wargame narnia writeup

转载

weixin_34392843 于 2018-03-08 10:56:09 发布 170 收藏
文章标签： shell python
原文链接：https://juejin.im/post/5aa116c9f265da23870e68df

> litao3rd · 2015/04/09 11:02

前言

这一期的*wargame*难度明显比之前的*leviathan*要高，而且已经涉及到相对完善的*Linux溢出*相关知识了。但是在overthewire上这才居然只是*2/10*的难度，看来我差得很远啊。

在narnia首页，有如下提示，使用初始账号和密码登陆到目标机器，关于本wargame的所有文件都在/narnia文件夹下面。Let's go

```
To login to the first level use:

Username: narnia0
Passowrd: narnia0
Data for the levels can be found in /narnia/.
复制代码
```

level 0

从目标机器的文件夹中我们可以看到，每个 level 都给了源代码和编译后的可执行文件，每个可执行文件都有*set-uid*权限。只要溢出该可执行文件，得到下一个 level 的 shell，就可以在*/etc/narnia_pass/*文件夹下面得到下一个 level 的密码了。

首先正常执行一下*narnia0*这个文件，看看有什么提示没有。

从执行结果来看，应该是溢出缓冲区，然后修改栈中的另外一个自动变量，以此来过后面的逻辑判断。从下面的源代码文件也可以看到，我的猜想是正确的。

```
#!c
#include <stdio.h>
#include <stdlib.h>

int main(){
        long val=0x41414141;
        char buf[20];

        printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
        printf("Here is your chance: ");
        scanf("%24s",&buf);

        printf("buf: %s\n",buf);
        printf("val: 0x%08x\n",val);

        if(val==0xdeadbeef)
                system("/bin/sh");
        else {
                printf("WAY OFF!!!!\n");
                exit(1);
        }

        return 0;
}
复制代码
```

可以看到，输入缓冲区buf，和待溢出变量都在main函数的栈帧中，这题很简单，只要正确构造输入就可以了。一开始，我只是构造了这样的一个输入：

很明显，明明正确溢出修改了val的值，但是没有得到想要的 shell，后来，发现原来是管道输出给了程序之后，就会自动关闭了，造成程序返回的 shell 无法打开。于是，修改shellcode如下，发现正确得到了密码。

level 1

执行可执行程序，可以得到一个很有用的提示如下。

好像只要把shellcode放到正确的环境变量中就可以了，从程序代码中，看到环境变量EGG的地址被作为函数指针调用了。

```
#!c
#include <stdio.h>

int main(){
        int (*ret)();

        if(getenv("EGG")==NULL){
                printf("Give me something to execute at the env-variable EGG\n");
                exit(1);
        }

        printf("Trying to execute EGG!\n");
        ret = getenv("EGG");
        ret();

        return 0;
}
复制代码
```

构造如下的带有shellcode的环境变量就可以正确溢出了。关于环境变量的地址计算，很多溢出类书籍都会提到，上百度google一下就可以得到想要的方法。

level 2

从程序执行结果来看，似乎需要给一个输入作为main函数的参数，估计是通过这个输入来做为溢出点。

```
#!c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
        char buf[128];

        if(argc == 1){
                printf("Usage: %s argument\n", argv[0]);
                exit(1);
        }
        strcpy(buf,argv[1]);
        printf("%s", buf);

        return 0;
}
复制代码
```

这也是一个很基础的溢出题目，正确的覆盖main函数的返回地址就可以了。解法如下图所示。我在这里，将shellcode放在了EGG这个环境变量中，所以只要使用EGG的地址覆盖main函数的返回地址就可以了。

level 3

从这道题目开始，难度开始慢慢加大了，不过依然都在控制之内。

从程序执行结果来看，似乎将某个文件作为参数传给程序，然后程序打开输出到/dev/null这个设备中去。

```
#!c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv){

        int  ifd,  ofd;
        char ofile[16] = "/dev/null";
        char ifile[32];
        char buf[32];

        if(argc != 2){
                printf("usage, %s file, will send contents of file 2 /dev/null\n",argv[0]);
                exit(-1);
        }

        /* open files */
        strcpy(ifile, argv[1]);
        if((ofd = open(ofile,O_RDWR)) < 0 ){
                printf("error opening %s\n", ofile);
                exit(-1);
        }
        if((ifd = open(ifile, O_RDONLY)) < 0 ){
                printf("error opening %s\n", ifile);
                exit(-1);
        }

        /* copy from file1 to file2 */
        read(ifd, buf, sizeof(buf)-1);
        write(ofd,buf, sizeof(buf)-1);
        printf("copied contents of %s to a safer place... (%s)\n",ifile,ofile);

        /* close 'em */
        close(ifd);
        close(ofd);

        exit(1);
}
复制代码
```

