Applied Software Security

Tutorial and Practice
Week 1

What?

- Digital devices are complicated
- Software is complicated
- More complexity → More bugs
- Unexpected behavior + malicious actors = trouble
- Complicated enough to warrant an entire discipline

- Knowledge is power

How?

Disciplines and knowledge:

- Coding, assembly, programming languages
- Computers and operating systems
- Networking and hardware interfaces
- Debugging (as a discipline)
- Code auditing
- Cryptography
- And more...

How?

- Reverse engineering* / “white-box” analysis
  - High-level code, decompiled and disassembled code
- Binary analysis
  - Files, formats, protocols, filesystems, DBs, data structures
- Dynamic analysis
  - Debugging†, “black-box” analysis, testing and automation
- Network and system analysis
  - Packets, logs, messages, etc.

* Technically everything here falls under “reverse engineering”, but in our world the term is used mostly for white-box analysis, sometimes even in a stricter sense of binary code analysis

† Here referring to monitored execution of a program using a *debugger*, and **not** the general idea of finding bugs

Why?

- Defense
  - Know your enemy
  - Analyze malicious code
  - Assess vulnerabilities
  - Get there before the bad guys
- Offense
  - Can be done for good deeds
  - Law enforcement, etc.
- Malicious reasons
  - Don’t even think about it
  - Malware/cracking/illegal espionage

Responsibility

With great power comes great responsibility

Assembly Recap

- We will be working with 32-bit and 64-bit x86 assembly (most commonly encountered today)

What does the following code do? (32-bit)

- input is in eax and output is in eax

```
    mov     edx, eax
    mov     bl, [eax]
    test    bl, bl
    jz      loop_end
loop_start:
    inc     eax
    mov     bl, [eax]
    test    bl, bl
    jnz     loop_start
loop_end:
    sub     eax, edx
```

What does the following code do? (32-bit)

- input is in `eax` and output is in `eax`

```
    mov     edx, eax      ; original input is now in edx
    mov     bl, [eax]     ; load byte at eax to bl
    test    bl, bl        ; check if bl is zero
    jz      loop_end      ; if true, go to end
loop_start:
    inc     eax           ; eax++
    mov     bl, [eax]     ; load byte at eax to bl
    test    bl, bl        ; check if bl is zero
    jnz     loop_start    ; repeat until bl=0
loop_end:
    sub     eax, edx      ; return diff between eax and original input
```

- Answer: finds the index of first zero byte (`strlen`)

What does the following code do? (32-bit)

– input is in `eax` and output is in `eax`

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx
mov     eax, ecx
```

What does the following code do? (32-bit)

- input is in `eax` and output is in `eax`

```
mov     edi, eax        ; initialize edi with input
xor     eax, eax        ; zero eax
xor     ecx, ecx        ; zero ecx
not     ecx             ; ecx = 0xFFFFFFFF
cld                     ; ???
repne scasb             ; ???
inc     ecx             ; what?
not     ecx             ; why?
mov     eax, ecx        ; move result from ecx to eax
```

What do `cld` and `repne scasb` do?

- We know they do something with `edi`, `eax` and `ecx`
- Time to check the manual...

CLD

- From Intel's official instruction set manual

CLD—Clear Direction Flag

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

```
DF := 0;
```

Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

REPNE

– From Intel's official instruction set manual

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.)

Table 4-22. Repeat Prefixes

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

NOTES:

* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Operation

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
    Implicit Source/Dest operand for memory use of SI/DI;
  ELSE IF AddressSize = 64
    THEN Use RCX for CountReg;
    Implicit Source/Dest operand for memory use of RSI/RDI;
  ELSE
    Use ECX for CountReg;
    Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
   DO
       Service pending interrupts (if any);
       Execute associated string instruction;
       CountReg := (CountReg – 1);
       IF CountReg = 0
           THEN exit WHILE loop; FI;
       IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
       or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
           THEN exit WHILE loop; FI;
   OD;
```

SCASB

– From Intel's official instruction set manual

SCAS/SCASB/SCASW/SCASD—Scan String

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of status flags. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Operation

Non-64-bit Mode:

```
IF (Byte comparison)
    THEN
        temp := AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (E)DI := (E)DI + 1;
                ELSE (E)DI := (E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp := AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI := (E)DI + 2;
                ELSE (E)DI := (E)DI – 2; FI;
        FI;
    ELSE IF (Doubleword comparison)
        THEN
            temp := EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI := (E)DI + 4;
                ELSE (E)DI := (E)DI – 4; FI;
        FI;
FI;
```

tl;dr:

- `cld` – Clear Direction Flag (`DF`)
  - Part of the `FLAGS` register
- `repne` – Repeat String Operation Prefix
  - Repeats a string instruction (`scasb` in our code) the number of times specified in `ecx`
  - Decrements `ecx` with each iteration
  - Stops when `ecx=0` or until `ZF=1`
- `scasb` – Scan String (Byte)
  - Compares a byte pointed to by `edi` to the value of `al` and sets flags accordingly (OF, SF, ZF, AF, PF, CF)
  - Increments `edi` by one if `DF=0`, decrements by one if `DF=1`
- Analogous instructions exist for 64-bit addressing (using `rdi`, `rcx`), and to compare words (`ax`), dwords (`eax`) and qwords (`rax`) (the latter in 64-bit mode only)

What does the following code do? (32-bit)

- input is in `eax` and output is in `eax`

```
mov     edi, eax        ; initialize edi with input
xor     eax, eax        ; zero eax
xor     ecx, ecx        ; zero ecx
not     ecx             ; ecx = 0xFFFFFFFF
cld                     ; DF=0
repne scasb             ; repeat compare to al=0
inc     ecx             ; ecx++
not     ecx             ; not ecx
mov     eax, ecx        ; move result from ecx to eax
```

What happens to `ecx`?

