Adaptive DLL Hijacking

written by Nick Landers | February 19, 2020
DLL hijacking has been a centerpiece of our operations for many years. During that time we've explored the deep caveats which make this technique difficult to actually use in the real world. Our implementations have expanded to include export table cloning, dynamic IAT patching, stack walking, and run time table reconstruction. We explore the details of these techniques extensively in our Dark Side Ops courses and we'd like to share some of that knowledge here.

If you've ever "understood" DLL hijacking, only to return to your lab and fail to get it working properly, this post is for you.

TLDR? Check out Koppeling. You really should read it though

Refresher

This post won't cover the basics of DLL hijacking. We expect you are familiar with module search order, KnownDLLs, "safe search", etc. If you need a refresher, here are some links:

- https://resources.infosecinstitute.com/dll-hijacking-attacks-revisited/
- https://pentestlab.blog/2017/03/27/dll-hijacking/
- https://liberty-shell.com/sec/2019/03/12/dll-hijacking/
- https://astr0baby.wordpress.com/2018/09/08/understanding-how-dll-hijacking-works/
- https://www.sans.org/cyber-security-summit/archives/file/summit-archive-1493862085.pdf
- https://posts.specterops.io/lateral-movement-scm-and-dll-hijacking-primer-d2f61e8ab992

In addition, some tooling designed to discover/exploit hijacks:

- https://github.com/cys3c/Siofra [https://github.com/Cybereason/siofra]
- https://github.com/cyberark/DLLSpy
- https://github.com/MojtabaTajik/Robber

When you first learned about DLL hijacking, you were likely shown a fairly primitive example which is trivial to exploit. Something like this:

```
void BeUnsafe() {
        HMODULE module = LoadLibrary("functions.dll");
        // ...
}
```

Here, we simply need to get some evil code into the correct location as "functions.dll". **LoadLibrary** will ultimately trigger the execution of our DllMain function, where we might write something like this:

```
BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
        if (reason != DLL_PROCESS_ATTACH)
                return TRUE;

        // Do evil stuff
        system ("start calc.exe");

        return TRUE;
}
```

There are a few critical reasons exploitation is so trivial here. We'll go through them here and then look at each one in more detail throughout the post.

1. **We don't maintain the stability of the source process.** In most instances, it will exit, crash, or otherwise misbehave as a result of our hijack. After all, it's likely loading this DLL for a reason.
2. **We don't maintain code execution in the source process.** As an extension of 1, we are simply executing

calc externally. We don’t care if the process stays up, or even what happens after we “pop our shell”.
3. **We don’t care about loader lock.** Because our entry point is so simple, we don’t have to worry about executing complex code inside DllMain while the loader lock is held (which can be dangerous).
4. **We don’t have to worry about export names.** Because this hijack occurs as a result of **LoadLibrary**, our malicious DLL doesn’t need to include any specific export names or ordinals.

If you’ve ever attempted to hijack in the real world, and something broke/failed, it was likely because of one (or many) of the 4 points above. Our time spent hijacking has yielded many tools and snippets which we’ll share throughout the post, so let’s get smarter.

Execution Sinks

There are two primary “sinks” from which DLL execution can originate. The names aren’t important, but we need consistent terminology to stay on the same page. Both of these sinks are provided by the module loader (LDR) within **ntdll.dll**. If an actor is interested in gaining execution as part of a DLL load, they require a call to **ntdll!LdrpCallInitRoutine**, triggering execution of **evil!DllMain**.

Static Sink (IAT)

The most obvious cause for DLL initialization is the result of its inclusion in a dependency graph. Specifically, **it’s membership of a required module’s import address table (IAT)**. This will most likely occur during process initialization (**ntdll!LdrpInitializeProcess**), but can also occur as a result of dynamic loading.