> 很神奇的事情，字符数组char ofile[16] = "/dev/null";居然可以这样初始化，我记得当年的谭老师的课本里不是这样写的啊。。。。

从源代码来看，之前的猜测是对的。我一开始的想法是，重定向/dev/null设备到某个文件，这样，将密码的存储文件作为参数传给程序，程序将密码文件中的内容输出到我重定向的目标文件中，就可以正确得到了。后来google了半天，没有找到有效的重定向的方法。另辟蹊径，我想到可以溢出缓冲区的内容，修改输出文件路径，这样就可以将结果输出到某个文件中去了。我构造了/tmp/narnia3/AAAAAAAAAAAAAAAAAAA/tmp/pass这个路径下的一个文件，该文件被软链接到密码文件。同时，该字符串被作为参数输入给了程序，/tmp/pass这部分子串被溢出到输出文件路径的存储缓冲区中，这样输入是密码文件的一个软链接，输出是/tmp/pass这样的一个文件。具体操作如下图所示，因为待溢出程序具有set-uid权限，所以执行是的有效用户是下一个 level，注意/tmp下文件夹的访问权限问题。

level 4

这一关程序执行居然没有输出，看来只能通过分析源代码来找溢出点了。

```
#!c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

extern char **environ;

int main(int argc,char **argv){
        int i;
        char buffer[256];

        for(i = 0; environ[i] != NULL; i++)
                memset(environ[i], '\0', strlen(environ[i]));

        if(argc>1)
                strcpy(buffer,argv[1]);

        return 0;
}
复制代码
```

从代码来看，程序清空了所有的环境变量，这样在环境变量中存放shellcode的方法不可用了，不过，程序将输入的main函数参数无限制拷贝到了buffer中，这样就给了我们缓冲区溢出的漏洞，很基础的一个缓冲区溢出题目，只要将shellcode放入到栈中，然后正确覆盖函数返回地址就可以了。在猜测shellcode地址的时候，可能是因为栈偏移的问题，导致gdb中的栈地址和shell中运行时的实际地址有所偏移，不过添加一些NOP Sled就可以了。结果如下图所示。

level 5

这一关从程序执行来看，也是通过溢出修改某个变量的值，但是从源代码看，并不是简单的溢出就可以修改了。

```
#!c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv){
        int i = 1;
        char buffer[64];

        snprintf(buffer, sizeof buffer, argv[1]);
        buffer[sizeof (buffer) - 1] = 0;
        printf("Change i's value from 1 -> 500. ");

        if(i==500){
                printf("GOOD\n");
                system("/bin/sh");
        }

        printf("No way...let me give you a hint!\n");
        printf("buffer : [%s] (%d)\n", buffer, strlen(buffer));
        printf ("i = %d (%p)\n", i, &i);
        return 0;
}
```
复制代码

从来看，变量i和缓冲区buffer在栈中相邻，但是，缓冲区输入的时候使用了安全的snprintf()函数，这导致不能通过溢出来覆盖变量i的值，但是snprintf()在调用的时候的格式化字符串是由用户作为main()函数的输入，我们可以控制这个格式化字符串，导致了格式化字符串漏洞。

验证的确有格式化字符串漏洞，利用这个漏洞，可以读写任意地址的值，所以我们构造一个合适的格式化字符串，就可以修改变量i的值，得到一个高权限的shell。具体操作如下图所示。

level 6

这一关需要两个输入作为main()函数的参数，猜测应该有很明显的溢出点，难度就在于如何构造合适的溢出字符串。

```
#!c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern char **environ;

// tired of fixing values...
// - morla
unsigned long get_sp(void) {
       __asm__("movl %esp,%eax\n\t"
               "and $0xff000000, %eax"
               );
}

int main(int argc, char *argv[]){
        char b1[8], b2[8];
        int  (*fp)(char *)=(int(*)(char *))&puts, i;

        if(argc!=3){ printf("%s b1 b2\n", argv[0]); exit(-1); }

        /* clear environ */
        for(i=0; environ[i] != NULL; i++)
                memset(environ[i], '\0', strlen(environ[i]));
        /* clear argz    */
        for(i=3; argv[i] != NULL; i++)
                memset(argv[i], '\0', strlen(argv[i]));

        strcpy(b1,argv[1]);
        strcpy(b2,argv[2]);
        //if(((unsigned long)fp & 0xff000000) == 0xff000000)
        if(((unsigned long)fp & 0xff000000) == get_sp())
                exit(-1);
        fp(b1);
        exit(1);
复制代码
```

