**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>

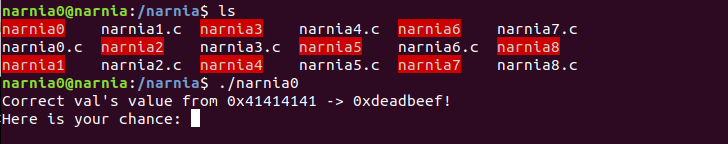
<https://www.youtube.com/watch?v=1S0aBV-Waeo>

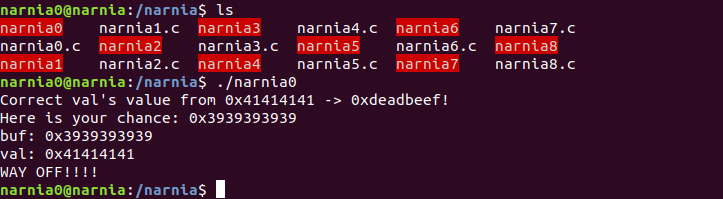
GDB cheat sheet:

<https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>

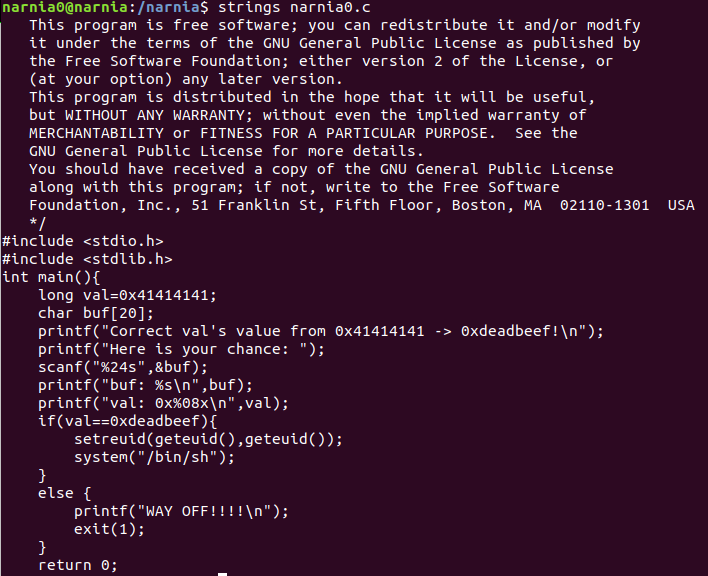
**Narnia 0 → 1:**

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:

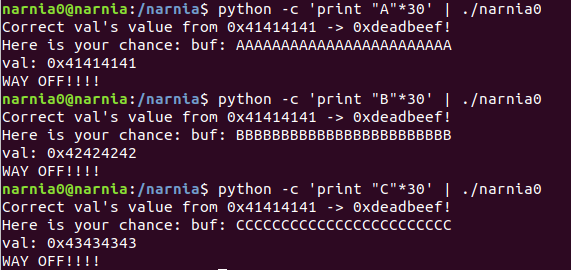




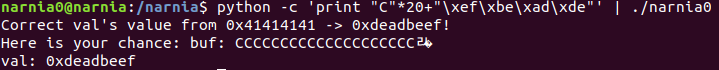
So from this I was able to guess what the program wants us to do. The value val is in a number notation known as hexadecimal, which is just base 16 (just like binary is base 2). So it wants us to correct the value of val by inputting a number, so I assumed that it just would add our input to the value 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:



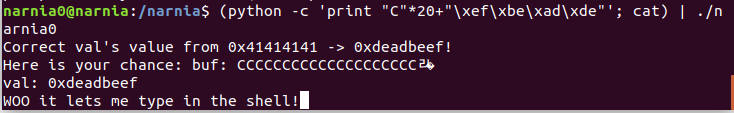
The source code shows that there is no addition happening. So we need to figure out how exactly we can change the value of val even when the program itself does nothing to change val. I did some research and found out about buffer overflows. A buffer overflow is when you exceed the allotted memory bounds 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):

****

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:



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:

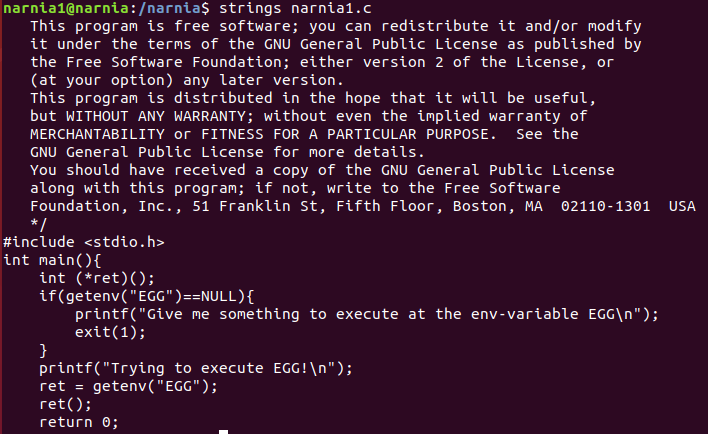


Using the id command we see that we are user narnia1, and we can access the password file now by just doing cat /etc/narnia\_pass/narina1.

**Narnia 1 → 2:**

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

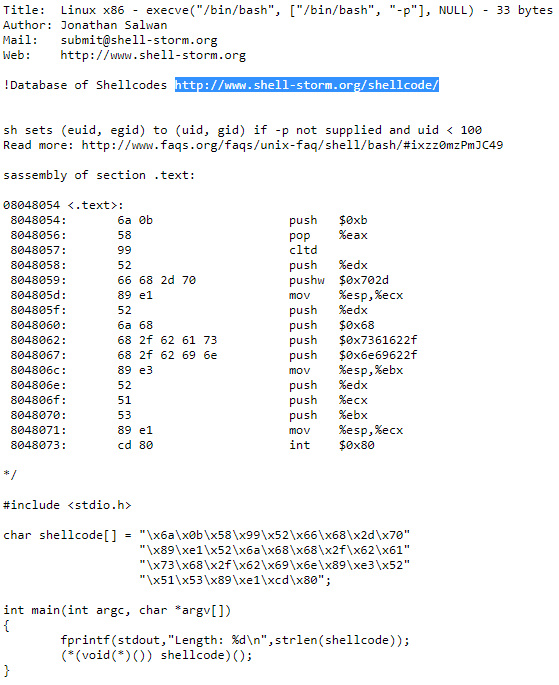
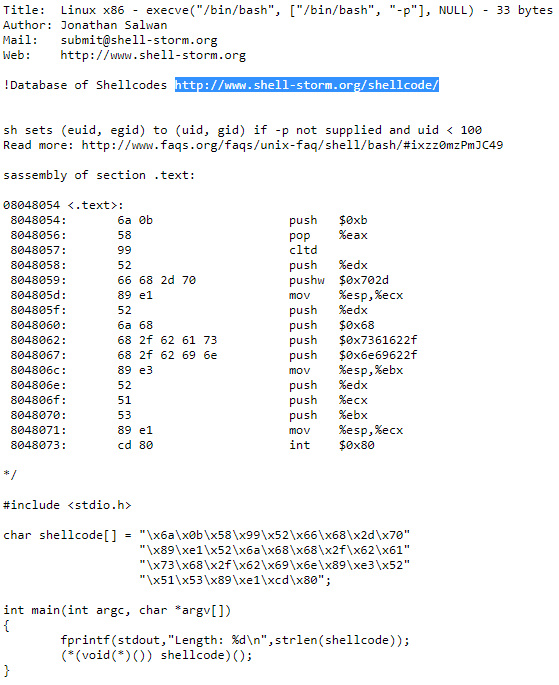




