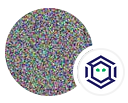
mavinject.exe Functionality Deconstructed



Matt Graeber · Follow
Published in Posts By SpecterOps Team Members · 6 min read · May 29, 2018

Attackers have been abusing mavinject.exe for some time to perform DLL injection. I first heard about mavinject from this tweet (and here), then later my interest was piqued again when Casey Smith started tweeting about practical use cases for abusing it; however, I lost interest for a time as mavinject ultimately doesn't permit execution of arbitrary, unsigned code (with application whitelisting enforced) — a frequent target of my research. Recently, I was running some scripts that identify signed applications which utilize the same APIs commonly employed by malware: CreateRemoteThread, VirtualAllocEx, etc., and mavinject.exe stood out (as it uses each of these APIs). This re-ignited my interest in mavinject and, while I was unable to fully abuse mavinject to achieve arbitrary code execution, I want this post to serve as documentation of my efforts, should anyone else decide to investigate further. Additionally, I'd like to offer some basic mavinject detection advice.

Among the suspicious APIs I scanned for, mavinject makes use of the following injection-related functions used commonly by malware:

- VirtualProtectEx — used to change memory page permissions in another process
- CreateRemoteThread — used to launch a thread in another process
- VirtualAllocEx — used to allocate memory in another process using the page permission of the developer's choosing
- WriteProcessMemory — used to write data into the virtual address space of another process.

While mavinject.exe obviously is not malware, I wanted to investigate if it could be abused for malicious purposes beyond just DLL injection.

. . .

mavinject for Traditional DLL Injection

The well-known method of abusing mavinject is to use it for traditional DLL injection via the following invocation:

```
mavinject.exe PROCESSID /INJECTRUNNING Path\To\Payload.dll
```

When used for DLL injection, mavinject performs the following actions:

1. Calls OpenProcess to get a handle to the target process. It requests the following access: 0x10043A (SYNCHRONIZE | PROCESS_QUERY_INFORMATION | PROCESS_VM_WRITE | PROCESS_VM_READ | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD). From a detection perspective, for those familiar with Sysmon, it would be a reasonable assumption to build a ProcessAccess rule, but be mindful that requested access will not always match the GrantedAccess field. Additionally, I am not aware of the uniqueness of this process access but that would be easy enough to find out by letting Sysmon capture ProcessAccess events for a while.
2. Calls VirtualAllocEx and WriteProcessMemory to write the DLL path to the target process. This string will be used as an argument to LoadLibraryW.
3. Calls CreateRemoteThread, which gets the target process to call LoadLibraryW on the DLL using the path that was written in step #2.

If this sounds like a familiar process, that's because this injection technique has been around forever and is extensively documented.

. . .

mavinject for Import Descriptor Injection

Upon looking at mavinject in IDA for a bit, the `/HMODULE=0x` command-line parameter stood out to me, as it hasn't been documented anywhere. I eventually figured out that it prepends an import table entry (an IMAGE_IMPORT_DESCRIPTOR structure) with the DLL name and ordinal number of your choosing using a variation of the Detours UpdateImports32/64 function. The `/HMODULE=0x` parameter would be used as follows:

```
mavinject.exe 4964 /HMODULE=0x013C0000 foo.dll 4
```

In this example, the 32-bit mavinject injects an import table entry consisting of “foo.dll” that “exports” a function with an ordinal of 4 into a 32-bit process (PID 4964) into the module at base address 0x013C0000 (the powershell.exe base address in this case). To be clear, there is no foo.dll that exports an ordinal of 4. This is just a fictitious example highlighting the fact that you can inject any string and ordinal number of your choosing. Upon injecting the import table addition, you can use WinDbg to validate the import entry addition:

```
0:009> !dh -i 013C0000
  _IMAGE_IMPORT_DESCRIPTOR 04860000
    foo.dll
              048600D0 Import Address Table
              048600C8 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference
80000004      Ordinal     4
```

In the output above, the 8 in the 80000004 refers to IMAGE_ORDINAL_FLAG32, which is used to tell the Windows loader to import a function by ordinal instead of by name. For more information about the import table, read this amazing article.

I found this functionality to be very interesting because ideally, I thought I might be able to somehow redirect import address table (IAT) entries. In practice, I was unable to achieve that goal and I’m not convinced that using mavinject *on its own* can be used to achieve arbitrary import table hooking. There *might* be some abuse potential mixing import table injection with other AppV-related DLLs that are intended to be injected like AppVEntSubsystems[32|64].dll, but I didn’t investigate that in too much depth.

