http://www.nologin.org

Remote Library Injection

skape
mmiller@hick.org

Jarkko Turkulainen
jt@klake.org

Contents

1 Foreword **2**

2 Introduction **4**

3 Loading a Library **6**
3.1 Linux 6
3.2 Windows 11

4 Library Injection Methods **12**
4.1 On-Disk 13
4.1.1 Linux 13
4.1.2 Windows 15
4.2 In-Memory 16
4.2.1 Linux 17
4.2.2 Windows 21

5 Potential Impacts **29**
5.1 Worm/Rootkit Deployment Automation 29
5.2 Operating System Independence 30
5.3 Anti-Virus Nightmares 30

6 Prevention and Detection **31**
6.1 Linux 32
6.1.1 Inspecting Loaded Libraries 32
6.1.2 Detecting Function Hooks 33
6.2 Windows 34
6.2.1 Inspecting Loaded Libraries 34
6.2.2 Detecting Function Hooks 36

7 Conclusion **37**

Chapter 1

Foreword

Disclaimer: This document was written in the interest of spreading knowledge. Like humans held in bondage, information too wants to be free. As such, the ideas conveyed in this document have been transferred to text to aide information in its patriotic quest. Like the blueprints to constructing a bomb, the text in this document might too be seen as a blueprint to doing something that is seemingly bad. However, like the blueprints to a bomb, there is also a side that is good; namely, the education of the masses. Though one person might make a bomb in the interest of doing damage to others, another person might take the blueprints to gain an understanding of how to defuse a bomb should the need ever arise. It was in this spirit that this document was written. In short, the authors of this document cannot and will not be held liable for how this information is used or abused, much like the people who design bombs are not held responsible for the people they indirectly kill.

In light of the potential danger of reading code that implements the concepts discussed in this document, the authors have opted to not include complete examples of working code. Proof of concept code has, however, been developed for both Linux and Windows.

Please forward any comments, questions, corrections, love letters, or flames to one or both of the authors.

With that, on with the show...

The authors would like to thank:

nologin	For continued enthusiasm, motivation, and editing assistance.
H D Moore, spoonm, thief	For theorizing with the authors and offering always insightful perspectives.
"The Motivated"	Everyone who is internally motivated and driven to learn for their own satisfaction.
Family	For understanding and support (in general, not assembly language :-)

This document was last modified: **April 06, 2004**.

Chapter 2

Introduction

Abstract: The common methods currently employed to compromise computers are ineffective and easily detected by standard Anti-Virus practices. Despite this, worm authors continue to use these same approaches, blindly hoping that at least some of the hosts will remain infected long enough for the worm author to make use of them. An alternative to the standard methods of computer compromise involves making use of a more complicated, yet high-yield, solution: library injection. When used in conjunction with a remote vulnerability, such as the DCOM[1] vulnerability, library injection can lead to an undetectable compromise at the host level as far as current Anti-Virus detection mechanisms are concerned. The impact from this is far-reaching; so much so that a completely automated, high-retention, operating system independent super-worm is an ever approaching reality.

`Library Injection` is the process by which a dynamically linked library is injected, or forcibly loaded, into a process' address space. Once loaded, the library exists like any other standard library in that its initialization routines are called and its exported symbols can be resolved through the platform's symbol resolution interfaces. In addition, the loading process resolves all of the library's dependencies, much like the process taken when an application is launched. This provides the library will all the tools commonly exposed to an executable. In short, an injected library has the same amount of flexibility associated with an executable and is capable of running in the context of an existing process.

Unlike executing an application, some methods of library injection are not externally noticeable without non-standard tools, such as `Process Explorer`[14] for Windows. The reason it is not easily noticeable is directly tied to the platform on which the library is injected, but suffice to say that the authors are

aware of no currently employed methodologies by which this can be detected[1]. This topic will be discussed in more detail in the `Prevention and Detection` chapter (6).

The basic process used to perform library injection is directly dependent on the context from which the library is injected from. This means that the methods used to inject a library from the local machine versus injection from a remote connection, such as an exploit, are done by different means. The focus of this paper will be on the injection of libraries over remote connections as it emphasizes the danger of being exposed to a remote exploit that could in turn be exploited by something that makes use of the topics discussed in this document.

At a high level, the approach used to inject a library through a remote exploit is relatively straight forward. An exploit author would employ what is referred to as `Multi-Stage Shellcode`, or multi-stage payloads, to allow himself the added flexibility of being able to execute arbitrarily large payloads[13]. The first stage would make use of a second topic, known as `File Descriptor Re-use`, whereby the exploit attempts to locate the file descriptor from which the exploit originated. Upon successfully locating the file descriptor, the first stage payload would then read in the second, arbitrarily sized, payload and execute it[13]. It is in this second stage that an exploit author would send the payload for downloading and injecting the library into the process that the exploit has targeted. After the library has been loaded, all bets are off. The potential impacts of a library being injected are discussed in depth in the `Potential Impacts` chapter (5).

Without yet understanding the *how* associated with library injection, it is pertinent to consider potential prevention and detection mechanisms. These would allow a person to defend or acknowledge a compromise that incorporates library injection. These two points will be discussed in the `Prevention and Detection` chapter (6).

Upon completion of this document the authors hope that the reader will have a complete understanding regarding the concept of `Library Injection`, thus enabling the reader to make educated and intelligent decisions as it pertains to the topic at hand. The following chapters will vary in levels of technical detail, but one should not be surprised to see code snippets and other very low-level details.

[1] This does not mean that all library injection methods cannot be detected; rather, it means that current implementations do not have the ability to do so. On-Disk library injection, as discussed later, can and will be detected by Anti-Virus scanners. However, In-Memory library injection will not be.

Chapter 3

Loading a Library

Before understanding how library injection works, one must understand how a library is loaded in the first place. The interfaces used to do this vary from platform to platform and as such will be analyzed separately for the two platforms of focus in this document: Linux and Windows.

