University of BRISTOL

COMS20012: Software Security and Assembly Programming

Joseph Hallett

Next week…

- Next week we're going to cover classical binary exploit techniques
  - (i.e. proper computer hacking)

- To ~~understand~~ *be comfortable with* these techniques you need to know how to ~~program in~~ *read and not be scared by* assembly code

- Assembly programming is hard and confusing
- The x86 ISA is ridiculous and confusing
  - The more you do with it the more comfortable you'll be
  - Even if you're not interested in security, knowing a bit will help you debug your programs throughout your career

This week…

- Overview of software security!

- Introduction to x86 assembly and memory layout

Software Security

- Software has bugs

- You can have the best cryptography in the world…
  - …but if your theoretically secure algorithm is implemented insecurely…

- *"Security is a chain; it's only as secure as the weakest link."*
  - Bruce Schneier

Software has stages

- Code has *syntax* (grammar) and *semantics* (meaning)
- Which gets written by a human
- Which gets compiled into machine code by a compiler
- Which gets linked with other machine code
- Which gets executed by a CPU

- Any (and all) of these stages can have bugs
  – Sometimes these bugs can lead to security properties being violated…

WYSINWYX

Compiler bugs can be extremely tricksy!

```
memset(pass,0,len);
free(pass);
```

Is the compiler free to optimize away the call to `memset`?

Can you assume the compiler will always *do the right thing?*

WYSINWYX:
What You See Is Not What You eXecute

GOGUL BALAKRISHNAN
NEC Laboratories America, Inc.
and
THOMAS REPS
University of Wisconsin and GrammaTech, Inc.

Over the last seven years, we have developed static-analysis methods to recover a good approximation to the variables and dynamically-allocated memory objects of a stripped executable, and to track the flow of values through them. The paper presents the algorithms that we developed, explains how they are used to recover intermediate representations (IRs) from executables that are similar to the IRs that would be available if one started from source code, and describes their application in the context of program understanding and automated bug hunting.

Unlike algorithms for analyzing executables that existed prior to our work, the ones presented in this paper provide useful information about memory accesses, even in the absence of debugging information. The ideas described in the paper are incorporated in a tool for analyzing Intel x86 executables, called CodeSurfer/x86. CodeSurfer/x86 builds a system dependence graph for the program, and provides a GUI for exploring the graph by (i) navigating its edges, and (ii) invoking operations, such as forward slicing, backward slicing, and chopping, to discover how parts of the program can impact other parts.

To assess the usefulness of the IRs recovered by CodeSurfer/x86 in the context of automated bug hunting, we built a tool on top of CodeSurfer/x86, called Device-Driver Analyzer for x86 (DDA/x86), which analyzes device-driver executables for bugs. Without the benefit of either source code or symbol-table/debugging information, DDA/x86 was able to find known bugs (that had been discovered previously by source-code-analysis tools), along with useful error traces, while having a low false-positive rate. DDA/x86 is the first known application of program analysis/verification techniques to industrial executables.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verifi-

Interesting software security bugs…

- Integer overflow and underflow
  - What happens when a number gets too big or too small?
- Buffer overflow
  - What happens when you go beyond the bounds of a data structure?
- Control flow corruption
  - What happens if an attacker can corrupt the structures used to interpret the machine code?

We will talk more about these next week!

University of BRISTOL

Memory Layout

…or: more than you ever wanted to know about the X86 ISA and binary executables

bristol.ac.uk

What is a binary program?

Or what does a compiler actually produce?

- Varies by operating system… but some generalities
  - Linux/BSD: ELF file
  - Windows: PE file
  - MacOS: Mach-O file

- Contains information on how to link and load the program
- Contains code and data needed for the program
  - This gets copied into the process's memory by the OS/Link-loader
  - Instruction pointer gets pointed at a function (usually `_start`)

Sections in a program's memory

- Executable code (.text segment)
- Initialised data (.data segment)
- Uninitialised data (.bss segment)

- The heap (dynamic memory; i.e. stuff you malloc)
- The stack (local memory; i.e. function variables)