Here, the subsystem is simply calculating all required dependencies for a particular load event, and sequentially initializing them. However, before passing execution to the

new module, **it’s export table will be examined to ensure it provides the expected functionality**. This is done by comparing the EAT of the child module and patching those addresses into the IAT of the parent module. A typical call stack looks something like this:

```
ntdll!LdrInitializeThunk &lt;- New process starts
ntdll!LdrpInitialize
ntdll!_LdrpInitialize
ntdll!LdrpInitializeProcess
ntdll!LdrpInitializeGraphRecurse &lt;- Dependency graph is built
ntdll!LdrpInitializeNode
ntdll!LdrpCallInitRoutine
evil!DllMain &lt;- Execution is passed to external code
```

Dynamic Sink (LoadLibrary)

In a similar, but distinctly different process, active code is requesting a new module be initialized without specifying required functions. As a result, ntdll!LdrLoadDll will happily **ignore the export table of the target module**. This will likely be followed by **GetProcAddress** in an attempt to identify a particular function for run time use, but not always.

The dependency graph will be calculated with the requested module at its root and load events will occur as described above. This call stack looks something like this:

```
KernelBase!LoadLibraryExW &lt;- Dynamic module load is requested
ntdll!LdrLoadDll
ntdll!LdrpLoadDll
ntdll!LdrpLoadDllInternal
ntdll!LdrpPrepareModuleForExecution
ntdll!LdrpInitializeGraphRecurse &lt;- Dependency graph is built
ntdll!LdrpInitializeNode
ntdll!LdrpCallInitRoutine
evil!DllMain &lt;- Execution is passed to external code
```

Takeaway

Hijacks are more complicated to implement when part of a static sink. We need to ensure our export table supplies the required import names of our parent module **before we have control over execution**. In addition, by the time we have control of execution the addresses in our EAT will have already been patched into the parent module. This complicates any solution which would just rebuild the export table at run time.

Function Proxying

Maintaining stability in our source process demands that we proxy functionality to the real DLL (if there is one). This essentially means, through one means or another, linking our export table to the export table of the real DLL. Game hackers have been using this for a long time, but like hikers and hunters, the knowledge was slow to propagate to network security spheres. Here are some references links that tackle proxying through different methods:

- https://itm4n.github.io/dll-proxying/
- https://dl.packetstormsecurity.net/papers/win/intercept_apis_dll_redirection.pdf
- https://www.shysecurity.com/post/20130111-Dll%20Proxy
- https://kevinalmansa.github.io/application%20security/DLL-Proxying/
- https://googleprojectzero.blogspot.com/2018/04/windows-exploitation-tricks-exploiting.html
- https://www.synack.com/blog/dll-hijacking-in-2014/

And here are some projects that implement these methods:

- https://github.com/kevinalmansa/DLL_Wrapper
- https://github.com/maluramichael/dll-proxy-generator
- https://github.com/oranke/proxy-dll-generator
- https://github.com/mcryptzzz/ProxyDllGenerator
- https://github.com/advancedmonitoring/ProxyDll

- https://www.codeproject.com/Articles/1179147/ProxiFy-Automatic-Proxy-DLL-Generation
- https://www.codeproject.com/Articles/16541/Create-your-Proxy-DLLs-automatically

These techniques all accomplish the same outcome through slightly different means. Let's take a quick look at some strategies for better understanding.

Export Forwarding

PE files provide a simple mechanism for redirecting exports to another module. We can take advantage of this and simply point our names at the same export from the real DLL. You can either rename the real file or just use the full path. Most do this using linker directives like so:

```
#pragma comment(linker,"/export:ReadThing=real.ReadThing")
#pragma comment(linker,"/export:WriteThing=real.WriteThing")
#pragma comment(linker,"/export:DeleteThing=real.#3")
#pragma
comment(linker,"/export:DoThing=C:\\Windows\\real.DoThing")
// ...
```

Very easy, and we offload the work to the loader subsystem. It might look a bit obvious that we are attempting a hijack (e.g. every export is forwarded), but the advantage lies in its simplicity. One downside is the requirement to modify source code and/or build processes to prepare a DLL for hijacking, we'll solve this later.

