

# Behemoth Level - 5

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 Class:

**Insecure data exfiltration via local network socket (logic flaw / information disclosure)**

The binary reads a privileged file and **transmits its contents over an unauthenticated local TCP socket** without encryption or access control.

### Core Concept:

**Privileged file read + plaintext network transmission = passive credential disclosure**

Instead of preventing access, the program:

- successfully reads `/etc/behemoth_pass/behemoth6`
- then sends the contents to `localhost` over a predictable port
- assuming no attacker is listening

## Methodology:

### 1. Static & Dynamic Analysis

- Disassembled `main()` in GDB
- Identified `fopen()` reading `/etc/behemoth_pass/behemoth6`
- Traced file handling ( `fseek` , `ftell` , `rewind` , `malloc` , `fgets` )
- Observed network-related libc calls:
  - `gethostbyname()`
  - `socket()`
  - `htons()`
  - `sendto()`

### 2. Network Behavior Identification

- Resolved target host: `localhost`
- Extracted hardcoded port value (via `atoi` + `htons` )
- Confirmed TCP socket usage
- Determined plaintext transmission (no encoding / encryption)

### 3. Exploit Strategy

- Let the binary run normally (no need to interfere with file access)
- Passively listen on the target port
- Capture the password as it is transmitted over the loopback interface

## Final Working Payload:

### Terminal 1 — Listener

```
nc -lvnp 1337
```

## Terminal 2 — Execute binary

```
./behemoth5
```

### Result:

The password for `behemoth6` is received directly via the listening socket.

## 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 `./behemoth5` script and then we will see as to how it behaves:

```
behemoth5@behemoth:~$ cd /behemoth
behemoth5@behemoth:/behemoth$ ls
behemoth0 behemoth1 behemoth2 behemoth3 behemoth4 behemoth5 behemoth6 behemoth6_reader behemoth7
behemoth5@behemoth:/behemoth$ ./behemoth5
behemoth5@behemoth:/behemoth$ ltrace ./behemoth5
__libc_start_main(0x804917d, 1, 0xffffd464, 0 <unfinished ...>
fopen("/etc/behemoth_pass/behemoth6", "r") = 0
perror("fopen"fopen: Permission denied
) = <void>
exit(1 <no return ...>
+++ exited (status 1) +++
behemoth5@behemoth:/behemoth$ |
```

The program tries to read the contents of `/etc/behemoth_pass/behemoth6` [which we want to access as well] but since the file: `/etc/behemoth_pass/behemoth6` require `behemoth6` permissions to read, the program fails to read it and quits with exit code 1.

Let's analyze the same in gdb and disassemble the main function:

gdb behemoth5  
(gdb) disassemble main

```
behemoth5@behemoth:/behemoth$ gdb behemoth5
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from behemoth5...

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Download failed: Permission denied. Continuing without separate debug info for /behemoth/behemoth5.
(No debugging symbols found in behemoth5)
(gdb) disassemble main
Dump of assembler code for function main:
   0x08049266 <+0>:    lea     0x4(%esp),%ecx
   0x0804926a <+4>:    and     $0xfffffffff0,%esp
   0x0804926d <+7>:    push    -0x4(%ecx)
   0x08049270 <+10>:   push    %ebp
   0x08049271 <+11>:   mov     %esp,%ebp
   0x08049273 <+13>:   push    %ecx
   0x08049274 <+14>:   sub     $0x44,%esp
```

```

0x08049277 <+17>:    mov     %ecx,%eax
0x08049279 <+19>:    mov     0x4(%eax),%eax
0x0804927c <+22>:    mov     %eax,-0x3c(%ebp)
0x0804927f <+25>:    mov     %gs:0x14,%eax
0x08049285 <+31>:    mov     %eax,-0xc(%ebp)
0x08049288 <+34>:    xor     %eax,%eax
0x0804928a <+36>:    movl    $0x0,-0x34(%ebp)
0x08049291 <+43>:    sub     $0x8,%esp
0x08049294 <+46>:    push    $0x804a008
0x08049299 <+51>:    push    $0x804a00a
0x0804929e <+56>:    call    0x80490e0 <fopen@plt>
0x080492a3 <+61>:    add     $0x10,%esp
0x080492a6 <+64>:    mov     %eax,-0x30(%ebp)
0x080492a9 <+67>:    cmpl    $0x0,-0x30(%ebp)
0x080492ad <+71>:    jne     0x80492c9 <main+99>
0x080492af <+73>:    sub     $0xc,%esp
0x080492b2 <+76>:    push    $0x804a027
0x080492b7 <+81>:    call    0x8049090 <perror@plt>
0x080492bc <+86>:    add     $0x10,%esp
0x080492bf <+89>:    sub     $0xc,%esp
0x080492c2 <+92>:    push    $0x1
0x080492c4 <+94>:    call    0x80490b0 <exit@plt>
0x080492c9 <+99>:    sub     $0x4,%esp
0x080492cc <+102>:   push    $0x2
0x080492ce <+104>:   push    $0x0
0x080492d0 <+106>:   push    -0x30(%ebp)
--Type <RET> for more, q to quit, c to continue without paging--c
0x080492d3 <+109>:   call    0x8049080 <fseek@plt>
0x080492d8 <+114>:   add     $0x10,%esp
0x080492db <+117>:   sub     $0xc,%esp
0x080492de <+120>:   push    -0x30(%ebp)
0x080492e1 <+123>:   call    0x80490d0 <ftell@plt>
0x080492e6 <+128>:   add     $0x10,%esp
0x080492e9 <+131>:   mov     %eax,-0x34(%ebp)
0x080492ec <+134>:   addl    $0x1,-0x34(%ebp)