- Before `repne scasb` we have `ecx = 0xFFFFFFFF = (-1)`
- After `repne scasb`, we have `ecx = -1 - index - 1`
  - The extra `(-1)` is because `ecx` is decremented once more when the zero byte is found
- After `inc ecx` we have `ecx = -(index + 1)`
- Remember 2’s complement means applying `not` then incrementing by one to get a sign change
  - If we `not ecx` but don’t increment by one, we get the number immediately preceding `(index + 1)`
- So after `not ecx` we have `ecx = index`

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

eip

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx
mov     eax, ecx
...
```

Registers/Flags

```
eax = 0x00010000
edi = 0x????????
ecx = 0x????????

  DF: ?   ZF: ?
```

Memory

eax

```
10000: 4A
10001: 00
10002: 00
10003: 00
 ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
     mov     edi, eax
eip  xor     eax, eax
     xor     ecx, ecx
     not     ecx
     cld
     repne scasb
     inc     ecx
     not     ecx
     mov     eax, ecx
     ...
```

Registers/Flags

```
eax = 0x00010000
edi = 0x00010000
ecx = 0x????????

  DF: ?   ZF: ?
```

Memory

```
eax, edi  10000: 4A
          10001: 00
          10002: 00
          10003: 00
            ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx
mov     eax, ecx
...
```

eip

Registers/Flags

```
eax = 0x00000000
edi = 0x00010000
ecx = 0x????????

  DF: ?   ZF: 1
```

Memory

edi

```
10000: 4A
10001: 00
10002: 00
10003: 00
 ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

eip →

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx
mov     eax, ecx
...
```

Registers/Flags

```
eax = 0x00000000
edi = 0x00010000
ecx = 0x00000000

  DF: ?   ZF: 1
```

Memory

edi →

```
10000: 4A
10001: 00
10002: 00
10003: 00
 ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

eip

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx
mov     eax, ecx
...
```

Registers/Flags

```
eax = 0x00000000
edi = 0x00010000
ecx = 0xffffffff

  DF: ?   ZF: 1
```

Memory

edi

```
10000: 4A
10001: 00
10002: 00
10003: 00
  ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

eip →

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx
mov     eax, ecx
...
```

Registers/Flags

al

```
eax = 0x00000000
edi = 0x00010000
ecx = 0xffffffff

  DF: 0   ZF: 1
```

Memory

edi →

```
10000: 4A
10001: 00
10002: 00
10003: 00
  ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
      mov    edi, eax
      xor    eax, eax
      xor    ecx, ecx
      not    ecx
      cld
eip → repne scasb
      inc    ecx
      not    ecx
      mov    eax, ecx
      ...
```

Registers/Flags

```
                    al
eax = 0x00000000
edi = 0x00010001
ecx = 0xfffffffe

  DF: 0    ZF: 0
```

Memory

```
edi → 10001

10000: 4A
10001: 00
10002: 00
10003: 00
  ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
mov    edi, eax
xor    eax, eax
xor    ecx, ecx
not    ecx
cld
repne scasb
inc    ecx
not    ecx
mov    eax, ecx
...
```

eip

Registers/Flags

```
                 al
eax = 0x00000000
edi = 0x00010002
ecx = 0xfffffffd

  DF: 0   ZF: 0
```

edi

Memory

```
10000: 4A
10001: 00
10002: 00
10003: 00
  ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
mov     edi, eax
xor     eax, eax
xor     ecx, ecx
not     ecx
cld
repne scasb
inc     ecx
not     ecx      <- eip
mov     eax, ecx
...
```

Registers/Flags

```
eax = 0x00000000
edi = 0x00010002
ecx = 0xfffffffe

  DF: 0    ZF: 0
```

Memory

```
10000: 4A
10001: 00
10002: 00   <- edi
10003: 00
  ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
      mov    edi, eax
      xor    eax, eax
      xor    ecx, ecx
      not    ecx
      cld
      repne scasb
      inc    ecx
      not    ecx
eip   mov    eax, ecx
      ...
```

Registers/Flags

```
eax = 0x00000000
edi = 0x00010002
ecx = 0x00000001

  DF: 0    ZF: 1
```

edi

Memory

```
10000: 4A
10001: 00
10002: 00
10003: 00
  ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
mov    edi, eax
xor    eax, eax
xor    ecx, ecx
not    ecx
cld
repne scasb
inc    ecx
not    ecx
mov    eax, ecx
...
```

eip

Registers/Flags

```
eax = 0x00000001
edi = 0x00010002
ecx = 0x00000001

  DF: 0   ZF: 1
```

edi

Memory

```
10000: 4A
10001: 00
10002: 00
10003: 00
 ... : ...
```

Assembly

So, what does the code do?

Answer: finds the index of the first zero byte in a string. In other words: `strlen`, again!

It is often useful to check simple cases, for example, a string of length 1 (2 bytes including the null terminator)

```
mov     edi, eax        ;
xor     eax, eax        ;
xor     ecx, ecx        ; ecx = 0
not     ecx             ; ecx = 0xFFFFFFFF = -1
cld                     ;
repne scasb             ; ecx-- ; (repeat once, hece ecx = -2)
inc     ecx             ; ecx = -1 = 0xFFFFFFFF
not     ecx             ; ecx = 0
mov     eax, ecx        ; result = 0
```

- Question: why did we have to start with `ecx = 0xFFFFFFFF`? What happens if we start with `ecx=0`?

Disclaimer

Actual implementations of `strlen` are different. Take the GNU libc implementation for example: (C code; comments omitted)