3.1 Linux

The standard approach to loading a library in Linux, at least for most distributions, involves making use of the library `libdl.so` which exports a small number of functions for interfacing with dynamically loaded libraries. These functions are actually wrappers for functions that are exported in `libc.so`. The three core functions that `libdl.so` provides are:

1. `void *dlopen (const char *filename, int flag);`

   This function opens a library, specified by `filename` and takes one or more flags that control things such as imported symbol resolution. Upon success, `dlopen` will return an opaque pointer to the context associated with the library. In the case of `glibc`, this opaque pointer is actually a `struct link_map` pointer as found in `link.h`. Upon failure, the return will be `NULL`.

   One thing to note is that `dlopen` will indirectly cause the calling of the `_init` symbol, or more correctly the symbol marked as a constructor, in the library that is loaded. The constructor symbol can be seen as being analogous to `main` in an application. Unlike main, however, the constructor symbol must not block.

2. `void *dlsym(void *handle, char *symbol);`

This function takes the opaque pointer retuned from `dlopen` as the `handle` argument and the name of a symbol (e.g. `gethostbyname`) as the `symbol` argument. On success, a pointer to the absolute VMA of the symbol is returend. If the symbol does not exist in the library passed in, the return value will be `NULL`.

3. `int dlclose (void *handle);`

   This function will unload a previously loaded library by passing the opaque pointer that was returned from `dlopen` as the `handle` argument. Upon success, zero will be returned. Otherwise, non-zero is returned.

   Like `dlopen`, `dlclose` has the property of indirectly calling the `_fini` symbol, or more correctly the symbol marked as a destructor, in the library that was loaded. This can be seen as analogous functionality to registering a handler with `atexit`, but instead of running at process exit, the destructor runs when the library is unloaded.

Though these three functions provide the basic functionality needed to interface with dynamically loaded libraries, they are not linked to or used by all applications. Many applications have no need to interface with the dynamic loader outside of initially resolving and loading dependent libraries which is taken care of behind the scenes during the initialization portion of execution. As such, one cannot assume that `libdl.so` will be loaded in the context of a given process. This fact will become important later during the chapter on `Library Injection Methods` (4).

Fortunately, it is possible to interface with the dynamic loader without having to worry about whether or not `libdl.so` is loaded. Instead, one can use a set of similar functions that are exported from `libc.so`. Those functions and their descriptions are as follows:

1. `void *_dl_open (const char *file, int mode, const void *caller);`

   This function supplies the exact same functionality that `dlopen` from `libdl.so` provides. However, its calling convention differs such that instead of using `cdecl` like `libdl.so` does, _dl_open uses `fastcall` whereby arguments are passed in registers instead of on the stack. For `IA-32`, arguments are passed in the following registers:

   `eax` = `file`
   `edx` = `mode`
   `ecx` = `caller`

   The return value is exactly the same as `dlopen`.

2. `void *_dl_sym (void *handle, const char *name, void *who);`

   This function supplies the exact same functionality that `dlsym` from `libdl.so` provides. Like `_dl_open`, _dl_sym uses `fastcall` linkage. For `IA-32`, arguments are passed in the following registers:

`eax` = `handle`
`edx` = `name`
`ecx` = `who`

The return value is exactly the same as `dlsym`.

3. `void _dl_close (void *_map);`

   This function supplies the exact same functionality that `dlclose` from `libdl.so` provides. This function also uses `fastcall`. The `_map` argument is passed in the `eax` register on `IA-32`.

Though it is not a requirement that an application link to `libc.so`, nearly every dynamically linked application does. Given this fact and combining it with the knowledge that `libdl.so` is not always loaded, `libc.so` can be seen as the least common denominator for interacting with the dynamic loader, at least through a set of exposed functions. As such, this concept needs to be understood before any attempts are made to write reliable code for loading a library in the context of an arbitrary process. In reality, the only function that is necessary for loading a library is `_dl_open`.

With the function necessary to load a library identified, it would seem like everything is complete. However, this is not the case. One consideration that needs to be made is the fact that the VMA for `_dl_open` may vary from installation to installation depending on a number of things, not the least of which is the fact that the machine may make use of `Address Space Layout Randomization`, or ASLR[9]. When ASLR is in use, library base addresses will be randomized such that they do not load at the same address during every execution of a given application. This means that hard-coding a static VMA for `_dl_open` is out of the question as it will never work properly. Given this hurdle a non-trivial process can be taken to locate the VMA of `_dl_open` that works regardless of whether or not ASLR is in use. That process is outlined as follows:

1. Walk process address space searching for ELF signatures

   The first step in the long trek involves walking process address space in `PAGE_SIZE` increments and comparing the first four bytes of a given page with `ELFMAG`, the four byte ELF signature. The problem here is that since valid address ranges are unknown in the given context the code must be written such that it can test for invalid addresses and not crash the application. The solution to this problem comes in the form of system call abuse.

   System call abuse is a method by which one uses a system call in a way that it was not intended to be used. In this case, a system call can be used to validate a memory address due to the inherent fact that some system calls will return the error `EFAULT` if an invalid pointer is passed in. One such system call is the `access` system call. The access system call is prototyped as:

```
int access(const char *pathname, int mode);
```

The `pathname` argument, which is passed as `ebx` on `IA-32`, can be used as the point of address validation. By passing in an invalid address, the `access` system call will return `EFAULT` in the case that the pointer happens to be unreadable. If it is not unreadable, another error code will be returned and that is a clear indication that the address is valid. If an unreadable address is encountered or a readable address that does not match the ELF signature, the current address being tried should be incremented by `PAGE_SIZE` and the loop should repeat itself. If a readable address is found that matches the ELF signature, the next step is taken. This address will henceforth be referred to as the `absolute base address`.

2. Check ELF image type

   Upon finding a readable page that matches the ELF signature, the next step in the process is to verify that the ELF image at the absolute base address is a library, not an executable. This is done by checking to see if the `e_type` field of the ELF header is set to `ET_DYN`. If it is not, the current base address is incremented by `PAGE_SIZE` and the the loop goes back to step 1. If the ELF image at the absolute base address is a library, the next step is taken.