*The traditional format for the module name was *without* the ".dll" extension when defining a forward. Nowadays this doesn't matter as the LDR subsystem has learned to ignore it. However, older systems like Windows 7 / Server 2008 will still fail if an extension is included. They also might crash when error reporting is attempted due to LdrUnloadDll being called too early.*

Stack Patching

An equally elegant, but more dynamic approach is to walk the stack backward from DllMain and replace the return value for the LoadLibrary call above us with a different module handle. As a result, any future calls to lookup functions will simply bypass us completely. It should be no surprise to the reader at this point, but **this technique will only work for dynamic sinks.** With static sinks, the LDR subsystem has already validated our export table and patched IATs with its values, nor does it care what we have to say about module handles.

Preempt mentions this in [a post about Vault 7 techniques](), but they don't go into much detail. Luckily we're crazy enough to try this stuff, so we've written a small PoC which should demo the trick nicely for anyone who wants to run with it.

https://gist.github.com/monoxgas/b8a87bec4c4b51d8ac671c7ff245c812

Run Time Linking

Here we create a hollow list of function pointers, and compile our export table to reference them. The names will be there, but the functions themselves won't go anywhere useful. When we gain control in DllMain, we load the real DLL dynamically and remap all of the function pointers at run time. This is essentially re-implementing export forwarding…. but with more code. We still have the same disadvantage of modifying source and/or build processes.

hijack.def<code>EXPORTS

ReadThing=ReadThing_wrapper @1
WriteThing=WriteThing_wrapper @2
DeleteThing=DeleteThing_wrapper @3</code>

hijack.asm<code>.code
extern ProcList:QWORD
ReadThing_wrapper proc

```
        jmp ProcList[0*8]
ReadThing_wrapper endp
WriteThing_wrapper proc
        jmp ProcList[1*8]
WriteThing_wrapper endp
DeleteThing_wrapper proc
        jmp ProcList[2*8]
DeleteThing_wrapper endp</code>
```

hijack.cpp<code>

```
extern "C" UINT_PTR ProcList[3] = {0};

extern "C" void ReadThing_wrapper();
extern "C" void WriteThing_wrapper();
extern "C" void DeleteThing_wrapper();

LPCSTR ImportNames[] = {
   "ReadThing",
   "WriteThing",
   "DeleteThing"
}

BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID
reserved)
{
        if (reason != DLL_PROCESS_ATTACH)
                return TRUE;

        HANDLE real_dll = LoadLibrary( "real.dll" );
        for ( int i = 0; i &lt; 3; i++ ) {
                  ProcList[i] = GetProcAddress(real_dll ,
ImportNames[i]);
        }

        return TRUE;
}</code>
```

Run-Time Generation

We could also go crazy and just re-build the entire export address table at run time. Here we need not know what DLL we are going to hijack when we write our code, which is nice. We can also add a basic function which re-implements the Windows

search order to try and locate the real DLL dynamically. It could also perform basic alterations like .old and .bak within the current directory just in case.

hijack.cpp<code>HMODULE FindModule(HMODULE our_dll)
{
	WCHAR our_name[MAX_PATH];
	GetModuleFileName(our_dll, our_name, MAX_PATH);

	// Locate real DLL using our_name

	if (our_dll != module){
		return module;
	}
}

void ProxyExports(HMODULE module)
{
	HMODULE real_dll = FindModule(module);

	// Rebuild our export table with real_dll

	return;
}

BOOL WINAPI DllMain(HMODULE module, DWORD reason, LPVOID reserved)
{
	if (reason != DLL_PROCESS_ATTACH)
		return TRUE;

	ProxyExports(module);

	return TRUE;
}</code>

This strategy, while elegant, suffers from being so dynamic. We no longer include the export names in our static table unless we explicitly add them (re: static sinks). In addition, we receive execution after the import tables (IATs) of other modules might already contain references to our old export

table (static sinks again). There is no easy fix for the former that keeps us dynamic unless we simply add every export name we might expect to need across all DLLs. To fix the latter, we need to iterate loaded modules and patch in addresses to the real DLL. Nothing some code can't solve, but a convoluted solution to some eyes. The bulk of this strategy can be found in the Koppeling project below.

Another caveat is that references to, and within, the export table are relative virtual addresses (RVAs). Because of their size (DWORD), we are limited to placing our new export table somewhere within 4GB of the PE base unless it can fit inside the old one. Not an issue on x86, but certainly on x64.

