# FATKit: Detecting Malicious Library Injection and Upping the "Anti"

AAron Walters
4TΦ Forensic Research
Washington, DC, USA
{fatkit}@4TΦ.net

**Abstract**

In this white paper, we discuss how the Forensic Analysis ToolKit (FATKit) can facilitate the process of enumerating suspicious artifacts manifested as a result of remote library injection. We discuss a number of techniques that have proven effective at elucidating artifacts that are by-products of advanced exploitation methods frequently characterized as anti-forensic or stealthy. One significant differentiator from the majority of previous work is the fact that we do not rely on the integrity of the potentially compromised operating system, but instead perform our analysis offline on a trusted capture of volatile memory (RAM) [10, 22, 7] [1]. While many of the previously published techniques have focused on detecting attacks in real time, we are focused on facilitating the forensic analyst's ability to extract memory-resident evidence from the information system under investigation.

## 1 FATKit: Forensic Analysis ToolKit

The Forensic Analysis ToolKit (FATKit) is a cross-platform, modular, and extensible digital investigation framework for analyzing volatile system memory [20, 19]. This modularity was designed to support multiple operating systems, but also with the goal of being able to support various hardware architectures as well. For example, we currently have profiles for Windows 2000, Windows XP, Windows Server 2003, Windows Vista, and Linux. The framework is intended for advanced researchers, law enforcement professionals, and forensics analysts who are interested in extracting and interpreting relevant information in the wake of a crime or incident. Unlike previous work in this area, the FATKit project is working to combine the latest research in volatile memory forensics, memory informatics, static analysis, and multi-relational data mining. FATKit offers a unique ability to automatically correlate information across multiple data stores (packet captures, volatile memory, filesystem, etc.), which facilitates an ability to enumerate suspicious artifacts on a system. Recently, the offensive communities have begun to focus time and effort on "anti-forensics" and stealthy advanced exploitation techniques. Many of these techniques leverage the complexities associated with physical memory analysis and rely on the fact that volatile memory is often an opaque component of information systems. These techniques have also exploited the closed nature of investigatory tools that have pigeon-holed digital forensics examiners. Unlike previous work, which introduced FATKit's unique visualization capabilities [19], in this white paper we will demonstrate FATKit's powerful ability to support analysis modules and the ease of building those modules using supporting tools and APIs. This will be the first paper in a series addressing anti-forensic and stealthy exploitation techniques.

[1] Until these mechanism are uniformally adopted forensic examiners must often leverage a potentially compromised operating system to create the memory image [30]

## 2 Overview: Remote Library Injection

Many operating systems provide mechanisms for dynamic linking, which allows a program to load and unload routines at runtime. Microsoft Windows implements this functionality using shared objects called Dynamic-Link Libraries (DLLs) [18]. DLLs are dynamically loadable objects that are mapped into the address space of a running process. While this has proven to be a very important feature of modern operating systems, attackers have also developed methods to exploit dynamic linking in order to reduce their obtrusiveness in the allocation of system resources and thwart conventional forms of filesystem forensic analysis. A remote library injection attack occurs when an adversary, typically after taking advantage of a remotely exploitable vulnerability, injects a DLL into the address space of an active process [15]. This technique has commonly been labeled as "anti-forensic," since data is not written to any non-volatile storage and because any indication of the injected DLL would typically be lost during common incident response procedures [14].

In this paper, we will demonstrate how FATKit can be used to implement an analysis module capable of enumerating suspicious data found in volatile memory. Specifically, FATKit will be used to search for in-memory artifacts left as a result of the sophisticated remote library injection techniques implemented by Miller and Turkulainen for the Metasploit Framework [17]. The Metasploit Meterpreter creates an extensible command interpreter server inside of an exploited process on a remote machine. One reason the aforementioned techniques are particularly sophisticated relates to the fact that a number of actions are taken to reduce their obtrusiveness on the system. For example, efforts are made to only temporarily hook system APIs and then undo the changes until the functionality is needed again. While other techniques can be coupled with remote library injection to make them even stealthier (e.g., object manipulation attacks [12, 13]), the purpose of this paper is to demonstrate how FATKit can augment the forensic analyst's ability to elucidate suspicious artifacts on a system under investigation. This will include artifacts that are intrinsic to particular attack techniques, those that are simply by-products of specific implementations, and those that are needed to support advanced functionality (e.g., Meterpreter). Regardless of their origin, all three types of artifacts are important, as they have the potential to draw the suspicion of the analyst, focus his or her attention, and possibly lead to the discovery of other forensic evidence.

## 3 Experimental Setup and Evidence Acquisition

The target system used for this discussion was a VMWare Workstation 5.5 virtual machine with 256 MB of RAM. The demonstrated analysis techniques are not dependent on this configuration and work similarly on images taken from machines running operating systems natively using a number of different acquisition mechanisms. The vulnerable machine was running an unpatched version of Microsoft Windows 2000. In this scenario, the attacking machine was running the latest version of the Metasploit Framework (framework-3.0-alpha-r3) [17]. The targeted Windows 2000 machine was compromised by exploiting the MS03_026 vulnerability [21] in Microsoft RPC DCOM. This was the exploit vector used for delivering the `reverse_tcp` Meterpreter payload. Once the machine was exploited and the Meterpreter server had been installed, volatile memory was captured from the virtual machine. The image of volatile memory collected in this step will be the basis for the forthcoming discussion and analysis.