3. Enumerate the Program Header Table

   After determining that the image at the address is not only an ELF binary, but also an ELF library, it becomes pertinent to locate the dynamic linkage information that the library provides to the dynamic loader in order to facilitate the resolving of dynamic symbols and the names of said symbols. The way that the library does this is by having a mappable segment for the dynamic section of the binary[1]. This mapping information is stored in the `Program Header Table` and each mappable segment has a type that is used to instruct the interpreter of the library as to how to interpret the contents of the mapped segment. In the case of the dynamic section, the program header type, or the `p_type` field, is `PT_DYNAMIC`.

   In order to locate the `PT_DYNAMIC` entry, the `Program Header Table` must be enumerated. The base address of the table is calculated by adding the absolute base address with the `e_phoff` attribute of the ELF header. The number of entries in the table is stored in the ELF header in the `e_phnum` attribute. Enumeration is then done with standard pointer path as would be expected during the enumeration of an array of a given data structure. During the enumeration process, each entry's `p_type` attribute is compared to `PT_DYNAMIC`. If a match is not located, the loop increments to the next index and continues. Otherwise, if a match is found, the dynamic section mapping information has been located. The field of interest for the dynamic entry is the `p_offset` field which holds that file offset to the dynamic section entry's content. This offset is where the actual dynamic

[1] The dynamic section is sometimes referred to as '.dynamic'.

linkage information is stored. To convert it to an absolute address, all that is necessary is to add the absolute base address with the offset specified in `p_offset`.

If no `PT_DYNAMIC` entry is located, the absolute base address is incremented by `PAGE_SIZE` and the loop starts over at step 1.

4. Enumerate the dynamic section

   Once the dynamic section entry's content has been located, dynamic linkage information can then be extracted such that it can later be used to resolve the symbol `_dl_open`. There are two values that need to be extracted from the section. The first of these values is the offset to the dynamic symbol table. This table holds an array of `ElfXx_Sym` structures that make up the group of exported symbols that the library allows external pieces of code to interact with. This value is identified by the `DT_SYMTAB` identifier. The second of the two values is the string table associated with the dynamic symbol table. This is needed due to the fact that the name of the symbol must be compared with `_dl_open` in order to determine if it is the right symbol or not. This value is identified by the `DT_STRTAB` identifier.

   The dynamic section entry's content is composed of an array of `ElfXx_Dyn` structures. Each array entry correlates an identifier with a given value. In the case of `DT_SYMTAB` and `DT_STRTAB`, this value is an offset from the start of the file to their respective contents. If neither of the two identifiers can be located or only one of the two can be located, the absolute base address is incremented by `PAGE_SIZE` and the loop starts over at step 1. Once both identifiers are located, the absolute base address should be added to both of them in order to convert them into absolute addresses.

5. Enumerate the dynamic symbol table

   The last step of the process involves enumerating the each of the dynamic symbol table entries and comparing each entry's symbol name with `_dl_open`. This is done by adding the symbol's `st_name` attribute to the address of the dynamic symbol string table. The result should give back a null terminated string for the name of the symbol. If the symbol names do not match, the loop is repeated. Otherwise, the `_dl_open` symbol has been found and the absolute address can be calculated by adding the `st_value` attribute to the absolute base address of the library. The result is a direct virtual address that can be used to call the function.

   One consideration that must be kept in mind during this phase is that it is possible that a the symbol will not be located in a library. For instance, if `ld-linux.so` is loaded before `libc.so`, which it nearly always is, the symbol resolution code will encounter `ld-linux.so` symbols before `libc.so` symbols. As such, the symbol resolution code must be robust enough to handle the scenario where it does not find any matching symbols. One approach to doing this involves doing a check at the beginning of the symbol

enumeration loop to see if the current symbol's address has gone past or is equal to the dynamic symbol string table's address. In the event that this is true, the absolute base address is incremented by `PAGE_SIZE` and the loop starts back at step 1. If the current symbol address is not greater than or equal to the dynamic symbol string table's address, it is safe to assume that it is still still within the symbol table[2].

With the absolute memory address of `_dl_open` located, all that's left to do is load the library itself by calling the function.

3.2 Windows

Loading a library on Windows, at least in the context of a payload, is nowhere near as complex as Linux. This is due to the fact that Windows exposes a position independent method for enumerating the loaded library list. This is important because it allows one the ability to walk the symbol tables in all of the libraries or in a specific library with relative ease. The specific implementation for doing this is described in *Understanding Windows Shellcode*[13] and in *Windows Assembly Components*[2].

[2]This assumes that the dynamic symbol string table occurs after the dynamic symbol table in memory. The current linker implementation on all the versions of Linux known to the authors links libraries in this fashion. If this were not the case, an alternative solution would be necessary.

Chapter 4

Library Injection Methods

This document will discuss two methods by which a library can be injected remotely. The two methods only differ in approach, but have the same desired goal as outlined in the introduction. The first of these methods is known as `On-Disk Library Injection` which, as the name implies, means that the library is written to disk and then loaded into the process' address space. The second of these methods is known as `In-Memory Library Injection` which entails loading the library entirely from memory without any disk activity at all.

The On-Disk method is the easiest of the two methods but also has the highest risk of detection. At the time of this writing, Anti-Virus software is capable of performing On-Access virus scanning which means that the virus scanner will perform virus checks when a file is accessed, such as when editing or execution occurs[11]. This means that when the payload used during the conceptual exploit writes the library to disk, those writes will undergo analysis by the virus scanner and potentially be detected. Not only that, but the library will also potentially undergo scanning upon opening and reading of the library during the loading phase. If the library is detected as a virus, the show stops there. As such, the On-Disk method should be seen as an inferior method as it suffers from the same problems that plague the downloading and subsequent execution of an actual executable.

The second method, In-Memory injection, is far less detectable. In fact, as stated in the introduction, the authors' are aware of no virus scanners that, as they stand at the time of this writing, are capable of detecting this method. Though virus scanners may detect earlier phases, such as the exploit transmission over the network, they cannot currently detect the actual library loading which is the focus of this document.

The following sections will discuss the two implementations in detail and explain

how one might approach implementing them across multiple platforms.