```c
size_t
STRLEN (const char *str)
{
  const char *char_ptr;
  const unsigned long int *longword_ptr;
  unsigned long int longword, himagic, lomagic;
  for (char_ptr = str; ((unsigned long int) char_ptr
                        & (sizeof (longword) - 1)) != 0;
       ++char_ptr)
    if (*char_ptr == '\0')
      return char_ptr - str;
  longword_ptr = (unsigned long int *) char_ptr;
  himagic = 0x80808080L;
  lomagic = 0x01010101L;
  if (sizeof (longword) > 4)
    {
      himagic = ((himagic << 16) << 16) | himagic;
      lomagic = ((lomagic << 16) << 16) | lomagic;
    }
  if (sizeof (longword) > 8)
    abort ();
  for (;;)
    {
      longword = *longword_ptr++;
      if (((longword - lomagic) & ~longword & himagic) != 0)
        {
          const char *cp = (const char *) (longword_ptr - 1);
          if (cp[0] == 0)
            return cp - str;
          if (cp[1] == 0)
            return cp - str + 1;
          if (cp[2] == 0)
            return cp - str + 2;
          if (cp[3] == 0)
            return cp - str + 3;
          if (sizeof (longword) > 4)
            {
              if (cp[4] == 0)
                return cp - str + 4;
              if (cp[5] == 0)
                return cp - str + 5;
              if (cp[6] == 0)
                return cp - str + 6;
              if (cp[7] == 0)
                return cp - str + 7;
            }
        }
    }
}
```

This will compile to significantly more instructions, but will be much more run-time efficient (looks at 4/8 bytes at a time, instead of just one)

Disclaimer

Usually an architecture-specific implementation overrides the generic c code for `strlen` and other useful functions.

- On modern x86 PCs for example, GNU libc will use instructions and registers that are part of the SSE2 (Streaming SIMD Extensions 2) supplementary instruction set, or later extensions.

In general, simple tasks often become much more complicated, from both the programmer’s and reverse engineer’s perspective

- To make things worse for RE, some decompilers don’t handle architecture-specific implementations well
- Sometimes it’s quicker to use other methods rather than following the code instruction by instruction

Assembly

What does the following code do?

- Input is in eax and output is in eax
- Warning: there may be a bug here

```
      mov     ecx, eax
      xor     ebx, ebx
      mov     eax, 1

  _loop_start:
      add     eax, ebx    ;eax = eax + ebx  | a’ = a + b
      sub     ebx, eax    ;ebx = ebx – eax  | b’ = b – a’ = -a
      neg     ebx         ;ebx = -ebx       | b’’ = -b’ = a
      loop _loop_start
```

Assembly

Answer: computes the n-th Fibonacci number

n is the input (ecx) and the result is in eax

Input (ecx)	1	2	3	4	5	6	7	8	9
Output (eax)	1	2	3	5	8	13	21	34	55

*note: the first two Fibonacci numbers are sometimes taken to be [0, 1] or [1, 1]

Questions:

- What happens when the input (ecx) is 0?
- Why in such a roundabout way (sub, neg)?

You may encounter weird ways of computing stuff

Usually, though, the computation will be straightforward

Assembly

A more straightforward approach for the same code will be:

```
    mov     ecx, eax
    xor     ebx, ebx
    mov     eax, 1

_loop_start:
    mov     edx, eax
    add     eax, ebx
    mov     ebx, edx
    loop _loop_start
```

What would you do to compute the following values for the given inputs instead?

Input	0	1	2	3	4	5	6	7	8	9	10	11	12
Output	0	1	1	2	3	5	8	13	21	34	55	89	144

Try it yourself! (you will probably need more instructions)

Assembler Practical Info

In this course we will be using nasm

- https://www.nasm.us/

Example files will be provided for assembly with a C function wrapper and for assembly to naked binaries

Functions and the stack

The basic concept of a function or subroutine has to translate to machine code somehow

In C a function can accept a fixed number of parameters (0 or more), or even a variable number of parameters

It may also return some value (but only one)

After a function is done, execution returns seamlessly to the code that called the function

Functions can be nested and even recursive

An elegant solution to accomplish all of this is the program stack

The stack, in general terms (reminder):

- An area of memory with a “top” pointer
- Each function uses its own “stack frame”
- Saves local variables, some function parameters, and the code return address when calling a function.
- Fulfills all the requirements of the functional programming abstraction

Calling conventions

Let’s start with 32-bit x86:

- Historically fewer general-purpose registers than x86_64
- Modern x86 needs to be backwards-compatible to code written for early processors
- Therefore, by default (using `stdcall`), all the params are passed on the stack, and all registers except `eax` are saved by the callee
- `eax` holds the return value
- The return address is the last thing pushed onto the stack, when the `call` instruction is run (this is part of the instruction)
- Other conventions exits (e.g. fastcall, which uses general purpose registers for params)
- We glossed over some details here, e.g. how structures are passed and returned, what happens with floating-point numbers, OOP (C++) member functions (methods), etc.

In x86-64 there is no shortage of general purpose registers, so all popular conventions use the registers for the first arguments

- There still is a difference in which and how many registers are used
- Practical info can be found on Wikipedia:

  https://en.wikipedia.org/wiki/X86_calling_conventions

- Let’s look at some examples of functions and how they compile to different architectures and conventions (all will be using GCC with POSIX conventions)
- We’ll show the C code together with disassembly of the compiled machine code

First, the difference between a returning and a non-returning function: (64-bit code)

`void no_args()`
`{`
`   return;`
`}`
	`401126: 55`
	`401127: 48 89 e5`
	`40112a: 90`
	`40112b: 5d`
	`40112c: c3`
	`40112d: _`
		`push   rbp`
		`mov    rbp,rsp`
		`nop`
		`pop    rbp`
		`ret`

`int no_args2()`
`{`
`   return 1;`
`}`
	`40112d: 55`
	`40112e: 48 89 e5`
	`401131: b8 01 00 00 00`
	`401136: 5d`
	`401137: c3`
	`401138: _`
		`push   rbp`
		`mov    rbp,rsp`
		`mov    eax,0x1`
		`pop    rbp`
		`ret`

`…`
`no_args();`
`x = no_args2();`
`…`
	`401343: b8 00 00 00 00`
	`401348: e8 d9 fd ff ff`
	`40134d: b8 00 00 00 00`
	`401352: e8 93 fe ff ff`
	`401357: 89 45 fc`
	`40135a: _`
		`mov    eax,0x0`
		`call   401126 <no_args>`
		`mov    eax,0x0`
		`call   40112d <no_args2>`
		`mov    DWORD PTR [rbp-0x4],eax`

Note that eax is restored after the first function, even though it doesn’t change eax

The 32-bit code is analogous so we won’t show it here

Now let's look at a function with three arguments in 64-bit code:

`int sum_of_3(`
`    int a, int b, int c)`