So the program seems to make a function pointer called ret, which points to a function which returns an int, and then the program goes on to make sure the environment variable EGG actually has something in it, then goes on to get the environment variable if it exists. So I tried just throwing something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

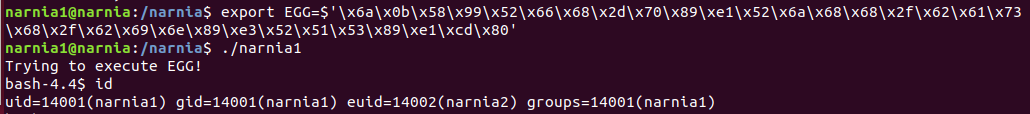


It seems that I ran into the problem that it was trying to execute whatever was in EGG, and because EGG was in reality just a string, nothing was executed. So basically I had to put something executable into EGG. That was when I found shellcode. In a previous guide I used shell script to run and execute 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:



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 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:



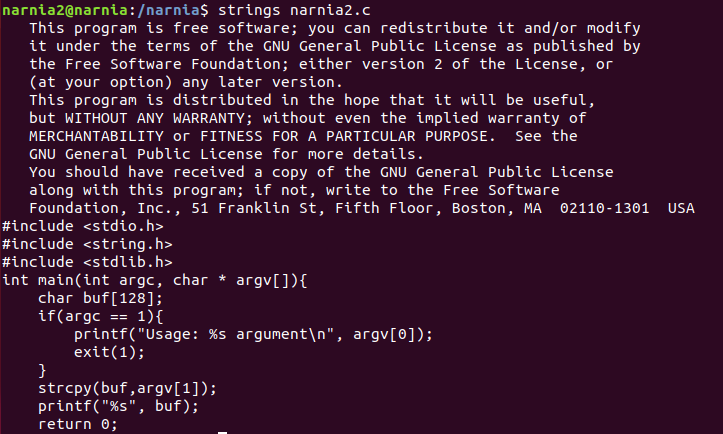
And there we go! That is a bash for the user narnia2, and from there we can just get the password from the password directory, and move on to the next level.

**Narnia 2 → 3:**

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

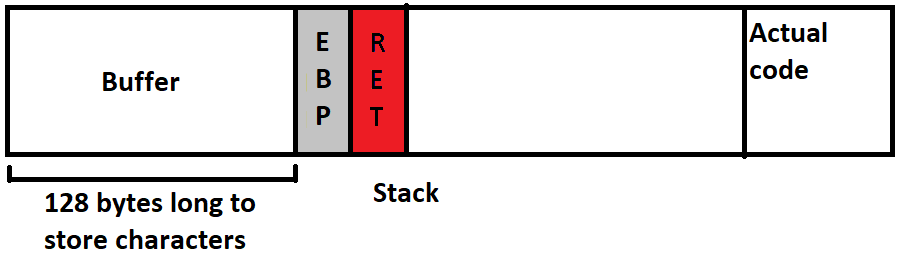


So it's looking for an argument, though I’m not sure what kind of argument so let's look at the source code:



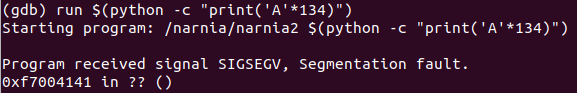
So it looks like it is just taking the argument we pass in, putting it into a buffer, then printing the buffer.

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:

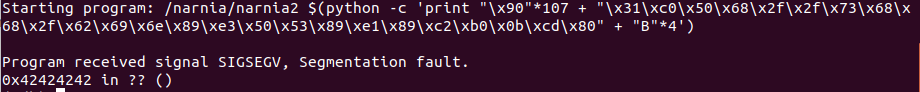


This may look bad because we got a segfault, but this is exactly what we wanted. We know that the hex code for the ASCII character ‘A’ is 41, and when a program segfaults, it shows where the return pointer was pointing at and where it went to get a segfault. We can see that there are two 41’s that are 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

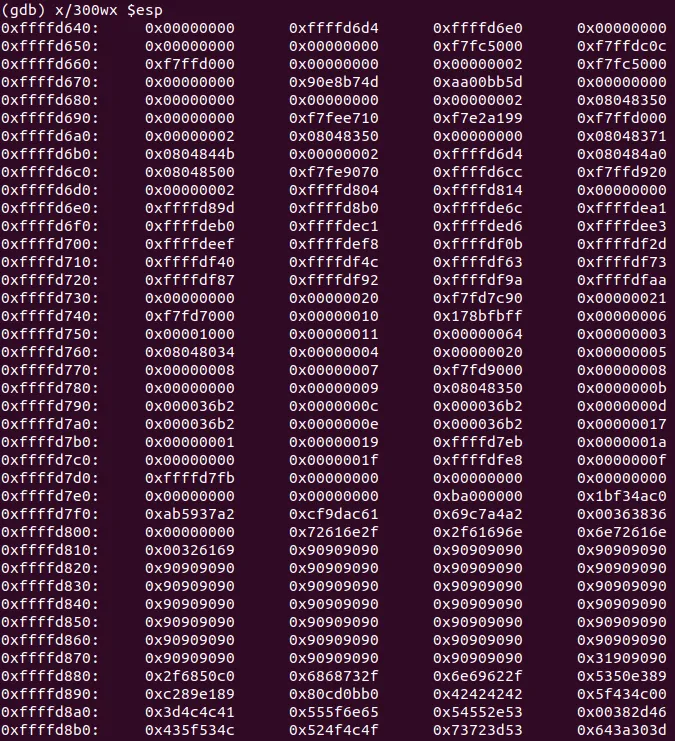
Shellcode (without getting too technical) is a series of “opcodes” that when a computer reads them will execute certain commands. This shell code will execute the shell for us and will allow us to become the user narnia3. So lets make what is called a “payload” which is just our program and some extra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