4.1 On-Disk

As noted above, the On-Disk method is the process by which a library is written to disk and subsequently loaded into a process' address space. The logical steps taken to do this are exactly the same between Linux and Windows, but varies greatly in implementation due to some hurdles that must be jumped due to the fact that the injection payload, or the things that actually perform the library injection, is running as shellcode. This means that standard library functions and most luxuries afforded to programmers of a given platform cannot be used, at least not directly. Regardless of how detectable the On-Disk approach is, it is nonetheless a viable method of injecting a library.

On both UNIX and Windows, the general approach to implementing the On-Disk method involves writing a payload that reads in the library from a file descriptor and then writes it somewhere on disk. Once the library is completely downloaded, the payload would then call the respective library loading methods for a given platform.

The following subsections will outline the implementation of the On-Disk method for both Linux and Windows. The concepts applied to Linux are common to most UNIX variants.

4.1.1 Linux

The Linux implementation of the On-Disk library injection method on Linux involves the steps listed below. The steps are written as if they would be implemented in the context of the payload sent in the second stage of an exploit as described in the introduction. The payload begins after the first stage has jumped into it. The file descriptor that the second stage was read in from, henceforth referred to as the "socket", is available to the second stage for continued re-use.

1. Open a file on disk to hold the library

   Open an arbitrary pathname, such as `/tmp/a.so`, for writing and preserve the file descriptor for later use. This is where the library will be downloaded to. To do this, the `open` system call is used.

2. Read the library's length from the socket

   Read the four byte length of the library and store it for later use. This step can be optimized by passing the `MSG_WAITALL` flag to the socket `recv`

system call and passing 4 bytes for the `len` argument. This step makes it possible to know exactly how many bytes should be read from the socket.

3. Read a chunk of the file from the socket

   Read an arbitrarily sized chunk no larger than the amount of data left to read into an intermediate buffer, such as a stack allocated buffer. Save the number of bytes read as returned from the socket `recv` system call for use in the next two steps.

4. Write the chunk to the opened file on disk

   Write the contents of the buffer that was read in from the socket to the file descriptor that was opened for the library. This write operation, using the `write` system call, should use the number of bytes actually read from the socket as the `len` parameter.

5. Subtract the number of bytes read from the library's length

   Subtract the number of bytes that were read from the socket from the length of the library that was read in during step 2. This is used to track how many bytes are actually left to be read from the socket.

6. If the length is non-zero, repeat steps 3-6

   If the amount of length of the library left to be read is not yet zero, that indicates that there is more data to be read. As such, steps 3-6 should be repeated until the length does drop to zero.

7. Close the library's file descriptor

   After the entire library has been read in, the file descriptor for the library should be closed.

8. Find the VMA for _dl_open

   Before loading the library, the `_dl_open` function must be resolved. The process to do this is discussed in the section on `Loading a Library` for Linux (3.1).

9. Call _dl_open with the path to the library

   After the address of `_dl_open` has been determined, all that remains is to actually load the library from the disk. The `path` argument should be set to the name of the file that was passed in to step 1. The `mode` parameter should be set to `0x80000000` OR'd with the binding mode desired, such as `RTLD_NOW` or `RTLD_LAZY`.

   Once `_dl_open` returns the library will either have been loaded successfully or will have failed to load. Validation as to what happened can be done by analyzing the return value.

4.1.2 Windows

The Windows implementation of On-Disk library injection is conceptually the same as the Linux approach. The steps outlined below are written in the context of a second stage payload that re-uses the file descriptor, henceforth referred to as the socket, from the first stage loader[13]. The basic process involves reading a library in from the socket, writing it to disk, and finally loading it via the dynamic library loading interface exposed by the operating system. The approach is compatible with both Windows 9X and NT-based versions of Windows.

1. Load required libraries and resolve required symbols

   In order to be able to read from the socket, write to disk, and eventually load the library, the required libraries and symbols must be loaded and resolved. This step is common to most Windows payloads due to the fact that using system calls directly is discouraged and unreliable[1]. The libraries that the On-Disk technique is directly dependent on are `KERNEL32.DLL` and `WS2_32.DLL`. The first DLL exposes standard file operation functions as well as providing the dynamic library loading interface. The second DLL exposes the `Windows Socket API` for use when reading from the socket. The functions depended upon and the library that they are exported from are listed below:

Library	**Required Function**

`KERNEL32.DLL`	`LoadLibraryA`
	`CreateFileA`
	`WriteFile`

`WS2_32.DLL`	`recv`

2. Create the library on disk

   Before downloading the library, a file must first be created on disk to store it. This is accomplished by using the `CreateFileA` function which, on success, returns a handle to the opened file. The filename associated with the library is arbitrary, but for the purposes of this description will be referred to as `inject.dll`.

3. Read the length of the library from the socket

   A four byte length should be read in from the socket before downloading the library. This makes it possible to know exactly how many bytes should be downloaded. This is done by calling `recv` on the socket that the first stage loader passed in. An important thing to note is that it may be possible for `recv` to return less than four bytes. Windows implementations of `recv` do not support the `MSG_WAITALL` flag and as such do not allow

[1] *Understanding Windows Shellcode*[13] has more details on this topic.

the caller a mechanism by which they can read an exact number of bytes from the socket.

4. Read the library from the socket and write it to disk

   At this point, everything is accounted for and the library can be read in from the socket and written to the file. Due to the fact that Windows does not support the `MSG_WAITALL` flag, an alternative approach can be used based on the code displayed below. This code snippet will read the library in as chunks and write those chunks to the file until the total number of bytes left to read is zero:

   ```
   char buffer[1024];
   int bytes = 0, bytesLeft = libraryLength,
       written;

   for (;bytesLeft > 0; bytesLeft -= bytes) {

       if ((bytes = recv(sock, buffer, bytesLeft, 0)) < 0) {
           break;
       }

       WriteFile(injectFile, buffer, bytes, &written);
   }
   ```

5. Load the library

   With the library completely downloaded, all that is left to do is load it. This is accomplished by calling the `LoadLibraryA` function with the `lpFileName` argument set to filename that was passed to `CreateFileA` in step 2, which in this case is `inject.dll`.

