A Very Short Introduction to CTF - Part I

Wu Xiuheng

February 22, 2022

Outline

Overview

Overview

- The purpose is to get you familiar with Jeopardy! style of CTF competitions.
- There are lots of links (in blue color) to online resources within the slides.
- Although lots of tools and challenges are mentioned, you do not need to understand or solve all of them.
- Think and try before rushing to provided hint or solutions.

The Name of the Game

Capture the Flag or *CTF* comes from an outdoor game where two teams compete for capturing the opponent's flag and keeping their own flag safe.

(Capture the Flag, Fahne im Spiel – Ronnie Berzins – CC BY-SA 3.0)

The Name of the Game

When it comes to the context of cyber security, two forms exist:

- Attack-defense style
- *Jeopardy!* style

The Name of the Game

When it comes to the context of cyber security, two forms exist:

- Attack-defense style
- *Jeopardy!* style

Discover flags (usually a text string) by using techniques including:

- reverse-engineering,
- forensics,
- web exploitation,
- cryptoanalysis...

When it comes to the context of cyber security, two forms exist:

- Attack-defense style
- *Jeopardy!* style

Discover flags (usually a text string) by using techniques including:

- reverse-engineering,
- forensics,
- web exploitation,
- cryptoanalysis...

You are expected to

- have some basic knowledge,
- be able to program,
- learn new things quickly

A Simple Example

Puzzles in CTF competitions are usually categorized and arranged by a progression of difficulty.

Warmup challenge of PicoCTF 2018

Cryptography doesn't have to be complicated, have you ever heard of something called rot13? `cvpbPGS{guvf_vf_pelcgb!}`

A Simple Example

Puzzles in CTF competitions are usually categorized and arranged by a progression of difficulty.

Warmup challenge of PicoCTF 2018

Cryptography doesn't have to be complicated, have you ever heard of something called rot13? `cvpbPGS{guvf_vf_pelcgb!}`

Solution: Wikipedia tells us that ROT13 is a simple substitution cipher similar to Caesar cipher.

Input	`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`
Output	`NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm`

(solution continues on next page)

A Simple Example (cont'd)

Using `tr` (a UNIX utility which can be found on Linux/macOS):

```
tr 'A-Za-z' 'N-ZA-Mn-za-m' <<< "cvpbPGS{guvf_vf_pelcgb!}"
=> picoCTF{this_is_crypto!}
```

Or python:

```
u = ''.join([chr(x) for x in range(65,91)])
l = ''.join([chr(x) for x in range(97,123)])
trpair = str.maketrans(u[13:]+u[:13]+l[13:]+l[:13], u+l)
print('cvpbPGS{guvf_vf_pelcgb!}'.translate(trpair))
```

A Simple Example (cont'd)

Using `tr` (a UNIX utility which can be found on Linux/macOS):

```
tr 'A-Za-z' 'N-ZA-Mn-za-m' <<< "cvpbPGS{guvf_vf_pelcgb!}"
=> picoCTF{this_is_crypto!}
```

Or python:

```
u = ''.join([chr(x) for x in range(65,91)])
l = ''.join([chr(x) for x in range(97,123)])
trpair = str.maketrans(u[13:]+u[:13]+l[13:]+l[:13], u+l)
print('cvpbPGS{guvf_vf_pelcgb!}'.translate(trpair))
```

ROT_{13} is self-inverse, i.e., $\text{ROT}_{13}(\text{ROT}_{13}(x)) = x$

Some Basics

Binary to Text Encoding

Some binary-to-text encoding are very common in various ctf problems. Be familiar with them so that you can recognize them when they appear in the problem.

- Base32: strings encoded in base32 contains capital alphabets `A-Z`, numerals `2-7` (variations exists) and = for padding.

Try decode this `NFVW433XL5UG6527MJQXGZJTGJPXO33SNNZQ====` (answer in white below)

```
echo -n "NFVW433XL5UG6527MJQXGZJTGJPXO33SNNZQ====" | base32 -d
```

Binary to Text Encoding

Some binary-to-text encoding are very common in various ctf problems. Be familiar with them so that you can recognize them when they appear in the problem.