从源代码中可以看到，环境变量和多余的`main()`函数参数都被清空了，导致无法在其中安放`shellcode`。缓冲区`b1`和`b2`在栈中紧邻，接下来是一个指向`puts()`函数的函数指针，于是有了覆盖这个函数指针的思路。函数指针以`b1`为参数，进行函数调用，于是思路是用`system()`的地址覆盖`fp`的值，然后在缓冲区`b1`中填充`/bin/sh`字符串，这样在程序结束的时候就会有`system("/bin/sh")`这个函数调用，得到一个高一级的`shell`。在程序中，首先`strcpy(b1)`，然后再`strcpy(b2)`，我们在构造带有`/bin/sh`这个子串的字符串时需要考虑到字符串的长度问题，使得字符串能够正常结束。这样，先使用缓冲区`b1`溢出覆盖`fp`的值，使用`system()`的地址覆盖该值，然后使用缓冲区`b2`溢出往`b1`中添加`/bin/sh`这样的子串，`b2`的长度需要考虑。实际操作如下。

level 7

简单的程序输出已经提供不了太多的有效信息了，但是还是可以看到有输入，就有可能有缓冲区溢出问题。

```
#!c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int goodfunction();
int hackedfunction();

int vuln(const char *format){
        char buffer[128];
        int (*ptrf)();

        memset(buffer, 0, sizeof(buffer));
        printf("goodfunction() = %p\n", goodfunction);
        printf("hackedfunction() = %p\n\n", hackedfunction);

        ptrf = goodfunction;
        printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

        printf("I guess you want to come to the hackedfunction...\n");
        sleep(2);
        ptrf = goodfunction;

        snprintf(buffer, sizeof buffer, format);

        return ptrf();
}
int main(int argc, char **argv){
        if (argc <= 1){
                fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
                exit(-1);
        }
        exit(vuln(argv[1]));
}
int goodfunction(){
        printf("Welcome to the goodfunction, but i said the Hackedfunction..\n");
        fflush(stdout);

        return 0;
}
int hackedfunction(){
        printf("Way to go!!!!");
        fflush(stdout);
        system("/bin/sh");

        return 0;
}
```

这个代码有点点长，不过思路还是很清楚的，在`vuln()`函数中，有我们的输入，有一个函数指针紧邻着缓冲区，使用了安全的`snprintf()`函数来复制我们的输入到缓冲区中，依然有格式化字符串漏洞。不过这题的难度在于，我们输入的格式化参数没有打印出来结果，导致我们无法根据输出来调整输入的格式化参数，而且由于栈偏移的问题，导致gdb中的地址和`shell`中实际执行的地址差距很大，基本上不能利用。好字啊程序打印出来了足够的地址信息，我们知道修改后的值和待修改的地址。于是，我先构造了含有目的地址和目标长度的格式化字符串`\x0c\xd5\xff\xff.%134514432d.`，然后在该字符串后面添加写入的格式化参数`%n`，依次尝试猜测，运气不错，猜到了第六个就得到了`shell`。

> 本来是写了一个*Python*脚本来尝试爆破的，但是技术太烂，导致脚本执行的结果不太好，还是人工爆破来做的。其实这里猜测的风险很大，因为如果字符串的存储地址不是四字节对齐的话，这样我们在字符串中存放的地址就需要调整偏移，但是因为没有输出，导致无法知道这个偏移到底存在与否。好在题目设计得不是太难。

level 8

既然程序执行已经无法提供太多有效的信息了，还是直接看代码吧。

```
#!c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// gcc's variable reordering fucked things up
// to keep the level in its old style i am
// making "i" global unti i find a fix
// -morla
int i;

void func(char *b){
        char *blah=b;
        char bok[20];
        //int i=0;

        memset(bok, '\0', sizeof(bok));
        for(i=0; blah[i] != '\0'; i++)
                bok[i]=blah[i];

        printf("%s\n",bok);
}

int main(int argc, char **argv){

        if(argc > 1)
                func(argv[1]);
        else
        printf("%s argument\n", argv[0]);

        return 0;
}
```

复制代码

看起来似乎很简单，只是一个很简单的缓冲区溢出，但是实际操作的时候发现，变量blah和缓冲区bok在栈中是相邻的，导致如果输入的字符串太长的话，就会覆盖blah这个变量，这个变量又是我们的输入字符串的基地址指针，如果被修改了，就无法正确访问我们输入的字符串。

从执行结果来看，输入太长的字符串，都会导致后面的字符串没有复制到缓冲区中，这样也就无法覆盖函数的返回地址，溢出失败。于是改变了思路，既然太长的输入字符串会修改原来的基地址值，那么就用原来的基地址值再覆盖回去，这样就相当于没有修改了。只需要猜测原来的基地址值就可以了。从程序中可以看到，缓冲区被复制之后，没有正确的结束符，这样给了我们打印blah变量原来的值的可能。

缓冲区长度是20，所以输入长度为20的字符串正好可以覆盖缓冲区，同时又没有正确的结束符，就可以看到blah变量的值了。在这里是0xffffd7c7，根据测试，我们输入的字符串长度每增加1，这个基地址值就会减少1，通过计算，就可以得到正确的基地址值了。具体操作过程如下图所示。

end of the game

终于结束了这次的wargame，想到这一期的wargame难度只有*2/10*，我就知道后面还会有更多更好玩的东西。毕竟这里还有没涉及到ASLR、stack canary等缓冲区溢出保护策略，不过这些在后面的游戏都会有的。尽请期待～