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

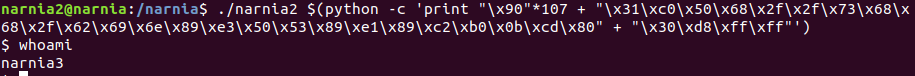
address. We can now take a look at the registers:

****

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:

****

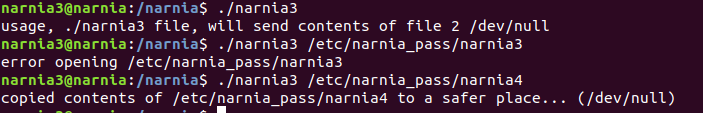
Hey! It seems to work in gdb! Lets try it in the terminal:



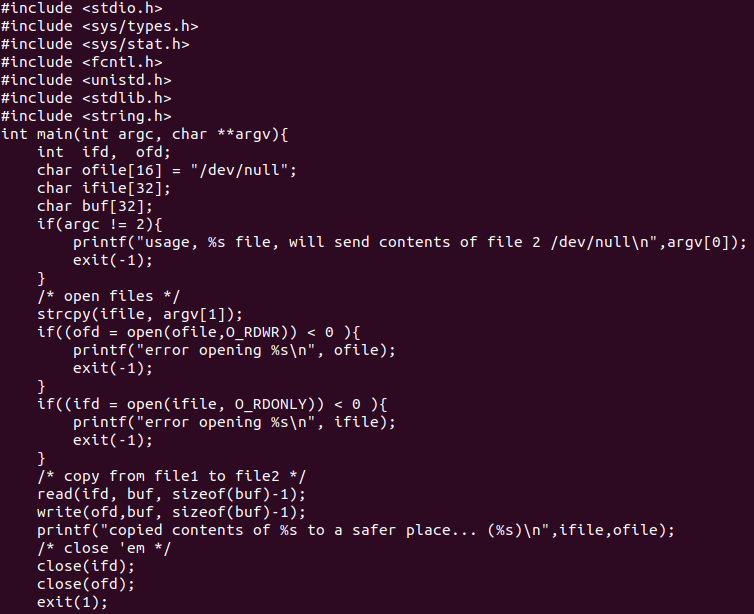
AWESOME! We got a shell! Lets cd to the password directory and get the password for narnia3!

**Narnia 3 → 4:**

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



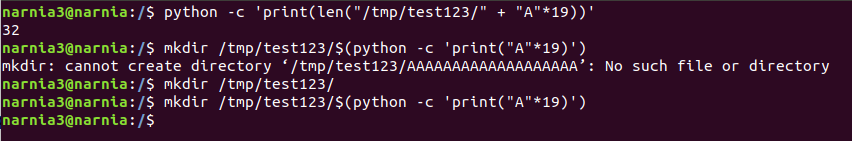
So I’m not sure why it had an error opening the narnia3 password, but when I passed in the narnia4 password file, it says it copies the content of the file to the /dev/null i.e. the null device. For now let's just take a look at the source code:



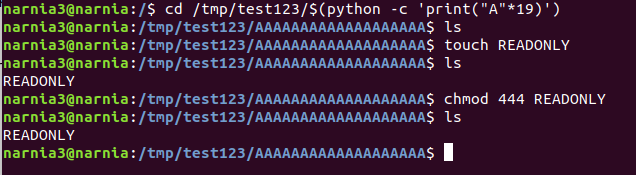
So it seems that the program takes our file name, puts it in a character array called ifile (with a max length of 32 characters) and sends the contents of that file to whatever is in ofile, which in our case is /dev/null. Any data you send to /dev/null otherwise known as the null machine is silently 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?

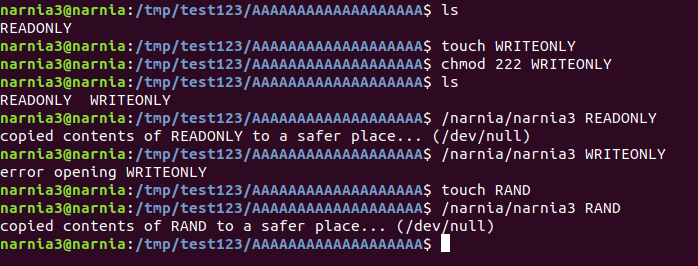
Well our input needs to be bigger than 32 characters, so lets make a payload. We have to first make a file that has a name larger than 32 characters. If we just passed in a string that was longer than 32 characters for a file that doesn’t exist, we will get stuck on the open file check after the strcpy(). 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:



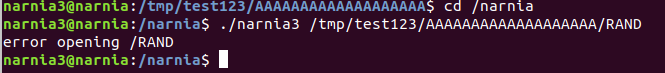
Since the path to whatever file is 32 + the length of the file name, we will always overflow. So now let’s make a file and try to see if it can open it:



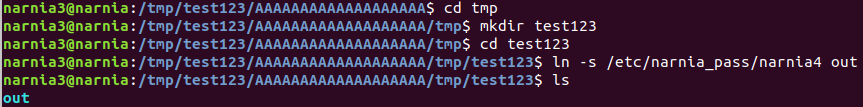
The first file I want to test is one that has chmod code so that it is read only, cause a section of the code checks to make sure the file is read only. Now let's also make a file that is write only, and one that just has base permissions so that we can see if the chmod is necessary:



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:

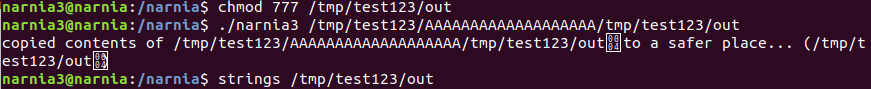


Interesting. It said that it had an error opening RAND, but rand was the part that should have overwritten data. Maybe it isn’t our original file that it is having trouble opening, but the outfile. It's possible that we have overwritten the outfile, and that the /RAND is what overwrote it. What we could 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:





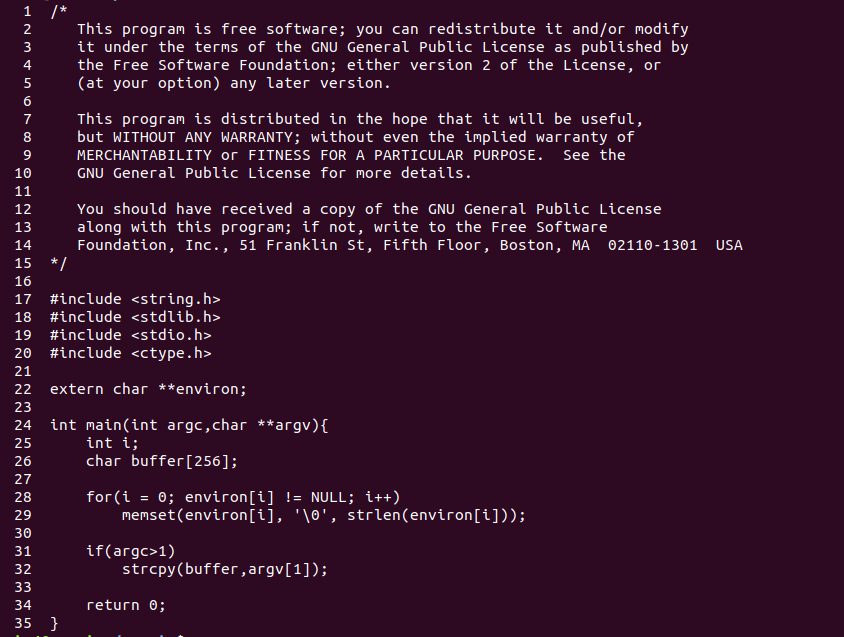
NOTE, you have to make a file **out** for the password to go to.



