Extending Control, API Hooking

Author: Dr Craig S Wright GSE GSM LLM MStat

Abstract / Lead

This article is going to follow from previous articles as well as going into some of the fundamentals that you will need in order to understand the shellcode creation process, how to use Python as a launch platform for your shellcode and that the various system components are.

We will continue here with DLL injection before starting on API hooking. At this point we have learnt the basics of DLL injection and are ready to move onto applying it. This article will include a section on functions and calls, extending DLL injection and then move to the actual API hooking process (that we will extend) in coming articles. When all of this is put together, we will have the foundations for creating shellcode for exploits and hence an understanding of the process that penetration testers and hackers use in exploiting systems. With these skills, you will see how it is possible to either create your own exploit code from scratch or even to modify existing exploit code to either add functionality or in order to bypass signature based IDS/IPS filters.

This article continues a monthly series designed to take the reader from a novice to being able to create and deploy their own shellcode and exploits.

Introduction

In previous articles, we have covered a number of topics to do with the creation of shellcode and assembly language. We continue with an introduction of one of the primary exploitation processes used against a Windows system. In subsequent articles this will be expanded into the creation of standalone exploit kits and in the deployment of a rootkit and in the last article we started to explore the use and concept of a DLL.

We follow DLL injection with API hooking. This process is used by attackers and is also incorporated into automated frameworks (including Metasploit) as a part of the testing and exploitation process. API Hooking is one of the more common methods used by malware such as a rootkit to load it into the host's privileged processes. Once injected, code can be inserted into functions being transmitted between the compromised code and a library function. This step extends DLL injection and though API hooking the malicious code is used to vary the library function calls and returns by replacing the valid function calls with one of the attackers choosing. In this, we shall also elucidate the functions used by attackers in more detail. This is a useful skill when reversing malware as well as a good way to learn from the existing code base and even to leverage some of the various tools that are freely available already.

Why do we want to inject code?

There are many reasons to want to inject code into an application; some of them are even valid. Others are malicious. Developers frequently and legitimately use these techniques to:

- Subclass a window created by another process,
- In debugging as hooking can help to determine which dynamic link libraries (DLLs) are in use within a specific (remembering that some applications do not list all of the functions they end up calling and more, many malicious applications specifically hide the DLLs that they use to make reversing and analysing them more difficult),
- Extend existing code.

The last example is particularly useful when code samples have not been supplied with source code and cannot be altered. Here, a developer can take legacy programs and incorporate new calls and network libraries extending the codes useful lifespan. Many pen testing tools have been extended in this manner allowing the tester to take code developed solely for IPv4 networks and to incorporate IPv6 support into a favourite but no longer supported tool.

The "*hooked*" process will run with the full access and privileges of the original application that it has been injected into. That is, an external process can be made to run as the security context of another user or application.

This is of particular interest to the developers of malicious code. If the developer of such code can inject their DLL into a process running with system or administrative level privileges or one that has access to a lower level security context (especially if it can access the kernel or Ring Zero) they can escalate their privileges and run functions they would not usually be able to run.

In doing this, a process can be made to run as another user (hiding the attacker) or to even give the attacker administrative access to the system. Most commonly, these processes are used to access protected data in applications and to exfiltrate this information. With UIPI in Windows Vista and Windows 7, an attacker can no longer hoot into a higher integrity level. They can still capture information and applications entered by a user on a system.

This data can include:

- Credit card numbers and other forms of PII[1],
- Passwords (as well as usernames),
- Keystrokes and mouse clicks, or
- Documents (email for instance can be captured).

Basically, an attacker who can hook an API can do nearly anything that the user or process they have attacked can do. It is a good start to creating a Trojan.

DLL's in more depth

In the last article [1] we introduced DDLs or dynamic link libraries stating that *A DLL is a Dynamically Linked Library of executable code*" [2]. The reason for the use of a DLL is simple, it allows for code reuse. This makes patching and updates far simpler than when static linked code is used as well as reducing the amount of code loaded into the system. The reason for this is as dynamic libraries (usually in the form of a DLL in Windows) increase the maintainability of the program by removing redundancy. This is extremely beneficial as the user can patch a single file in place of hundreds (or more) statically linked files.