- Base32: strings encoded in base32 contains capital alphabets `A-Z`, numerals `2-7` (variations exists) and = for padding.
- Base64: strings encoded in base64 contains alphabets `A-Za-z`, numerals `0-9`, 2 more special characters +/ (variations exists) and = for padding.

Try decode this `YmFzZTY0X2lzX3Nob3J0ZXI=` (answer in white below)

```
echo -n "YmFzZTY0X2lzX3Nob3J0ZXI=" | base64 -d
```

Binary to Text Encoding

Some binary-to-text encoding are very common in various ctf problems. Be familiar with them so that you can recognize them when they appear in the problem.

- Base32: strings encoded in base32 contains capital alphabets `A-Z`, numerals `2-7` (variations exists) and = for padding.
- Base64: strings encoded in base64 contains alphabets `A-Za-z`, numerals `0-9`, 2 more special characters +/ (variations exists) and = for padding.
- Percent encoding: common for URI/URL, use hex ASCII values, and UTF-8 for non-ASCII characters. E.g. `%20` for space and `%2F` for slash (/)

E.g. `https://zh.wikipedia.org/wiki/%E6%96%B0%E5%8A%A0%E5%9D%A1` is the Chinese Wikipedia page for Singapore.

Binary to Text Encoding

Some binary-to-text encoding are very common in various ctf problems. Be familiar with them so that you can recognize them when they appear in the problem.

- Base32: strings encoded in base32 contains capital alphabets `A-Z`, numerals `2-7` (variations exists) and = for padding.
- Base64: strings encoded in base64 contains alphabets `A-Za-z`, numerals `0-9`, 2 more special characters +/ (variations exists) and = for padding.
- Percent encoding: common for URI/URL, use hex ASCII values, and UTF-8 for non-ASCII characters. E.g. `%20` for space and `%2F` for slash (/)

Challenge: Some problems use relative uncommon encodings. Try to solve this. (Hackergame2018 of USTC, released under CC BY-NC-SA 4.0 license.)

Hint: come back to this after reading page 23 if you cannot solve it now.

Endianness

Endianness (or *byte ordering*) refers to the order of bytes within a multi-bytes number in memory.

C-type	variable	value	Memory at offset							
			Big Endian				Little Endian			
			+0	+1	+2	+3	+0	+1	+2	+3
`int32_t`	longVar	0x0a0b0c0d	0a	0b	0c	0d	0d	0c	0b	0a
`int16_t`	shortVar	0x0c0d	0c	0d			0d	0c		

Endianness

Endianness (or *byte ordering*) refers to the order of bytes within a multi-bytes number in memory.

			Memory at offset							
			Big Endian				Little Endian			
C-type	variable	value	+0	+1	+2	+3	+0	+1	+2	+3
`int32_t`	longVar	0x0a0b0c0d	0a	0b	0c	0d	0d	0c	0b	0a
`int16_t`	shortVar	0x0c0d	0c	0d			0d	0c		

Little-Endian: the **Least significant** byte goes into the **Lowest-address**.

Binaries of today's PC are mostly little-endian and network packets are big-endian.

Endianness (cont'd)

```c
// simple.c
int main(void) {
    long a = 1234567890;
    return 0;
}
```

```
python3 -c "print(hex(1234567890))"
=> 0x499602d2

# compile and run with debugger
gcc -g simple.c -o simple && gdb simple
```

```
(gdb) break 3
Breakpoint 1 at 0x1131: file simple.c, line 3.
(gdb) run
Breakpoint 1, main () at simple.c:3
(gdb) p/d a
$1 = 1234567890
(gdb) p &a
$2 = (long *) 0x7fffffffdba8
(gdb) x/4xb &a
0x7fffffffdba8: 0xd2    0x02    0x96    0x49
```

GDB Manual: Examine Memory

Endianness (cont'd)

Example: a BMP image file beginning with `42 4D EA 9B 22 00` (first 6 bytes)

- the first 2 bytes (`42 4d`: BM) are file signature (magic bytes)
- bytes 3-6 (`EA 9B 22 00`) represents file size.
  - the file size is actually `00229BEA` (2268138) bytes.

hexdump is a utility for "display file contents in hexadecimal, decimal, octal, or ascii". When using without any options, it will display bytes as they are 2-bytes (16-bits) words in little-endian.