## 4 Utilizing FATKit for Analysis

Once the compromised image had been successfully acquired, it was loaded into the FATKit framework along with the profile for Windows 2000. These profiles are automatically generated offline and contain

important meta-information about the system under investigation including relevant symbols, data structures, object signatures, and hashes of shared libraries and executables. In previous work involving Linux, we demonstrated the ability of the FATKit framework to enumerate lists of kernel objects based on pointers found in the symbol table [19]. In this example, we begin by leveraging a complementary technique – performing a linear scan of the physical address space to allow our accumulators to find objects of interest (e.g., devices, drives, processes, threads, memory-mapped files). While others have used similar techniques [1, 4, 26], the main difference is that our signatures for these objects are automatically developed using advanced techniques in memory informatics that are based on mutation rates developed through a consensus technique for ungapped data streams. Our work on memory informatics was inspired by the recent work in protocol informatics [3]. We also have the ability to automatically perform these scans across address spaces. Accumulators are used to return the set of Python objects that describe the low-level instances in the kernel's physical address space. Each of these potential kernel objects is then evaluated with respect to their semantic integrity [23] in relation to other objects and the operating system's semantics in order to determine their likelihood of being a valid object. In this example, we will focus primarily on process and thread objects, as they are the most relevant for the detecting the remote library injection technique under consideration.

In the remainder of this section, we will discuss the important data structures and the relationships between those data structures of which we will make extensive use. In order to facilitate this discussion, we have included pseudo-code which demonstrates how DLLs can be enumerated using the FATKit interface in Figure 1 and Figure 2. We have also tried to illustrate the pertinent relationships in Figure 3. Much of the utilized information was extracted from numerous sources [25, 29]. Processes in Windows are represented using an executive process block ( **EPROCESS** ) structure. These **EPROCESS** structures, which reside in kernel address space, contain information about processes on the system. In this discussion we will focus on two important members of this structure – the *Pcb* and the *Peb*. The first field of the **EPROCESS** structure is the *Pcb* (process control block). The *Pcb* is a kernel process block of type **KPROCESS** and contains information related to scheduling. From the *Pcb* substructure, we find the *DirectoryTableBase* member, which is a pointer to the physical address of the process page directory. Using the address of the process page directory, we create a new Flat-paged virtual address space for each process using our `create_address_space()` function. Once the address space is created, we can interact with the underlying physical memory completely within the context of the emulated address space of each individual process. This also allows us to differentiate the pages in physical memory associated with each virtual address space. The other important member of the **EPROCESS** structure is the *Peb*, which is a pointer to an object of type **PEB**, the process environment block. This object is typically stored in user address space since it needs to be modified from user space by such things as DLLs. The **PEB** holds information about the current state of the process.

Using the previously instantiated address space for the current process, we use the `vtop()` function to perform a virtual to physical address translation to determine if the pointer value is still in memory and, if so, its offset in the physical address space. If the address is successfully translated, we use `create_object()` to create a **PEB** object in the process' address space. The **PEB** also has two fields which are extremely useful in detecting remote library injections. The first field is *ImageBaseAddress*, which is a pointer to the virtual address of the memory-mapped PE (Portable Executable) image of the program. PE is the file format used by the Windows operating system for representing such things as executables, DLLs, device drivers, etc. We will put off the discussion of the processing performed by the PE module until we discuss extracting DLLs.

Another important field is *Ldr*, which is a pointer to the virtual address location of the **_PEB_LDR_DATA** object for this process. The **_PEB_LDR_DATA** maintains information filled-in by the loader and is updated when DLLs are loaded or unloaded. Once a check is performed to make sure the *Ldr* address is still paged in, `create_object()` is once again employed to instantiate the object at this address within the

```
1
2  def ListModules(analyzer):
3      # The analyzer is the central analysis engine. It
4      # maps a profile to an image.
5      profile = analyzer.get_profile(profile)
6      # Access the linear physical address space
7      phys = analyzer.get_address_space("Physical")
8      # Perform a linear scan for process objects
9      pobjs=linear_search(phys,profile.get_signature('_EPROCESS'))
10     # Validate the semantic integrity of those objects
11     vobjs=validate_processes(pobjs)
12     # Iterate through the validated objects
13     for x in vobjs:
14         # Access the Directory Table Base associated with
15         # this process
16         directorytablebase=x.member('Pcb').member('DirectoryTableBase')
17         directorytableval=directorytablebase.get_member(0).value()
18         # Reconstruct the processes virtual address space
19         analyzer.create_address_space('process','Flat-paged', \
20                                       directorytableval)
21         proca = analyzer.get_address_space('process')
22         # Perform a virtual to physical address translation
23         pebp = proca.vtop(x.member('Peb').value())
24         # Check if virtual page is memory resident
25         if pebp != PAGE_OUT:
26             # Instantiate object in the process' address space
27             peb = profile.create_object(['_PEB'], proca, \
28                                         x.member('Peb').value())
29             # Validate that executable is memory mapped
30             imgp=proca.vtop(peb.member('ImageBaseAddress').value())
31             if imgp != PAGED_OUT:
32                 # Process the PE Image
33                 validate_pe(imgp)
34                 # Check if loader information is memory resident
35                 peb_ldr_p=proca.vtop(peb.member('Ldr').value())
36                 if peb_ldr_p != PAGED_OUT:
37                     # Instantiate object
38                     ldr_obj = profile.create_object(['_PEB_LDR_DATA'], proca, \
39                     peb.member('Ldr').value())
40                     module_base = ldr_obj.member('InLoadOrderModuleList').member('Flink').value()
41                     init_module = profile.create_object(['_LDR_MODULE'], proca, \
42                                                         module_base)
43                     list_do_check(init_module, ['InLoadOrderModuleList', \
44                                                 'Flink'], ProcessModule ,module_base)
```