Some codes, such as cryptographic and algorithmic libraries are particularly difficult and having a library of tested and validated code makes the creation of software with these functions far simpler. More, it reduces the instances of bugs and exploits[2] against common code. Not all DLLs come from Microsoft, with many third part code in use (such as RSA's crypto libraries) being used on nearly all systems.

DLLs resemble standard Windows executables in the implementation of the PE format with the distinction that they cannot be called as an executable would be and require that a true executable calls them to run. OBC (Object Orientated Code) allows the creation and use of common libraries in place of statically linked code. As such, a single DLL can be called from numerous programs.

Most Windows executables maintain a list of the libraries that load. This is not essential and some (mainly malicious) code will in reality load a dynamic library that is not included in any import tables hiding the execution of these additional functions.

Back to the IAT (Import Address Table)

The import address table (IAT) is used as a lookup table when the application is calling a function in a different module. It can be in the form of both import by ordinal and import by name. As a consequence of a compiled program being unable to know the memory location of the libraries it depends upon, an indirect jump is required whenever an API call is made. As the dynamic linker loads modules and joins them together, it writes actual addresses into the IAT slots, so that they point to the memory locations of the corresponding library functions [3].

The dynamic linker moves to the IAT after the PE Header. The system uses the IAT as a lookup table to find functions that are located in different modules used by the application. The IAT exists as the system does not have the memory location for all of the libraries it uses. Rather, an indirect jump is necessary whenever an API call is completed. The dynamic linker loads the various modules into memory and connects them together and then writes jump instructions into the IAT slots.

---

[1] PII is Personally Identifiable Information (see http://en.wikipedia.org/wiki/Personally_identifiable_information)
[2] Although it should be noted that the impact of any exploit against a bug in a widely used DLL increases as the use of the library increases.

The system is then configured such that it is positioned at the memory locations of the consequent library functions. This does have a negative impact on the performance of the system as additional jumps are made outside the calling executable (in place of intra-module calls). Some examples of calls made by the IAT include those files set from the code as external calls.

For instance, a C# program using the following statement could call the "Sleep()",GetDisk(), FreeSpace or " GetCommandLine()" functions:

```
using System;
```

Using hooks to inject DLLs

We have seen that Windows provides a means to intercept (hook) messages sent within the system (e.g. mouse clicks). Attackers or Malicious software can also use this process such that they can inject a DLL into a remote process. Kuster [4] uses the example of "HookSpy" to have code "*executed in the address space of another process"*.

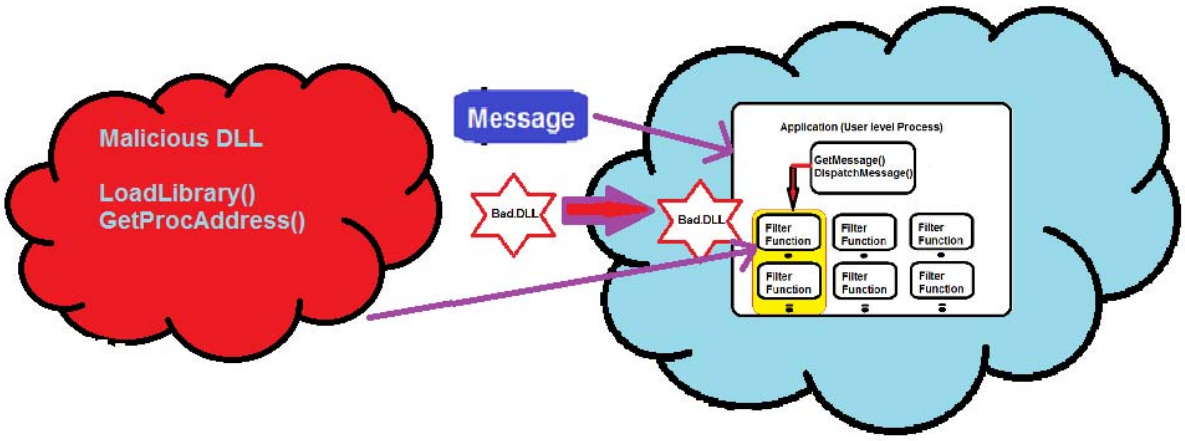

Figure 1: Using Hooks for DLL Injection.