```

```

0x080492f0 <+138>:  sub    $0xc,%esp
0x080492f3 <+141>:  push   -0x30(%ebp)
0x080492f6 <+144>:  call   0x8049060 <rewind@plt>
0x080492fb <+149>:  add    $0x10,%esp
0x080492fe <+152>:  mov    -0x34(%ebp),%eax
0x08049301 <+155>:  sub    $0xc,%esp
0x08049304 <+158>:  push   %eax
0x08049305 <+159>:  call   0x80490a0 <malloc@plt>
0x0804930a <+164>:  add    $0x10,%esp
0x0804930d <+167>:  mov    %eax,-0x2c(%ebp)
0x08049310 <+170>:  sub    $0x4,%esp
0x08049313 <+173>:  push   -0x30(%ebp)
0x08049316 <+176>:  push   -0x34(%ebp)
0x08049319 <+179>:  push   -0x2c(%ebp)
0x0804931c <+182>:  call   0x8049040 <fgets@plt>
0x08049321 <+187>:  add    $0x10,%esp
0x08049324 <+190>:  sub    $0xc,%esp
0x08049327 <+193>:  push   -0x2c(%ebp)
0x0804932a <+196>:  call   0x80490c0 <strlen@plt>
0x0804932f <+201>:  add    $0x10,%esp
0x08049332 <+204>:  mov    -0x2c(%ebp),%edx
0x08049335 <+207>:  add    %edx,%eax
0x08049337 <+209>:  movb   $0x0,(%eax)
0x0804933a <+212>:  sub    $0xc,%esp
0x0804933d <+215>:  push   -0x30(%ebp)
0x08049340 <+218>:  call   0x8049050 <fclose@plt>
0x08049345 <+223>:  add    $0x10,%esp
0x08049348 <+226>:  sub    $0xc,%esp
0x0804934b <+229>:  push   $0x804a02d
0x08049350 <+234>:  call   0x8049130 <gethostbyname@plt>
0x08049355 <+239>:  add    $0x10,%esp
0x08049358 <+242>:  mov    %eax,-0x28(%ebp)
0x0804935b <+245>:  cmpl   $0x0,-0x28(%ebp)
0x0804935f <+249>:  jne    0x804937b <main+277>
0x08049361 <+251>:  sub    $0xc,%esp