- Shared libraries and their data (.got/.plt tables)

```
[joseph@fedora ~]$ r2 $(command -v cat) <<<iS=
[0x000037a0]> iS=

0*  0x0        ------------------------------ 0x0        ---   0
1   0x318      █----------------------------- 0x334      r--   28    .interp
2   0x338      █----------------------------- 0x388      r--   80    .note.gnu.property
3   0x388      █----------------------------- 0x3ac      r--   36    .note.gnu.build-id
4   0x3ac      █----------------------------- 0x3cc      r--   32    .note.ABI-tag
5   0x3d0      █----------------------------- 0x3ec      r--   28    .gnu.hash
6   0x3f0      ███--------------------------- 0x9d8      r--   1.5K  .dynsym
7   0x9d8      --█--------------------------- 0xcaf      r--   727   .dynstr
8   0xcb0      --█--------------------------- 0xd2e      r--   126   .gnu.version
9   0xd30      --█--------------------------- 0xda0      r--   112   .gnu.version_r
10  0xda0      --██-------------------------- 0x1010     r--   624   .rela.dyn
11  0x1010     ---██------------------------- 0x14d8     r--   1.2K  .rela.plt
12  0x2000     -------█---------------------- 0x201b     r-x   27    .init
13  0x2020     -------█---------------------- 0x2360     r-x   832   .plt
14  0x2360     -------██--------------------- 0x2690     r-x   816   .plt.sec
15* 0x2690     --------█████████████--------- 0x5c32     r-x   13.4K .text
16  0x5c34     --------------------█--------- 0x5c41     r-x   13    .fini
17  0x6000     ---------------------████----- 0x6daf     r--   3.4K  .rodata
18  0x6db0     ------------------------█----- 0x6e64     r--   180   .eh_frame_hdr
19  0x6e68     ------------------------██---- 0x72c0     r--   1.1K  .eh_frame
20  0x8a90     --------------------------█--- 0x8a98     rw-   8     .init_array
21  0x8a98     --------------------------█--- 0x8aa0     rw-   8     .fini_array
22  0x8aa0     --------------------------██-- 0x8c00     rw-   352   .data.rel.ro
23  0x8c00     ---------------------------█-- 0x8df0     rw-   496   .dynamic
24  0x8df0     ---------------------------██- 0x8ff0     rw-   512   .got
25  0x9000     ----------------------------█- 0x9068     rw-   104   .data
26  0x9080     ----------------------------█- 0x91c0     rw-   0     .bss
27  0xb1c0     ----------------------------█- 0xb428     ---   616   .gnu.build.a
28  0x0        ----------------------------█- 0x20       ---   32    .gnu_debugli
29  0x0        ----------------------------██ 0x454      ---   1.1K  .gnu_debugda
30  0x0        -----------------------------█ 0x13e      ---   318   .shstrtab
=>  0x000037a0 ------------------------------ 0x0000379f
```

bristol.ac.uk

Once loaded into memory…

- Heap goes above the preloaded section and grows towards the top of virtual memory space

- Program arguments/environment variables go at the top of memory
- Stack beneath it growing down towards the heap

X86 Instruction Set Architecture

- First developed for the Intel 8086 chip back in 1976
- Dominant PC ISA today
- **Lots** of different instructions
- 16/32/64-bit modes
- Several different registers (each with multiple addressing modes)
  - AH/AL Register A high 8 bits, low 8 bits
  - AX 16 bit Register A
  - EAX 32 bit Register A (AX is low 16 bits) (E for extended!)
  - RAX 64 bit Register A (EAX is low 32 bits) (R for register/really extended?!)

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325462-075US
June 2021

There is a manual!

- It is available online, print copies can be bought too
- It is a mere 4778 pages
- It is *mostly somewhat* right

Assembly Programming

Bad news…

- No variables or functions
- No control flow or loops (well… *kinda*)
- Infamously difficult to get right
- Sometimes written just as big lists of hex values!

Instead…

- Registers for holding values
- Pointers in registers for referring to memory (with offsets)
- Pointers to instructions instead of functions
- Pointers to non-writable memory instead of constants
- Randomly named instructions for doing stuff

Good bit!

You don’t really need to be able to write it *(well a little bit will help…)*

Reading assembly is really useful

- Sooner or later you’ll have to debug something or reverse engineer some weird system

Some stuff *only* works via assembly ;-)

GNU/AT&T vs Intel Assembly

There are two big syntaxes for writing/displaying X86 assembly code… both are confusingly different!

