HARMONY SECURITY
RESEARCH AND CONSULTANCY

www.harmonysecurity.com
info@harmonysecurity.com

Reflective DLL Injection

v1.0

By Stephen Fewer

31st October 2008

Contents

Disclaimer

Introduction

Under the Windows platform, library injection techniques both local and remote have been around for many years. Remote library injection as an exploitation technique was introduced in 2004 by Skape and JT[1]. Their technique employs shellcode to patch the host processes ntdll library at run time and forces the native Windows loader to load a Dynamic Link Library (DLL) image from memory. As an alternative to this technique I present Reflective DLL Injection.

Reflective DLL injection is a library injection technique in which the concept of reflective programming is employed to perform the loading of a library from memory into a host process. As such the library is responsible for loading itself by implementing a minimal Portable Executable (PE) file loader. It can then govern, with minimal interaction with the host system and process, how it will load and interact with the host. Previous work in the security field of building PE file loaders include the bo2k server by DilDog[2].

The main advantage of the library loading itself is that it is not registered in any way with the host system and as a result is largely undetectable at both a system and process level. When employed as an exploitation technique, Reflective DLL Injection requires a minimal amount of shellcode, further reducing its detection footprint against host and network based intrusion detection and prevention systems.

Overview

The process of remotely injecting a library into a process is two fold. Firstly, the library you wish to inject must be written into the address space of the target process (Herein referred to as the host process). Secondly the library must be loaded into that host process in such a way that the library's run time expectations are met, such as resolving its imports or relocating it to a suitable location in memory. For any of these steps to occur you should have some form of code execution in the host process, presumably obtained through exploitation of a remote code execution vulnerability.

Assuming we have code execution in the host process and the library we wish to inject has been written into an arbitrary location of memory in the host process, (for example via a first stage shellcode), Reflective DLL Injection works as follows.

- Execution is passed, via a tiny bootstrap shellcode, to the library's ReflectiveLoader function which is an exported function found in the library's export table.

- As the library's image will currently exists in an arbitrary location in memory the ReflectiveLoader will first calculate its own image's current location in memory so as to be able to parse its own headers for use later on.

- The ReflectiveLoader will then parse the host processes kernels export table in order to calculate the addresses of three functions required by the loader, namely LoadLibraryA, GetProcAddress and VirtualAlloc.

- The ReflectiveLoader will now allocate a continuous region of memory into which it will proceed to load its own image. The location is not important as the loader will correctly relocate the image later on.

- The library's headers and sections are loaded into their new locations in memory.

- The ReflectiveLoader will then process the newly loaded copy of its image's import table, loading any additional library's and resolving their respective imported function addresses.

- The ReflectiveLoader will then process the newly loaded copy of its image's relocation table.

- The ReflectiveLoader will then call its newly loaded image's entry point function, DllMain with DLL_PROCESS_ATTACH. The library has now been successfully loaded into memory.

- Finally the ReflectiveLoader will return execution to the initial bootstrap shellcode which called it.

Implementation

A skeleton Reflective DLL project for building library's for use with Reflective DLL Injection is available under the three clause BSD license[7]. Support for Reflective DLL Injection has also been added to Metasploit[6] in the form of a payload stage and a modified VNC DLL.

The ReflectiveLoader itself is implemented in position independent C code with a minimal amount of inlined assembler. The ReflectiveLoader code has been heavily commented and for a thorough understanding you should read through this code. A good understanding of the PE file format[3] is recommended when reading through the source code. Matt Pietrek has written a number of good articles detailing the PE file format[4] [5] which are also recommended reading.

Under Metasploit, the Ruby module Payload::Windows::ReflectiveDllInject implements a new stage for injecting a DLL which contains a ReflectiveLoader. This stage will calculate the offset to the library's exported ReflectiveLoader function and proceed to generate a small bootstrap shellcode, as show below in Listing 1, which is patched into the DLL images MZ header. This allows execution to be passed to the ReflectiveLoader after the Metasploit intermediate stager loads the entire library into the host process.

```
dec ebp                 ; M
pop edx                 ; Z
call 0                  ; call next instruction
pop ebx                 ; get our location (+7)
push edx                ; push edx back
inc ebp                 ; restore ebp
push ebp                ; save ebp
mov ebp, esp            ; setup fresh stack frame
add ebx, 0x???????? ; add offset to ReflectiveLoader
call ebx                ; call ReflectiveLoader
mov ebx, eax            ; save DllMain for second call
push edi                ; our socket
push 0x4                ; signal we have attached
push eax                ; some value for hinstance
call eax                ; call DllMain( somevalue,
DLL_METASPLOIT_ATTACH, socket )
push 0x????????     ; our EXITFUNC placeholder
push 0x5                ; signal we have detached
push eax                ; some value for hinstance
call ebx                ; call DllMain( somevalue,
DLL_METASPLOIT_DETACH, exitfunk )
                        ; we only return if we don't set a valid
EXITFUNC
```

Listing 1: Bootstrap Shellcode.

The bootstrap shellcode also lets us attach Metasploit by passing in the payloads socket and finally the payloads exit function via calls to DllMain for compatibility with the Metasploit payload system.

Detection

Successfully detecting a DLL that has been loaded into a host process through Reflective DLL Injection will be difficult. Previous methods of detecting injected libraries will be largely obsolete, such as inspecting a process's currently loaded library's for irregularities or detecting hooked library functions.

As a reflectively loaded library will not be registered in the host processes list of loaded modules, specifically the list held in the Process Environment Block (PEB) which is modified during API calls such as kernel32's LoadLibrary and LoadLibraryEx, any attempt to enumerate the host processes modules will not yield the injected library. This is because the injected library's ReflectiveLoader does not register itself with its host process at any stage.

Similarly, detecting hooked library functions is redundant as no library functions are required to be hooked for the ReflectiveLoader to work.

At a process level the only indicators that the library exists is that their will be a chunk of allocated memory present, via VirtualAlloc, where the loaded library resides. This memory will be marked as readable, writable and executable. Their will also be a thread of execution which will be, periodically at least, executing code from this memory chunk.

By periodically scanning all threads of execution and cross referencing the legitimately loaded libraries in the process with a call trace of each thread it may be possible to identify unusual locations of code. This will inevitably lead to many false positives as there are many reasons to execute code in a location of memory other then a library's code sections. Examples of this are packed or protected executables and languages that use techniques like just in time (JIT) compilation, such as Java or .NET.

At a system level when using Reflective DLL Injection in remote exploitation, the library should be undetectable by file scanners, such as Anti Virus, as it never touches disk. This leaves the main detection surface being the network, although many IDS evasion techniques are available[8] to aid circumventing detection such as polymorphic shellcode.

References

[1] Skape and JT, *Remote Library Injection*
http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf

[2] DilDog, *Back Orifice 2000*
http://sourceforge.net/projects/bo2k/

[3] Microsoft, *Microsoft Portable Executable and Common Object File Format Specification*
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[4] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*
http://msdn.microsoft.com/en-gb/magazine/cc301805.aspx

[5] Matt Pietrek, *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*
http://msdn.microsoft.com/en-gb/magazine/ms809762.aspx

[6] Metasploit LLC, *Metasploit*
http://www.metasploit.com

[7] Stephen Fewer, *Reflective Dll Injection*
http://www.harmonysecurity.com/ReflectiveDllInjection.html

[8] Kevin Tim, *IDS Evasion Techniques and Tactics*
http://www.securityfocus.com/infocus/1577