```

```

0x08049361 <+251>: sub    $0xc,%esp
0x08049364 <+254>: push   $0x804a037
0x08049369 <+259>: call   0x8049090 <perror@plt>
0x0804936e <+264>: add    $0x10,%esp
0x08049371 <+267>: sub    $0xc,%esp
0x08049374 <+270>: push   $0x1
0x08049376 <+272>: call   0x80490b0 <exit@plt>
0x0804937b <+277>: sub    $0x4,%esp
0x0804937e <+280>: push   $0x0
0x08049380 <+282>: push   $0x2
0x08049382 <+284>: push   $0x2
0x08049384 <+286>: call   0x8049120 <socket@plt>
0x08049389 <+291>: add    $0x10,%esp
0x0804938c <+294>: mov    %eax,-0x24(%ebp)
0x0804938f <+297>: cmpl   $0xffffffff,-0x24(%ebp)
0x08049393 <+301>: jne    0x80493af <main+329>
0x08049395 <+303>: sub    $0xc,%esp
0x08049398 <+306>: push   $0x804a045
0x0804939d <+311>: call   0x8049090 <perror@plt>
0x080493a2 <+316>: add    $0x10,%esp
0x080493a5 <+319>: sub    $0xc,%esp
0x080493a8 <+322>: push   $0x1
0x080493aa <+324>: call   0x80490b0 <exit@plt>
0x080493af <+329>: movw   $0x2,-0x1c(%ebp)
0x080493b5 <+335>: sub    $0xc,%esp
0x080493b8 <+338>: push   $0x804a04c
0x080493bd <+343>: call   0x8049110 <atoi@plt>
0x080493c2 <+348>: add    $0x10,%esp
0x080493c5 <+351>: movzwl %ax,%eax
0x080493c8 <+354>: sub    $0xc,%esp
0x080493cb <+357>: push   %eax
0x080493cc <+358>: call   0x8049070 <htons@plt>
0x080493d1 <+363>: add    $0x10,%esp
0x080493d4 <+366>: mov    %ax,-0x1a(%ebp)
0x080493d8 <+370>: mov    -0x28(%ebp),%eax

```

```

0x080493d8 <+370>:  mov    -0x28(%ebp),%eax
0x080493db <+373>:  mov    0x10(%eax),%eax
0x080493de <+376>:  mov    (%eax),%eax
0x080493e0 <+378>:  mov    (%eax),%eax
0x080493e2 <+380>:  mov    %eax,-0x18(%ebp)
0x080493e5 <+383>:  sub    $0x4,%esp
0x080493e8 <+386>:  push   $0x8
0x080493ea <+388>:  push   $0x0
0x080493ec <+390>:  lea    -0x1c(%ebp),%eax
0x080493ef <+393>:  add    $0x8,%eax
0x080493f2 <+396>:  push   %eax
0x080493f3 <+397>:  call   0x80490f0 <memset@plt>
0x080493f8 <+402>:  add    $0x10,%esp
0x080493fb <+405>:  sub    $0xc,%esp
0x080493fe <+408>:  push   -0x2c(%ebp)
0x08049401 <+411>:  call   0x80490c0 <strlen@plt>
0x08049406 <+416>:  add    $0x10,%esp
0x08049409 <+419>:  sub    $0x8,%esp
0x0804940c <+422>:  push   $0x10
0x0804940e <+424>:  lea    -0x1c(%ebp),%edx
0x08049411 <+427>:  push   %edx
0x08049412 <+428>:  push   $0x0
0x08049414 <+430>:  push   %eax
0x08049415 <+431>:  push   -0x2c(%ebp)
0x08049418 <+434>:  push   -0x24(%ebp)
0x0804941b <+437>:  call   0x8049100 <sendto@plt>
0x08049420 <+442>:  add    $0x20,%esp
0x08049423 <+445>:  mov    %eax,-0x20(%ebp)
0x08049426 <+448>:  cmpl   $0xffffffff,-0x20(%ebp)
0x0804942a <+452>:  jne    0x8049446 <main+480>
0x0804942c <+454>:  sub    $0xc,%esp
0x0804942f <+457>:  push   $0x804a051
0x08049434 <+462>:  call   0x8049090 <perror@plt>
0x08049439 <+467>:  add    $0x10,%esp
0x0804943c <+470>:  sub    $0xc,%esp

```

```

0x0804943c <+470>:  sub    $0xc,%esp
0x0804943f <+473>:  push   $0x1
0x08049441 <+475>:  call   0x80490b0 <exit@plt>
0x08049446 <+480>:  sub    $0xc,%esp
0x08049449 <+483>:  push   -0x24(%ebp)
0x0804944c <+486>:  call   0x8049140 <close@plt>
0x08049451 <+491>:  add    $0x10,%esp
0x08049454 <+494>:  sub    $0xc,%esp
0x08049457 <+497>:  push   $0x0
0x08049459 <+499>:  call   0x80490b0 <exit@plt>
End of assembler dump.
(gdb) |

```

Quite a large main function, in fact if I'm not wrong largest disassembling ever in all the OverTheWire series I have attempted so far.

