2015第一届强网杯

转载

R茜R 于 2018-03-25 21:10:00 发布 525 收藏 1
文章标签： shell c/c++ 数据结构与算法
原文链接：http://www.cnblogs.com/clingyu/p/8646916.html

第一届强网杯pwn题

shellman

首先看看题目的大致信息，看看哪里会出现问题。

list函数，首先可以看到一些关于内存块的信息，其中堆表的基址在bss段，在堆中有一个类似于堆块的结构，三个64位数据作为一个堆块的标识，第一个数据位应该是标识当前是否有堆块，为1则表示有堆块，为0则没有，第三个数据应该就是这个堆块标识对应的堆的地址，可以从该地址直接访问堆中的数据。其中堆块数目也存放在bss段，以及最多只能申请256个堆块这些，都是一目了然的。

一般来说，new或者create这些新建堆块函数就是用来了解堆的详细结构，以及堆中的数据的。

这里可以很明显的看到，在刚才所说的bss段分别放了什么，其中堆块标识的第二个数据就应该是堆块的大小，不过是用户申请的大小，并不是堆块的真实大小。

而堆块中的内容很简单，就是用户输入的数据，按照固定长度写入堆块。

还有一个注意点就是，堆块长度不能超过1023字节，这可能会导致某些攻击方式不能用。

这个题目的问题就出现在edit函数中，个人觉的这个漏洞其实还是不容易发现的，虽然很简单。

问题主要是选择堆块要进行修改时，在把新的数据读入堆块中时，并未限制读入的长度，而是用户输入多长就读入多长，这就造成了堆溢出。

堆块释放一般就是漏洞的多灾区，这里同样有一个问题，堆块释放后指针没有清零，可能会造成UAF，不过由于堆表的其他数据都清零了，并不具有方便的利用条件，而且本题的堆溢出已经足以getshell，就没必要花精力去想这方面了。

程序的基本情况了解了，程序的问题也发现了，接下来就是通过GDB和脚本尝试利用来getshell。

本题的利用方式其实很简单，由于堆中的数据过于简单，而且程序本身没有开启针对于堆的保护措施，所以我们尝试了两种思路：

第一种：fast bin伪造

大致思路就是通过堆溢出伪造fast bin链，伪造出一个不存在的fast bin块，并将它指向任意地址。

这个很简单，首先申请两个fast bin大小的块，大小无所谓，释放第二个，通过第一个溢出到第二个，修改指向下一个的指针为我们的堆表地址，当我们第二次申请该大小的块时，就获得了位于堆表地址的堆块，当修改堆表第三个数据位为动态库函数got地址时，再次list就会泄露出函数真实地址，从而可以获得任意函数地址，也就能通过一定的构造getshell。

图中的时释放第二个堆块后的堆表，堆块，以及bins中的情况，可以看到，我们可以通过堆溢出很轻易的修改fast bin中的内容。

这时候就很明显了，我们成功的修改了fast bin链中的信息，接下来通过两次申请内存就可以获得堆表中的地址，还可以造成任意修改。

接下来就是一般的套路了，这里就不再过于详细的叙述了，主要就是利用了这个程序对输入进行处理的一个函数，修改成system，通过输入"/bin/sh\n"来getshell。

第二种：small bin的unlink攻击

unlink攻击大家都很清楚，任意地址写，我们通过堆溢出来伪造堆块造成unlink，跟上面的思路一样，也是为了获得堆表地址的堆块，只能写一个地址，所以我们就选择堆表中的堆块地址，将其覆盖为堆表地址，也就是相当于获得了一个位于堆表的块。

这是伪造前的堆中的数据，我们所要覆盖的地址就是0x6016d0，而图中黄色的部分就是我们需要构造的部分。

我们构造的内容很简单，就是根据chunk的复用，以及在查看前一个块的缺陷，我们在原来的堆中伪造一个小一点的块，这样在释放第二块的时候，就会查看前一块是否空闲，然而只发现了我们伪造的前一块，标识位被置为了空闲，所以触发unlink合并。

这里需要注意的是，Unlink攻击时需要绕过检查，假设我们要写入的地址是point，前一块的前指针=point-0x18，后指针=point-0x10，就可以绕过unlink检查。

我们成功获得了位于堆表处的堆块，接下来只需要通过edit修改就可以进行正常利用了。

imdb

先看一下程序，查询movies和TV的，可以添加TV，Movie，也可以删除，可以查看这些，也就是一般的pwn题的基本逻辑，为了构造出可以利用的简单漏洞。

简单看一下程序的主要功能，跟一般的堆利用题目大差不差，按某种数据结构建立堆块，根据位于bss段的堆表遍历堆块，释放堆块这些。

本题的问题就是UAF，删除所有同名堆，但只清零了一个指针。

接下来我们看一下程序的细节，以便于我们进行利用。

> Movie堆块在有一个附加的堆块Actor，大小可任意，且内容只有用户输入，不出意外的话，这里我们可以通过这个Actor块伪造Movie或者TV块进行利用。TV块中没有其他块，可以用于申请出我们将要处理的内存，也可以防止bin中的块被合并。
>
> show函数中，通过的是虚函数进行的遍历，也就是堆块中有一个函数指针，这个指针用来打印堆块中内容。
>
> 堆块通过名称对块进行查询。

对程序有了大致的了解之后，我们就可以开始进行利用了，我们的首要任务是泄露出库函数的真实地址，用的方法就是通过Actor堆，可以分为两种思路进行泄漏：

第一种：

通过Movie大块和Actor块之间的大小之差，在Actor之后在安排一个堆块，恰好使得通过Actor伪造的Movie中的虚表指针恰好可以访问到Actor之后的块中的内容，由于是通过指针访问，所以可以造成任意地址读，这样就可以泄露出库函数地址。

用黄色标注出来的就是我们的Actor伪造的Movie块，当show函数通过堆表遍历时，会认为该块是一个Movie块然后就会调用堆表函数打印Actor中的信息，可以看出，堆表函数要去查看的地址正好位于下一个块中，因此，我们就可以任意设定这个地址，来泄漏库函数地址。

第二种：

这一种相对来说就很寻常而且稳定简单，不需要关注堆块大小，由于我们伪造了Movie块，就干脆把它的Actor指针同时写入该块中，就不需要下一块来做辅助了。

不过相比于第一种方法的缺陷就是，每次进行泄漏都需要对堆环境进行铺垫，而第一种只需将辅助块释放后再申请就可以无限泄漏了。

统一说一下对堆环境的铺垫：

首先申请三个TV块，同名块，然后释放，这样堆表中就会有两个可以利用的堆指针，第一个作为Movie，第二个就指向Actor，这样才使得我们的泄漏有可能。

> 如果不希望释放同名块后堆块被合并，申请第四个堆块不释放即可。

泄露完地址之后就需要想办法getshell，这里我们依然通过Actor伪造的Movie块来进行劫持程序流。

删除C++对象后，会调用虚函数，所以很显然，我们的Actor伪造的Movie就是一个对象，而它的虚表函数是我们伪造的，如果写上one-gadget地址，就可以直接getshell。

还有其他的一些思路，不过在本地都不能正确获得shell，所以不再详细描述，总之，本题的泄漏以及利用思路，都是通过Actor块伪造Movie块，并任意指定虚函数，然后通过各种调用虚函数的方法来getshell。

注：

题目以及解题脚本：https://github.com/LJRosemary/ctf

转载于:https://www.cnblogs.com/clingyu/p/8646916.html