That's all there is to On-Disk library injection in Windows. An alternative approach to On-Disk library injection that follows a similar train of thought is to call `LoadLibraryA`, or the function responsible for loading a library, with a UNC path[5]. This effectively loads a library over an SMB connection. The biggest problem with this approach is that it requires outbound SMB ports to be open, passable, and not blocked on the target machine. Regardless, both methods accomplish the desired goal of loading a library from disk, whether it's gotten to via the local filesystem or by a SMB share.

4.2 In-Memory

Of the two methods used to perform library injection, the In-Memory method is by far the most advanced and dangerous. The benefits of In-Memory library

injection include the ability to avoid detection from On-Access virus scanners due to the fact that the library itself never actually touches the disk. The process used to achieve this varies from platform to platform, but the general approach is to hook the underlying file operations that the dynamic loader uses to load a library. Hooking is the process by which a call to a given function is routed through an intermediate step, such as a custom function that can emulate or perform a different operation than was originally intended for the original function[3]. This concept plays a critical role in achieving the goal of In-Memory library injection.

The following sections will detail how In-Memory library injection can be implemented across Linux and Windows.

4.2.1 Linux

1. Read the length of the library from the socket

   Read the four byte length of the library and store it for later use. This step can be optimized by passing the `MSG_WAITALL` flag to the socket `recv` system call and passing 4 bytes for the `len` argument. This step makes it possible to know exactly how many bytes should be read from the socket.

2. Anonymously map memory that is at least the length of the library

   In order to be able to store a dynamically sized library, one has two options. Either the stack can be used, which is quite limited as to the amount of space it has available for storage, or an anonymously mapped memory range can be used. The heap is also an option, but it involves a more tedious process than would otherwise be necessary. The `mmap` system call exposes an interface that allows for associating a memory range with a file descriptor. It also allows for mapping an arbitrary memory range that is not tied to a file descriptor; this is referred to as an *Anonymous Map*. It is the latter of the two capabilities that are of use for In-Memory library injection. The `mmap` function is prototyped as follows:

   ```
   void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
   ```

   In order to map anonymous memory, the arguments should be set to the following:

Argument Name	Argument Value
`start`	`NULL`
`length`	The length of the library read in from the socket
`prot`	`PROT_READ | PROT_WRITE | PROT_EXEC`
`flags`	`MAP_PRIVATE | MAP_ANON`
`fd`	`-1`
`offset`	`0`

The return value will be the VMA of the mapped memory, or `-1` on failure. This address should be saved for later use.

3. Read the entire library into the mapped memory

   With the memory allocated to store the library in, the next step entails reading the actual contents into it. This is done by calling `recv` with the `buf` argument set to the VMA that was returned from `mmap`. The `len` argument should be set to the length of the library. Finally, the `flags` argument should be set to `MSG_WAITALL` in order to read the whole library in one swoop. On success, `recv` will return the number of bytes read which should be equal to the length of the library.

4. Hook file operation functions

   This step is the most complex and requires some understanding of how the dynamic loader operates. During normal operation, the dynamic loader (`ld-linux.so`) makes use of a subset of the file operation functions in order to open, read, and map the library into the process' address space. In order to load a dynamic library that does not exist on disk, a program, or payload in this case, must layer itself in-between the dynamic loader and said file operation functions. This is done through hooking, as mentioned earlier in the chapter. The actual file operation functions that dynamic loader uses, at least at the time of this writing, are as follows[2]:

[2]This list only includes the file operations that are required to be hooked in order to successfully load a library that exists only in memory.

File Operation	Usage
`open`	Used to open a library for subsequent file operations.
`read`	Used during the initial loading phase to validate the library as being an ELF image and, potentially, to read in the program header table.
`lseek`	In scenarios where the program header table exceeds the number of bytes initially read in during the validation phase with `read`, the dynamic loader will use this operation to seek to the start of the program header table
`mmap`	Used to associate a memory range with the contents of the library.
`fxstat64`	Used to get information about the file, such as its size and mode.

Hooking the aforementioned functions allows one to emulate their described purpose and thus make the dynamic loader think that the operations are actually being performed on a file on disk when in fact they are merely emulating the operations against the memory range that was anonymously mapped to hold the library. The actual implementations of the hook functions are broken down by the function that is being hooked:

open

The "open" hook involves checking to see if the pathname that was passed into the function matches the "fake" library name that the hook expects to see. If it does match, a virtual file descriptor should be returned that does not conflict with any existing file descriptors and can be used by subsequent file operations to identify it as being special. The virtual file descriptor should store information such as the current virtual file offset, the size of the library in memory, and the base address at which the library was loaded. This information is then used by subsequent file operation functions when reading, seeking, and for the other file operations as well. If the pathname passed in does match the fake library name, the call should simply be passed to the real open function.

read

The "read" file hook should check to see if the file descriptor passed in as the `fd` argument is a virtual file descriptor or a real file descriptor. If it's a virtual file descriptor a logical read operation should be emulated against the memory range. This means that up to `count` number of bytes should be copied from the mapped memory range to the buffer passed in as `buf`. If the current file offset is equal to the length of the library, zero bytes should be copied. After a successful read operation the current file

offset should be updated by adding the number of bytes actually copied to the original offset. If the file descriptor passed into read is not a virtual file descriptor, the call should simply be passed to the real read function.

lseek

The "lseek" file hook, if needed, should emulate file seeking operations against the mapped memory range, but only if the file descriptor passed in as the `fd` argument is a virtual file descriptor. There are three types of seeking operations: `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`. The first of the three, `SEEK_SET`, is a way by which a caller can set the file descriptor's offset. In the case of a virtual file descriptor, this would involve setting the current file offset to the argument passed in by `offset`. The second of the three, `SEEK_CUR`, is a way by which a caller can update the file offset relative to its current position. In the case of a virtual file descriptor, this would be emulated by adding the offset passed in as `offset` to the current file offset. In theory, sanity checks are not necessary in this context as it is unlikely that the dynamic loader will pass invalid offsets. Finally, the third seek operation, `SEEK_END`, is used when the caller wants to update the file descriptor's offset relative to the end of the file. In the case of a virtual file descriptor, this is emulated by adding the offset passed in as `offset` to the size of the library itself and storing sum as the current file offset. If the file descriptor passed into lseek is not a virtual file descriptor, the call should simply be passed to the real lseek function.

