Function hooking

Function hooks refer to the broad technique of changing the regular execution flow of a program or operating system to execute the attackers code instead. This class of techniques is as old as the hills, and while it is usually easy to detect, a little bit of ingenuity can make life difficult for security software.

One of the biggest challenges when dealing with function hooks from a defenders perspective is that legitimate software (like firewalls, anti-virus etc) also use function hooking to accomplish their goals. This makes determining whether a 'hook' is benign or malicious virtually impossible. We do not make this decision for you, but provide you with all the possible information to track down the hook and the code it calls so that you can decide for yourself whether it is malicious or not. Helios allows you to 'unhook' certain types of function hooks.

There are two basic ways that the flow of execution can be changed with hooks, they are hooks that modify function pointers and hooks that patch the actual bytes of the function (detour patching). These two techniques are explained below and then illustrated both in userland and kernel mode.

Function Pointer Hooks:

Windows operations rely on arrays of function pointers to call code. An attacker picks a function that he wants to hook, and modifies the function pointer address to point to his code. As a result, everytime the function is called, the function pointer points to the attackers code and it gets called in place of the legitimate function. Examples of this are IAT hooks and SSDT hooks in userland and kernel mode respectively.

Function pointer hooks are usually easy to detect as all one needs to do is query the appropriate function pointer tables and determine whether they point to the legitimate code. This requires that one is aware of where the original code resides in the first place, but sometimes it is enough to check that the code being called falls into the address space of the correct module. It is difficult to determine whether a function pointer hook is benign or malicious as there are many legitimate uses for these hooks.

Detour patches:

Detour patches (also known as in-line hooks) are the other form of function hooking that can be used generically from userland or kernel mode. Instead of modifying a pointer to a function (like the SSDT hooks or IAT hooks), the detour hook 'patches' or overwrites the actual code start of the function to call into the attackers code. Once the attackers code is called, he can call the original function (minus his overwrite) and then modify the return results.

Once again, detour patches can be used benignly, and Microsoft has used them with hotfixes to solve bugs. In order to make life easier for themselves, Windows XP Service

Pack 2 onwards has a slightly different function prolog so that when the prolog bytes are overwritten, a perfect 5 bytes will be overwritten without any alignment problems.

Original prolog
Push ebp
Mov ebp, esp

Modified prolog
Mov edi, edi
Push ebp
Mov ebp, esp

Detour hooks can be difficult to detect as an attacker does not necessarily have to implement them within the first five bytes (it is of course, far easier and more convenient to do this). There are many ways to detect detour patches, listed below are a few common techniques:

1. Compare the start of a function in memory with the bytes on disk. This is useless if the attacker is able to change the bytes on disk to reflect his patch in memory.

2. Parse the initial bytes to determine whether they contain a code branch such as a jump or call operation. This is highly effective, but can generate false positives when a legitimate patch is made.

3. Calculate a complete cryptographic hash of the function bytes on disk and compare it against the bytes loaded in memory. This method is effective because it can inform you of a detour patch or modification anywhere in the function. However, it suffers because it cannot pinpoint the actual location where the detour patch occurs (this can be solved by then comparing the functions at the byte level).

4. In certain cases where one is aware that the function cannot branch to code outside of its own module space, one can check that all code branches are intra-modular. Any jump that targets the space of another module can be considered malicious.

As we stated before, function hooks can occur either in userland or kernel mode depending on the level of information that needs to be controlled. Occasionally, a userland hook can be more effective and harder to detect than a kernel mode hook. This is a typical case of the coolest technique not necessarily being the most powerful.

Userland hooks:
As an example of how userland hooks work, let's consider programs that wants to take control of the send() function in Winsock. In order to do this, the malicious program needs to ensure that its code replaces any calls to send() and after copying the buffer data, passes the data to send.