Here's the interesting bit at our disposal though:

```

0x08049288 <+34>:  xor     %eax,%eax
0x0804928a <+36>:  movl    $0x0,-0x34(%ebp)
0x08049291 <+43>:  sub     $0x8,%esp
0x08049294 <+46>:  push    $0x804a008
0x08049299 <+51>:  push    $0x804a00a
0x0804929e <+56>:  call    0x80490e0 <fopen@plt>
0x080492a3 <+61>:  add     $0x10,%esp
0x080492a6 <+64>:  mov     %eax,-0x30(%ebp)
0x080492a9 <+67>:  cmpl    $0x0,-0x30(%ebp)
0x080492ad <+71>:  jne     0x80492c9 <main+99>
0x080492af <+73>:  sub     $0xc,%esp
0x080492b2 <+76>:  push    $0x804a027

```

```

0x08049294 <+46>:  push    $0x804a008
0x08049299 <+51>:  push    $0x804a00a
0x0804929e <+56>:  call    0x80490e0 <fopen@plt>

```

So this `fopen` function definitely accesses the behemoth6 file, let's see what the two registers `0x804a008` and `0x804a00a` store values as:

```

End of assembler dump.
(gdb) x/s 0x804a008
0x804a008:      "r"
(gdb) x/s 0x804a00a
0x804a00a:      "/etc/behemoth_pass/behemoth6"
(gdb) |

```

So;

`0x804a00a` → stores the behemoth6 password file

`0x804a008` → "r" is the read permission

Alright, the program is quite lengthy, I will look up everything I don't understand clearly and also establish talking point for the one's equally lost and overwhelmed as me:

```

0x080492ad <+71>:    jne     0x080492c9 <main+99>
0x080492af <+73>:    sub     $0xc,%esp
0x080492b2 <+76>:    push    $0x804a027
0x080492b7 <+81>:    call    0x08049090 <perror@plt>
0x080492bc <+86>:    add     $0x10,%esp
0x080492bf <+89>:    sub     $0xc,%esp
0x080492c2 <+92>:    push    $0x1
0x080492c4 <+94>:    call    0x080490b0 <exit@plt>

```

`<perror@plt>` is for error generation it prints: `s: <human-readable error message>` based on the current value of `errno`.

And if the error exists it exits: `<exit@plt>`

Next, this:

```

--Type <RET> for more, q to quit, c to continue without paging--c
0x080492d3 <+109>:   call    0x08049080 <fseek@plt>
0x080492d8 <+114>:   add     $0x10,%esp
0x080492db <+117>:   sub     $0xc,%esp
0x080492de <+120>:   push    -0x30(%ebp)
0x080492e1 <+123>:   call    0x080490d0 <ftell@plt>

```

If the program can read the file it gets the file contents using `<fseek@plt>` and writes it using `<ftell@plt>`

Next this one:

```
0x080492f6 <+144>: call 0x8049060 <rewind@plt>
0x080492fb <+149>: add $0x10,%esp
0x080492fe <+152>: mov -0x34(%ebp),%eax
0x08049301 <+155>: sub $0xc,%esp
0x08049304 <+158>: push %eax
0x08049305 <+159>: call 0x80490a0 <malloc@plt>
0x0804930a <+164>: add $0x10,%esp
0x0804930d <+167>: mov %eax,-0x2c(%ebp)
0x08049310 <+170>: sub $0x4,%esp
0x08049313 <+173>: push -0x30(%ebp)
0x08049316 <+176>: push -0x34(%ebp)
0x08049319 <+179>: push -0x2c(%ebp)
0x0804931c <+182>: call 0x8049040 <fgets@plt>
0x08049321 <+187>: add $0x10,%esp
0x08049324 <+190>: sub $0xc,%esp
0x08049327 <+193>: push -0x2c(%ebp)
0x0804932a <+196>: call 0x80490c0 <strlen@plt>
```

I don't know what is `<rewind@plt>`, let's see if we can read it:

```
End of assembler dump.
(gdb) x/s 0x804a008
0x804a008: "r"
(gdb) x/s 0x804a00a
0x804a00a: "/etc/behemoth_pass/behemoth6"
(gdb) x/s 0x8049060
0x8049060 <rewind@plt>: "\377%\f\300\004\bh\030"
(gdb)
```

That does not make a lot of sense now does it?

Okay so here's what it does:

- Sets the file position indicator to the **beginning of the file**

- Clears the error and EOF flags for that stream

Equivalent to:

```
fseek(stream,0, SEEK_SET);
clearerr(stream);
```

`<malloc@plt>` must be storing the values.