While xxd gives a human-friendly order by default.

```
echo -n "abcd" > tmp
# hexdump tmp
0000000 6261 6463
# hexdump -C tmp
00000000 61 62 63 64 |abcd|
# xxd tmp
00000000: 6162 6364  abcd
```

Reverse Engineering

Assembly Language

Two syntax exist for x86 assembly. web resource: a detailed comparison

```
movl  (%eax), %ecx             mov  ecx, DWORD PTR [eax]
addl  $12, %edx                add  edx, 12
movl  %ecx, -12(%edx)          mov  DWORD PTR [edx-12], ecx
```

AT&T syntax (left), used by GNU Assembler

- instructions suffixed with letters indicating operand sizes
- `$` before intermediate numbers
- `%` before register names, src operands before dest
- order of operands is *src, dst*

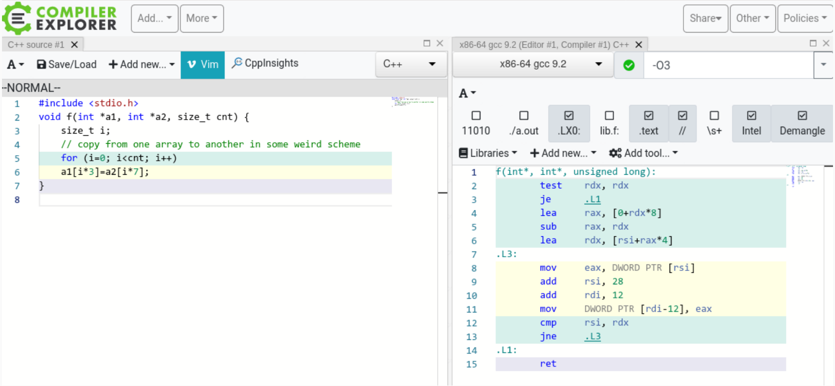
Intel syntax (right)

- order of operands is *dst, src*
- different indirect addressing format

x86 assembly: Wikibooks:X86 Assembly, a one-page quick tutorial.

Tool: Godbolt Compiler Explorer

Whenever you are interested in code generated by compiler *x* in *y* architecture with *z* options, check Godbolt Compiler Explorer.



Basic Tools

A list of command line tools for basic inspection of binary files.

- `file` shows filetype and some basic information about the file
- `strings` prints the sequences of printable characters in files, can help solve easy warmup challenges quickly.
- `xxd, hexdump` can create hexdump of files.
- `bvi` is a hex editor, there is also a good non-free GUI hex editor 010 Editor.
- `objdump` displays information from object files.
- `readelf` displays information about ELF files.
- checksec checks the properties of executables.

Ghidra

Ghidra is a powerful reverse-engineering tool, released under Apache-2.0 license.

Note: On a recent version of macOS, you may need to deal with gatekeeper. (refer to github/ghidra/issue/1559)

Resources:

- In-application help appears automatically the first time you start ghidra, or access it through [menu]->[help]->[contents].
- Presentations by NSA introduce features.
- Cheat Sheet of shortcuts.

Some alternatives are:

- Radare2 is an open source toolset.
- IDA Pro, the long time state-of-the-art product, but proprietary and expensive. An old feature-limited version is provided for free.
- Binary Ninja is also not free but comes with a demo version.

Usage of Ghidra

Can you write C programs without using headers such as stdio, stdlib?
Try to get flag from this file.