`{`
`     return a+b+c;`
`}`
	`401138: 55`
	`401139: 48 89 e5`
	`40113c: 89 7d fc`
	`40113f: 89 75 f8`
	`401142: 89 55 f4`
	`401145: 8b 55 fc`
	`401148: 8b 45 f8`
	`40114b: 01 c2`
	`40114d: 8b 45 f4`
	`401150: 01 d0`
	`401152: 5d`
	`401153: c3`
	`401154: _`
		`push   rbp`
		`mov    rbp,rsp`
		`mov    DWORD PTR [rbp-0x4],edi`
		`mov    DWORD PTR [rbp-0x8],esi`
		`mov    DWORD PTR [rbp-0xc],edx`
		`mov    edx,DWORD PTR [rbp-0x4]`
		`mov    eax,DWORD PTR [rbp-0x8]`
		`add    edx,eax`
		`mov    eax,DWORD PTR [rbp-0xc]`
		`add    eax,edx`
		`pop    rbp`
		`ret`

`…`
`y = sum_of_3(x, 7, 945);`
`…`
	`40135a: 8b 45 fc`
	`40135d: ba b1 03 00 00`
	`401362: be 07 00 00 00`
	`401367: 89 c7`
	`401369: e8 ca fd ff ff`
	`40136e: 89 45 f8`
	`401371: _`
		`mov    eax,DWORD PTR [rbp-0x4]`
		`mov    edx,0x3b1`
		`mov    esi,0x7`
		`mov    edi,eax`
		`call   401138 <sum_of_3>`
		`mov    DWORD PTR [rbp-0x8],eax`

The parameters are passed using edi, esi, and edx. Note that these are 32-bit registers since int is 32 bits in this implementation of a 64-bit C compiler (and it usually is).

Also note that without optimization, the compiler stores the arguments on stack and then immediately retrieves them. While storing them on stack is redundant, it's useful for more complex functions.

Question: Why can we write to addresses less than `rbp` when `rbp=rsp`? (that is, we haven't decremented `rsp`)

And the same function compiled as 32-bit code:

`int sum_of_3(`
`    int a, int b, int c)`
`{`
`     return a+b+c;`
`}`
	`8049186: 55`
	`8049187: 89 e5`
	`8049189: 8b 55 08`
	`804918c: 8b 45 0c`
	`804918f: 01 c2`
	`8049191: 8b 45 10`
	`8049194: 01 d0`
	`8049196: 5d`
	`8049197: c3`
	`8049198: _`
		`push   ebp`
		`mov    ebp,esp`
		`mov    edx,DWORD PTR [ebp+0x8]`
		`mov    eax,DWORD PTR [ebp+0xc]`
		`add    edx,eax`
		`mov    eax,DWORD PTR [ebp+0x10]`
		`add    eax,edx`
		`pop    ebp`
		`ret`

`…`
`y = sum_of_3(x, 7, 945);`
`…`
	`80492a7: 68 b1 03 00 00`
	`80492ac: 6a 07`
	`80492ae: ff 75 f4`
	`80492b1: e8 d0 fe ff ff`
	`80492b6: 83 c4 0c`
	`80492b9: 89 45 f0`
	`80492bc: _`
		`push   0x3b1`
		`push   0x7`
		`push   DWORD PTR [ebp-0xc]`
		`call   8049186 <sum_of_3>`
		`add    esp,0xc`
		`mov    DWORD PTR [ebp-0x10],eax`

Even though edi, esi, and edx all exist in the 32-bit architecture, here we see that the default way to pass parameters is on the stack. Also note that the parameters are pushed in reverse order. This is helpful for variadic functions (first parameter is a fixed distance from the stack frame).

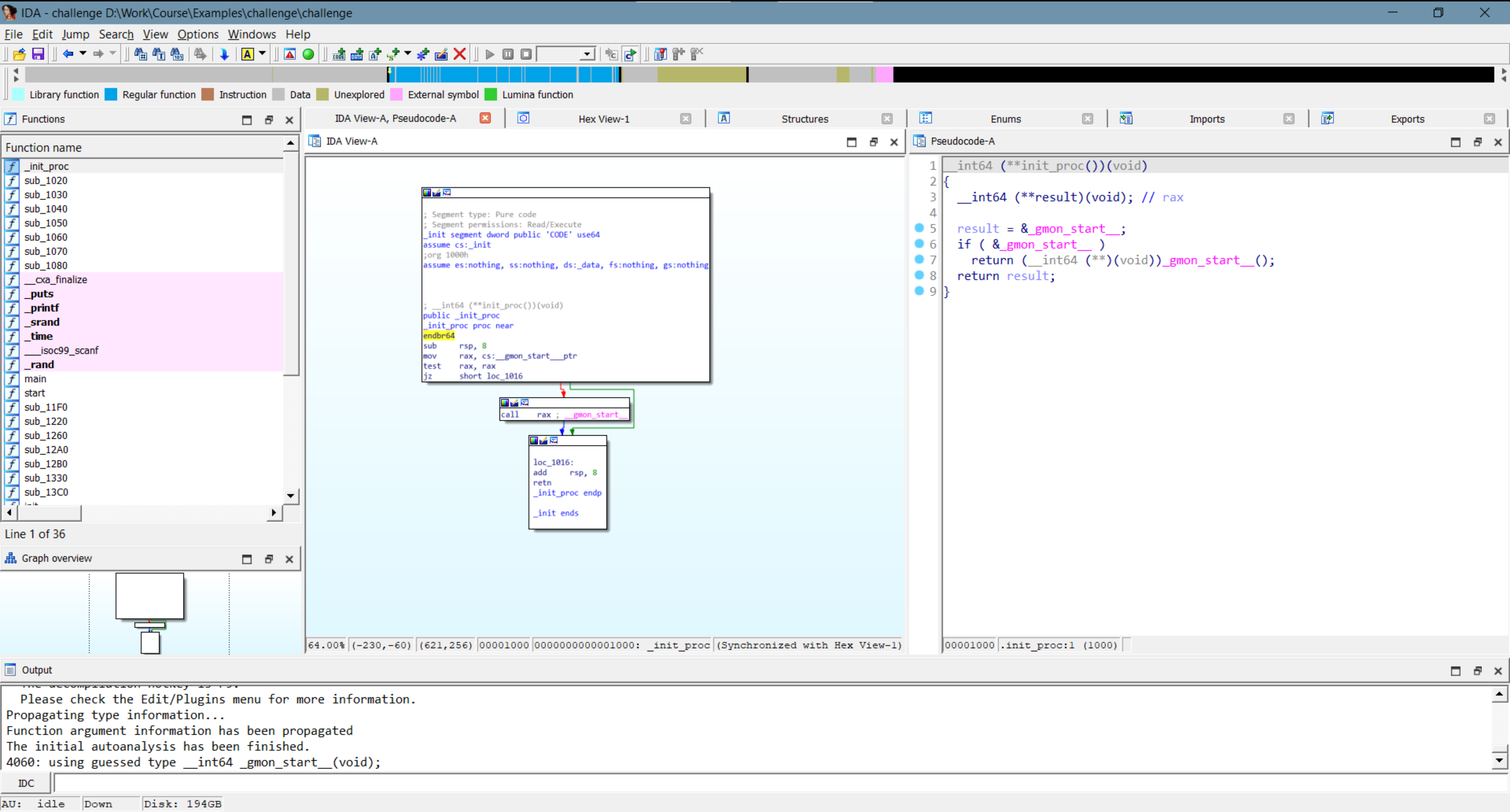
We'll look at other examples of functions in a second, when we'll introduce reversing tools

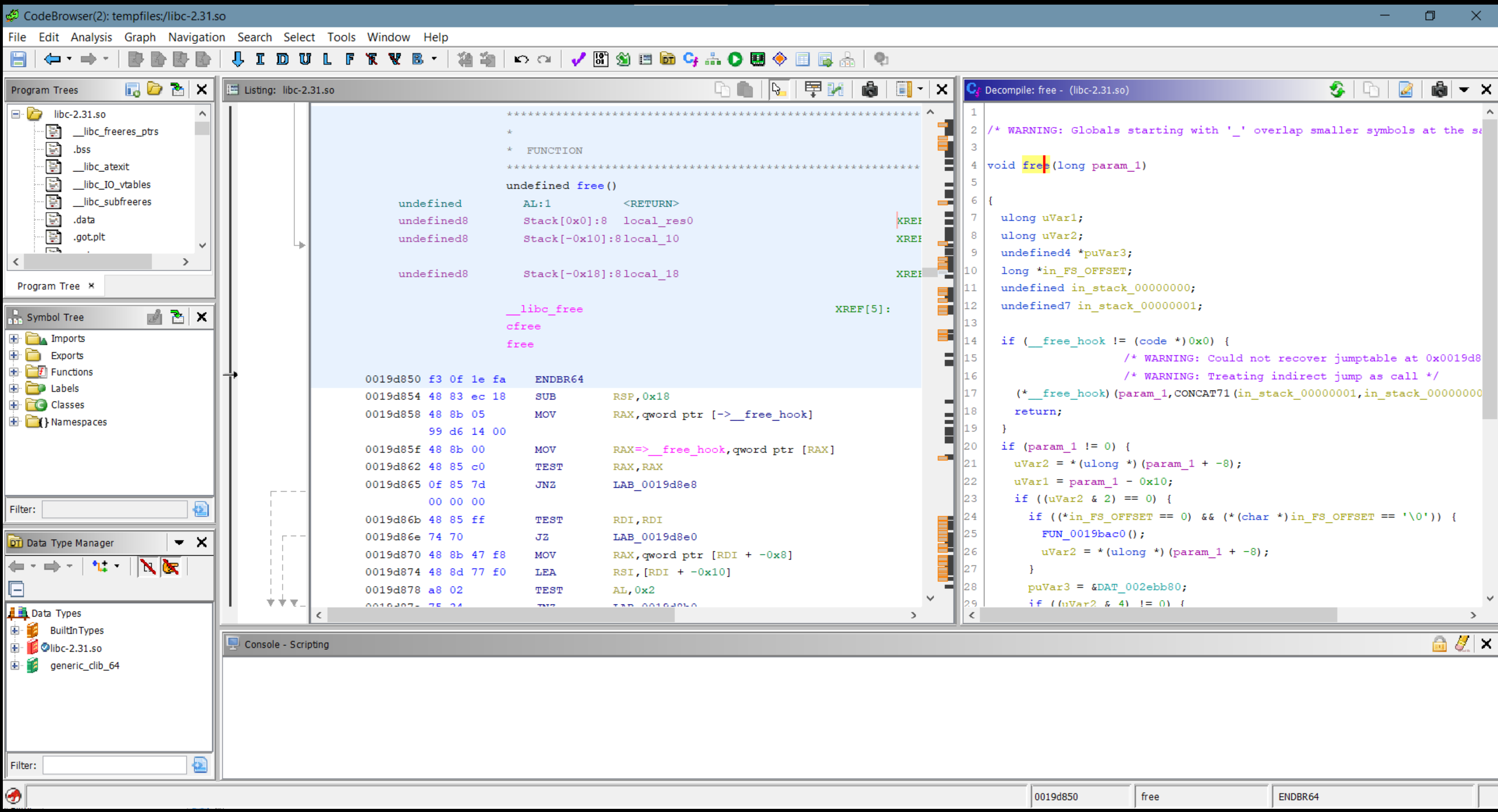
Tools of the trade

Reverse engineering

- The Interactive Disassembler (IDA)

  - Pros: versatile, easy to use, industry standard, good decompilers
  - Cons: closed source, **crazy** expensive (but the free version is very good too!)
- Ghidra
  - Initially a secret NSA tool; released as open source code to the infosec community in 2019
  - Pros: open source, versatile decompiler, free
  - Cons: still buggy years after release, scripting is a mess, decompiler is meh, JAVA
- Other tools exist (e.g. Radare)
- Plugins exist for both
  - Worth mentioning is GhIDA, which integrates Ghidra decompilers in IDA to get the best of both worlds


IDA - challenge D:\Work\Course\Examples\challenge\challenge
File Edit Jump Search View Options Windows Help
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Functions
Function name
_init_proc
sub_1020
sub_1030
sub_1040
sub_1050
sub_1060
sub_1070
sub_1080
__cxa_finalize
_puts
_printf
_srand
_time
___isoc99_scanf
_rand
main
start
sub_11F0
sub_1220
