
Analyzing DLL Injection

James Shewmaker
GSM Presentation - 2006

Analyzing DLL Injection - © 2006 James Shewmaker

Today we will take a look at what DLL injection is, how it works, what it looks like, and how to defend against it.

Objectives

- Define DLL Injection
- Injection Techniques
- Process Monitoring
- Develop Defenses
- Conclusions

Analyzing DLL Injection - © 2006 James Shewmaker

We will start by establishing what DLLs are and what we mean when we say “DLL injection.” We will take a look at a benign injection: its source code, methods of injection, and removal. We will end with how to increase defenses against such attacks.

What is DLL Injection?

- Dynamically Linked Library
- Designed this way on purpose
 - Code reuse
 - Modular programming
- Can be loaded into memory
- Can be called from another process

Analyzing DLL Injection - © 2006 James Shewmaker

A DLL is a Dynamically Linked Library of executable code. The idea with a DLL is that programmers do not need to reproduce common functions and can reuse code. It also makes it much easier to have standards such as a documented API that can be followed. For example, not each programmer needs to know how Windows needs for every minor detail, just that there is a DLL it can use that contains a function. The program merely references the external function from the DLL, and it is loaded into memory and available for use.

Today when we talk about injection, we are talking about a DLL that is loaded into a running process's memory. Windows as we know it now was designed for this, and injection techniques can be used by any application. Some applications use it to add features to a closed-source program. However, these or similar techniques are now a huge problem with spyware so prevalent.

Injection via Windows API

- VirtualAllocEx()
 - Not for Windows 9x/Me
- ReadProcessMemory() and WriteProcessMemory()
 - All Versions of Windows

Analyzing DLL Injection - © 2006 James Shewmaker

Windows provides a few different ways to call external functions in the DLL files. VirtualAllocEx() only works on NT based systems, so in the past it was not used as much. The universal way to deal with DLLs in memory is to use ReadProcessMemory() and WriteProcessMemory(). These work in all released versions of Windows, so you would be reasonably safe to use them. That is, you are as safe as can be considering you are writing to running memory . . . but that is besides the point.

Injection via Debugging API

- `GetThreadContext()` and `SetThreadContext()`
 - Saves backup of registers
 - Uses breakpoint to resume original
- Useful for returning control back to the host process gracefully

Analyzing DLL Injection - © 2006 James Shewmaker

Another approach that we see with DLL injection is to use the Windows provided API for debugging. This allows us to keep the thread that we are dealing with in scope. Keeping it in scope means we can save the registers, save the instruction pointer (EIP), and save the state of the stack. This prevents stomping on variables and such issues that would cause problems when the injected code is completed.

It would not be very interesting to us as an attacker if the moment we start to own the box, it became unusable. As a supplement to an application, we would not want to crash the original application either. This is the case for any injection attempt, but the `ThreadContext()` functions help us accomplish this easier.

The debugging features can be troublesome when you are actually trying to debug such an application, which as an attacker is a good thing for us. While analyzing, we will have to pick our battles and not set arbitrary breakpoints.

Tools Useful for Analysis

- My favorites
 - Windows XP service pack 1
 - VMWare Workstation
 - Ollydbg
 - Process Explorer
 - RemoteDLL
 - SysInternals

Analyzing DLL Injection - © 2006 James Shewmaker

Now to set up our analysis environment, we will need to prepare our system and install a few tools. Of course we need a host, I'll use Windows XP with service pack 1; it is fairly predictable with regards to injection. VMWare Workstation is extremely handy if only for the fact I can revert back and try a similar without rebooting the entire machine. Other tools that are useful:

- Any SysInternals style monitoring tools such as Regmon, Filemon, TDImon, Process Explorer
- Ollydbg is my debugger of choice, have one handy that suits you
- RemoteDLL was the first tool I found that claimed to inject and remove cleanly, so I stuck with it
- Any other tools you prefer that help you understand what precisely is happening on the drive, on the CPU, or in memory.

We will be focusing specifically on the injection, so you will not need an industrial strength lab, so stick to the tools you are comfortable with.

Hello World Injection

- Source code
 - Simple DLL
 - “Hello World” message
- Compiler
- Explorer.EXE as a host process
- Inject then Observe

Analyzing DLL Injection - © 2006 James Shewmaker

It will be best to start with a simple, predictable, and benign example for our DLL injection project. I have chosen one from a link at the end of the presentation. This DLL only provides one message box when directly called and one when loaded into memory. I’ve used the Dev-C++ IDE packaged with gcc compiler, as it is free and referenced in most of the URLs at the end of this presentation.

We can inject into any DLL, but for this purpose, we will use Explorer.EXE. I picked Explorer.EXE because it is commonly used by malware that injects itself. There are many static and dynamic libraries, so it is easy to hide. Also, killing Explorer.EXE will cause it to respawn, so we are not too worried about damage.