The up-side to performing this sort of hook in userland is that you can effectively change who is responsible for sending data. Consider a keylogger that needs to transfer a logfile to a remote server. If the system has a personal firewall, it will alert the user when the keylogger process attempts to access the Internet. Instead, the keylogger can hook send() in iexplore.exe (Internet Explorer) and when Internet Explorer makes a legitimate request (allowed by the firewall), the keylogger first sends out its data and then sends IE's request. This effectively bypasses the personal firewall as it sees that the owner of the send request is Internet Explorer.

Function pointer hooks in userland:
Whenever code is called in another library, the calling process 'imports' that code by making an entry in its IAT (import address table) to point to the function that it is calling. Whenever calls are made to the imported function, they are made through the IAT address. Thus a rootkit that modifies the IAT address of a function to point to its code will get called every time that function is called. This is perhaps the simplest form of userland hooking, however it can be problematic because some applications may dynamically load the library using a call to loadlibrary() and getprocaddress(), in this case, there will be no entry for the function in the IAT.

Detour hooks in userland:
From an attackers perspective, detour hooking (overwriting the function bytes) is prefered to function pointer hooking as it is more difficult to detect. The first thing to understand with this technique is that every DLL gets loaded only once in physical memory. After that, every process linked with that library gets an entry in its page directory table that points to this one copy. Thus, even though every process believes it has a different copy of ws2_32.dll, there is actually only one physical copy in memory that is mapped via a virtual address with every process that wants to use it.

An attacker can choose to hook send on a process specific or system wide basis. If he wants to modify it at the process level, he needs to inject his code into the victim process. This can be accomplished by setting a CBT hook that gets called every time a process gets created (technically, a CBT hook could be used system-wide, but this is inefficient), or by injecting his code into the victim process using a method like CreateRemoteThread (which allows the loading and execution of a library in another process space). Once his code is injected, he modifies the initial bytes of send with a detour hook (see below).

When Windows sees the request to overwrite the bytes in ws2_32.dll, it realises that this DLL is shared between many processes and that the write will affect all of them. So it initiates a copy-on-write operation and creates another copy of ws2_32.dll in physical memory. It then changes the directory entry of the process that asked for the write to point to the new copy of the DLL. This way, only the process that requested the write gets a modified version of the DLL.

If the attacker wants to modify the send() call system-wide, the copy-on-write operation poses a problem, as he will receive a unique copy of the DLL which will not affect other processes. To accomplish his task, he can however access physical memory and change the bytes in the original single copy of ws2_32.dll in physical memory. This way, every process that uses ws2_32.dll will receive the modified DLL.

Kernel mode hooks:
The same two principles for hooks apply in kernel mode. We'll give an example of both function pointer hooks and detour patches in kernel mode:

Function pointer hooking in kernel mode (System call table hooking)
The concept and implementation of system calls differs from O/S to O/S. For example, in the UNIX world, system calls are made by setting up the appropriate registers and then calling an interrupt (such as INT 80) to invoke the system call.

System call table hooks refer to hooks that modify function pointers to Windows native API functions. These functions are called in kernel mode to perform all the basic tasks that Windows needs to undertake to operate.

They are usually wrapped by a higher level function operating in userland. So, for example, when you list the contents of a directory using the userland FindNextFile() API, Windows will internally call the NtQueryDirectoryFile native API in kernel mode to pass those results to explorer. Hooking these functions can be extremely powerful as Windows relies on the data returned very heavily. An attacker in control of these functions can decide what Windows 'sees' on the system, while a defender can control 'choke-points' or functions that malware is likely to use.

In the Windows world, when a userland function is called, it makes a context switch into ring-0 to call the appropriate system call. This happens by the userland code calling a 'stub' function in NTDLL.DLL / USER32.DLL / GDI32.DLL or a similar module which moves the appropriate system call number into the EAX register and then performs the transition to ring-0 using the INT 2E or SYSENTER instruction.

The system call number that is moved into the EAX register is actually an offset into a series of system call tables (there are 4 of them) the most popular of which are table 0 – the SSDT (system service dispatch table) and the GDI function table. The SSDT functions

reside in NTOSKRNL.EXE (or NTKRNLPA.EXE for systems with physical address extensions) whereas the GDI functions reside in WIN32K.SYS.