For remote hooks where the thread is not directly associated with the initial process, the hook is either:

- *thread-specific, to monitor the message traffic of a thread belonging to another process;*
- *system-wide, to monitor the message traffic for all threads currently running on the system* [4]

As such, the hook procedure must reside in a dynamic-link library (DLL) and we need to use the hooking process to load (inject) the desired DLL into the control and hence the address space of the hooked thread. When this occurs, the entire DLL, which includes the hooked process as well as all other code in the DLL is mapped. The result, an attacker can use a Windows hook to inject code into the address space of a space application to itself and to capture information such as keystrokes.

In figure 1, we see the attacker first load the DLL to be used in the attack into a separate memory space (one not used by the process under attack). The attack relies on being able to determine the address used by the filter function of the attacked process. Here, the Application will have some means of calling a filter function as a remote process (an external library or DLL).

Using the inbuilt Windows function, *SetWindowsHookEx()*, the attacker sets a hook into the remote process (the application under attack). The Malicious DLL stipulates a DLL and particular function in the same to act as a filter function. The next part is all about waiting.

The application under attack has to receive an external message that would be processed using an external function. This is why we see so many attacks where the attacker tries to lure the victim into clicking on an icon or something similar. On receiving this message, the application loads the malicious DLL, injecting it inside the applications address space. Once this occurs, the replacement filter function from "Bad.DLL" processes the message allowing it to execute the attacker's code with the rights and privileges of the application under attack.

Functions and calls

For these methods, the main functions we should understand are:

- *SetWindowHookEx*() - For GUI based injections. [5]
- *UnhookWindowsHookEx*() – To release our code

In the next article in this series we will investigate the following additional methods:

- *CreateRemoteThread()* – [6]
- *WriteProcessMemory()* – [7]
- *LoadLibrary*() - [8]
- *FreeLibrary*() - [9]

Problems with DLL injection

As with anything in life, nothing is perfect (not even for the attacker). One of the more common methods we have discussed, the *SetWindowsHookEx()* function call works exclusively with GUI based applications. This means that it cannot be used against background services. The attacker also cannot use this to infect a compiler, linker or other such code. It does have the ability to inject multiple applications with a single call. If the target thread ID parameter is set to zero (0), then all GUI applications running can be infected in one go.

Although this can be a powerful form of exploiting a system, it is also risky. As this attack is not selective, it will inject code into far more applications than would generally be desired. Some applications are more robust than others. As the attack injects code into many GUI applications at once the change of any particular application crashing the system (or at least an application) increases significantly. Hence this technique is far more likely to crash the system than one that selectively targets an application or thread. For an attacker, it is best to limit where code is injected into as it also makes it simpler to detect the attack.

Further to this, if the code is universally targeted at thread ID parameter 0, the attacker's DLL will be injected into each GUI-based application for the complete lifetime of each of these applications. This makes forensic analysis of the attack and detection by anti-malware software simpler.

Vista and Windows 7 also change this process. Starting with Windows Vista, user-processes and privileged system processes run in separate sessions. This means that user-processes can no longer simply exchange messages. This limits the ability of code to escalate its privilege using API hooking significantly. More, another feature, UIPI (or User Integrity Privilege Isolation) was introduced to stop a lower privilege process from calling a higher privilege process using the *SetWindowsHookEx()* function call. The *CreateRemoteThread()* function call was similarly limited with the system stopping unprivileged processes from using this function call against a protected process.

The simplest means to inject a DLL involves substituting an alternate DLL with one that will be called by the application. Here, renaming the attack library to the original (and as yet unloaded DLL) and waiting. The difficulty is that the substituted library needs to export all of the symbols that were exported within the original DLL. This can be achieved through the use of function forwarders and which of course simplify the process used to hook functions[3].

There are many reasons for an attacker not to use this method (even if it is far simpler). First, it leaves a file on the system being attacked making forensics and anti-malware processes far simpler. Next, it is far from version-resilient. When Microsoft (or another vendor as the case may be) patches a system DLL used by this library, the substituted DLL will lack any added functions. That is the substitute DLL will lack function forwarders. The application that calls this DLL will at best be unable to load and execute and at worst will crash the system.