`<fgets@plt>` must be used for standard input i.e. the behemoth6 password file.

`<strlen@plt>` is used to get the length of string, pretty basic.

Next:

```
0x08049340 <+218>: call    0x8049050 <fclose@plt>
0x08049345 <+223>: add     $0x10,%esp
0x08049348 <+226>: sub     $0xc,%esp
0x0804934b <+229>: push    $0x804a02d
0x08049350 <+234>: call    0x8049130 <gethostbyname@plt>
0x08049355 <+239>: add     $0x10,%esp
0x08049358 <+242>: mov     %eax,-0x28(%ebp)
0x0804935b <+245>: cmpl    $0x0,-0x28(%ebp)
0x0804935f <+249>: jne     0x804937b <main+277>
0x08049361 <+251>: sub     $0xc,%esp
0x08049364 <+254>: push    $0x804a037
0x08049369 <+259>: call    0x8049090 <perror@plt>
0x0804936e <+264>: add     $0x10,%esp
0x08049371 <+267>: sub     $0xc,%esp
0x08049374 <+270>: push    $0x1
0x08049376 <+272>: call    0x80490b0 <exit@plt>
0x0804937b <+277>: sub     $0x4,%esp
0x0804937e <+280>: push    $0x0
0x08049380 <+282>: push    $0x2
0x08049382 <+284>: push    $0x2
0x08049384 <+286>: call    0x8049120 <socket@plt>
0x08049389 <+291>: add     $0x10,%esp
0x0804938c <+294>: mov     %eax,-0x24(%ebp)
0x0804938f <+297>: cmpl    $0xffffffff,-0x24(%ebp)
0x08049393 <+301>: jne     0x80493af <main+329>
0x08049395 <+303>: sub     $0xc,%esp
0x08049398 <+306>: push    $0x804a045
0x0804939d <+311>: call    0x8049090 <perror@plt>
```

This one is quite interesting than before:

`<fclose@plt>` simply quits or closes the file i.e. behemoth6.

`<gethostbyname@plt>` I genuinely don't know what this is either, let's look it up:

```
End of assembler dump.
(gdb) x/s 0x804a008
0x804a008:      "r"
(gdb) x/s 0x804a00a
0x804a00a:      "/etc/behemoth_pass/behemoth6"
(gdb) x/s 0x8049060
0x8049060 <rewind@plt>: "\377%\f\300\004\bh\030"
(gdb) x/s 0x8049130
0x8049130 <gethostbyname@plt>:  "\377%\@300\004\bh\200"
(gdb) |
```

This is not readable as well, okay so here's the misconception I had- we can only read registers with push instruction not `call`, not `jmp` not `mov` no other only `push` let's see some register before or after this:

```
End of assembler dump.
(gdb) x/s 0x804a008
0x804a008:      "r"
(gdb) x/s 0x804a00a
0x804a00a:      "/etc/behemoth_pass/behemoth6"
(gdb) x/s 0x8049060
0x8049060 <rewind@plt>: "\377%\f\300\004\bh\030"
(gdb) x/s 0x8049130
0x8049130 <gethostbyname@plt>:  "\377%\@300\004\bh\200"
(gdb) x/s 0x804a02d
0x804a02d:      "localhost"
(gdb) |
```

So `localhost` is being used, I don't know why, but it is being used- so there must be some network related activity carried out by the program, here's what

`<gethostbyname@plt>` does:

- Takes a hostname (e.g. `"example.com"` )
- Resolves it to IP address(es)
- Returns a `struct hostent *`

Example:

```
struct hostent *h = gethostbyname("google.com");
```

Notes:

- Uses **DNS** / `/etc/hosts`
- **Not thread-safe**

I think it might be sending the contents of behemoth6 password file to the localhost, because the next interesting function executed is:

```
<socket@plt>
```

**socket(2)**

System Calls Manual

**socket(2)**

## NAME [top](#)

socket - create an endpoint for communication

## LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

## SYNOPSIS [top](#)

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

## DESCRIPTION [top](#)

**socket()** creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

**socket()** is a libc function that creates a network socket:

```
int socket(int domain, int type, int protocol);
```

examples:

```
socket(AF_INET, SOCK_STREAM, 0); // TCP
socket(AF_INET, SOCK_DGRAM, 0);  // UDP
```