- People have strong opinions about which is better…

- Intel: (Destination first, no prefix/suffixes)
  - `add eax, 0xa`
  - `mov [rbp-3], 0x1234`

- GNU/AT&T: (Destination last, instruction takes a length suffix, registers marked with %, values marked with $)
  - `addl $0xa, %eax`
  - `movl $0x1234, -3(%rbp)`

(I'll try and be consistent but no promises ;-))

X86 Assembly (32 bit)

- 6 general purpose registers:
  - `eax, ebx, ecd, edx, esi, edi`
- 2 special purpose registers (for making the stack)
  - `esp, ebp`
- 1 instruction pointer (says what to do next)
  - `eip`

- Extensions for doing extra stuff
  - Parallelism, floating point, cryptography, security…

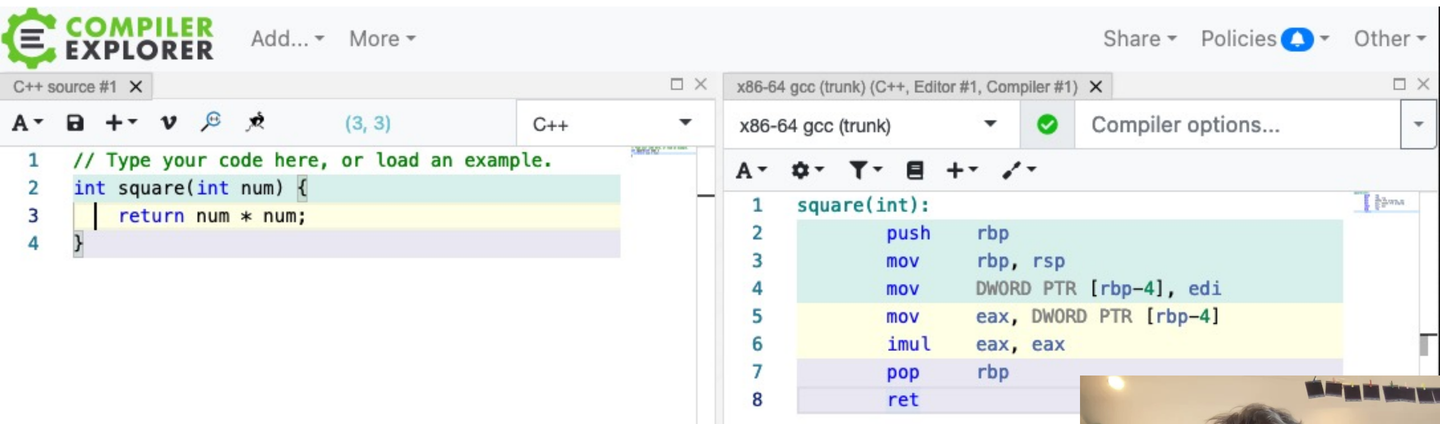
X86 Assembly (64 bit)

- 14 general purpose registers:
  - `rax, rbx, rcd, rdx, rsi, rdi`
  - `r8, r9, r10, r11, r12, r13, r14, r15`
- 2 special purpose registers (for making the stack)
  - `rsp, rbp`
- 1 instruction pointer (says what to do next)
  - `rip`

- Extensions for doing extra stuff
  - Parallelism, floating point, cryptography, security…

Functions (and https://godbolt.org)

- In C we have *functions* that we call to make our programs
- In Assembly we have *procedures*



So how do you call a function?

- Put the arguments in registers and/or on the stack…
- Save where you were before you called the function on the stack…
  - ret is equivalent to rip = *(rsp--);
- Set the Instruction Pointer to the start of the procedure

But which arguments go where?

Calling conventions (32 bit)

- cdecl
  - Arguments go on the stack; caller cleans up the stack
- stdcall
  - Arguments go on the stack; callee cleans up the stack
- fastcall
  - Arguments go in EAX, EDX, ECX; and then the rest on the stack
- thiscall
  - Pointer to class object in ECX; and then arguments on the stack

Calling conventions (64 bit)

- Microsoft x64 calling
  - RCX, RDX, R8 and R9 for the first four integer or pointer arguments
  - XMM0…XMM3 are for floating point arguments
  - Additional arguments are pushed on the stack *in reverse order*

- X86-64 calling convention
  - RDI, RSI, RDX, RCX, R8, R9 are used for the first six integer or pointer arguments
  - XMM0…XMM7 are for *some* floating point arguments
  - Additional arguments are pushed on the stack
  - Return value in RAX and RDX

Calling conventions (caveats)

These are conventions not rules!

