# Session 4

CTF Track: Reverse Engineering

# Reminder

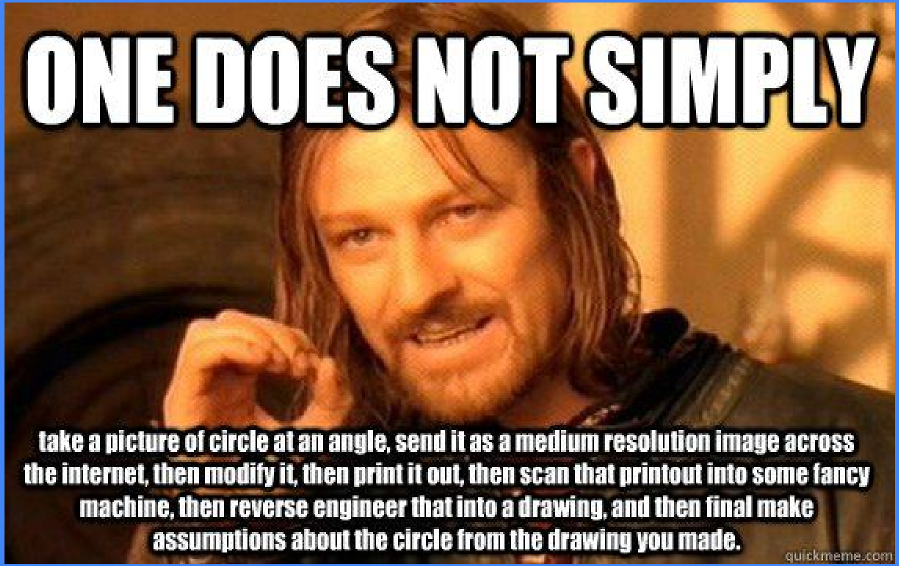
As announced last time, we will participating in TU CTF on Nov 26th from 10am to 3pm.

# Contents

- What is reverse engineering?
- Basic skills
- Practice problems

# What is reverse engineering?

Oops.



# What is reverse engineering?

(in general)

Wikipedia has a pretty abstract definition:

*Reverse engineering, also called back engineering, is the processes of extracting knowledge or design information from a product and reproducing it or reproducing anything based on the extracted information.*

# What is reverse engineering?

(for software)

Most software is licensed to you under the premise that you do not have the rights to reverse engineer it. It basically means you are not allowed to run it inside a debugger to discover how it works and to modify it.

M. No Reverse Engineering. You may not, and you agree not to or enable others to, copy (except as expressly permitted by this License or by the Usage Rules if they are applicable to you), decompile, reverse engineer, disassemble, attempt to derive the source code of, decrypt, modify, or create derivative works of the Apple Software or any services provided by the Apple Software or any part thereof (except as and only to the extent any foregoing restriction is prohibited by applicable law or by licensing terms governing use of Open-Sourced Components that may be included with the Apple Software).