An earlier method of API injection used to exist as well. In older systems (such as 8 and 16 bit X86 hosts) that did not support multithreading (and multithreading makes this technique catastrophically foolish other than a s a means to crash a system) could use the method known as "*API Hooking by overwriting code*" [10]. In this, the attacker would overwrite the initial bytes of the targeted function with a JUMP CPU instruction. This instruction would be constructed to jumps to the memory location of the supplemented function that has already been loaded into memory.

The difficulty is that the substituted function be required to possess precisely the equivalent signature as the function that has just been hooked. This would require that any parameters in the initial function have been recreated exactly. This would require that the return value and the calling convention used by the substituted function remain the same as those used within the original function.

The main issues with this on a single threaded system (and note again that this does not work well at all in multi-threaded environments) comes from the differences in system architectures. Systems with x86, x64, IA-64, or alternative processors each use different JUMP instructions and it can lead to unexpected if not catastrophic results when JUMPs from one system are coded into another architecture.

Conclusion

In the next instalment in the series of articles we will continue with DLL injection. In addition to API hooking, there are a few other means to have code executed in the address space of a separate process. We will cover attacks against remote processes using the using the *CreateRemoteThread & LoadLibrary* method and the *WriteProcessMemory* / *CreateRemoteThread* method in coming articles. The later of these methods actually copies the malicious code directly inside the remote process.

At this point we have learnt the basics of DLL injection and are ready to move onto applying it. When we then put all of this together, we will have the foundations for creating shellcode for exploits and hence an understanding of the process that penetration testers and hackers use in exploiting systems. With these skills, you will see how it is possible to either create your own exploit code from scratch or even to modify existing exploit code to either add functionality or in order to bypass signature based IDS/IPS filters. We will also look at "*trampolines*" including that used in code such as "QuietRIATT"[4] in next month's article.

---

[3] There is a MSDN blog entry on this topic at:
http://blogs.msdn.com/b/oldnewthing/archive/2006/07/19/671238.aspx

[4] See http://www.blackhat.com/presentations/bh-dc-09/Krumheuer_Raber/BlackHat-DC-09-Krumheuer-Raber-QuietRIATT-Slides.pdf for details of this code.

With this knowledge, you will learn just how easy it is for sophisticated attackers to create code that can bypass many security tools. More, armed with this knowledge you will have the ability to reverse engineer attack code and even malware allowing you to determine what the attacker was intending to launch against your system. In this way, you can improve your forensic and incident response skills.

Author's bio

About the Author:

Craig Wright (Charles Sturt University)is the VP of GICSR in Australia. He holds the GSE, GSE-Malware and GSE-Compliance certifications from GIAC. He is a perpetual student with numerous post graduate degrees including an LLM specializing in international commercial law and ecommerce law, A Masters Degree in mathematical statistics from Newcastle as well as working on his 4th IT focused Masters degree (Masters in System Development) from Charles Stuart University where he lectures subjects in a Masters degree in digital forensics. He is writing his second doctorate, a PhD on the quantification of information system risk at CSU.

References

1. Wright, C.S., *Taking control, Functions to DLL injection.* Hakin9, Exploiting Software, 2012. **2**(4): p. 22-27.

2. Shewmaker, J. *Analyzing DLL Injection*. in *NS2006, GSM Presentation*. 2006. SANS.

3. Wiki. *Portable Executable*. Available from: http://en.wikipedia.org/wiki/Portable_Executable.

4. Kuster, R. *Three Ways to Inject Your Code into Another Process*. 2003 [cited 2012 15/05/2012]; Available from: http://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces.

5. MSDN. *SetWindowHook Function*. 19/05/2012]; Available from: http://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx.

6. MSDN. *CreateRemoteThread function*. 19/05/2012]; Available from: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx.

7. MSDN. *WriteProcessMemory function*. 20/05/2012]; Available from: http://msdn.microsoft.com/en-us/library/windows/desktop/ms681674(v=vs.85).aspx.

8. MSDN. *LoadLibrary function*. 21/05/2012]; Available from: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx.

9. MSDN. *FreeLibrary function*. 15/05/2012]; Available from: http://msdn.microsoft.com/en-us/library/windows/desktop/ms683152(v=vs.85).aspx.

10. Madshi. *API Hooking Methods* 20/05/2012]; Available from: http://help.madshi.net/ApiHookingMethods.htm.