I did find an interesting bug, however, in how mavinject applies IMAGE_ORDINAL_FLAG32 to the ordinal supplied at the command line. Ideally, IMAGE_ORDINAL_FLAG32 should be binary AND’ed (i.e. masked on) to an ordinal value instead of added to it. mavinject accepts a DWORD value at the command line as the ordinal number. The code then adds IMAGE_ORDINAL_FLAG32 (0x8000000) to the value specified which can result in an overflow. If an overflow occurs, then technically code that reads the import table would treat the import as an import *by name* rather than by *ordinal*. To highlight the overflow, I’ll have the import table appear to import the `This program cannot be run in DOS mode` import (a bogus “import” used to highlight the bug) — offset 0x4E of the PE header (0x80000000 + 0x4E == 2147483726 — which will cause the overflow and wrap back to 0x4E):

```
mavinject.exe 4964 /HMODULE=0x013C0000 foobar.dll 2147483726
```

Upon performing the import table addition, WinDbg confirms the confusion caused by the overflow upon listing imports:

```
0:009> !dh -i 013C0000
  _IMAGE_IMPORT_DESCRIPTOR 04870000
    foobar.dll
              048700F8 Import Address Table
              048700F0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

0000004E 6854 is program cannot be run in DOS mode.
```

I have yet to find a compelling way to abuse this in any useful fashion but I thought it was interesting. Also, a notable side effect of the UpdateImports32/64 function is that the PE checksum of the module for which an import table entry is added will be set to zero. This is notable because malware and tools like Cobalt Strike Beacon are known to clear/update/modify PE checksums as well.

. . .

One other command line invocation of mavinject.exe that's supported is the following:

```
mavinject.exe PROCESSID DLL_FULL_PATH_OR_FILENAME
```

This option is similar to the one that accepts `/HMODULE=0x` but instead, it adds an import entry to the main process module using the specified DLL that imports from ordinal #1. An additional side effect is that this method calls DetourCopyPayloadToProcess from the Microsoft Detours library, and saves the contents of the PE header in the main module in a `.detours` section of memory. This article explains what the `.detours` section is used for. There appears to be no way to introduce custom hook functions into the `.detours` section using mavinject alone, though.

So again, ultimately, I was unable to abuse import table injection as a useful primitive for anything interesting (aside from arbitrary data injection) even with the wraparound bug. I am documenting this functionality, though, in the hopes that it might inspire someone else to invest the time into abusing Detours functionality in signed code. For those interested in identifying signed code that uses the Detours library (since the library is intended to be statically linked into code), you would need to obtain symbols for all the modules you want to search through and then search within the PDB files for references to Detours functions.

. . .

mavinject Abuse Detection Suggestions

- Where command-line logging can be captured, look for "INJECTRUNNING". Attackers can and will rename abusable applications to evade detections so don't build detections off "mavinject.exe". "INJECTRUNNING" is required to have mavinject perform DLL injection so this would form the basis for a good DLL injection detection.
- To detect the execution of mavinject.exe in general, do not assume that an attacker won't rename it. Build a detection based on attributes in the Version Info resource instead like Description or OrginalFileName. Note however that an attacker can modify these fields but in doing so, the signature will be invalidated.
- Here is a sample set of Sysmon rules to detect mavinject usage:

```
<ProcessCreate onmatch="include">
    <!-- Catch mavinject DLL injection -->
    <CommandLine condition="contains">INJECTRUNNING</CommandLine>
    <!-- Catch mavinject regardless of its filename and command
line usage -->
    <!-- Note: an attacker can modify this field in the binary. It
will render its signature invalid but it will also evade this
rule. -->
    <Description condition="is">Microsoft Application
Virtualization Injector</Description>
</ProcessCreate>
<CreateRemoteThread onmatch="include">
    <!-- Catch mavinject DLL injection or any other DLL injection
where LoadLibraryW was called. May be FP prone. -->
    <StartFunction condition="is">LoadLibraryW</StartFunction>
</CreateRemoteThread>
```

. . .

Conclusion

In writing this post, I hope to have achieved any of the following:

- You arrived to this post also curious about the `/HMODULE=0x` command-line switch and you now no longer feel obligated to reverse that functionality yourself.
- You are now inspired to investigate Detours abuse using mavinject and AppV functionality which is, admittedly, an area I currently know nothing about but which I am *very* confident will have a lot of abuse/tradecraft potential.
- I encouraged you to consider blogging about your research findings and methodology even if you weren't ultimately successful. After all, failure breeds inspiration which ultimately, leads to eventual success.

Mavinject Reverse Engineering Dll Injection

 --

Written by Matt Graeber

635 Followers · Writer for Posts By SpecterOps Team Members

Threat Researcher

Follow