Takeaway

Export forwarding is the easiest solution when it comes to proxying functionality. It's preparatory (we need to create the DLL with a specific hijack in mind), but the loader subsystem does the heavy lifting. We can make some nice improvements to the preparation process itself which we'll look at later. We like the flexibility of run-time generation, but it has weaknesses regarding static-sinks and their requirement for export names to be included in the file on disk. When it comes down to it, we might as well automate export forwarding.

Loader Lock

The LDR subsystem holds a single list of loaded modules for the process. To solve any thread sharing issues, a "loader lock" is implemented to ensure only one thread is ever modifying a module list at one time. This is relevant for hijacking as we typically gain code execution inside DllMain, which occurs while the LDR subsystem is still working on the module list. In other words, ntdll has to pass execution to us while the loader lock is still being held (not ideal). As a consequence, Microsoft provides a big list of [things you](#)

certainly SHOULD NOT DO while inside DllMain.

- Call LoadLibrary or LoadLibraryEx (either directly or indirectly). This can cause a deadlock or a crash.
- Call GetStringTypeA, GetStringTypeEx, or GetStringTypeW (either directly or indirectly). This can cause a deadlock or a crash.
- Synchronize with other threads. This can cause a deadlock.
- Acquire a synchronization object that is owned by code that is waiting to acquire the loader lock. This can cause a deadlock.
- Initialize COM threads by using CoInitializeEx. Under certain conditions, this function can call LoadLibraryEx.
- Call the registry functions. These functions are implemented in Advapi32.dll. If Advapi32.dll is not initialized before your DLL, the DLL can access uninitialized memory and cause the process to crash.
- Call CreateProcess. Creating a process can load another DLL.
- Call ExitThread. Exiting a thread during DLL detach can cause the loader lock to be acquired again, causing a deadlock or a crash.
- Call CreateThread. Creating a thread can work if you do not synchronize with other threads, but it is risky.
- Use the memory management function from the dynamic C Run-Time (CRT). If the CRT DLL is not initialized, calls to these functions can cause the process to crash.
- Call functions in User32.dll or Gdi32.dll. Some functions load another DLL, which may not be initialized.
- Use managed code.

Scary list, right?

In our experience, however, this list is not as bad as it might appear. For example, LoadLibrary is typically safe to

call within DllMain. In fact during static sinks, the loader lock is not re-acquired as long as the same thread is still in initialization. The call to LdrLoadDll will simply re-trigger dependency graph calculation and initialization. Does this mean that Microsoft is wrong to publish the list above? Absolutely not. They are just trying to prevent issues wherever possible.

The real answer to "Can I do <questionable thing> inside DllMain?" is typically "it depends, but avoid trying it". Let's check out one example where LDR synchronization can cause a deadlock:

hijack.cpp<code>DWORD ThreadFunc(PVOID param) {
	printf("[+] New thread started.");
	return 1;
}

BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
	if (reason != DLL_PROCESS_ATTACH)
		return TRUE;

	DWORD dwThread;
	 HANDLE hThread = CreateThread(0, 0, ThreadFunc, 0, 0, &dwThread);

	// Deadlock starts here
	WaitForSingleObject(hThread, INFINITE);

	return TRUE;
}</code>

Regardless of the sink we use, our DllMain will get stuck waiting for the new thread to finish, but the new thread will be waiting for us to finish. You can see this in the two call stacks for the threads:

DllMain<code>...
ntdll!LdrpCallInitRoutine

Theif!DllMain
KernelBase!WaitForSingleObjectEx
ntdll!NtWaitForSingleObject <- Waiting for the thread</code>

ThreadFunc<code>ntdll!LdrInitializeThunk
ntdll!LdrpInitialize
ntdll!_LdrpInitialize
ntdll!NtWaitForSingleObject <- Waiting for LdrpInitCompleteEvent
(can also be NtDelayExecution/LdrpProcessInitialized != 1)</code>

Inside a dynamic sink, you'll probably see the deadlock occur in LdrpDrainWorkQueue (as the process has already been initialized by then).