AND BOOM! This reveals the password!

**Narnia 4 → 5:**

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:



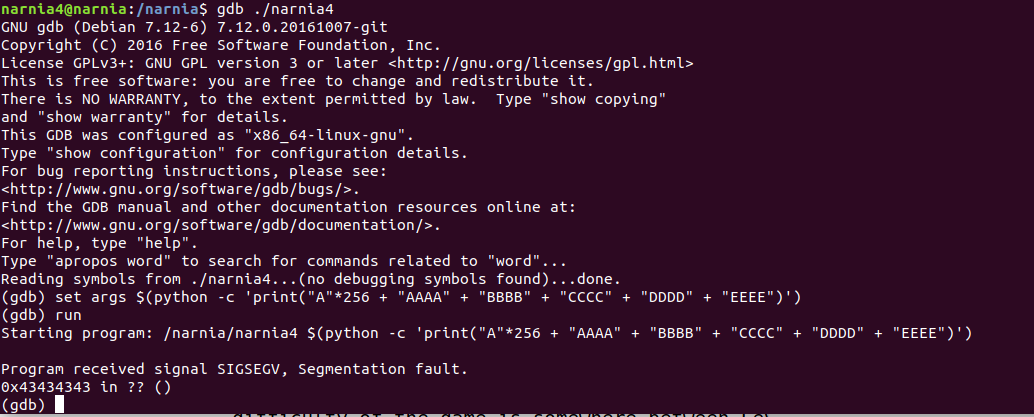
(I learned how to print line numbers)

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:



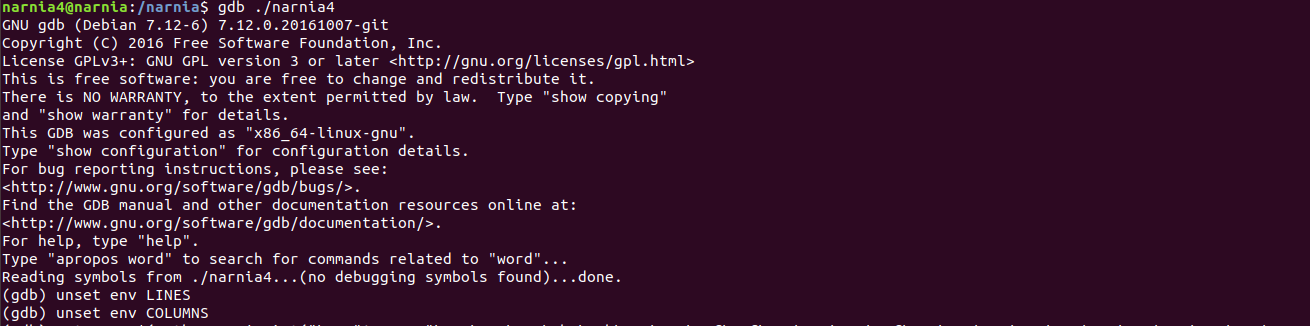
Using that payload we get a seg fault, so let’s take it into gdb and see if we can figure out what exactly is going on:



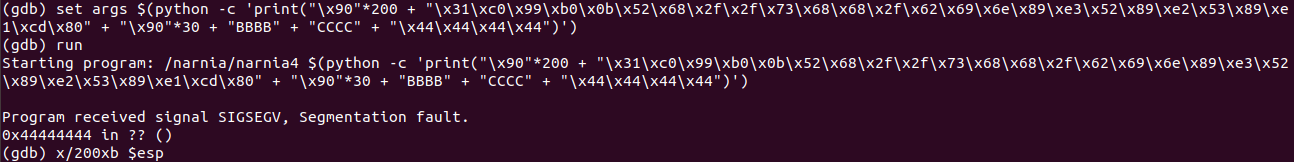
It seems that our string of C’s is overwriting the return value in the program, so I wonder if we can just use some shellcode to try and get a shell.

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.

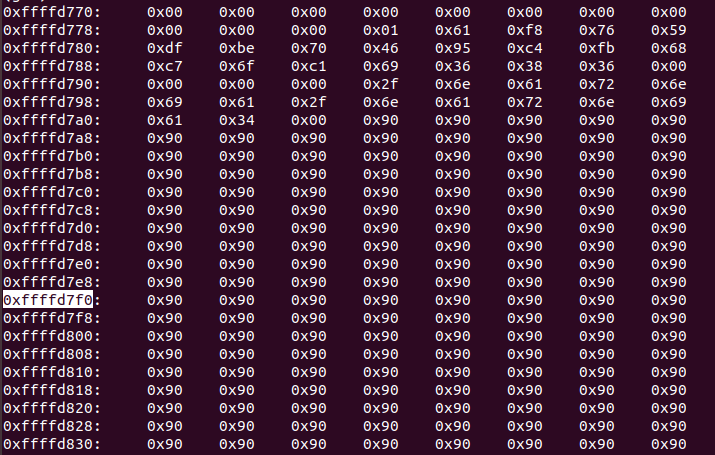
So let's use gdb and see if we can use some shellcode to attain a shell, but first things first let's line up gdb’s execution with execution outside of gdb by running the following commands:



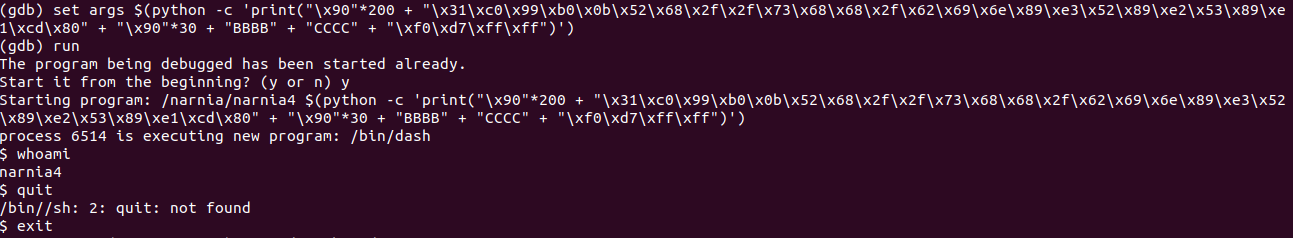
Now we can go onto figure out what our memory address should be that we have to jump to:



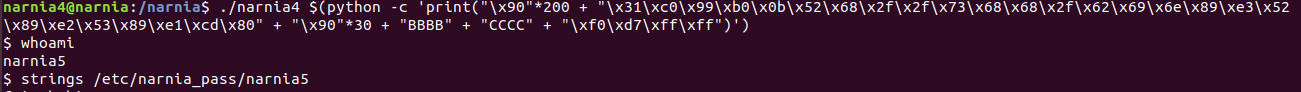
If when you run x/200xb $esp you don’t see your NOP sled, just hit enter a couple times, it will take you through the registers, and you will eventually find your sled:



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



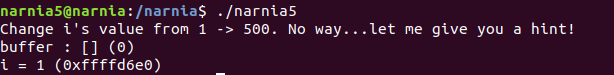
Works in gdb! (it says we are user narnia4 because gdb only has access to what the user running gdb has access too) Let’s see if it works outside of gdb:



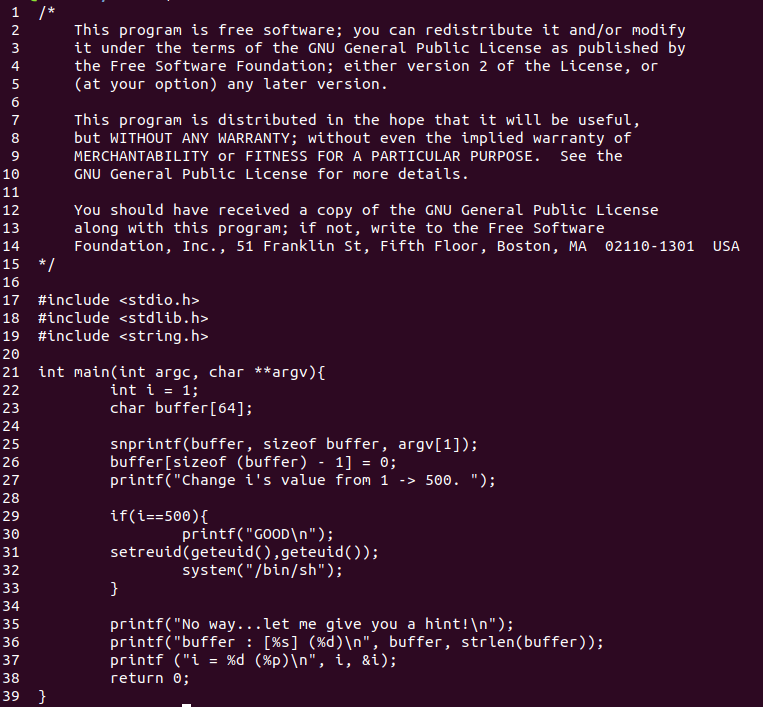
Yay! We got a shell! So I just ran strings to get the password.

**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:

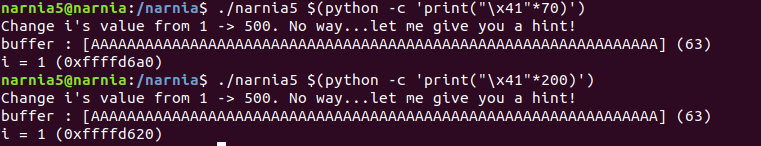


Hmm. “Change i’s value from 1 -> 500.” I’m a little confused by this statement so let's look at the source code:

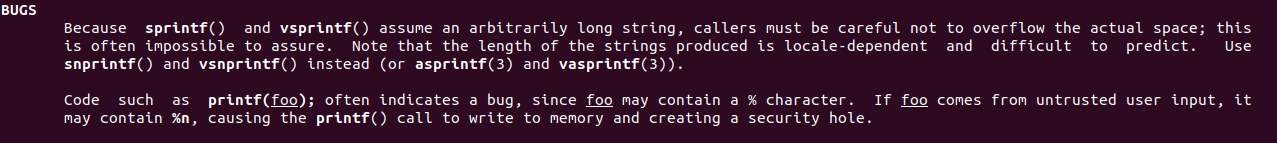


I think I understand now what the original prompt wanted from us, so let’s analyze the code:

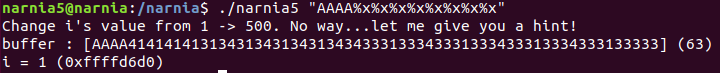
On line 25, the snprintf() function is going to be what is the main problem. All the code before now was using strcpy() to get our characters from the argument into a buffer, but now “the snprintf() function formats and stores a series of characters and values in the array buffer. The snprintf() function 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:



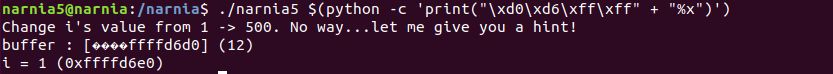
I’m not sure yet how the hint is helpful, but I had an idea. With many functions that I researched before, a lot of places said that the man page for these c functions tell what kind of bugs the functions have, so I decided to check the manpage for snprintf() and see if it had any information about any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



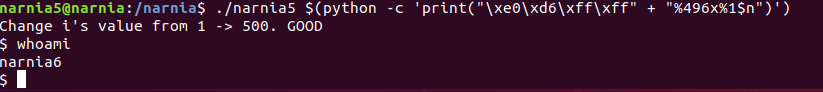
Interesting, so apparently just doing a print with an unknown string foo is dangerous because foo may contain a %n character, which would cause a write to memory? Well when we run the program, it does show us the address of where the variable i is, we can see that both from the code and from running 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:



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:



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:



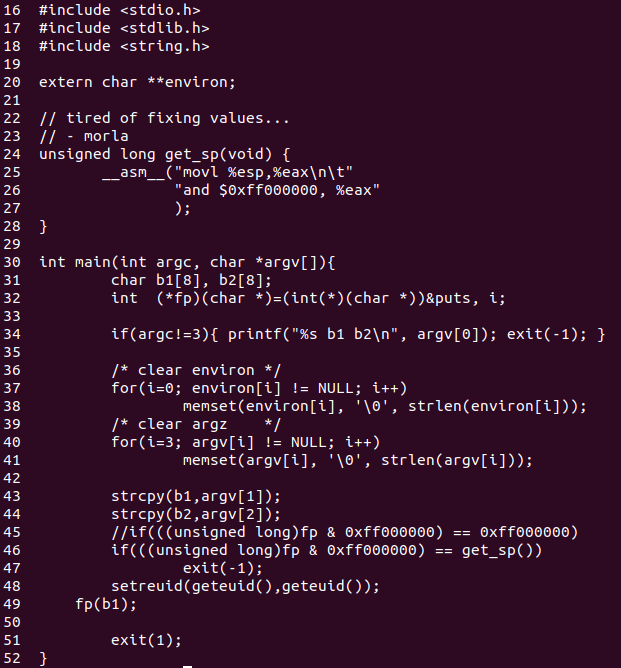
The %496x will print 496 paddings, and with those plus the address we have a total of 500 characters, then the %1$n just means overwrite at the 1st stack address. And look at that we got a shell!