Figure 1: Illustrative example of listing loaded DLLs (Python).

virtual address space. From this object, we have access to three sets of linked lists that relate three different orderings of the DLLs that are loaded in this process' address space. Each of these doubly-linked lists (InLoadOrderModuleList, InMemoryOrderModuleList, InInitializationOrderModuleList) are composed of pointers to objects of type **_LDR_MODULE**. We iterate through these embedded list pointers using methods designed to facilitate walking lists. Once each **_LDR_MODULE** object has been instantiated, we can use the information found in these objects to look for suspicious artifacts associated with the DLLs loaded in each of the process.

While there are a number of members of **_LDR_MODULE** that will be discussed in the following section, we will focus on the *BaseAddress*. As with the *ImageBaseAddress* found in the **PEB**, *BaseAddress* is a pointer to the virtual address of the memory mapped PE (Portable Executable) image of the program. Both of these objects are processed using the FATKit PE module which is used for such things as extracting the memory mapped files, providing accumulators for important data, evaluating the semantic integrity of the PE data structures, and creating a Relative Virtual Address Space (RVAS) for each PE image. A Relative

```
1
2  def ProcessModule(x):
3      proca = analyzer.get_address_space('process')
4      # Access members of the obect
5      str=unicode_string(x.member('FullDllName'))
6      base_addr = x.member('BaseAddress').value()
7      timestamp = x.member('TimeDateStamp').value()
8      loadcnt = x.member('LoadCount').value()
9      # Output module info
10     print "Name: %s BaseAddress: (0x%08x) TimeDateStamp:" \
11         "(0x%08x) LoadCount %d"%(str,base_addr,timestamp,loadcnt)
12     # Validate DLL PE
13     dllimg=proca.vtop(base_addr)
14     if dllimg != PAGED_OUT:
15         validate_pe(base_addr)
16     # Perform a linear scan for module objects in process address space
17     pmobjs=linear_search(proca,profile.get_signature('_LDR_MODULE'))
```

Figure 2: A function to print information about loaded DLLs called from Figure 1.

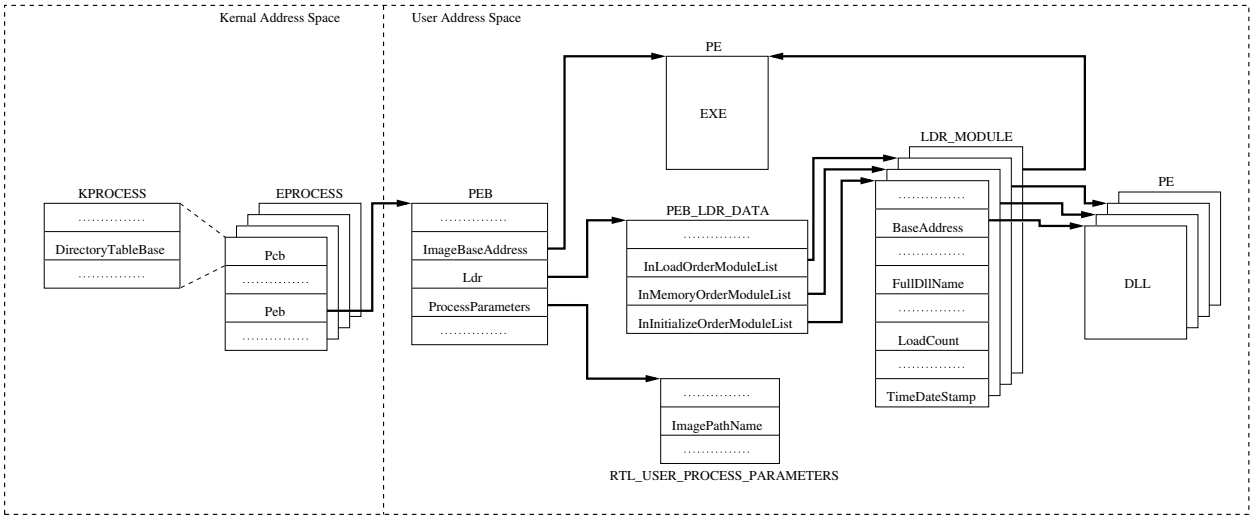


Figure 3: Pertinent EPROCESS Memory Structures and Relationships

Virtual Address (RVA) allows address pointers to be specified despite the fact that a PE file can be loaded arbitrarily within the processes address space. Thus, an RVA is typically an offset from the base address of where the PE is loaded in memory.