sub_1260
sub_12A0
sub_12B0
sub_1330
sub_13C0
Line 1 of 36
Graph overview
IDA View-A, Pseudocode-A
Hex View-1
Structures
Enums
Imports
Exports
IDA View-A
; Segment type: Pure code
; Segment permissions: Read/Execute
_init segment dword public 'CODE' use64
assume cs:_init
;org 1000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
; __int64 (**init_proc())(void)
public _init_proc
_init_proc proc near
endbr64
sub rsp, 8
mov rax, cs:__gmon_start___ptr
test rax, rax
jz short loc_1016
call rax ; __gmon_start__
loc_1016:
add rsp, 8
retn
_init_proc endp
_init ends
64.00% (-230,-60) (621,256) 00001000 0000000000001000: _init_proc (Synchronized with Hex View-1)
Pseudocode-A
__int64 (**init_proc())(void)
{
__int64 (**result)(void); // rax
result = &_gmon_start__;
if (&_gmon_start__)
return (__int64 (**)(void))_gmon_start__();
return result;
}
00001000 .init_proc:1 (1000)
Output
Please check the Edit/Plugins menu for more information.
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
4060: using guessed type __int64 _gmon_start__(void);
IDC
AU: idle Down Disk: 194GB


CodeBrowser(2): tempfiles:/libc-2.31.so
File Edit Analysis Graph Navigation Search Select Tools Window Help
Program Trees
libc-2.31.so
__libc_freeres_ptrs
.bss
__libc_atexit
__libc_IO_vtables
__libc_subfreeres
.data
.got.plt
Program Tree
Symbol Tree
Imports
Exports
Functions
Labels
Classes
Namespaces
Filter:
Data Type Manager
Data Types
BuiltInTypes
libc-2.31.so
generic_clib_64
Filter:
Listing: libc-2.31.so
* FUNCTION
undefined free()
undefined AL:1 <RETURN>
undefined8 Stack[0x0]:8 local_res0
undefined8 Stack[-0x10]:8 local_10
undefined8 Stack[-0x18]:8 local_18
__libc_free XREF[5]:
cfree
free
0019d850 f3 0f 1e fa ENDBR64
0019d854 48 83 ec 18 SUB RSP,0x18
0019d858 48 8b 05 99 d6 14 00 MOV RAX,qword ptr [->__free_hook]
0019d85f 48 8b 00 MOV RAX=>__free_hook,qword ptr [RAX]
0019d862 48 85 c0 TEST RAX,RAX
0019d865 0f 85 7d 00 00 00 JNZ LAB_0019d8e8
0019d86b 48 85 ff TEST RDI,RDI
0019d86e 74 70 JZ LAB_0019d8e0
0019d870 48 8b 47 f8 MOV RAX,qword ptr [RDI + -0x8]
0019d874 48 8d 77 f0 LEA RSI,[RDI + -0x10]
0019d878 a8 02 TEST AL,0x2
Decompile: free - (libc-2.31.so)
/* WARNING: Globals starting with '_' overlap smaller symbols at the sa
void free(long param_1)
{
ulong uVar1;
ulong uVar2;
undefined4 *puVar3;
long *in_FS_OFFSET;
undefined in_stack_00000000;
undefined7 in_stack_00000001;
if (__free_hook != (code *)0x0) {
/* WARNING: Could not recover jumptable at 0x0019d8
/* WARNING: Treating indirect jump as call */
(*__free_hook)(param_1,CONCAT71(in_stack_00000001,in_stack_0000000
return;
}
if (param_1 != 0) {
uVar2 = *(ulong *)(param_1 + -8);
uVar1 = param_1 - 0x10;
if ((uVar2 & 2) == 0) {
if ((*in_FS_OFFSET == 0) && (*(char *)in_FS_OFFSET == '\0')) {
FUN_0019bac0();
uVar2 = *(ulong *)(param_1 + -8);
}
puVar3 = &DAT_002ebb80;
Console - Scripting
0019d850
free
ENDBR64

Tools of the trade

- In this course we will be using IDA and Ghidra for reversing
- We will not be using their decompilers for the first sets of exercises
- Switching between the two can be annoying (especially the keyboard shortcuts – cheat sheets will be provided)
- But each has its strengths so it’s best to familiarize ourselves with both

Tools of the trade

Debugging:

Many debuggers out there, but we will be using **gdb** (GNU debugger)

- GNU/Linux standard
- Works very well without sources
- Command-line interface (GUI/Text UI front-ends exist)
- Many plugins
  - Of particular note is GEF (GDB Enhanced Features)
- We’ll show the basics but in general, the **help** command is your friend
- Command cheat sheets are very helpful (we’ll provide a decent one)

Example gdb session

(slightly edited for clarity)