mmap

The "mmap" hook is arguably the easiest of the set. When a virtual file descriptor is passed in, the mmap hook should simply call the real mmap function and map an anonymous memory range based on the arguments passed in. Once the range has been mapped successfully, the contents of the library at the offset specified as the `offset` argument for `length` bytes should be copied into the newly mapped memory range. If the file descriptor passed in is not a virtual file descriptor, the call should simply be passed to the real mmap function.

fxstat64

The "fxstat64" hook is responsible for giving the caller information about the file descriptor passed in, such as its size, atime, ctime, among other things. In the case of emulating this sort of operation on a virtual file descriptor, all that is really necessary is to attempt to provide the caller with as much accurate information as possible. For instance, the `st_size` attribute of the `struct stat64` argument passed into the function should be set to the size of the library. The `st_uid` and `st_gid` attributes should be set to the uid and gid of the current process, respectively. The `st_mode` needs to be at least initialized to zero in order to avoid having it be indicated as something other than a normal file. If the file descriptor passed in is not a virtual file descriptor, the call should simply be passed to the real fxstat64 function.

5. Find the VMA for _dl_open

   The process to do this is discussed in the section on `Loading a Library` for Linux (3.1).

6. Call _dl_open with a "fake" library name

   Once `_dl_open` has been successfully located, the next step is to call it with the `path` argument set to a unique library name that the hook functions will know to expect as symbolizing the library that exists in memory. This will then indirectly call the hook functions described in the previous step and eventually lead to the loading of the library, even though it does not reside on disk.

4.2.2 Windows

The In-Memory library injection implementation on Windows can subjectively be seen as the most impacting of all the platforms due to the relative number of exposed machines that run Windows. The implementation is conceptually similar to the approach taken on Linux in that function hooking is used to emulate file operations against a memory range instead of referencing a file on disk. The following steps describe the approach for NT-based versions of Windows and is portrayed in the context of a second stage payload:

1. Load required libraries and resolve required symbols

   In order to be able to download the library from the socket, map it into memory, and finally coerce the dynamic loader into loading it requires that a certain subset of the Windows API be used. The functions can be used to complete the end-goal are listed below along with their respective libraries[3]:

[3]Some of the functions listed above were used in the specific implementation of the proof of concept and may not be required depending on the approach taken. These functions include VirtualQuery, VirtualProtect, FlushInstructionCache, and RtlUnicodeStringToAnsiString.

Library	Required Function
KERNEL32.DLL	LoadLibraryA VirtualAlloc VirtualQuery VirtualProtect FlushInstructionCache WriteProcessMemory
NTDLL.DLL	NtOpenSection NtCreateSection NtMapViewOfSection NtQueryAttributesFile NtOpenFile RtlUnicodeStringToAnsiString
WS2_32.DLL	recv

2. Read the length of the library from the socket

   A four byte length should be read in from the socket before downloading the library can proceed. This length is used to allocate a buffer that will be used to hold the contents of the library and also to manage the downloading phase in such a way that only the exact number of bytes necessary are read from the socket. As noted in the On-Disk implementation of library injection, Windows' implementation of `recv` does not support `MSG_WAITALL`. As such, the possibility exists that fewer than four bytes could be read from one call to `recv`. Once the entire length has been read in it should be saved in some manner for subsequent steps.

3. Allocate memory to store the library in

   In order to load the library it must first be stored locally in some manner. Given that the library cannot be stored on disk, the only other viable option is to store it in memory. The method used to allocate memory is to call the `VirtualAlloc` function which is conceptually synonymous with `malloc`, though it provides more flexibility. It is also easier to obtain due to the fact that it exists in `KERNEL32.DLL`. The prototype for `VirtualAlloc`[4] is:

   ```
   LPVOID VirtualAlloc(
     LPVOID lpAddress,
     SIZE_T dwSize,
     DWORD flAllocationType,
     DWORD flProtect
   );
   ```

   To allocate memory for storing the library, the `lpAddress` argument should be set to `NULL`, the `dwSize` argument should be set to the size of the library, the `flAllocationType` argument should be set to `MEM_COMMIT`, and finally

the `flProtect` argument should be set to `PAGE_READWRITE`. On success, `VirtualAlloc` should return a pointer to the allocated buffer. Otherwise, `NULL` is returned. The pointer that is returned should be saved in some context for subsequent steps.

One item worth noting is that, by default, the pages in the loaded library may swap out to disk. If an Anti-Virus scanner were to support swap scanning it might be possible for it to detect the library. In order to avoid this, one can make use of the `VirtualLock` function to pin the allocated address range for the library in memory.

4. Read the library from the socket and write it to memory

   Once the buffer has been allocated to store the library, the next step is to actually download it from the socket that was passed in from the first stage loader. One method to doing this is outlined in the following example code:

```
int bytesLeft = libraryLength, bytesRead = 0, bytes = 0;

for (; bytesLeft > 0; bytesLeft -= bytes, bytesRead += bytes) {

    if ((bytes = recv(sock, libraryMemoryRegion + bytesRead,
            bytesLeft, 0)) < 0) {
        break;
    }
}
```

5. Manually map the image's sections for later use

   After downloading the library and storing it in memory, the payload must then set up and the logical sections of the binary must be initialized such that they can be returned from future calls of `MapViewOfSection`[7]. This can be done in two steps.

   The first step is to allocate a memory range, with `VirtualAlloc`, that uses the size of the image as specified in the `IMAGE_NT_HEADER` portion of the PE by the `OptionalHeader.SizeOfImage` attribute. The flags used for this allocation, as passed by the `flProtect` argument, should be set to `PAGE_EXECUTE_READWRITE` due to the fact that execution will be done on the memory range at some point. This buffer will be referred to as `targetLibraryBuffer` henceforth.