Source Code for Injection

```
DLLIMPORT void HelloWorld ()
{ MessageBox (0, "Hello World from
  DLL!\n", "Hi", MB_ICONINFORMATION); }

void HelloWorldMsg ()
{ MessageBox (0, "Hello World from an
  injected DLL!\n", "Hi",
  MB_ICONINFORMATION); }

if ((hThread =
  CreateThread(NULL, 0, HelloWorldMsg, NULL,
  0, &nThread)) != NULL)
```

Analyzing DLL Injection - © 2006 James Shewmaker

The first two functions in this slide are the functions called when directly calling the DLL and loading it into memory, respectively. The third item on the slide is the call when the DLL is attached to a thread, which is what we will see when injecting it into a running EXE. The complete source code is available via a link on the “More Information” slide at the end of the presentation.

The format of the parameters of each MessageBox call are dictated by the API that this DLL calls (much like how an EXE would call our HelloWorld function). To keep things short we will keep the title of the our popups to be simply “Hi” but with different contents in the message box itself.

We also have a header file that handles what we need to have the HelloWorld function imported from another process.

Executing the DLL

- Loading with User32.dll via the registry

```
HKLM\Software\Microsoft\  
Windows NT\CurrentVersion\Windows\  
AppInit_DLLs
```

- Running it manually

```
rundll32 dll_ex.dll,HelloWorld
```

- Injecting into a running Process

- Explorer.EXE
- Using RemoteDLL tool

Analyzing DLL Injection - © 2006 James Shewmaker

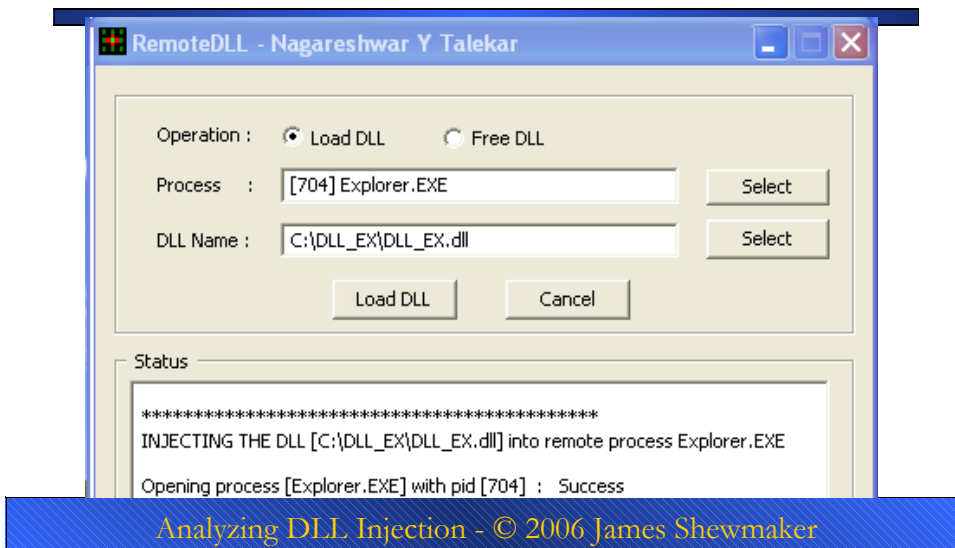
Now that we have our DLL source prepared, we save and compile, resulting in a shiny new DLL_EX.dll file. Now we have several different ways we can get it to run.

One we will not be doing today is by forcing it to load for the User32.dll library. If we edit this registry key and place the full path of the new DLL in the key, every process that loads User32.dll will also load our DLL_EX.dll file. Many calls to our new DLL would get messy quickly, and we would not get the chance to load our system tools first.

To test our DLL, we can use the rundll32.exe command included with Windows. This tool is essentially a stub loader that will run the DLL and routing provided on the command line. By running this command, we get a nice pop up that we would expect to see, telling us the DLL works as expected.

Now for the real test, we can use RemoteDLL to inject our working DLL into a running process. Upon loading, we should see a popup occur in the context of the EXE we injected into.