```
$ gdb challenge
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online
at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to
"word"...
Reading symbols from challenge...
(No debugging symbols found in challenge)
(gdb) set disassembly-flavor intel
(gdb) break * 0x555555555168
Breakpoint 1 at 0x555555555168
```

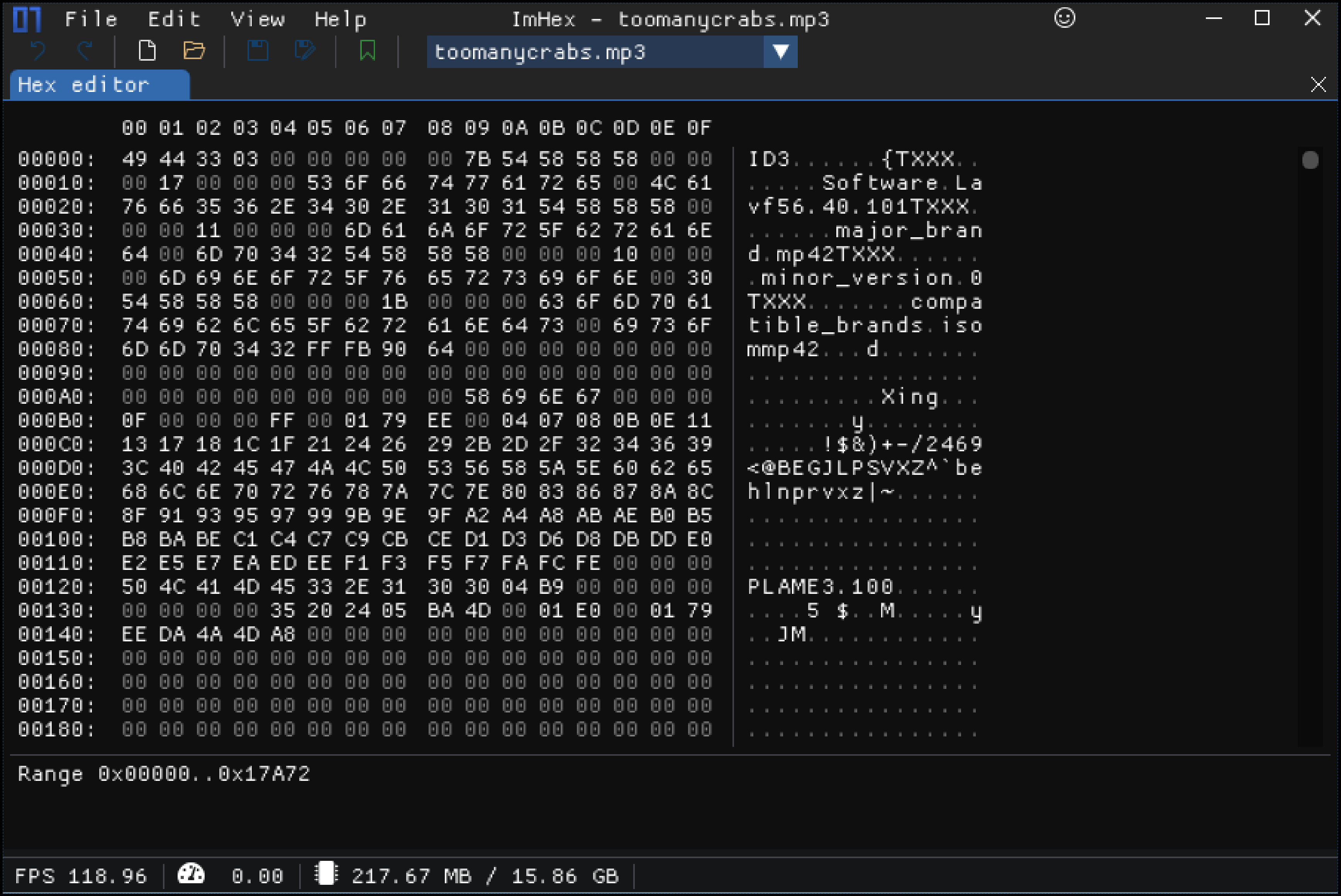
```
(gdb) run
Starting program: /mnt/d/Work/Course/Examples/challenge/challenge