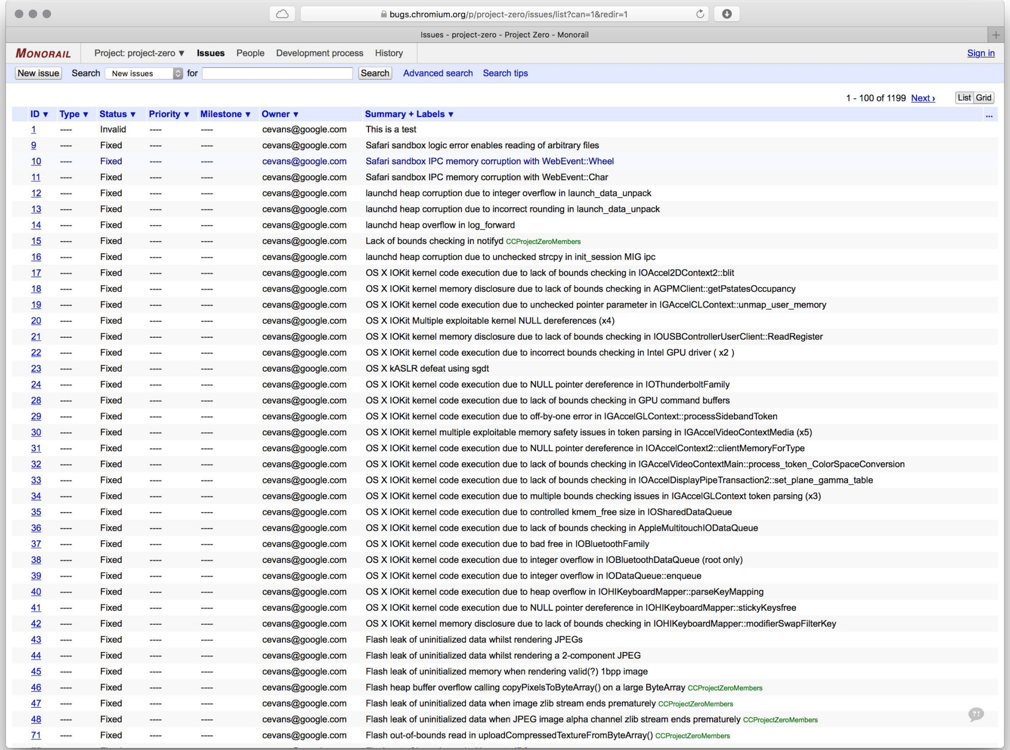
**An excerpt from the License Agreement of some Apple software**

# What is reverse engineering?

(for security researchers)

Security researchers ignore those rules.

They reverse engineer software as they see fit, usually to discover and report security vulnerabilities.



bugs.chromium.org/p/project-zero/issues/list?can=1&redir=1

Issues - project-zero - Project Zero - Monorail

MONORAIL Project: project-zero ▾ Issues People Development process History Sign in