   After allocating the memory, the next step is to begin initializing its contents. The first portion that needs to be initialized are the various headers in the PE. In order to initialize the contents, `WriteProcessMemory`[4] can be used which is prototyped as:

```
BOOL WriteProcessMemory(
```

```
  HANDLE hProcess,
  LPVOID lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesWritten
);
```

The headers are copied by calling `WriteProcessMemory` like so:

```
WriteProcessMemory(
  (HANDLE)-1,
  targetLibraryBuffer,
  downloadedLibraryBuffer,
  libraryNtHeader->OptionalHeader.SizeOfHeaders,
  NULL
);
```

Once the PE headers are populated, the next step is to populate each individual section of the image by enumeration. In order to enumerate the sections in the image the `FileHeader.NumberOfSections` attribute is used from the `IMAGE_NT_HEADER` portion of the PE. For each individual section, `WriteProcessMemory` should be called as follows[4]:

```
WriteProcessMemory(
  (HANDLE)-1,
  targetLibraryBuffer + sections[index].VirtualAddress,
  downloadedLibraryBuffer + sections[index].PointerToRawData,
  sections[index].SizeOfRawData,
  NULL
);
```

After the `targetLibraryBuffer` has been allocated and initialized it should be saved for later use.

6. Hook functions used during library loading

   Now that all of the buffers are initialized, it's time to actually hook the required set of functions that the dynamic loader uses when loading a library. Hooking, as discussed in previous chapters, is the process by which a call to a function is re-routed through an intermediate step, such as a custom function. This allows for adding extended error checking or functionality to an API without recompiling the original implementation. Microsoft's research group has an excellent implementation of function hooking as part of the *Detours* project[3]. The set of the functions that must be hooked in order to emulate their functionality against a memory

[4]The "sections" variable is of type PIMAGE_SECTION_HEADER.

range instead of a file on disk are listed below along with what their originally intended purpose is:

Function Name	Versions	Usage
`NtOpenSection`	2000+	Opens a section of a file.
`NtMapViewOfSection`	NT4+	Maps part or all of a section into memory.
`NtOpenFile`	NT4	Opens a file. This is similar to `open` in the Linux implementation.
`NtCreateSection`	NT4	Creates a section, or virtual memory block, that is associated with a file on disk. This is somewhat similar `mmap` in the Linux implementation.
`NtQueryAttributesFile`	NT4	Queries basic file attributes such as size. This is similar to `fxstat64` in the Linux implementation.

Each of the aforementioned functions are exported from `NTDLL.DLL`. The hook implementations of each of the functions emulate the expected file operations against the `targetLibraryBuffer` instead of a file on disk.

NtOpenFile

The "NtOpenFile" hook handles requests to the `NtOpenFile` function which is prototyped as[8]:

```
void NTAPI NtOpenFile(
  OUT PHANDLE FileHandle,
  IN ACCESS_MASK DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN ULONG ShareAccess,
  IN ULONG OpenOptions
);
```

The hook implementation must inspect the name of the file that is being passed in to see if it is the "fake" library name or not. This is done by checking the `ObjectName` attribute of the `ObjectAttributes` parameter. If the library name does match the "fake" library's name, a unique, identifiable handle should be returned in the `FileHandle` parameter. This file handle should then used by subsequent file operations. If the filename does not match, the original `NtOpenFile` shouldcalled.

NtQueryAttributesFile

The "NtQueryAttributesFile" hook handles requests to the `NtQueryAttributesFile` function which is prototyped as[8]:

```
NTSTATUS NTAPI NtQueryAttributesFile(
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  OUT PFILE_BASIC_INFORMATION FileAttributes
);
```

The hook implementation must do the same check that the `NtOpenFile` hook does by inspecting the `ObjectAttributes`' `ObjectName` attribute to see if it matches the "fake" name of the library that is being injected. If the name does match, the hook function should populate the `FileAttributes` argument with sane values, such as setting the `FileAttributes` attribute to `FILE_ATTRIBUTE_NORMAL`[5]. After the structure has been initialized, the hook function should return `STATUS_SUCCESS`. If the filename does not match, the original `NtQueryAttributesFile` should be called.

NtCreateSection and NtOpenSection

The "NtCreateSection" and "NtOpenSection" hooks handle requests to their respective functions and are prototyped as[8]:

```
NTSTATUS NTAPI NtCreateSection(
  OUT PHANDLE SectionHandle,
  IN ULONG DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  IN PLARGE_INTEGER MaximumSize,
  IN ULONG PageAttributes,
  IN ULONG SectionAttributes,
  IN HANDLE FileHandle
);

NTSTATUS NTAPI NtOpenSection(
  OUT PHANDLE SectionHandle,
  IN ACCESS_MASK DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes
);
```

Both of these functions emulate the same behavior as far as the hook routines are concerned. In the case of `NtCreateSection`, the hook function should check to see if the `FileHandle` argument matches a handle that may have been previously returned from the `NtOpenFile` hook. If it does not, the original `NtCreateSection` function is called. In the case of

[5]Other attributes, such as CreationTime, LastAccessTime, LastWriteTime, and ChangeTime should also be set to "sane" values.

NtOpenSection, the hook should simply check to see if the ObjectAttributes' ObjectName attribute matches the "fake" library name. If it does not, the orignial NtOpenSection is called.

In the case where the check passes for the two hook functions, the SectionHandle argument should be set to the targetLibraryBuffer that was initialized in the previous steps.

