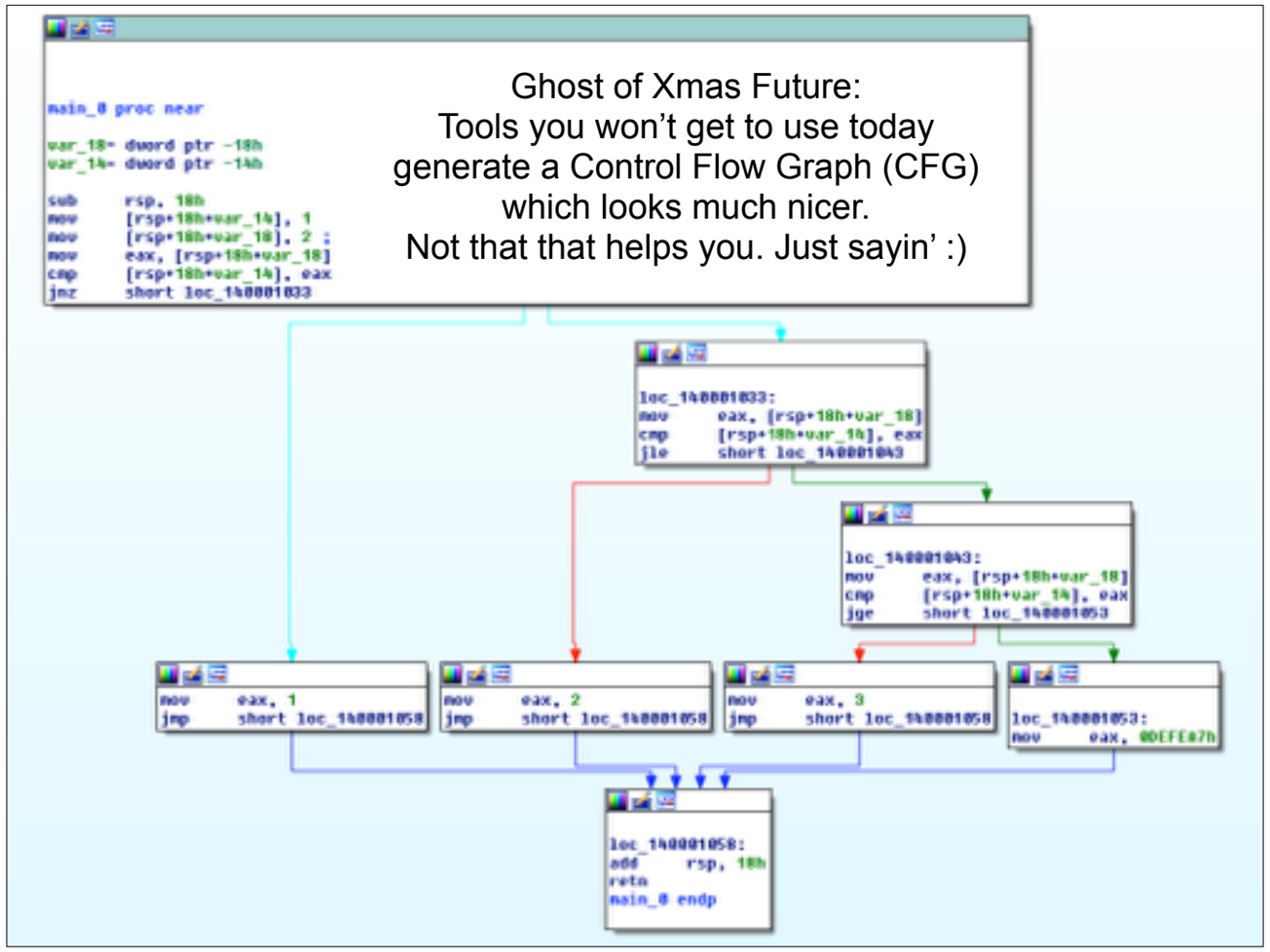
```
main:
0000000140001010  sub      rsp,28h
0000000140001014  jmp      0000000140001023
0000000140001016  lea      rcx,[40006000h]
000000014000101D  call     qword ptr [40008368h]
$mylabel:
0000000140001023  lea      rcx,[40006010h]
000000014000102A  call     qword ptr [40008368h]
0000000140001030  mov      eax,0F00Dh
0000000140001035  add      rsp,28h
0000000140001039  ret
```