New issue Search New issues for Search Advanced search Search tips

1 - 100 of 1199 Next › List Grid

| ID ▾ | Type ▾ | Status ▾ | Priority ▾ | Milestone ▾ | Owner ▾ | Summary + Labels ▾ |
|---|---|---|---|---|---|---|
| 1 | ---- | Invalid | ---- | ---- | cevans@google.com | This is a test |
| 9 | ---- | Fixed | ---- | ---- | cevans@google.com | Safari sandbox logic error enables reading of arbitrary files |
| 10 | ---- | Fixed | ---- | ---- | cevans@google.com | Safari sandbox IPC memory corruption with WebEvent::Wheel |
| 11 | ---- | Fixed | ---- | ---- | cevans@google.com | Safari sandbox IPC memory corruption with WebEvent::Char |
| 12 | ---- | Fixed | ---- | ---- | cevans@google.com | launchd heap corruption due to integer overflow in launch_data_unpack |
| 13 | ---- | Fixed | ---- | ---- | cevans@google.com | launchd heap corruption due to incorrect rounding in launch_data_unpack |
| 14 | ---- | Fixed | ---- | ---- | cevans@google.com | launchd heap overflow in log_forward |
| 15 | ---- | Fixed | ---- | ---- | cevans@google.com | Lack of bounds checking in notifyd CCProjectZeroMembers |
| 16 | ---- | Fixed | ---- | ---- | cevans@google.com | launchd heap corruption due to unchecked strcpy in init_session MIG ipc |
| 17 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to lack of bounds checking in IOAccel2DContext2::blit |
| 18 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel memory disclosure due to lack of bounds checking in AGPMClient::getPstatesOccupancy |
| 19 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to unchecked pointer parameter in IGAccelCLContext::unmap_user_memory |
| 20 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit Multiple exploitable kernel NULL dereferences (x4) |
| 21 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel memory disclosure due to lack of bounds checking in IOUSBControllerUserClient::ReadRegister |
| 22 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to incorrect bounds checking in Intel GPU driver ( x2 ) |
| 23 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X kASLR defeat using sgdt |
| 24 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to NULL pointer dereference in IOThunderboltFamily |
| 28 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to lack of bounds checking in GPU command buffers |
| 29 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to off-by-one error in IGAccelGLContext::processSidebandToken |
| 30 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel multiple exploitable memory safety issues in token parsing in IGAccelVideoContextMedia (x5) |
| 31 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to NULL pointer dereference in IOAccelContext2::clientMemoryForType |
| 32 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to lack of bounds checking in IGAccelVideoContextMain::process_token_ColorSpaceConversion |
| 33 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to lack of bounds checking in IOAccelDisplayPipeTransaction2::set_plane_gamma_table |
| 34 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to multiple bounds checking issues in IGAccelGLContext token parsing (x3) |
| 35 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to controlled kmem_free size in IOSharedDataQueue |
| 36 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to lack of bounds checking in AppleMultitouchIODataQueue |
| 37 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to bad free in IOBluetoothFamily |
| 38 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to integer overflow in IOBluetoothDataQueue (root only) |
| 39 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to integer overflow in IODataQueue::enqueue |
| 40 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to heap overflow in IOHIKeyboardMapper::parseKeyMapping |
| 41 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel code execution due to NULL pointer dereference in IOHIKeyboardMapper::stickyKeysfree |
| 42 | ---- | Fixed | ---- | ---- | cevans@google.com | OS X IOKit kernel memory disclosure due to lack of bounds checking in IOHIKeyboardMapper::modifierSwapFilterKey |
| 43 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash leak of uninitialized data whilst rendering JPEGs |
| 44 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash leak of uninitialized data whilst rendering a 2-component JPEG |
| 45 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash leak of uninitialized memory when rendering valid(?) 1bpp image |
| 46 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash heap buffer overflow calling copyPixelsToByteArray() on a large ByteArray CCProjectZeroMembers |
| 47 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash leak of uninitialized data when image zlib stream ends prematurely CCProjectZeroMembers |
| 48 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash leak of uninitialized data when JPEG image alpha channel zlib stream ends prematurely CCProjectZeroMembers |
| 71 | ---- | Fixed | ---- | ---- | cevans@google.com | Flash out-of-bounds read in uploadCompressedTextureFromByteArray() CCProjectZeroMembers |

# Google's "Project Zero" is basically all about reverse engineering to find vulns

# What is reverse engineering?

(for CTFs)

In CTFs, reverse engineering questions usually involve:

1. Being provided an unknown binary
2. Find out some way to coerce it into doing something

# Typical CTF Problem

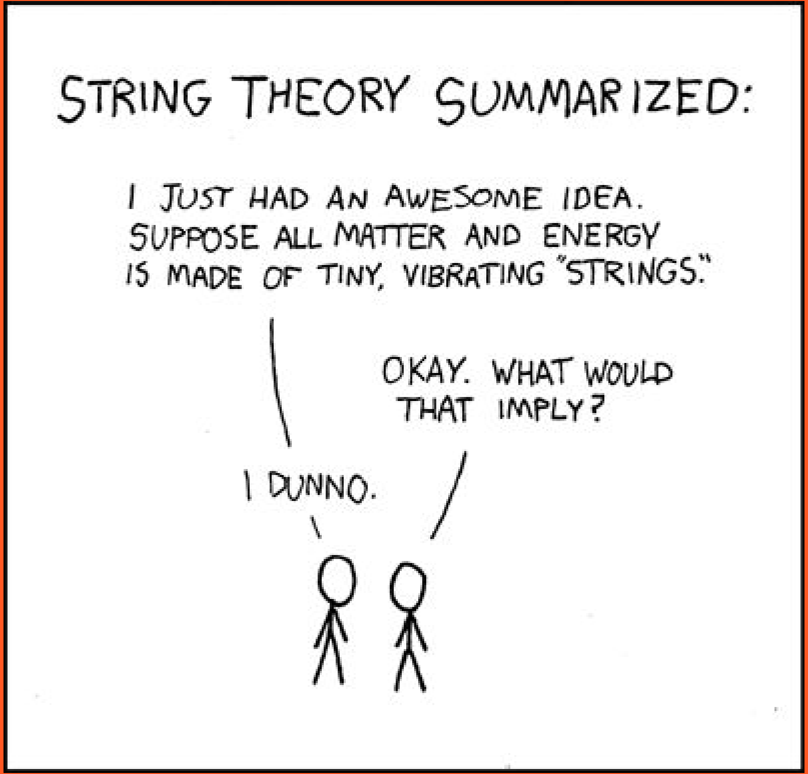
Here is a binary that performs some funky operations on a password you specify, and then after these funky operations it may give you the flag.