Breakpoint 1, 0x0000555555555168 in ?? ()
(gdb) x/10i $rip
=> 0x555555555168:      call   0x5555555550e0 <__isoc99_scanf@plt>
   0x55555555516d:      cmp    eax,0x1
   0x555555555170:      jne    0x5555555551aa
   0x555555555172:      mov    rsi,QWORD PTR [rsp+0x8]
   0x555555555177:      mov    rdi,rbp
   0x55555555517a:      call   0x555555555330
   0x55555555517f:      test   rax,rax
   0x555555555182:      jne    0x55555555519a
   0x555555555184:      lea    rdi,[rip+0xf13] # 0x55555555609e
   0x55555555518b:      call   0x5555555550a0 <puts@plt>
(gdb) stepi
0x00005555555550e0 in __isoc99_scanf@plt ()
(gdb) cont
Continuing.
[Inferior 1 (process 51) exited with code 0377]
(gdb)
```

Tools of the trade

Other tools:

- Hex editors
  - Used to display and edit binary files in hexadecimal representation
  - For our purposes any editor will do
    - you can consult Wikipedia:

      https://en.wikipedia.org/wiki/Comparison_of_hex_editors
- Scripts and one-off tools
  - We’ll encounter some later in the course


File Edit View Help
ImHex - toomanycrabs.mp3
toomanycrabs.mp3
Hex editor
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000: 49 44 33 03 00 00 00 00 00 7B 54 58 58 58 00 00 ID3......{TXXX..
00010: 00 17 00 00 00 53 6F 66 74 77 61 72 65 00 4C 61 .....Software.La
00020: 76 66 35 36 2E 34 30 2E 31 30 31 54 58 58 58 00 vf56.40.101TXXX.
00030: 00 00 11 00 00 00 6D 61 6A 6F 72 5F 62 72 61 6E ......major_bran
00040: 64 00 6D 70 34 32 54 58 58 58 00 00 00 10 00 00 d.mp42TXXX......
00050: 00 6D 69 6E 6F 72 5F 76 65 72 73 69 6F 6E 00 30 .minor_version.0
00060: 54 58 58 58 00 00 00 1B 00 00 00 63 6F 6D 70 61 TXXX.......compa
00070: 74 69 62 6C 65 5F 62 72 61 6E 64 73 00 69 73 6F tible_brands.iso
00080: 6D 6D 70 34 32 FF FB 90 64 00 00 00 00 00 00 00 mmp42...d.......
00090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
000A0: 00 00 00 00 00 00 00 00 00 58 69 6E 67 00 00 00 ....Xing...
000B0: 0F 00 00 00 FF 00 01 79 EE 00 04 07 08 0B 0E 11 ....y........
000C0: 13 17 18 1C 1F 21 24 26 29 2B 2D 2F 32 34 36 39 .....!$&)+-/2469
000D0: 3C 40 42 45 47 4A 4C 50 53 56 58 5A 5E 60 62 65 <@BEGJLPSVXZ^`be
000E0: 68 6C 6E 70 72 76 78 7A 7C 7E 80 83 86 87 8A 8C hlnprvxz|~......
000F0: 8F 91 93 95 97 99 9B 9E 9F A2 A4 A8 AB AE B0 B5 ....
00100: B8 BA BE C1 C4 C7 C9 CB CE D1 D3 D6 D8 DB DD E0 ....
00110: E2 E5 E7 EA ED EE F1 F3 F5 F7 FA FC FE 00 00 00 ....
00120: 50 4C 41 4D 45 33 2E 31 30 30 04 B9 00 00 00 00 PLAME3.100......
00130: 00 00 00 00 35 20 24 05 BA 4D 00 01 E0 00 01 79 ....5 $..M.....y
00140: EE DA 4A 4D A8 00 00 00 00 00 00 00 00 00 00 00 ..JM............
00150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
Range 0x00000..0x17A72
FPS 118.96
0.00
217.67 MB / 15.86 GB

A note about documentation

“We don’t need documentation

We don’t need no source control”

– Pink Floyd

A serious note about documentation

Documentation (both on-the-fly and as a final product) is as important in reverse engineering as it is in software development

- While reverse engineering
  - Give meaningful names to your functions and variables
    - Even if you’re not entirely sure what they do
      - IDA and Ghidra allow question marks as part of a symbol for a reason
  - Use the comment features of IDA/Ghidra
  - When you encounter complex code it’s worth writing down what it does step by step
    - Words, diagrams, flowcharts, pseudocode and actual code are all valid
  - Write a script any time you need to calculate something or for repetitive tasks
    - You’ll probably need to use it again
    - The script itself will serve as documentation
- Whenever you reach a milestone in your research it’s worth writing a thorough report

About your exercises

Each exercise has a specific requirement in its instructions

- The end result is usually a piece of code or a binary file

Even if not explicitly required, every step of the solution is part of the solution

- All source files, Python scripts, etc.
- You should be able to reproduce the solution from scratch

We’re available in person and by email for any question

Let’s look at some tools