IfExample.c

```
int main(){
        int a=1, b=2;
        if(a == b){
                return 1;
        }
        if(a > b){
                return 2;
        }
        if(a < b){
                return 3;
        }
        return 0xdefea7;
}
```

Jcc

```
main:
0000000140001010  sub     rsp,18h
0000000140001014  mov     dword ptr [rsp+4],1
000000014000101C  mov     dword ptr [rsp],2
0000000140001023  mov     eax,dword ptr [rsp]
0000000140001026  cmp     dword ptr [rsp+4],eax
000000014000102A  jne     0000000140001033
000000014000102C  mov     eax,1
0000000140001031  jmp     0000000140001058
0000000140001033  mov     eax,dword ptr [rsp]
0000000140001036  cmp     dword ptr [rsp+4],eax
000000014000103A  jle     0000000140001043
000000014000103C  mov     eax,2
0000000140001041  jmp     0000000140001058
0000000140001043  mov     eax,dword ptr [rsp]
0000000140001046  cmp     dword ptr [rsp+4],eax
000000014000104A  jge     0000000140001053
000000014000104C  mov     eax,3
0000000140001051  jmp     0000000140001058
0000000140001053  mov     eax,0DEFEA7h
0000000140001058  add     rsp,18h
000000014000105C  ret
```


main_0 proc near
var_18= dword ptr -18h
var_14= dword ptr -14h
sub rsp, 18h
mov [rsp+18h+var_14], 1
mov [rsp+18h+var_18], 2 ;
mov eax, [rsp+18h+var_18]
cmp [rsp+18h+var_14], eax
jnz short loc_140001033
Ghost of Xmas Future:
Tools you won't get to use today
generate a Control Flow Graph (CFG)
which looks much nicer.
Not that that helps you. Just sayin' :)
loc_140001033:
mov eax, [rsp+18h+var_18]
cmp [rsp+18h+var_14], eax
jle short loc_140001043
loc_140001043:
mov eax, [rsp+18h+var_18]
cmp [rsp+18h+var_14], eax
jge short loc_140001053
mov eax, 1
jmp short loc_140001058
mov eax, 2
jmp short loc_140001058
mov eax, 3
jmp short loc_140001058
loc_140001053:
mov eax, 0DEFE07h
loc_140001058:
add rsp, 18h
retn
main_0 endp

Jcc - Jump If Condition Is Met

- There are more than 4 pages of conditional jump types! Luckily a bunch of them are synonyms for each other.
- JNE == JNZ (Jump if not equal, Jump if not zero, both check if the Zero Flag (ZF) == 0)

Book page 93

Some Notable Jcc Instructions

- JZ/JE: if ZF == 1
- JNZ/JNE: if ZF == 0
- JLE/JNG : if ZF == 1 or SF != OF
- JGE/JNL : if SF == OF
- JBE/JNA: if CF == 1 OR ZF == 1
- JB: if CF == 1
- Note: Don’t get hung up on memorizing which flags are set for what. More often than not, you will be running code in a debugger, not just reading it. In the debugger you can just look at rflags and/or watch whether it takes a jump.

Some Notable Jcc Instructions

- Mnemonic translations
- A = above, unsigned notion
- B = below, unsigned notion
- G = greater than, signed notion
- L = less than, signed notion
- E = Equal (same a Z, zero flag set)
- N = Not (like “Not less than:” JNL)

Some Notable Jcc Instructions

- Mnemonic translations
- **B** = below, unsigned notion
- **A** = above, unsigned notion
- **N** = Not (like “Not less than:” JNL)
- **G** = greater than, signed notion
- **L** = less than, signed notion
- **E** = Equal (same a Z, zero flag set)

Final Fantasy 7
Crystal Bangle
Unlocked!