The 13th bit of the EAX register determines which of the four tables is to be looked up. If it is 0, the SSDT will be queried, whereas if it is 1, the GDI table will be queried.

In order to hook the system service tables, an attacker has to simply figure out the system call number of the function he wants to hook and then replace the function pointer at that offset into the appropriate table with his code.
The easiest way to discover the service number is to look at the stub function that sets up the switch to ring-0 as it moves the table number / service number combination into the EAX register.

As far as NTDLL.DLL is concerned, the stub functions are all prefixed with 'Zw'. They call corresponding code in NTOSKRNL.EXE which is prefixed with 'Nt'. So the system call 'NtQuerySystemInformation' in NTOSKRNL.EXE will be called by the stub 'ZwQuerySystemInformation' in NTDLL.DLL. The Zw* functions are only responsible for setting up the service number in the EAX register and then performing the context switch. Interestingly NTDLL.DLL also has Nt* functions, which have the same code as the Zw* stubs. Similarly, NTOSKRNL.EXE has Zw* functions which are identical to the stub in ring-3. This can be handy as having the Zw* stubs accessible in NTOSKRNL.EXE (which runs in ring-0) means you will have easy access to the service number.

To get the service number, all one needs to do is disassemble the Zw function, the first byte will be the MOV EAX instruction, the next DWORD will be the service number. Once this is obtained, one gains access to the appropriate system call table and replaces the function at [servicenumber] offset.

Finding the service tables can be accomplished in multiple ways. The easiest way to obtain the address of the SSDT table is to take it directly from NTOSKRNL.EXE where it is exported. A simple statement like:

```
__declspec(dllimport) sdt KeServiceDescriptorTable;
```

is enough to gain the tables address. However, importing the symbol is not particularly stealthy as there will be an entry in your modules import address table showing that this symbol is being loaded.

Luckily for us we can obtain the service table entries rather easily. If we call the function PsGetCurrentThread() we obtain the _ETHREAD structure of the currently executing thread. The first DWORD of this structure is a pointer to a _KTHREAD structure. At offset 0xE0 in the _KTHREAD is a pointer to a member called ServiceTable.

This ServiceTable entity (also called the shadow table) is four back to back structures each representing one of the four tables (NTOSKRNL.EXE, WIN32K.SYS and the other two unknown tables). Each table is represented by a 4 DWORD structure, the first DWORD being the pointer to the table and the third DWORD being the total number of entries (services) in the table.
It is also possible to gain access to the shadow table through the _KPRCB (Kernel's Processor Control Block) structure which is always located at 0xffdff120. At offset 0x04 is the currently executing thread _KTHREAD structure. Once again 0xE0 into this structure, we find the shadow table.

After finding the table, the attacker replaces the appropriate function pointer to point to his code. In his code, the attacker will usually call the 'real' original function and either filter the parameters passed to it (filtering the input data) or modify the return data of the function (filtering the output / result data). From a defensive perspective, one often curtails the functionality of the API by returning an access denied status code and not calling the original code (for example if one wishes to block access to \\device\\physicalmemory, one will monitor any requests to this device and then terminate the API with an access denied return code).

Defending against service table hooks is relatively easy as the actual code already exists in the relevant module such as ntoskrnl.exe (or ntkrnlpa.exe for systems with physical address extensions). One can compare the bytes in memory against the bytes in the physical file on disk, or ensure that each of the function pointers falls in the kernel memory space. Security software such as firewalls and antivirus products regularly hook the SSDT to perform their functions. It is not difficult however, to determine what kernel module's address space the hook points to and thus, one can determine which driver is actually hooking the table.

Functions that are regularly hooked by malware include NtQueryDirectoryFile to hide malicious files and NtQuerySystemInformation to hide processes from task manager.