**Narnia 6 → 7:**

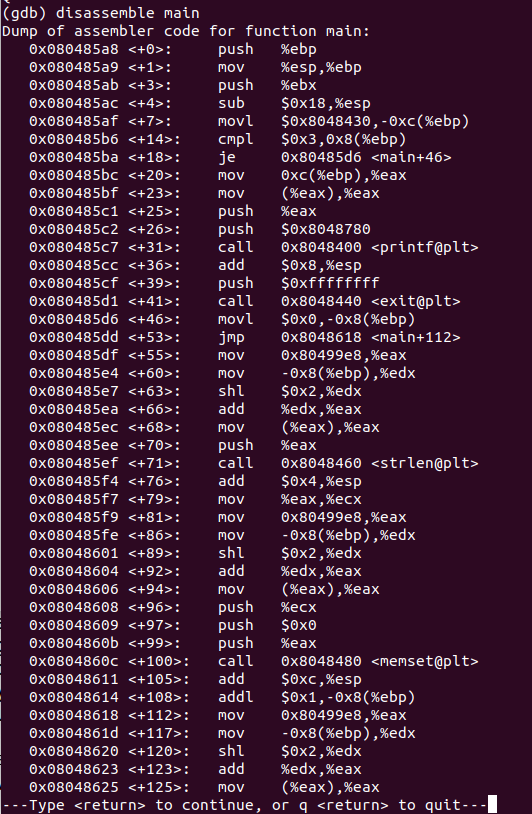
Just like always lets do a dry run of the program, toy around with some inputs, then move onto the source code:



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

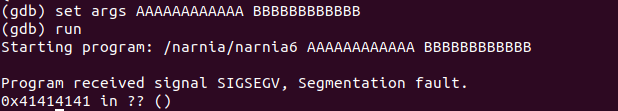


There is a lot going on here in this code that I don’t quite understand, such as the \_\_asm\_\_ function in get\_sp() however I did realize that the program creates a function pointer fp which points to puts. I also see that there is a strcpy() and maybe we could do something with that. It appears that 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:

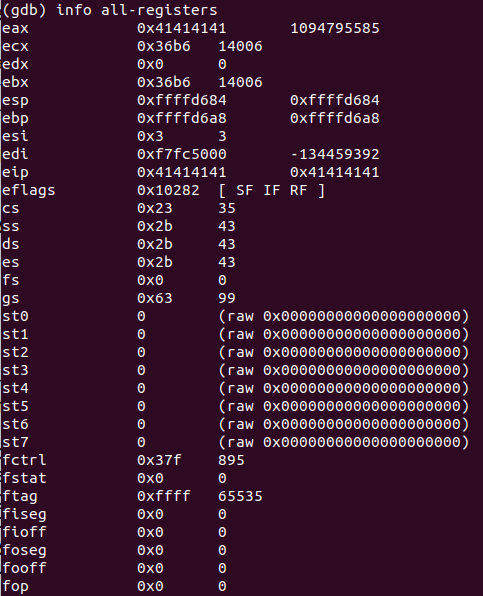


At this point I realized that I don’t know too terribly well what exactly I’m looking at when I read a disassembly, and I found a nice post from medium.com which describes in greater detail exactly what you are looking at when you open up a disassembly: <https://medium.com/@okaleniuk/how-to-read-x86-x64-disassembler-output-ebbbeb2ddf02>

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:



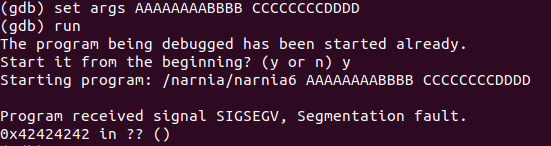
So it seems that the string we input b1 overruns the return value. Let's look at some registers and see if anything there was overwritten by b2:

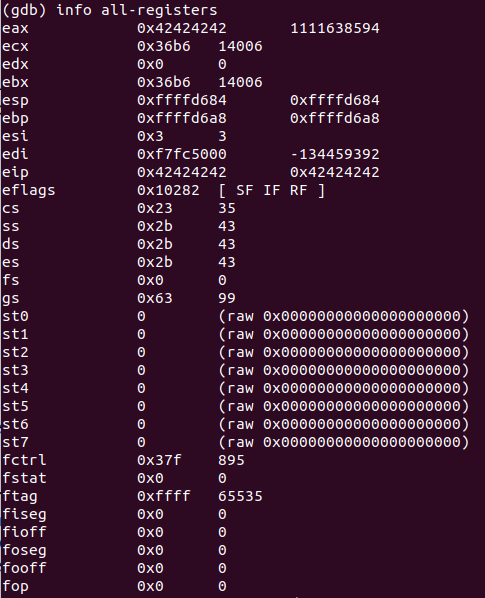


Interestingly enough, nothing here was overwritten by B’s, as there isn’t anything that's just written as a ton of 42’s, but it does seem that we have overwritten the eax register with all our A’s. The eax register is used in input/output and arithmetic instructions, is there any way we can utilize either 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:



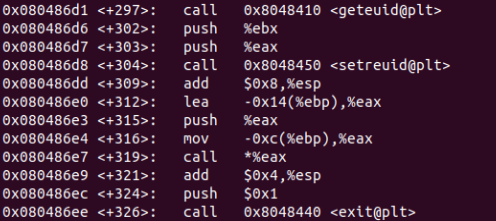
And there they are, along with a couple of our A’s. Let’s try using a different payload so we can see exactly which A’s are overwriting eax and which are overwriting eip:

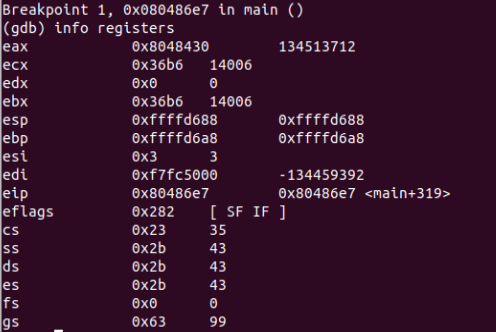




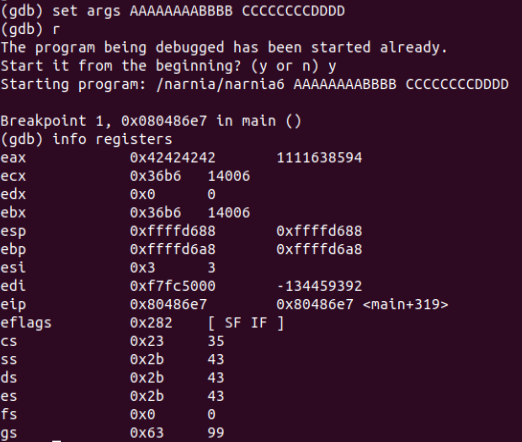
Well now I am super confused… How are the four B’s overwriting both the eax pointer and eip pointer? I had to look up some help as I have been stuck on this problem for about 4 days and I think I understand what is happening now. I will link where I found help but this blog spot post does a really good 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:





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



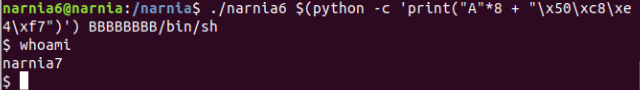
So here we see that the pointer has been overwritten and directed to 0x42424242 which is the hex code for ‘BBBB’. So now let's try and change the BBBB into something useful, some useful function call. Nicola Gatta suggests using system() to call for a shell (/bin/sh). So let's do that and overwrite the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



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

****

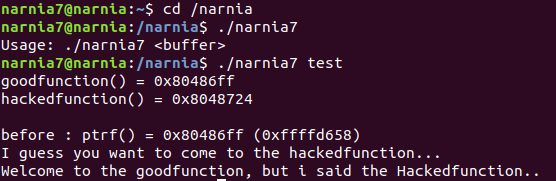
There we go! Now we can just replace our DDDD with the shell location:



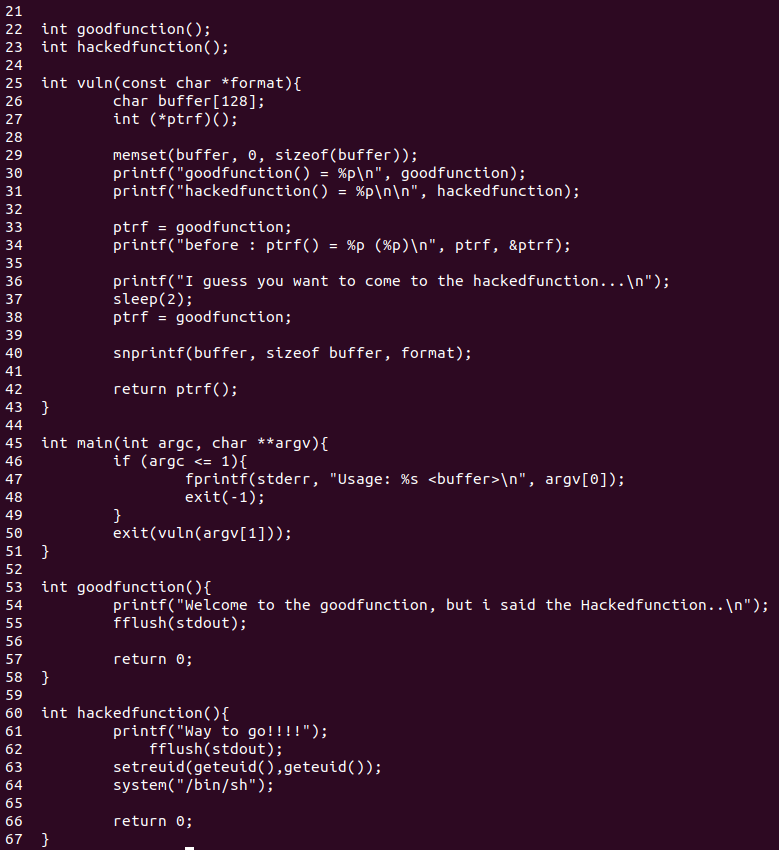
And now we have a shell!!!

**Narnia 7 → 8:**

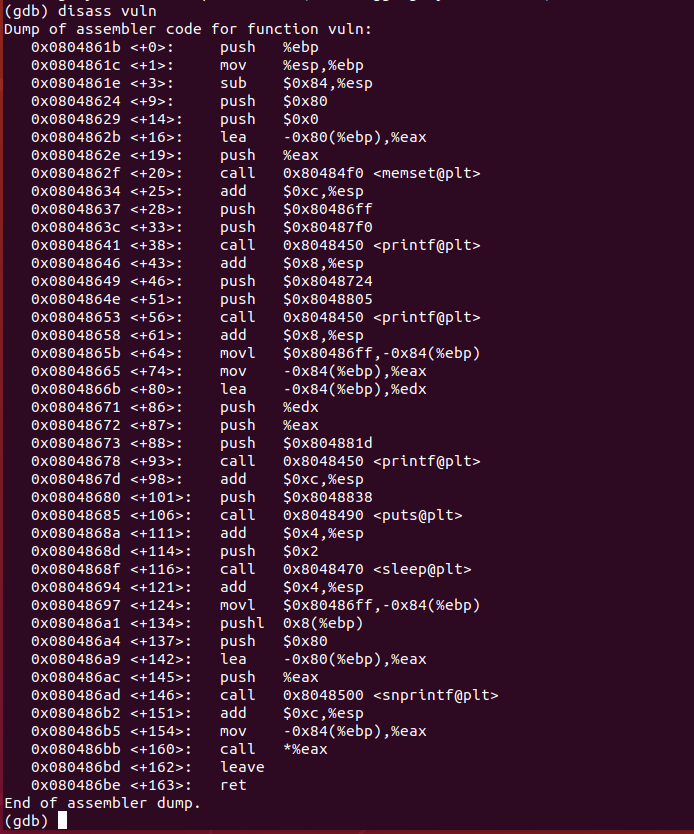
Let’s start with our usual test run:



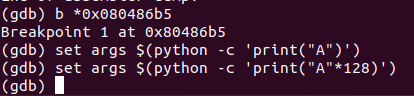
Okay let’s take a look at the code:



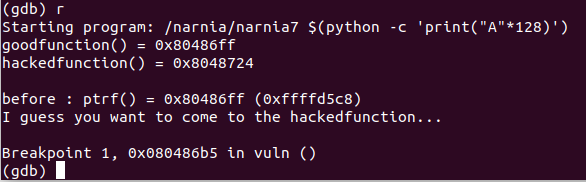
So after taking a good look at the code, we know that the only thing that we really control is the argument that gets passed into the function vuln(), and the fact is that we cannot just go ahead an overwrite the function pointer as snprintf() prevents us from writing pass the bounds of the memory available 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:



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



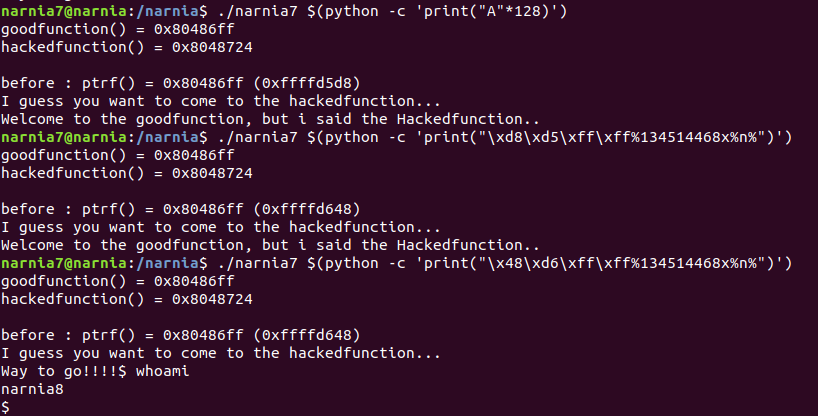
Now we want to run it and when we hit the breakpoint, it is important we notice that it shows us the address of ptrf:



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%”

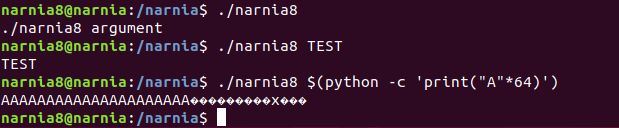
So let's run the program out of gdb and get our address to develop our payload:



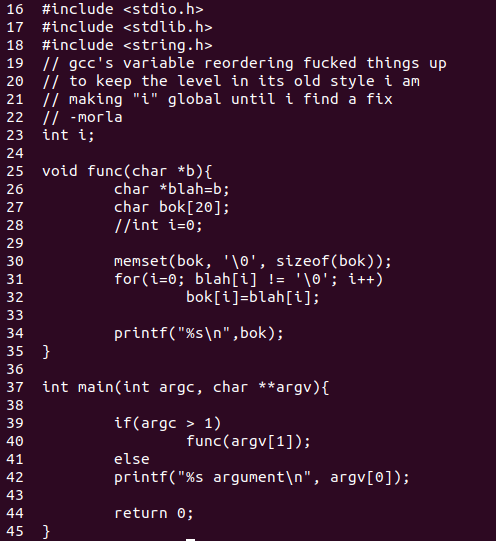
The first test I did with the 128 A’s was a mistake as I realized the string I put in should have been the length that my payload was going to be. But after I figured that out WE GOT IT!

**Narnia 8 → 9:**

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



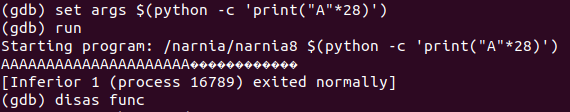
So on the surface level it looks like that it just takes our string that we pass as an argument and prints it. It also looks like that our string has a limit to how big it can be, but we didn’t achieve a seg fault so we can assume that our string of 64 didn’t overwrite the return value, so maybe there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

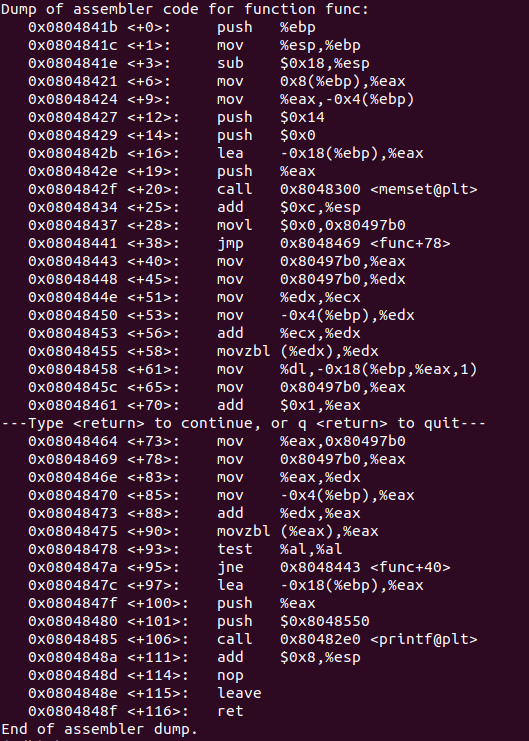


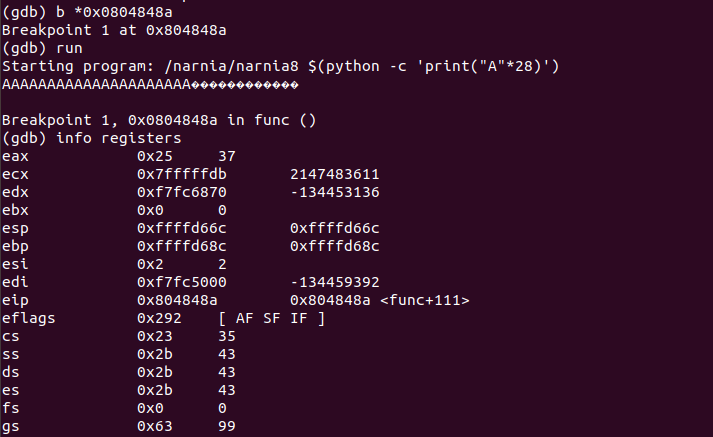
So it looks like what the code is actually doing is taking our input argument and copying it into a character array called blah. Then from there blah gets copied into bok 1 character at a time, and goes until we reach the null character of blah, signifying the end of the string. Then afterwards, it prints 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:



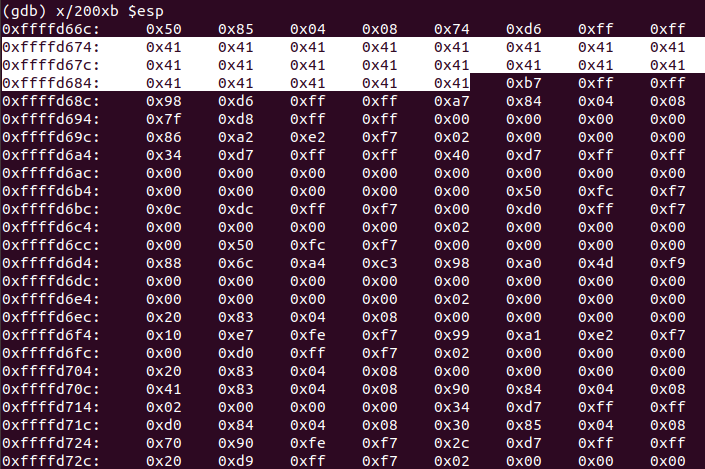
So clearly the vulnerability must lie in the function func, as they irresponsibly copy our entire argument into the character array bok, and if we have a string longer than bok’s 20 character bounds, we will overwrite memory that doesn’t belong to the array. Ideally I think we should aim to overwrite 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:







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:



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.