The information used to create the following modules was found in [24]. Figure 4 has been added to facilitate the discussion about the PE relationships. The PE module begins by using the base address as the frame of reference. The `create_object()` method is used to instantiate the **_IMAGE_DOS_HEADER** object within the virtual address space of the process. From this object, the module extracts the *e_lfanew* member, which provides the offset of the PE header from the base address. Using this address, the module instantiates a new object of type **_IMAGE_NT_HEADER** . The *FileHeader* member of this object is a structure of type **_IMAGE_FILE_HEADER**, which contains important general information about the file. This includes the *NumberOfSections* member, which indicates the number of sections that can be found in the section table. Another important member of **_IMAGE_NT_HEADER** is the *OptionalHeader* member, which is a structure of type **_IMAGE_OPTIONAL_HEADER**. Using this object, we are able to find the *DataDirectory* member of **_IMAGE_OPTIONAL_HEADER**, which is an array of structures of

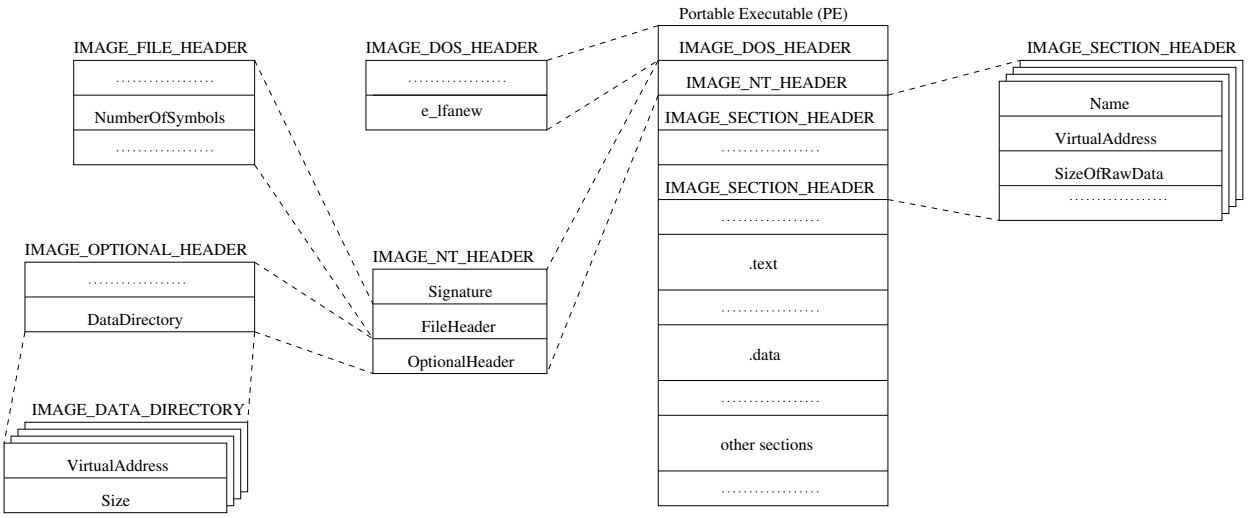


Figure 4: Pertinent PE Structures and Relationships

type **_IMAGE_DATA_DIRECTORY** that indexes important parts of the executable. This is accomplished with the two members of **_IMAGE_DATA_DIRECTORY** , *VirtualAddress* and *Size*. The *VirtualAddress* member is the RVA of the data and the *Size*, as expected, relates the amount of relevant data at that location. We will primarily focus on two such directories, the *Export table* and the *Import table* .

The export section is used for making symbols (functions, variables) available within the the processes address space that can be dynamically imported. Figure 5 provides an illustration of the important export relationships and will be used to facilitate this discussion. Using the RVA found in the *Virtual Address* member of the *Export table*, we find the **_IMAGE_EXPORT_DIRECTORY** structure. This is the first element in the array of **_IMAGE_DATA_DIRECTORY** structures. The *AddressOfFunctions* member relates to an RVA for the Export Address Table (EAT). The *AddressOfNames* member corresponds to the RVA for the Export Name Table (ENT) and the *AddressOfNameOrdinals* relates to the RVA for the Export Ordinal Table (EOT). The Export Address Table is composed of an array of RVAs, where each RVA points to the location of the exported symbol. The Export Name Table, on the other hand, is also an array of RVAs, but these are the addresses of the lexicographically sorted names of the symbols. Finally, the Export Ordinal Table is an array of 2-byte values used to index the EAT. Each elements in the EOT matches the element at that index in the EAT. The EOT provides the mapping between the name of the exported symbol and its ordinal index within the EAT. By traversing these tables, we are able to enumerate the symbols exported by this PE file.