# Basic Skills

In beginner CTFs, sometimes the source code to that unknown executable will be provided. So you will read the source code to figure out how things work.

Also in beginner CTFs, sometimes the flag as a string is inside the executable. This is when `strings` is helpful.

# I HEARD YOU LIKE STRINGS.



# Find the printable strings in a binary.

# Find the printable strings in a binary.

What is `strings(1)`?

# strings

Its man page is less than one page (on a Mac; two pages on Linux). Just read it.



**NAME**

strings – find the printable strings in a object, or other binary, file

**SYNOPSIS**

**strings** [ **–** ] [ **–a** ] [ **–o** ] [ **–t** *format* ] [ **–***number* ] [ **–n** *number* ] [--] [file ...]

**DESCRIPTION**

*Strings* looks for ASCII strings in a binary file or standard input. *Strings* is useful for identifying random object files and many other things. A string is any sequence of 4 (the default) or more printing characters [ending at, but not including, any other character or EOF]. Unless the – flag is given, *strings* looks in all sections of the object files except the (__TEXT,__text) section. If no files are specified standard input is read.

The file arguments may be of the form *libx.a(foo.o)*, to request information about only that object file and not the entire library. (Typically this argument must be quoted, *"libx.a(foo.o)"*, to get it past the shell.)

The options to *strings*(1) are:

**–a** This option causes *strings* to look for strings in all sections of the object file (including the (__TEXT,__text) section.

**–** This option causes *strings* to look for strings in all bytes of the files (the default for non-object files).

**––** This option causes *strings* to treat all the following arguments as files.

**–o** Preceded each string by its offset in the file (in decimal).

**–t** *format*
Write each string preceded by its byte offset from the start of the file. The format shall be dependent on the single character used as the format option-argument:

*d* The offset shall be written in decimal.

*o* The offset shall be written in octal.

*x* The offset shall be written in hexadecimal.

**–***number*
The decimal *number* is used as the minimum string length rather than the default of 4.

**–n** *number*
Specify the minimum string length, where the number argument is a positive decimal integer. The default shall be 4.

**–arch** *arch_type*
Specifies the architecture, *arch_type*, of the file for *strings*(1) to operate on when the file is a universal file. (See *arch*(3) for the currently know *arch_types*.) The *arch_type* can be "all" to operate on all architectures in the file.

**SEE ALSO**

od(1)

**BUGS**

The algorithm for identifying strings is extremely primitive.

# Simple Usage

```
$ strings -n 8 funny < /path/to/your/file
```

The strings utility is by default prints any 4 consecutive printable characters in a binary file.

But this frequently results in false positives because typical x86-64 machine code can contain many sequences of 4 consecutive printable characters.

We recommend passing `-n 8` (or maybe `-n 6` if you don't consider yourself lucky).

# Exploration Time!

Try to run the following and see what you can find:

- `strings -n 8 $(which cat) # find printable strings in the cat(1) utility`
- `strings -n 8 $(which ls) # find printable strings in the ls(1) utility`
- `strings -n 8 $(which cp) # find printable strings in the cp(1) utility`

What do you see? Why do you see those strings? What do you think these strings are used for?