You don't have to follow them

- But in practice if you want to link against a library you need to ensure you follow the same conventions

Sometimes other mechanisms get used…

- E.g. System calls on Linux put the syscall number in RAX and then arguments in RBX, RCX…

So how does a call work?

```c
#include <stdio.h>

int square(int num) {
    return num * num;
}

int main(void) {
    int n = 7;
    n = square(n);
    return n;
}
```

```asm
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 7
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
```

So how does a call work? (Calling square from main)

```
rsp ->
           | main+16 |
           | …       |
rbp ->     |         |
```

- BP holds the bottom of the previous function’s stack
- SP holds the current position in the stack
- main+14-15: set up args
- main+16: call square
  Push current IP onto stack and set IP to address of square

```
1  square:
2          push    rbp
3          mov     rbp, rsp
4          mov     DWORD PTR [rbp-4], edi
5          mov     eax, DWORD PTR [rbp-4]
6          imul    eax, eax
7          pop     rbp
8          ret
9  main:
10         push    rbp
11         mov     rbp, rsp
12         sub     rsp, 16
13         mov     DWORD PTR [rbp-4], 7
14         mov     eax, DWORD PTR [rbp-4]
15         mov     edi, eax
16         call    square
17         mov     DWORD PTR [rbp-4], eax
18         mov
19         leave
20         ret
```

So how does a call work? (In square)

```
rbp-4 ->     | num     |
rbp,rsp ->   | oldrbp  |
             | main+16 |
             | …       |
oldrbp ->    |         |
```

- square+2: save the old stack frame's base on the stack.
- square+3: move the current stack pointer onto the base pointer creating a new stack frame!
- square+4: store the function's argument on the stack—effectively a variable. (This will *probably* get optimized away later;
- square+6: finish up with the function: the result is in RAX.

```
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 7
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        mov     DWORD PTR [rbp-4], eax
        mov
        leave
        ret
```

So how does a call work (Returning)

- square+7: We're done. RAX has the result so we return. Lets restore the old stack frame
- square+8: Return! Pop SP into IP and increment. Execution goes to main+17

```
                | num      |
                | oldrbp   |
                | main+16  |
rsp ->          | …        |
rbp ->          |          |
```

```
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 7
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        mov     DWORD PTR [rbp-4], eax
        mov
        leave
        ret
```

Assembly is confusing!

- I really want to emphasize that assembly programming is hard!
- Whilst I do expect you to be able to stare at it and figure out/guess what it is doing… I don't expect you to be able to write it straight off

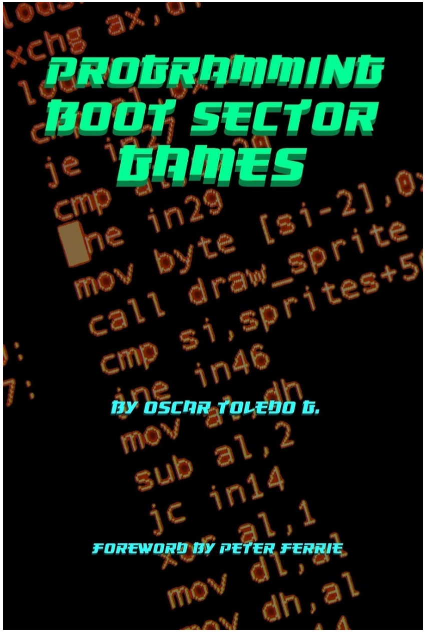
- That said it is *a really useful* skill

Learning assembly

xchg rax, rax

- Fun meditative puzzles to figure out what a small sequence of X86 assembly does

- Book available or online!
- https://www.xorpd.net/pages/xchg_rax/snip_00.html



Next week…

- Classical hacking for beginners ;-)
- http://phrack.org/issues/49/14.html