On the other hand, the Import table relates to those symbols which are dynamically imported by this PE file. Figure 6 illustrates important Import table relationships and will be used to augment the discussion. Using the *Virtual Address* member of the second element of the array of **_IMAGE_DATA_DIRECTORY** structures we find the RVA for an array of **_IMAGE_IMPORT_DESCRIPTOR** structures. Each element of this array corresponds to the PE file from which the current PE file imports symbols. The array is delimited by an element that is zeroed out. These **_IMAGE_IMPORT_DESCRIPTOR** elements each contain pointers to both the Import Name Table (INT) and the Import Address Table (IAT). The *OriginalFirstThunk* member contains the RVA of the INT and the *FirstThunk* member contains the RVA of the IAT. There is also a *Name* member, which is the name of the DLL that contains the symbols being imported. The IAT and INT are both composed arrays of objects type **_IMAGE_THUNK_DATA** . Each **_IMAGE_THUNK_DATA** corresponds to an imported symbol. The in-memory manifestation of the IAT contains the actual addresses

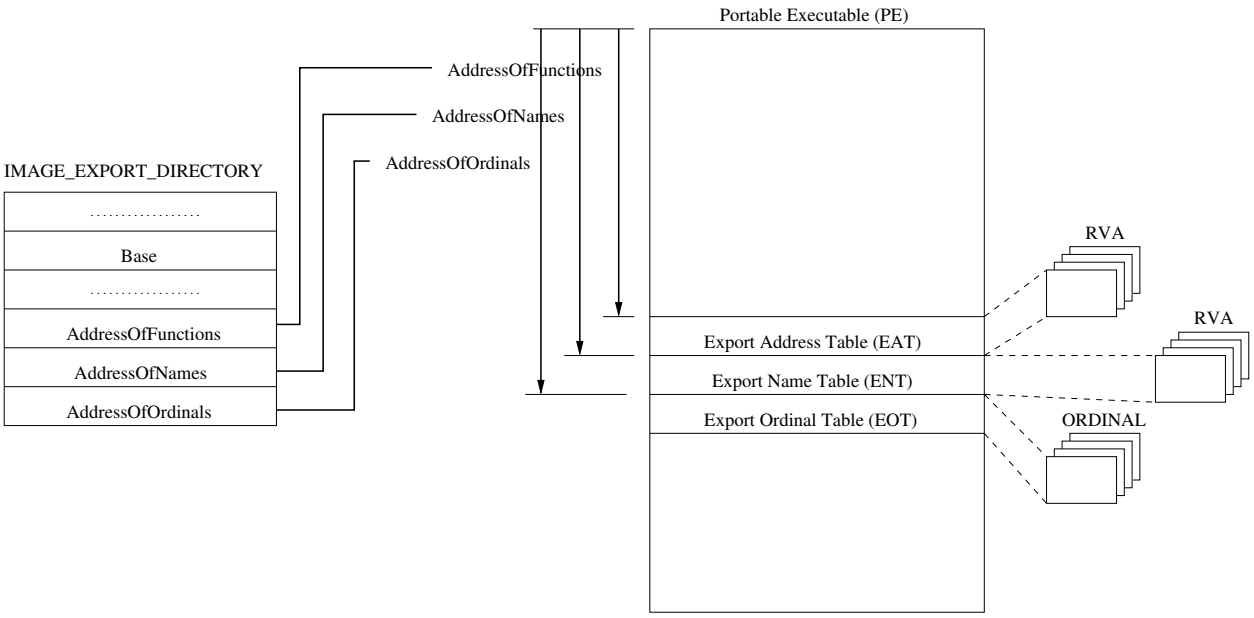


Figure 5: Exports Section

of the imported functions within the *Function* member of the **_IMAGE_THUNK_DATA** . The INT either contains the ordinal for the imported API in the EOT of the exporting DLL or an RVA to a **IMAGE_IMPORT_BY_NAME** structure. The **IMAGE_IMPORT_BY_NAME** structure contains two members. The *Hint* member corresponds to a suggested ordinal into the exporting DLLs EOT and the *Name* member contains the name of the symbol to be imported. Once we have parsed these data structures, we have the symbols imported by this DLL and where they are mapped in the virtual address space of the process.

As previously alluded to, another important aspect of the PE image is the Section Table. This table immediately follows the last **_IMAGE_DATA_DIRECTORY** . The section table is composed of an array of **IMAGE_SECTION_HEADER** objects which provides important meta-information about the sections and their locations in memory. We previously found the size of this table that was stored in the *NumberOfSections* member of **_IMAGE_FILE_HEADER** . The information stored in the **IMAGE_SECTION_HEADER** elements is very important for verifying the integrity of the information and rebuilding the sections in memory. For example, we can extract the `.text` section from this information and it gives us a recipe for rebuilding it in the RVAS. We can also verify the integrity of code sections.

Once this is finished, our accumulator modules have now given us all the memory objects (processes, threads, etc.) and DLLs associated with each process. It has also found all of the verified sections of `.text` that are mapped into the address space. Also included are symbols that are exported by the DLLs and the address of all the symbols that are imported.