(Hackergame2019 of USTC, released under CC BY-NC-SA 4.0 license.)

Usage of Ghidra

Can you write C programs without using headers such as stdio, stdlib?
Try to get flag from this file.

(Hackergame2019 of USTC, released under CC BY-NC-SA 4.0 license.)

First we can try to run this file.
It just prompts a string.

```
% ./tinyELF
please input flag:
```

`objdump` also gives nothing useful.

```
% objdump -D tinyELF
tinyELF:     file format elf64-x86-64
```

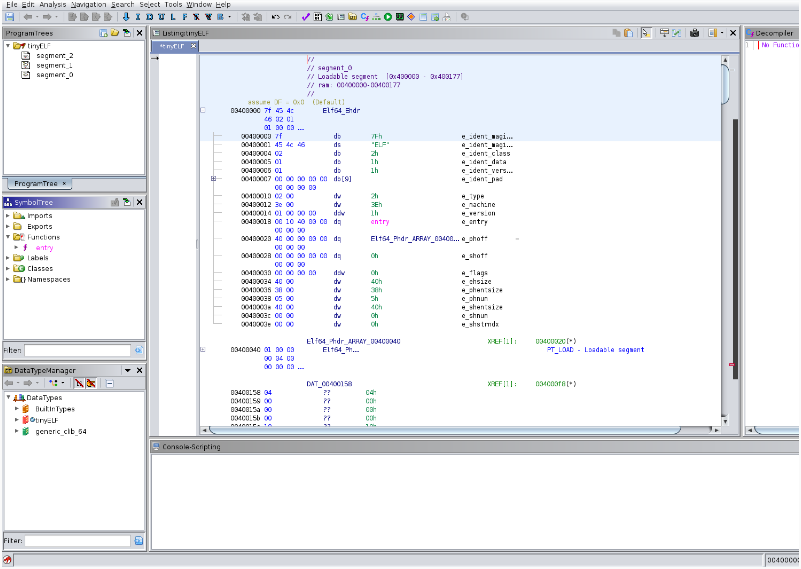
`strings` gives out some hint, possibly we need to reach the "correct".

```
% strings tinyELF
please input flag:
correct
```

Then start Ghidra:

- `[new project]->[non-shared project]`, select a path and name for your project.
- `[File]->[Import file]`, add `tinyELF` you just downloaded.
- In the active project window, double-click file you want to analyze, `tinyELF` here.
- A new code browser will appear and ask if you want to analyze this file, click `Yes`.
- Then you will see disassembler in the middle and decompiler on the right.
- To quickly find solutions, make use of the strings we found using `strings`.
- `[Search]->[For Strings]`, click `search`, and a new window pops up, click "correct" and go back to disassembler window.

Usage of Ghidra (cont'd)



Usage of Ghidra (cont'd)

StringSearch-2items-[tinyELF,Minimumsize=5,Align=1]

...	Location	Label	CodeUnit	StringView	Stri...	Le...	IsWord
A	00402000	s_please_in_put_flag:_0040...	ds "please in put flag:"	"pleaseinputflag:"	string	20	true
A	00402014	s_correct_00402014	ds "correct"	"correct"	string	8	true

After cliking "correct", ghidra will locate it in the disassembler.

```
                 s_correct_00402014         xREF[1]:    entry:004011ad(*)
63 6f 72             ds        "correct"
72 65 63
74 00
```

Double clicking `entry: 004011ad` will bring you to the address in the `entry` function.

Once you are in a function in the disassembler, ghidra will automatically show decompiled code on the right side window.

Usage of Ghidra (cont'd)

Go through the decompilied `entry`. Focus on the following part:

```
syscall(); syscall(); local_c = 0;
while(local_c < 0x2e) {
  bVar1 = (byte)local_c;
  local_48[local_c] = bVar1 * '\x02' + local_48[local_c];
  local_48[local_c] = bVar1 ^ local48[local_c];
  local_48[local_c] = local48[local_c] - bVar1;
  local_c = local_c + 1;
}
local_10 = 0;
while(local_10 < 0x2e) {
  if (local_48[local_10] != local78[local_10] {syscall();}
  local_10 = local_10 + 1;
}
syscall(); syscall();
```

In the code, `local_48` is an un-initialized array, while `local_78` is filled with specific value from `local_78[0]` to `local_78[45]`.