Detour Patching in kernel mode:
Instead of modifying a pointer to a function (like the SSDT hooks above), detour patches that overwrite the actual start of the function to call into the attackers code can be used. Once the attackers code is called, he can call the original function (minus his overwrite) and then modify the return results.

An example of the detour patch in kernel mode would be to take one of the system call functions, but instead of modifying its pointer in the SSDT, the attacker goes to the code of ntoskrnl.exe (or ntkrnlpa.exe for systems with physical address extensions) and changes the initial bytes to perform a jump to his code. This method can become slightly complicated as there are functions in kernel mode that don't have a prolog, thus making it more complicated for the attacker to overwrite the bytes as he has to be concerned

about what instructions are being modified. To perform this generically, the attacker will have to implement some form of disassembly functionality that allows her to figure out how many bytes each instruction is going to take and then overwrite the appropriate number of bytes after saving the legitimate instructions.

Direct Kernel Object Manipulation

Till now we've dealt with hooking functions. While this is an extremely effective and safe method of subverting security, it is not always very stealthy. A more stealthy but potentially less stable method of hiding information is by modifying the actual data that the kernel maintains. This technique is known as Direct Kernel Object Manipulation and it was pioneered by fuzen_op with the Fu rootkit.

The basic idea is that the kernel maintains the system state through various data structures. All the functions of the operating system query this data. If one were to remove information from these data structures, the kernel and the corresponding functions would be none the wiser. This is particularly effective when hiding processes.

The Fu rootkit demonstrated how to hide a process from Windows without any function hooking. The kernel internally maintains a circular linked list of structures known as EPROCESS structures. Each one of these EPROCESS nodes in the linked list represents a running process on the system and it stores all the information about the process such as its running threads, path etc. All Windows functions that deal with process data will at the lowest level query the EPROCESS structures. The Fu rooktit would hide a process by unlinking it from the linked list. It does this by pointing the next node pointer of the previous node to the node following it and pointing the previous node pointer of the following node to the previous node (excuse that complicated sentence, this is best seen diagramatically). This way, when the linked list is walked, there is no reference to the hidden process. Interestingly enough, since the EPROCESS structures are not connected with thread execution, the process is still able to get processor time and continue executing!

From the defender's point of view, the most obvious way to tackle processes hidden in this manner is to monitor the thread scheduling since the actual threads themselves must continue to execute. Each thread should have a corresponding EPROCESS structure, if there is a thread that is operating without an EPROCESS structure parent process, it has probably been hidden from the EPROCESS linked list. Furthermore, the information stored in the _ETHREAD structure for each thread gives us a pointer to the original EPROCESS entry (that is now orphaned and hanging in space) so we should be able to access all the information about the process and even 'relink' it into the list.

Physical Memory Tricks

All process memory in Windows for each process is referenced as virtual memory. In other words, each process believes that it has the entire RAM space in which to play. In

actual fact, the actual mapping to the system's physical memory happens at a much lower level. If an attacker can gain direct access to physical memory, she will be able to modify the process spaces of any arbitrary process as well as perform kernel level operations without having to be in ring-0. In short, giving an attacker access to physical memory is not a good idea.

The simplest way to protect the integrity of physical memory is to prevent the attacker from being able to open a handle to it. This can be accomplished by hooking the NtOpenSection() system service in NTOSKRNL.EXE. The third parameter to this function is a pointer to an ObjectAttributes structure that contains the name of the object being opened as a UNICODE string. If this object is \\device\\physicalmemory, one can block the call by returning a STATUS_ACCESS_DENIED.
This is insufficient as in Phrack '59, crazylord documented a simple bypass to this check. The basic problem is that one can create a symbolic link to \\device\\physicalmemory with an arbitrary unfiltered name and then pass this symbolic link to NtOpenSection. Windows will not resolve the symbolic link to the proper name and then call NtOpenSection but will call it with the symbolic name itself. Thus, one can bypass the filter.