Discuss with people around you (5 mins).

# The Hunt for Paul Eggert

Challenge: for all executables in your PATH, run `strings -n 8` on them. How many of them contains the string “Eggert”? Which?

# The Hunt for Paul Eggert: Example Solution

```
for p in $(echo $PATH | tr : $'\n'); do

    find $p -maxdepth 1 -type f -exec sh -c 'strings -n 8 {} | fgrep -q Paul\ Eggert' \; -print

done 2> /dev/null
```

On a SEASnet machine with typical PATH configuration, I find 20 of them.

Now manually run strings to find out why they have “Paul Eggert.”

# Alternatives to strings

Although strings is simple and popular, many more powerful utilities can perform similar functionalities of finding and extracting printed ASCII characters, including old good grep and its extended cousin egrep.

grep(1)/egrep(1) allows you to only print matched fragments within a file, without dumping all the binary and unprintable characters.

grep(1)/egrep(1) also allows recursive searching without using tools like find(1) or xargs(1).

# The Hunt for Paul Eggert, revisited

Challenge: for all files (recursively) in /usr/bin, which of them contains a string that has an uppercase P, followed by three alphabetical characters, a space, an uppercase “E”, followed by four alphabetical characters, followed by a lowercase “t”?

What are the matched strings in question?

# The Hunt for Paul Eggert, revisited: Example Solution

```
# To find which files:
egrep -rl 'P[[:alpha:]]{3} E[[:alpha:]]{4}t' /usr/bin 2>/dev/null

# To find what the matched strings are:
egrep -aro 'P[[:alpha:]]{3} E[[:alpha:]]{4}t' /usr/bin 2>/dev/null
```

# Packers (Executable compressor)

Even though a CTF reverse engineering problem could be more complicated than just running strings, sometimes it can reveal important clues.

Take for example flag from pwnable.kr. When you run strings, one of the lines is:

```
$Info: This file is packed with the UPX executable packer
http://upx.sf.net $
$Id: UPX 3.08 Copyright (C) 1996-2011 the UPX Team. All
Rights Reserved. $
```

This tells you the file is packed with UPX and you can use `upx -d` to unpack it.

**Is every CTF rev eng problem as simple as running strings? I sure hope so.**

(It's not.)

# What if source code is provided?

Since many of you haven't taken CS33 yet, let's first tackle the kind of problems where source code is given.

Take collision from pwnable.kr for example.

# Walkthrough: pwnable.kr collision

```c
#include <stdio.h>
#include <string.h>
unsigned long hashcode = 0x21DD09EC;
unsigned long check_password(const char* p){
    int* ip = (int*)p;
    int i;
    int res=0;
    for(i=0; i<5; i++){
        res += ip[i];
    }
    return res;
}

int main(int argc, char* argv[]){
    if(argc<2){
        printf("usage : %s [passcode]\n", argv[0]);
        return 0;
    }
    if(strlen(argv[1]) != 20){
        printf("passcode length should be 20 bytes\n");
        return 0;
    }
```

```c
    if(hashcode == check_password( argv[1] )){
        system("/bin/cat flag");
        return 0;
    }
    else
        printf("wrong passcode.\n");
    return 0;
}
```

# Understanding the `main` function

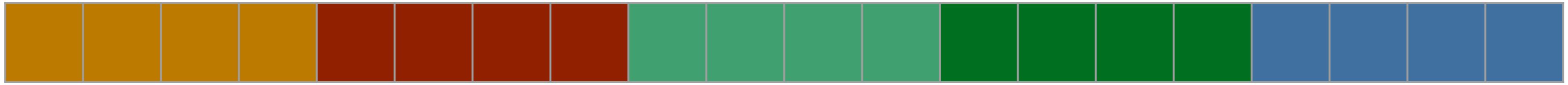
- It first makes sure one argument is provided.
- Then it makes sure it is exactly 20 bytes.
- Then it makes sure after doing some funky operations, the result is `0x21DD09EC`.
- Then it gives you the flag!