# 5 Suspicious Artifacts

Using these objects and the relationships between these objects, we are able to perform a number of checks to elucidate suspicious artifacts. Many of the checks we will leverage include advanced techniques in semantic integrity evaluation [23], outlier detection for evidence identification [8], cross view diff-based approaches [2, 28, 9], and validating static data [22]. The following set of checks is not an exhaustive list, but is only a

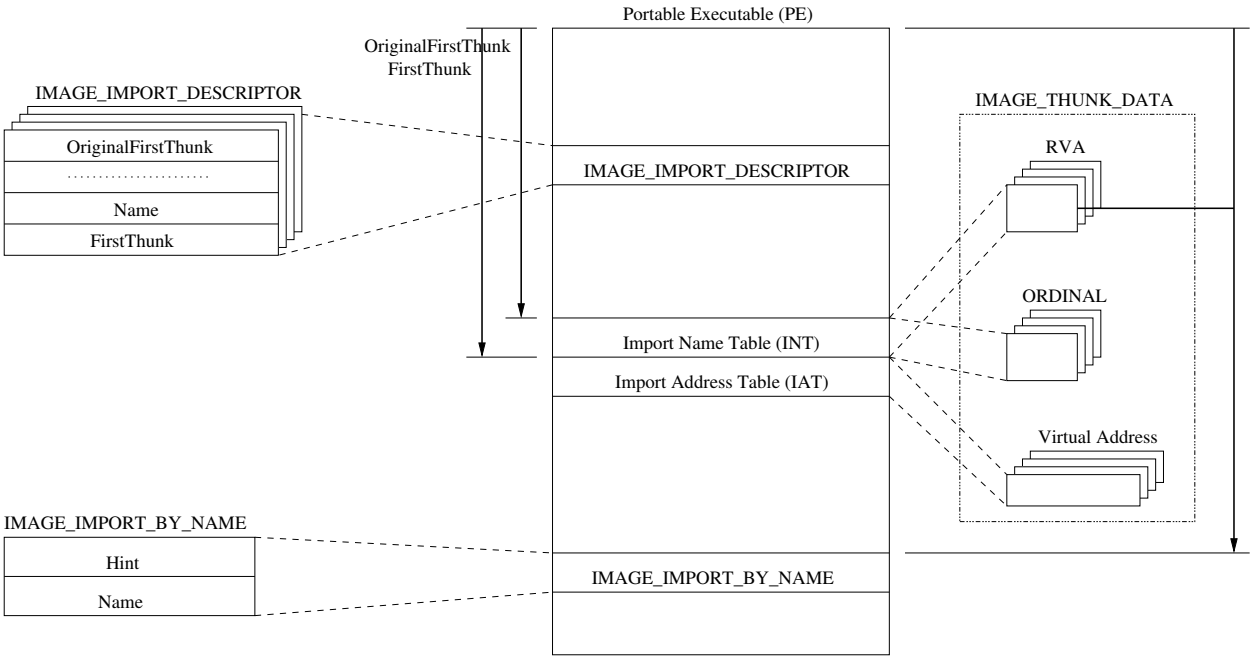


Figure 6: Imports Section

representative subset of the types of checks that are possible.

One set of checks we perform relates to correlating what is in memory with what is collected during the filesystem acquisition. Thus for each **_LDR_MODULE** we look to see if there is actually a DLL in the filesystem found at the path and name stored in the *FullDllName* field. In the case of our compromised system, there are two **_LDR_MODULE** objects (metsrv.dll, ext581787.dll) in the svchost.exe address space that cannot be found on disk at their specified locations in the C:\WINNT\system32\ directory. Thus these two **_LDR_MODULES** are considered suspicious artifacts that deserve further investigation. We also compare the list of modules loaded by each process in the compromised image with those typically loaded by the respective programs as found in the Windows 2000 profile, once again drawing attention to the aforementioned artifacts.

Another set of checks that is performed relates to correlating the in-memory image of the DLL's PE with the image stored on disk. The *BaseAddress* field of the **_LDR_MODULE** object is a pointer to the virtual address of the memory mapped PE image associated with this DLL. Previous research has demonstrated the ability to detect black listed modules by hashing the first 1024 bytes of this field [31]. Unfortunately, the first 1024 bytes only covers part of the memory-mapped PE header. Alternatively, using the *BaseAddress* we are able to parse the in-memory image of the DLL and reassemble the sections of the PE. This enables us to compare the hashes of static sections of the PE. For example, we can actually rebuild the text section of the file and compare the hash against the metsrv.dll used by metasploit.

| DLL | SHA-1 (.text) |
|---|---|
| metsrv.dll (framework-3.0-alpha-r3) | 0c3e67e1c02da875df13a084af0b4f29122e6d53 |

Since blacklisting has its obvious limitations, more importantly we hash the static sections of all the PE images and compare those with the values stored in the Windows 2000 profile and those generated from the filesystem information. As a result, we attempt to guarantee that every DLL loaded in memory must match the DLL found on disk. Metasploit's remote library injection technique actually undoes its changes so that

the code sections will not exhibit any changes unless the image is taken during the loading of a module. Special processing is also taken to handle self-modifying components [27].