Flag setting

- Before you can do a conditional jump, you need something to set the condition flags for you.
- Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag-setting side-effects

CMP - Compare Two Operands

- "The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction."
- What's the difference from just doing SUB? Difference is that with SUB the result has to be stored somewhere. With CMP the result is computed, the flags are set, but the result is discarded. Thus this only sets flags and doesn't mess up any of your registers.
- Modifies CF, OF, SF, ZF, AF, and PF
- (as we already saw, SUB modifies all those too)

Book page 93

IfExample.c takeaways

- Conditional logic, like if statements, manifests in assembly as conditional jumps (Jcc). “If condition true, jump there, else fall through”
- Conditions involving in/equality are often checked with a CMP instruction, which is the same thing as a SUB, but it just throws the results away after the relevant RFLAGS bits are set
- The RFLAGS bits are fundamentally what are checked by the Jccs

```
int main(){
        int a=1, b=2;
        if(a == b){
                return 1;
        }
        if(a > b){
                return 2;
        }
        if(a < b){
                return 3;
        }
        return 0xdefea7;
}
```

```
main:
0000000140001010 sub      rsp,18h
0000000140001014 mov      dword ptr [rsp+4],1
000000014000101C mov      dword ptr [rsp],2
0000000140001023 mov      eax,dword ptr [rsp]
0000000140001026 cmp      dword ptr [rsp+4],eax
000000014000102A jne      0000000140001033
000000014000102C mov      eax,1
0000000140001031 jmp      0000000140001058
0000000140001033 mov      eax,dword ptr [rsp]
0000000140001036 cmp      dword ptr [rsp+4],eax
000000014000103A jle      0000000140001043
000000014000103C mov      eax,2
0000000140001041 jmp      0000000140001058
0000000140001043 mov      eax,dword ptr [rsp]
0000000140001046 cmp      dword ptr [rsp+4],eax
000000014000104A jge      0000000140001053
000000014000104C mov      eax,3
0000000140001051 jmp      0000000140001058
0000000140001053 mov      eax,0DEFEA7h
0000000140001058 add      rsp,18h
000000014000105C ret
```

BitmaskExample.c

```
#define MASK 0x100

int main(){
        int a=0x1301;
        if(a & MASK){
                return 1;
        }
        else{
                return 2;
        }
}
```

```
main:
0000000140001010  sub       rsp,18h
0000000140001014  mov       dword ptr [rsp],1301h
000000014000101B  mov       eax,dword ptr [rsp]
000000014000101E  and       eax,100h
0000000140001023  test      eax,eax
0000000140001025  je        0000000140001030
0000000140001027  mov       eax,1
000000014000102C  jmp       0000000140001035
000000014000102E  jmp       0000000140001035
0000000140001030  mov       eax,2
0000000140001035  add       rsp,18h
0000000140001039  ret
```

TEST - Logical Compare

- “Computes the bit-wise AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result.”
- Like CMP - sets flags, and throws away the result

BitmaskExample.c takeaways

- Conditions depending on bit tests (which is often expressed with boolean logic instructions) will often see the RFLAGS set with the CMP instruction. CMP is like AND, but throws the results away
- The reason for the extraneous jmp here is because it's unoptimized code so it's following a simpler set of asm construction rules

```
#define MASK 0x100

int main(){
        int a=0x1301;
        if(a & MASK){
                return 1;
        }
        else{
                return 2;
        }
}
```

```
main:
0000000140001010  sub       rsp,18h
0000000140001014  mov       dword ptr [rsp],1301h
000000014000101B  mov       eax,dword ptr [rsp]
000000014000101E  and       eax,100h
0000000140001023  test      eax,eax
0000000140001025  je        0000000140001030
0000000140001027  mov       eax,1
000000014000102C  jmp       0000000140001035
000000014000102E  jmp       0000000140001035
0000000140001030  mov       eax,2
0000000140001035  add       rsp,18h
0000000140001039  ret
```

Instructions we now know (17)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB
- IMUL
- MOVZX/MOVSX
- LEA
- JMP/Jcc (family)
- CMP/TEST