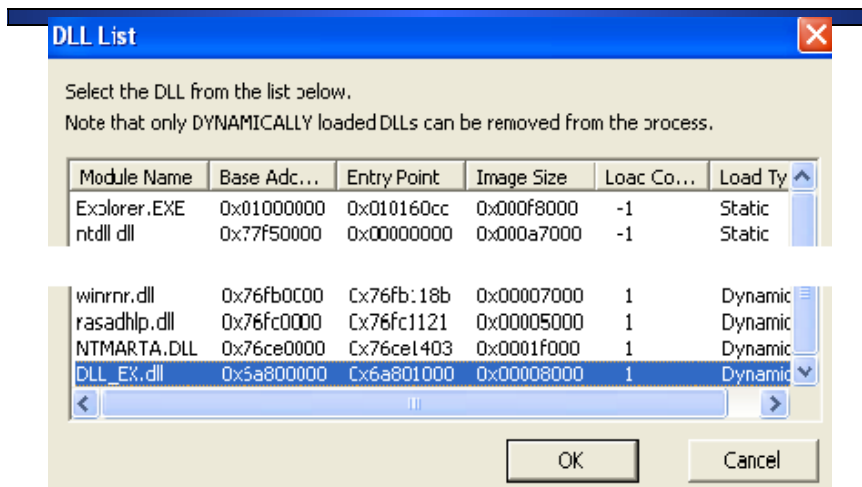
Loading into Explorer.EXE



The RemoteDLL tool requires selection of a running process, then selection of a file to inject. I picked Explorer.EXE and the newly compiled DLL file, which is loaded into the running process and we get a popup (though the popup arrives behind the window). We have now just completed a DLL injection.

Normally, malware will use a stub dropper to get the DLL loaded first. This could be practically anything—a binary loader, a RunOnce registry key, or maybe input into an application that can be manipulated (ala metasploit). If the author wanted to replicate the DLL, he only needs to complete the equivalent of our rundll32.exe command, and then the main function of the DLL can take care of anything needed. With this modular nature, many different things are possible.

Unloading from Explorer.EXE



Analyzing DLL Injection - © 2006 James Shewmaker

Here we see the top and bottom of the RemoteDLL interface when removing an injected DLL. It enumerates all of the libraries loaded by the selected process. By selecting the library and clicking ok we can remove the injection from memory.

More importantly, the memory locations provided here can be used to save the DLL to a file. If we were analyzing an unknown injection, we could take an entire memory dump and surgically cut out the library we are interested in to take a closer look at it.

How could we take a closer look at it? Well, we could disassemble the code, or to watch the behavior a little more closely we could inject this library we saved from memory to file, then inject into a less complicated EXE such as notepad.exe. Once such a library is loaded in notepad.exe, it would likely behave differently but at least we could attach a debugger to it in a way that we cannot do with Explorer.EXE.

DLL Injection Defenses

- Standard layered defense
 - OS and Application patches
 - Host firewall
 - Perimeter firewall
- Process behavior detection
- Host integrity checks

Analyzing DLL Injection - © 2006 James Shewmaker

So now we have seen a successful DLL injection. How could we protect ourselves? Standard Defense-in-Depth principles will help minimize the chances and impact of an event. By minimizing services installed and accessible there will be fewer attack vectors.

Behavior based detection engines would likely be the largest improvement. APIs do change, and applications cannot realistically check every library when called (after all, the point of the library is so the application does not need to know how that function works).

Detecting such a thing on the network would be difficult at best, since the same methods and approaches to injecting DLLs are standard in a variety of applications.

We would get some help from host integrity and host intrusion detection systems. The integrity checks would only be useful if the DLL actually touches the hard drive (which does not always happen). General host activity metrics would be of limited use as well—they would only catch the noisy injections.

Beyond Behavior Analysis

- Attach to something besides Explorer.EXE
 - Much easier to debug
 - Less likely to crash the environment
- Make use of VMWare's revert
- Use other tools
 - Attach debugger to host process
 - Set breakpoints at API calls

Analyzing DLL Injection - © 2006 James Shewmaker

Where do we go from here? We can perform similar tests by reloading our test DLL into different EXE files to confirm our results. Now that we have seen how Explorer.EXE works with an injection we can better isolate a test to inject into notepad.exe and try more useful injections. With this technique, it would be fairly easy to log any activity of that process.

With a smaller and less complicated, or maybe more familiar EXE, we can observe interactive tests. Attaching a debugger to the EXE would help us see precisely how any library is called.

As with most types of analysis, we want to make small, precise changes in a way we can reproduce our results later. Virtualization technologies are now a must during this kind of analysis.

More Information

- Hello World DLL
<http://craigheffner.com/dll.html>
- Methods to inject DLLs
http://www.codeproject.com/dll/DLL_Injection_tutorial.asp
- <http://users.ece.gatech.edu/~owen/Academic/ECE4112/Fall2004/Projects/dll%20injecting.doc>

Analyzing DLL Injection - © 2006 James Shewmaker

The HelloWorld DLL example came from <http://craigheffner.com/dll.html> and all three of these links had information regarding standard DLL usage as well as what we have been referring to as DLL injection.

There is also some material that Apex (<http://www.iamaphex.net>) has written, but as of this writing that site is still down. There are many other useful tutorials on DLLs and even injection that can be found with <http://www.google.com> queries.

Conclusions

- Define DLL Injection
- Injection Techniques
- Process Monitoring
- Develop Defenses
- More Information

Analyzing DLL Injection - © 2006 James Shewmaker

During this presentation, we covered what DLLs are, how to inject them, and how to monitor them. We talked briefly about a few defenses, and where to learn more about these techniques.

DLL injection is common occurrence with malware today, and understanding how injections occur will help any analyst track down such beasts.