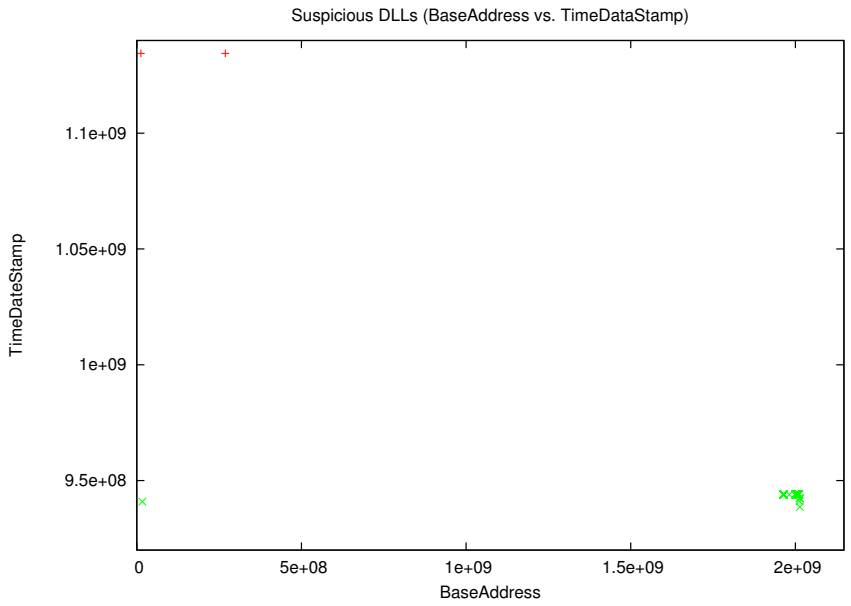


Figure 7: Anomalous Artifacts

We also look for anomalies among the set of DLLs loaded in the address space of each process. For example, we can check for set equality among the three doubly linked lists. We also correlate across the set of **_LDR_MODULE** structures for anomalous members. This includes looking for inconsistent values for the *BaseAddress* and *TimeDateStamp* across the set of **_LDR_MODULE** objects. For example, the graph in Figure 7 is a scatter-plot of the these values for the DLLs loaded into svchost.exe address space. The "X" data points in the lower right-hand corner represent the valid DLLs. The single "X" data point in the lower left corner represents the executable image. On the other hand, the "+" data points in the upper left corner represent the two anomalous DLLs loaded by Metasploit. While many of these features may not be intrinsic to the attack, they are used to emphasize that the attacker must make a concerted effort to avoid attracting suspicion.

# 6 Beyond the Basics

Our remote library injection detection module attempts to go beyond simply detecting the implementation artifacts of Metasploit. Thus we also take into consideration techniques that can be used to make remote library injection more stealthy. For example, let us assume that the attacker couples the remote library injection with an object manipulation technique similar to that presented in NTIllusion [13], which allows the attacker to hide the DLLs that have been injected into a process.

As seen in Figure 8, and in a manner similar to DKOM attacks, this is accomplished by manipulating the pointers to remove the **_LDR_MODULE** object from the doubly-linked lists. Once the **_LDR_MODULE** is removed, none of the previous detection methods will be useful since the aforementioned enumeration techniques will no longer find the object and thus have nothing to compare against.

In order to address these methods, the FATKit remote library injection model employs a number of techniques. First, the detection module performs a linear scan of the process' address space looking for

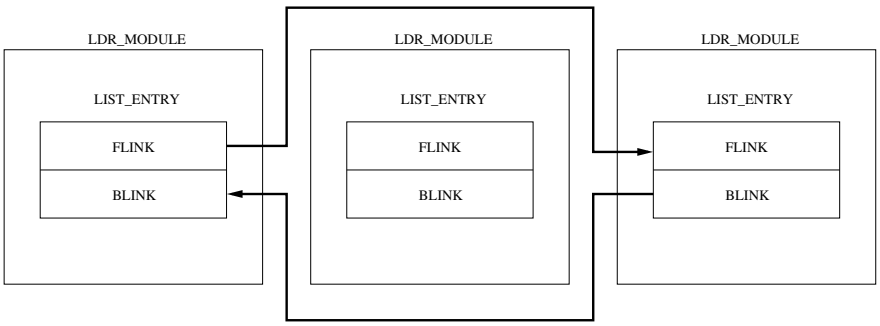


Figure 8: Direct User Object Manipulation

unlinked modules using a signature developed from memory informatics. Next, we assume the attacker has gone even a step further and zeroed out memory associated with the suspicious **_LDR_MODULE**, as suggested in NTIllusion [13]. In order to address this, the detection module leverages the PE module. As previously mentioned, we have already extracted locations of the `.text` sections associated with each PE and evaluated the integrity of those sections.

FATKit's ability to parse the PE headers associated with the image of the program and correlate import address tables with the address spaces of the DLLs that are represented by **_LDR_MODULE** objects in the embedded doubly-linked list make the injected library detectable. Thus, similar to [22, 27], we are able to verify the integrity of the trusted code sections in memory, locate any hooks [6] that do not point to these sections, and validate the semantic integrity of both the import and export tables [23].

One may now ask, "what if the attacker has undone any of these hooks," since they are now able to execute arbitrary code on the compromised machine. FATKit also has modules to detect this activity by tracing both the kernel and user-space stacks of the running threads and correlating the information stored on the stack with verified code sections. In this example, by tracing the stacks of the threads, it is possible to find the threads that are executing code found in the injected libraries. In the next white paper, we will discuss the details of finding this information and demonstrate how to map open ports to process object handles.