It returns:

- A file descriptor ( $\geq 0$ ) on success
- **1** on failure (sets **errno**)

Next we got something interesting looking although I don't know what it exactly is:

```
0x080493bd <+343>: call 0x8049110 <atoi@plt>
0x080493c2 <+348>: add $0x10,%esp
0x080493c5 <+351>: movzwl %eax,%eax
0x080493c8 <+354>: sub $0xc,%esp
0x080493cb <+357>: push %eax
0x080493cc <+358>: call 0x8049070 <htons@plt>
0x080493d1 <+363>: add $0x10,%esp
0x080493d4 <+366>: mov %ax,-0x1a(%ebp)
```

What is `atoi`

`atoi()` = **ASCII to integer**

```
int atoi(constchar *nptr);
```

It converts a string to an `int`.

Example:

```
int port = atoi("8080");// port = 8080
```

Important properties:

- **No error checking**
- Stops at first non-digit
- Returns `0` on failure (ambiguous)

In real binaries, it's often used for:

- Command-line args
- Config values
- User-supplied ports / sizes

`htons()` = **Host TO Network Short**

```
uint16_t htons(uint16_t hostshort);
```

What it does:

- Converts a 16-bit value from **host byte order** → **network byte order**
- Network byte order = **big-endian**

Example:

```
uint16_t port = htons(8080);
```

On little-endian systems:

```
0x1F90 (8080) → 0x901F
```

So this tells us that there for sure is some network activity happening the background of this process, and it definitely involves behemoth6 password file being shared/transported.

Let's actually see what this memory pushed before `atoi@plt` :

```
(gdb) x/s $0x804a04c
Value can't be converted to integer.
(gdb) x/s 0x804a04c
0x804a04c:      "1337"
(gdb) |
```

I don't know what this number is, allow me to refer to some walkthroughs or writeups and I can make sense of it, although it should be something of relevance and use.

Okay so this is the port number at which transmission takes place, pretty neat. Now we can make sense of this, so the program utilizes localhost at port 1337.

And that brings us to this really close one to our intentions:

```

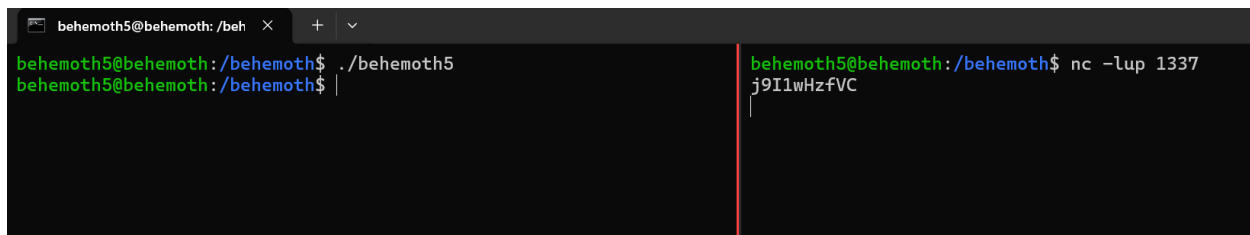
0x08049415 <+431>:  push    -0x2c(%ebp)
0x08049418 <+434>:  push    -0x24(%ebp)
0x0804941b <+437>:  call    0x8049100 <sendto@plt>
0x08049420 <+442>:  add     $0x20,%esp
0x08049423 <+445>:  mov     %eax,-0x20(%ebp)
0x08049426 <+448>:  cmpl    $0xffffffff,-0x20(%ebp)
0x0804942a <+452>:  jne     0x8049446 <main+480>

```

`0x2c` or `0x24` probably might be the password of behemoth 6 as the contents of behemoth6 file being transported; although that is an assumption i have based on `<sendto@plt>` function, I'm not too sure about it either.

Okay now it is making sense more than ever, we might be able to exploit this while its in network and read the contents of the file being transmitted- because there certainly doesn't seem to be any encryption methodology put to use here.

We will need two terminals one executing the program the other listening on port 1337:



```

behemoth5@behemoth: /beh
behemoth5@behemoth: /behemoth$ ./behemoth5
behemoth5@behemoth: /behemoth$ |
behemoth5@behemoth: /behemoth$ nc -lup 1337
j9I1wHzfVC

```

And there we go, it executed as thought around. The password for behemoth 6 is grabbed.

Onto next one.

## References:

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