# Understanding the `check_password` function

```
unsigned long check_password(const char* p){
    int* ip = (int*)p;
    int i;
    int res=0;
    for(i=0; i<5; i++){
        res += ip[i];
    }
    return res;
}
```

- It first casts a pointer to characters to a pointer to int. Essentially reinterpret_cast in C++.
- And it adds together the five integers that are contained in those 20 bytes.

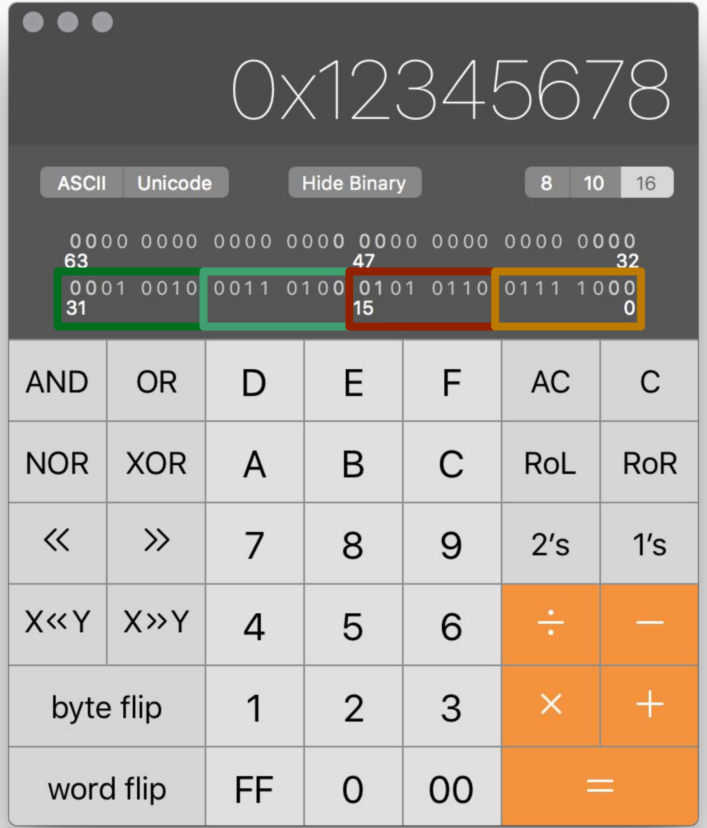
# Understanding the `check_password` function

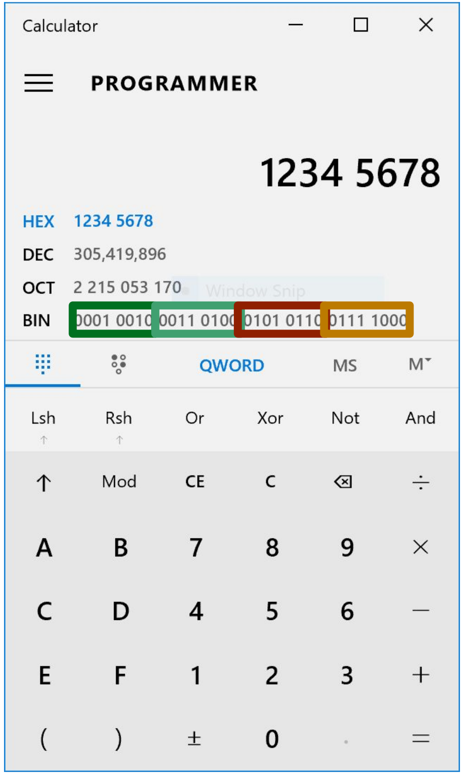


- Each small box above is a character you need to provide.
- Each colored region is a reinterpreted integer.