Usage of Ghidra (cont'd)

There are several `syscall()`s, we can check what they actually did in the assembly. Each syscall has a number, and is called by set specific register with the number. Since this is a x86_64 ELF, we have:

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
rax	rdi	rsi	rdx	r10	r8	r9

For example, look at assembly around the first syscall():

```
LEA  RDX,[s_please_in_put_flag]
MOV  RDI,0x1  ; param 1                      // /usr/include/asm/unistd_64.h
MOV  RSI,RDX  ; param 2                      #define __NR_read 0
MOV  RDX,0x13 ; param 3                      #define __NR_write 1
MOV  RAX,0x1                                 #define __NR_exit 60
SYSCALL
```

Recall the write system call or `man 2 write`:

```
write(int fd, const void *buf, size_t count);
```

So this syscall is actually `write(1, "please in put flag", 19)`.

fd 1 is stdout, so that is how the "please in put flag" appeared.

Usage of Ghidra (cont'd)

After working out all the `syscall()`s, we can have clearer code.

```
write(1, "please in put flag", 19)
read(0, user_input, 46)
for (i = 0; i < 46; ++i) {
  solve[i] += i*2;
  solve[i] ^= i;
  solve[i] -= i;
}
for (i = 0; i < 46; ++i) {
  if (solve[i] != goal[i])
    _exit(0);
}
write(1, "correct", 7)
_exit(0);
```

Values of `goal[]` are known, and can be found in assembly code.

We just need to find the 46 characters for `solve[0]` to `solve[45]`, so that they equal to `goal[]` after the transformations.

Try to write program to do the inverse calculations.

And the correct flag is `flag{Linux_Syst3m_C4ll_is_4_f4scin4ting_t00ls}`.

(This challenge can also be solved by using angr.)

Solving Puzzles with GDB

Try to get flag from this file. [from picoCTF 2021]

Solving Puzzles with GDB

Try to get flag from this file. [from picoCTF 2021]

```
.LC1: .string    "Correct! You entered the flag."
.LC2: .string    "No, that's not right."
.L2:
    ...
    leaq    -272(%rbp), %rcx ; load rbp-272 addr in %rcx
    leaq    -208(%rbp), %rax ; load rbp-208 addr in %rax
    movl    $49, %edx    ; set %edx to number 49
    movq    %rcx, %rsi   ; set %rsi to the content in %rcx
    movq    %rax, %rdi   ; set %rdi to the content in %rax
    call    memcmp@PLT   ; memcmp() compare %rsi and %rdi
    testl   %eax, %eax ; check return val of memcmp()
    jne     .L4  ; if memcmp() does not ret 0, i.e., %rsi != %rdi (49 bytes)
    leaq    .LC1(%rip), %rdi
    call    puts@PLT  ; print out "Correct"
    movl    $0, %eax

.L4:
    leaq    .LC2(%rip), %rdi
    call    puts@PLT  ; print out "No"
    movl    $1, %eax
```

By locating the two strings indicating a "correct" or "incorrect" branch, we discover the call to `memcmp()` which is condition determining which branch to take.

Solving Puzzles with GDB (cont'd)

Go back to check the content on stack, user inputs are read using `fgets()` and stored starting from `rbp-208`.

```
movq stdin(%rip), %rdx ; %rdx: FILE *st
leaq -208(%rbp), %rax ; load rbp-208 address in %rax movl $50, %esi ; %esi: int size
movq %rax, %rdi ; %rdi: char* s
call fgets@PLT ; fgets(char*s, int size, FILE *st)
```

So what contents do `rbp-272` hold?