The easiest way to prevent this is to stop anyone creating a symbolic link to physical memory. This can be be accomplished by hooking NtCreateSymbolicLinkObject and monitor the fourth parameter which is a pointer to a UNICODE_STRING that will contain \\device\\physicalmemory if someone is trying to create a symbolic link to physical memory.

Hidden Processes

Hiding a malicious process and its execution time from the system is one of the most important things for an attacker to do. If the process is visible, it is highly likely that a power user will be able to kill it. It is not exactly stealthy either to steal large amounts of CPU cycles and not have them accounted for.

Processes can be hidden using a variety of methods, including an SSDT hook or in-line hook on NtQuerySystemInformation. This function is queried by the Windows task manager to obtain the list of running processes. There is also a DKOM method of hiding processes. When a process starts, Windows creates an EPROCESS structure for it containing all the information related to that process. This EPROCESS structure is part of a linked list of similar structures that collectively give Windows information on all the running processes. An attacker can modify the linked list entries to 'unlink' his process' EPROCESS structure from the list. For all practical purposes, Windows will believe that this process does not exist. This is the technique that was pioneered by the Fu rootkit authored by fuzen_op.

Malware can also remove its entries from the handle tables or PspCidTable as this is virtually a single point of information that Windows relies on. The FuTo rootkit authored by Peter Silberman is a fine example of how effective this technique is. Detecting a process that is hidden in this manner can be cumbersome, but it is possible.

Detecting hidden processes can be done in a variety of ways.

1. Check for hooks on the NtQuerySystemInformation function to ensure that the function is returning reliable information.
2. Manually walk the EPROCESS linked list and compare that information with the results of NtQuerySystemInformation.
3. Brute-force the process ID space using the OpenProcess function and look at its return values, if a certain PID returns a valid handle but does not feature in the system's information lists, it has been hidden. This technique is the technique used by F-Secure's BlackLight tool.
4. Brute-force the process ID space in kernel mode. For all practical purposes this is identical to the check above, but it is performed in kernel mode.
5. Manually walk the threads linked list and determine that all threads have a valid EPROCESS attached to them. If the thread is 'orphaned', it has probably been hidden.
6. Take control of the Windows swapcontext function (the thread scheduler) and maintain a list of threads and their corresponding EPROCESS structures. If a thread doesn't have an EPROCESS structure, it has been hidden. This technique is credited to Kimo Kasslin and is by far one of the most effective checks for hidden processes simply because if the malicious process wants CPU time, it will have to have its thread swapped in and out.

Hidden files

For a rootkit to be succesful, it has to hide its files from the user on disk. Failure to do so will result in a quick discovery of the subversion and corresponding deletion of the malicious files. Hiding files is thus a standard feature of all rootkits. Files can be hidden in the following ways:

1. Hooking the NtQueryDirectoryFile native API function that is responsible for generating listings of files and directories. This can be accomplished with an SSDT or in-line hook.

2. Hiding the files in an NTFS alternate data stream. These alternate data streams are not generally visible to the system components such as explorer and can only be accessed by name, thus making them difficult to detect. An NTFS alternate data stream can hide any file 'behind' another file, for example, an executable PE file behind an ASCII text file.

Detecting hidden files can be complicated depending on what length the attacker has gone to to hide the data. Some ideas that can be used are:

1. Compare the results obtained by querying the regular Windows file functions against a list of files generated by parsing the low-level disk structures (such as the FAT and NTFS data). This form of cross-view detection is extremely powerful. The technique is credited to Mark Russinovich of SysInternals, and is used in his patent-pending tool Rootkit Revealer. However, a non-persistent rootkit (that doesn't write itself to disk) will not be detected this way. An attacker could also 'drop' code into an existing system file and change that files execution path, thus evading detection.

2. Monitor and block access to NTFS alternate data streams. This can be tricky as Windows itself has some legitimate uses for alternate streams as do certain anti-virus products. However, by white-listing known-good programs, one can effectively disable the alternate data stream 'feature' while having the system function normally.