# Understanding the `check_password` function


- Looking at one integer.
- Each colored region is one byte now.
- Little endian: least significant byte on the left, most significant byte on the right.
- Remember to use your calculator to help you!




**Use your calculator! Remember however that it uses big-endian so the order is reversed.**

# Advanced Reverse Engineering: x86-64 and disassemblers

In more advanced CTF problems, you will not be given the source code.

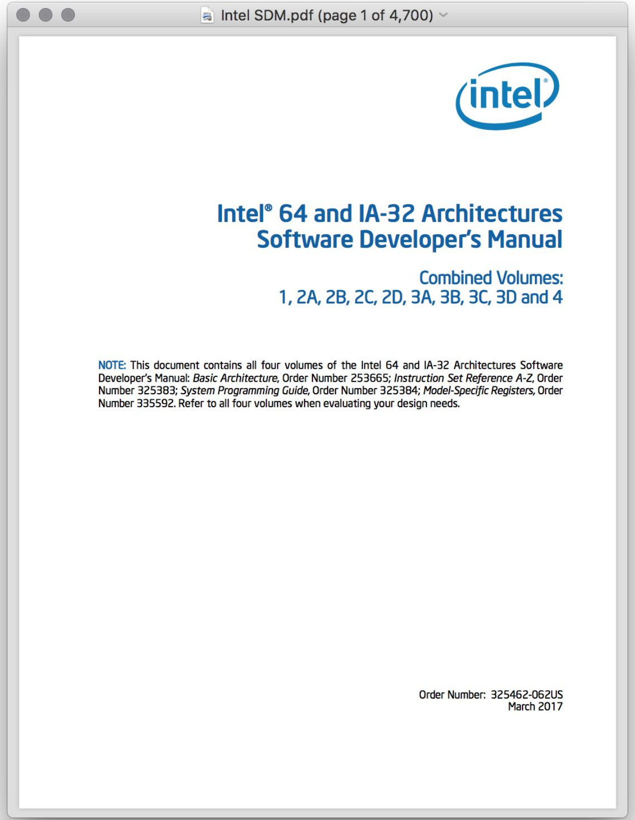
You then have to use disassemblers and/or debugger such as GDB.

You will also need to use GDB.

# The x86 and x86-64 architecture

Very few people on few know every detail of x86-64 systems. The full documentation is a staggering 4,700 pages.

(It was my bedtime reading when I was taking CS33).

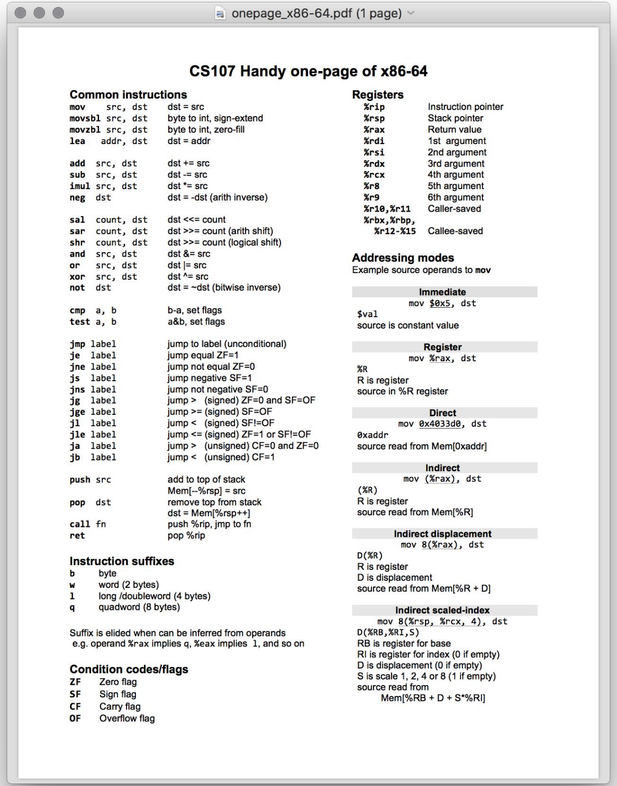


# The x86 and x86-64 architecture

Basic concepts:

- What is RAM?
  - What's the stack?
  - How do we address memory?
- What are registers?
- What is code?
- How do C control structures map to assembler?
  - If/else, function calls, loops, switches

Use this one-page summary of x86-64 from Stanford as a reference!



onepage_x86-64.pdf (1 page)

**CS107 Handy one-page of x86-64**

**Common instructions**

```
mov    src, dst      dst = src
movsbl src, dst      byte to int, sign-extend
movzbl src, dst      byte to int, zero-fill
lea    addr, dst     dst = addr

add  src, dst        dst += src
sub  src, dst        dst -= src
imul src, dst        dst *= src
neg  dst             dst = -dst (arith inverse)

sal  count, dst      dst <<= count
sar  count, dst      dst >>= count (arith shift)
shr  count, dst      dst >>= count (logical shift)
and  src, dst        dst &= src
or   src, dst        dst |= src
xor  src, dst        dst ^= src
not  dst             dst = ~dst (bitwise inverse)

cmp  a, b            b-a, set flags
test a, b            a&b, set flags

jmp label            jump to label (unconditional)
je  label            jump equal ZF=1
jne label            jump not equal ZF=0
js  label            jump negative SF=1
jns label            jump not negative SF=0
jg  label            jump >  (signed) ZF=0 and SF=OF
jge label            jump >= (signed) SF=OF
jl  label            jump <  (signed) SF!=OF
jle label            jump <= (signed) ZF=1 or SF!=OF
ja  label            jump >  (unsigned) CF=0 and ZF=0
jb  label            jump <  (unsigned) CF=1

push src             add to top of stack
                     Mem[--%rsp] = src
pop  dst             remove top from stack
                     dst = Mem[%rsp++]
call fn              push %rip, jmp to fn
ret                  pop %rip
```

**Instruction suffixes**

| | |
|---|---|
| b | byte |
| w | word (2 bytes) |
| l | long /doubleword (4 bytes) |
| q | quadword (8 bytes) |

Suffix is elided when can be inferred from operands
e.g. operand %rax implies q, %eax implies l, and so on

**Condition codes/flags**

| | |
|---|---|
| ZF | Zero flag |
| SF | Sign flag |
| CF | Carry flag |
| OF | Overflow flag |

**Registers**

| | |
|---|---|
| %rip | Instruction pointer |
| %rsp | Stack pointer |
| %rax | Return value |
| %rdi | 1st argument |
| %rsi | 2nd argument |
| %rdx | 3rd argument |
| %rcx | 4th argument |
| %r8 | 5th argument |
| %r9 | 6th argument |
| %r10,%r11 | Caller-saved |
| %rbx,%rbp, %r12-%15 | Callee-saved |

**Addressing modes**

Example source operands to mov

**Immediate**
`mov $0x5, dst`
`$val`
source is constant value

**Register**
`mov %rax, dst`
`%R`
R is register
source in %R register

**Direct**
`mov 0x4033d0, dst`
`0xaddr`
source read from Mem[0xaddr]

**Indirect**
`mov (%rax), dst`
`(%R)`
R is register
source read from Mem[%R]

**Indirect displacement**
`mov 8(%rax), dst`
`D(%R)`
R is register
D is displacement
source read from Mem[%R + D]

**Indirect scaled-index**
`mov 8(%rsp, %rcx, 4), dst`
`D(%RB,%RI,S)`
RB is register for base
RI is register for index (0 if empty)
D is displacement (0 if empty)
S is scale 1, 2, 4 or 8 (1 if empty)
source read from
Mem[%RB + D + S*%RI]

tinyurl.com/CTFproblems4

tinyurl.com/CTFfeedback4