We can analyze the rest assembly to work out the transformations on `rbp-272`, but a dynamic solution is easier for this puzzle.

Solving Puzzles with GDB (cont'd)

Go back to check the content on stack, user inputs are read using `fgets()` and stored starting from `rbp-208`.

```
movq stdin(%rip), %rdx ; %rdx: FILE *st
leaq -208(%rbp), %rax ; load rbp-208 address in %rax movl $50, %esi ; %esi: int size
movq %rax, %rdi ; %rdi: char* s
call fgets@PLT ; fgets(char*s, int size, FILE *st)
```

So what contents do `rbp-272` hold?

We can analyze the rest assembly to work out the transformations on `rbp-272`, but a dynamic solution is easier for this puzzle.

Note a difference between this puzzle and the previous ghidra one is:

In this puzzle, the transformations are done on the constants while in the ghidra one, the transformations are done on the user input. Therefore, this gdb method cannot be applied directly on that puzzle we solved using ghidra.

Solving Puzzles with GDB (cont'd)

```
% gdb chall
(gdb) run # program wait for user input, press Control-C here
^C Program received signal SIGINT, Interrupt.
(gdb) next
asdfasdf # random input
(gdb) finish # use finish until we are back in the main()
(gdb) disassemble # check the assembly code to get desired breakpoint
# You can find sth. like this:
#   00005555555552f0 <+379>:  call  0x555555555060 <memcmp@plt>
(gdb) break *0x555555555060
(gdb) c # continue to the breakpoint
(gdb) x/49xb $rsi  # refer to previous slides on why the length is 49
0x7fffffffd6b0: 0x70  0x69  0x63  0x6f  0x43  0x54  0x46  0x7b
0x7fffffffd6b8: 0x64  0x79  0x6e  0x34  0x6d  0x31  0x63  0x5f
0x7fffffffd6c0: 0x34  0x6e  0x34  0x6c  0x79  0x31  0x73  0x5f
0x7fffffffd6c8: 0x31  0x73  0x5f  0x35  0x75  0x70  0x33  0x72
0x7fffffffd6d0: 0x5f  0x75  0x73  0x33  0x66  0x75  0x6c  0x5f
0x7fffffffd6d8: 0x36  0x30  0x34  0x34  0x65  0x36  0x36  0x30
0x7fffffffd6e0: 0x7d
```

Convert hex to ASCII characters, you can get the flag:

`picoCTF{dyn4m1c_4n4ly1s_1s_5up3r_us3ful_6044e660}`.

Forensics

Steganography

Steganography usually involves finding information hidden in various types of files. The most common steganography challenges are

- including the flags in an archive file (try to solve this and this from Google CTF 2018 QualsBeginners Quest)
- modifing the length field in the metadata (similar to the challenge using an uncommon encoding on page 5).
- Hiding flags in image files is quite popular. Information can be inserted in not only metadata of images but also lower bits of "actual image data".(Click "how it works", if you do not understand the explanation, read about RGB)

Try this: You are given an image. Try to extract the flag. (Hint: use the lower bits trick) Solution [PDF]

Another Simple Challenge

Try to extract flag from this filesystem data.

(Google CTF 2018 Quals Beginners Quest.)

Another Simple Challenge

Try to extract flag from this filesystem data.

(Google CTF 2018 Quals Beginners Quest.)

- `file` tells that it is a gzip, `gzip -d challenge.ext4.gz`
- mount the ext4 file, `sudo mount -o loop challenge.ext4 /tmp/mount`
  (Always be careful when you use `sudo` and only when you know what you are doing.)
- See the `.mediapc_backdoor_password.gz` file, uncompress it using `gzip -d` and you get the flag.

Appendix

Other resources

- RE4B is a free book about reverse engineering.
- Refer to list of file signatures when checking file magic bytes.
- Awesome-ctf on github is a curated list of CTF-related resources
- A YouTube channel related to CMU picoCTF competition.
- TLDR is an utility for showing common usage of command line tools.