# 7 Discussion

The purpose of this white paper was to demonstrate how an analyst can leverage FATKit to elucidate artifacts of techniques that have been colloquially seen as "anti-forensic". The paper was not intended to present a complete detection algorithm, but rather to demonstrate the types of things that are possible and how FATKit facilitates this process. Unlike the other work being used to perform real time detection, we do not depend on the runtime integrity of the operating system. As a result, many of the subversion techniques used to defeat these tools do not apply. Furthermore, we are only concerned with detection as a means of finding suspicious artifacts. The goal of the FATKit project is to find as much evidence as possible from volatile memory once an information system is believed to have been compromised.

**Note:** As previously mentioned, some of the artifacts identified earlier in the whitepaper are not intrinsic to performing remote library injection but are merely suspicious artifacts of the implementation that would draw an analysts attention. For example, a Meterpreter stub could be written which maliciously modifies some of the memory resident objects (_LDR_MODULE, PE headers, etc) to scrub them of their suspicious data [16]. Thus, reducing the number or clustering of suspicious artifacts.

## 8 Acknowledgments

The author would like to thank those who contributed and reviewed this white paper. The FATkit project would also like to thank the Metasploit project for keeping things interesting and especially thank HD for his valuable comments. We would also like to thank MISSL, $DS^2$, Komoku, monkeys, and the BAD boys! Finally, we would like to thank Harlan Carvey [11], Andreas Schuster [1], and Mariusz Burdach [5] for doing interesting work and forcing us to write some of this stuff up.

## References

[1] Andreas Schuster. Computer Forensics Blog, June 2006. Available at: http://computer.forensikblog.de/en/.

[2] Doug Beck, Binh Vo, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 368–377, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Marshall Bedoe. The Protocol Informatics Project, June 2006. Available at: www.baselineresearch.net/PI/.

[4] bugcheck. GrepExec: Grepping Executive Objects from Pool Memory, June 2006. Available at: http://www.uninformed.org/?v=4&a=2&t=sumry.

[5] Mariusz Burdach. *Windows Memory Forensic Toolkit (WMFT)* , 2006. Available at: http://forensic.seccure.net/.

[6] James Butler. VICE- Catch the hookers! Info at: www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf.

[7] Brian D. Carrier and Joe Grand. A Hardware-Based Memory Aquisition Procedure for Digital Investigations. *Journal of Digital Investigations*, 1(1), 2004.

[8] Brian D. Carrier and Eugene H. Spafford. Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence. In *Proceedings of the 2005 Digital Forensic Research Workshop (DFRWS)*, 2005.

[9] Bryce Cogswell and Mark Russinovich. RootkitRevealer, February 2006. Available at: http://www.sysinternals.com/Utilities/RootkitRevealer.html.

[10] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *The 10th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, February 2003.

[11] Harlan Carvey. Windows Incident Response, June 2006. Available at: http://windowsir.blogspot.com.

[12] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, July 2005.

[13] Kdm. NTIllusion: A portable Win32 userland rootkit, July 2004. Available at: http://www.phrack.org/show.php?p=62&a=12.

[14] Vincent Liu. Metasploit Anti-Forensic Investigation Arsenal (MAFIA), July 2006. Available at: http://www.metasploit.com/projects/antiforensics/.

[15] Matt Miller and Jarkko Turkulainen. Remote Library Injection, June 2006. Available at: www.nologin.org/Downloads/Papers/remote-library-injection.pdf.

[16] H D Moore. Personal Communication.

[17] H D Moore and Skape. Metasploit Project, June 2006. Available at: http://www.metasploit.com.

[18] MSDN Library. Dynamic-link libraries, June 2006. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/dynamic_link_libraries.asp.

[19] Nick Petroni and AAron Walters. FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory, February 2006. Under Submisssion.

[20] Nick Petroni and AAron Walters. Forensic Analysis Toolkit (FATKit), June 2006. Available at: www.4tphi.net/fatkit.

[21] The Last Stage of Delirium Research Group. Microsoft Windows RPC DCOM Interface Overflow. Info at: http://www.osvdb.org/2100.

[22] Nick L Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[23] Nick L Petroni, Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *15th USENIX Security Symposium*, Vancouver, B.C., Canada, August 2006.

[24] Matt Pietrek. An In-Depth Look into the Win32 Portable Executable File Format, Feruary 2002. Available at: http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx.

[25] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals*. Microsoft Press, 2005.

[26] Joanna Rutkowska. modGREPER, June 2005. Available at: http://invisiblethings.org/tools/modGREPER/.

[27] Joanna Rutkowska. System Virginity Verifier, 2005. Available at: http://invisiblethings.org/tools/.

[28] Joanna Rutkowska. Thoughts about Cross-View based Rootkit Detection, June 2005. Available at: http://www.invisiblethings.org/papers/crossview_detection_thoughts.pdf.

[29] Sven B. Schreiber. *Undocumented Windows 2000 Secrets*. Addison Wesley Professional, 2001.

[30] Andreas Schuster. Personal Communication.

[31] Tobias Klein. Memory Parser (MMP), June 2006. Available at: www.trapkit.de/research/forensic/mmp.