ThreadFunc<code>ntdll!LdrInitializeThunk
ntdll!LdrpInitialize
ntdll!_LdrpInitialize
ntdll!LdrpInitializeThread
ntdll!LdrpDrainWorkQueue
ntdll!NtDelayExecution <- Waiting for LdrpWorkCompleteEvent</code>

This outcome is frustrating, because starting a new thread is the easiest way to avoid LDR conflicts. We can collect execution in DllMain, kick off a new thread, and let our malicious code run there once the process has finished initializing. To avoid the deadlock, we could remove the **WaitForSingleObject** call like so:

ThreadFunc<code>BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
if (reason != DLL_PROCESS_ATTACH)
return TRUE;

DWORD dwThread;
HANDLE hThread = CreateThread(0, 0, ThreadFunc, 0, 0, &dwThread);

```
        // WaitForSingleObject(hThread, INFINITE);

        return TRUE;
}</code>
```

This works if the process stays up long enough for our code to execute, but this is a rare occurrence. Most likely, we will return execution to the primary module and it will exit quickly or throw an error if we haven't done proxying properly. Our thread will never get a chance to do anything useful.

Hooking for Stability

Lucky for us, we do hold execution long enough to implement a hook, so we can try to take over primary execution once LDR is done. Where exactly we place this hook is going to depend on where in the execution chain we sit.

- **Pre-Load:** The process is still being initialized and execution has not been handed over to the primary module. In this case, we'd likely want to hook the **entry point** of the primary module.
- **Post-Load:** The process has already started core execution, and we might be loaded as a consequence of a LoadLibrary call. The most optimal is to just hook the **last function** in the call stack which is part of the primary module. Whatever issues/errors bubble up can be ignored then.

To differentiate between these two scenarios was can just keep walking backward in the stack. If we find a return address for the primary module, we are probably post-load. Otherwise, the process likely hasn't kicked off yet and the entry point is our best bet. Naturally, we've built a proof of concept already so you don't have to pull your hair out:

https://gist.github.com/monoxgas/5027de10caad036c864efb32533202ec

Takeaway

Loader lock represents some challenges, but nothing too difficult as long as we respect it. **Starting a separate thread for any significant code is the best option.** In situations where we need to keep the process alive so the thread can continue run, we can use function hooking.

Koppeling

We started this post by introducing various complexities of hijacking. Let’s review and pair them up with relevant solutions:

1. **Stability of the source process:** Use function proxying, avoid loader lock.
2. **Maintaining code execution inter-process:** Use proxying and/or function hooking.
3. **Complexities of loader lock:** Use new threads and/or function hooking.
4. **Static export names:** Use post-build cloning, static definitions, linker comments, etc.

If there is one thing to communicate however, the solution space is quite large and everyone will have preferences. Our current “best” implementation combines the simplicity of export forwarding with post-build patching for flexibility. The process goes like this:

1. We compile/collect our “evil” DLL for hijacking. It doesn’t need to be aware of any hijacking duty (unless you need to add hooking).
2. We clone the exports of a reference DLL into the “evil” DLL. The “real” DLL is parsed and the export table is duplicated into a new PE section. Export forwarding is used to correctly proxy functionality.
3. We use the freshly cloned DLL in any hijack. Stability is guaranteed and we can maintain execution inter-process.

We're releasing a project to demonstrate this, and some other, advanced hijacking techniques called **Koppeling**. Much like our sRDI project, **it allows you to prepare any arbitrary DLL for hijacking** provided you know the final path of the reference DLL. We hope you find use for it and contribute if you love hijacking as much as we do.

https://github.com/monoxgas/Koppeling

Wrap Up

Our team is very passionate about not only how to weaponize a technique, but how to do it with stability and poise. We want to avoid impact to customer environments at all costs. This kind of care demands hours of research, testing, and development. Our Slingshot toolkit maintains seamless integration with the techniques we've detailed here to ensure our team and others can take full advantage of hijacking. As mentioned earlier, we also dive deeper into these topics in our Dark Side Ops course series if you're hungry for more.

We hope this post has provided a deeper understanding of this often misrepresented technique. Till next time.

– Nick (@monoxgas)