NtMapViewOfSection

The "NtMapViewOfSection" hook handles requests to the NtMapViewOfSection function and is prototyped as[8]:

```
NTSTATUS NTAPI NtMapViewOfSection(
  IN HANDLE SectionHandle,
  IN HANDLE ProcessHandle,
  IN OUT PVOID *BaseAddress,
  IN ULONG ZeroBits,
  IN ULONG CommitSize,
  IN OUT PLARGE_INTEGER SectionOffset,
  IN OUT PULONG ViewSize,
  IN SECTION_INHERIT InheritDisposition,
  IN ULONG AllocationType,
  IN ULONG Protect
);
```

The hook implementation must check to see if the SectionHandle argument matches with one that is associated with the "fake" library as previously returned from either NtCreateSection or NtOpenSection. If the handle does match, the hook function should set the BaseAddress argument to the targetLibraryBuffer and return STATUS_IMAGE_NOT_AT_BASE such that the image can be relocated gracefully. If the SectionHandle argument does not match, the original NtMapViewOfSection is called.

7. Load the library with a "fake" pathname

   With everything set up, the "fake" library can now be loaded. This is done by simply calling LoadLibraryA with the predetermined fake library name that the hook functions know to expect. By calling LoadLibraryA, the hook functions are indirectly called and emulate the expected behavior against the memory range. The order in which the hooks are called is:

Windows Version	Function Name
Windows 2000+	NtOpenSection NtMapViewOfSection
Windows NT 4.0	NtQueryAttributesFile NtOpenFile NtCreateSection NtMapViewOfSection

The end result: the library is loaded, relocated, and initialized.

Chapter 5

Potential Impacts

With the *how* of library injection covered, it would seem prudent to consider the potential impacts of this technology being incorporated into exploits and malware. As identified in the introduction, library injection lowers the bar for exploit writers such that it is no longer a requirement that one know assembly in any form; rather, all that must be known is how to program in any language that supports being linked as a dynamically loadable library[1]. The following sections will discuss a number of potential impacts and attempt to analyze the severity of each

5.1 Worm/Rootkit Deployment Automation

One of the scarier impacts of remote library injection involves the possibility for writing highly automated worms. These worms would use an arbitrary exploit to inject one or more libraries. Once the library or libraries load on the target machine, it would be possible to do a number of things. For instance, the library or libraries could propagate themselves to every other process on the machine by replicating into the address space of other processes. This means that not only would one process be infected, but so too would every other process on the machine[2]. This makes killing the worm a much harder task in that there is not just one or two processes that can be killed.

Aside from the local propagation to other processes, worm infection techniques can be made more advanced and intelligent due to the fact that the library, depending on the method of injection used, will be loaded under the radar of current Anti-Virus solutions. This allows the worm to maintain a greater level

[1] Which, as scary as it seems, includes Visual Basic.
[2] This is dependent on the access rights of the infected process.

of retention when it comes to being removed by aggressive or passive virus scanning. Since the worm is less likely to be detected, at least the host level, it is then inherently possible to write arbitrarily complex host infection methodologies. Simply put, a worm author is afforded more luxury when it comes to writing non-deterministic infection patterns that make heuristic identification by virus scanners just that much more complicated.

5.2 Operating System Independence

The luxury of being able to develop an injectable library brings with it the potential for writing advanced, cross platform worms that are capable of infecting a wide array of operating systems. Granted, the binary format and runtime libraries between each operating system are typically different, but it is still possible to write fairly portable code that can then be compiled down into the binary format of the target machine. Combine this ability with the fact that library injection, at least In-Memory library injection, is not currently detected by Anti-Virus scanners and one gets a worm that targets not just one operating, but instead a number of operating systems. At the time of this writing, the trend for worm infection seems to be uni-platform in nature from what the authors have witnessed.

5.3 Anti-Virus Nightmares

As with all new methods of infection, Anti-Virus vendors will have to react and come up with a solution to the problem posed by library injection. Granted, On-Disk library injection already has a means by which it can be detected, but In-Memory on the other hand is a whole different problem. Potential methods of detection are discussed in the chapter on `Prevention and Detection` (6).

Chapter 6

Prevention and Detection

This chapter will discuss potential ways in which remote library injection might be prevented or detected, both passively and aggressively. The methods of detection, much like the methods of injection themselves, vary greatly from platform to platform and as such will be discussed separately from one another.

As far as prevention is concerned, the most logical and repeatedly emphasized solution by the security industry is to ensure that machines remain patched and up-to-date when it comes to security related issues. Indeed, this does not help prevent against the unreleased vulnerabilities, but it is a method of prevention nonetheless. The second method of prevention comes in the form of `Host Intrusion Prevention Systems`, or HIPS. These packages implement host level intrusion detection and prevention features such as system call logging and analysis, page execution enforcement such as no-exec stacks, and other security improvement features such as ASLR. Both Linux and Windows have HIPS or HIPS-like software components and can be used to help with the prevention of exploits, thusly preventing the injection of libraries.

In the event that prevention isn't an option, detection becomes a requirement. Without prevention or detection, a system is left vulnerable to attack and infection from library injection based techniques. The following sections will analyze potential detection mechanisms for both Linux and Windows.

6.1 Linux

6.1.1 Inspecting Loaded Libraries

An external application can inspect the loaded library list in one of two ways. The least accurate way involves using the `proc` filesystem and looking at the `maps` file which contains memory mapping information within a given process. In the case of On-Disk library injection, this method is adequate in detecting whether or not a potentially malicious library has been loaded. On the other hand, In-Memory library injection is not quite as simple. The memory mapping for the library will not show up as being associated with a given file. As such, one cannot directly correlate a memory range with a malware library.

The second, more accurate option involves walking the linked list of loaded libraries in the context of the process. This can be done by using the `ptrace` function to attach to the process and read memory from within it. In order to enumerate the linked list of loaded libraries, one must first locate the first entry in the list. Fortunately, some versions of `ld-linux.so` have a named symbol in the `bss` called `_dl_rtld_map`. By adding the `st_value` attribute to the base address of `ld-linux.so`, the VMA for `_dl_rtld_map` can be calculated. Once the address is known, `ptrace` can be used to read the contents of the global variable which happens to be a `struct link_map *` variable. The `link_map` structure has the following exposed definition (as found in `/usr/include/link.h`):

```
struct link_map {
    ElfW(Addr) l_addr;
    char *l_name;
    ElfW(Dyn) *l_ld;
    struct link_map *l_next, *l_prev;
};
```

The `l_name` attribute is a pointer to the name of the library that was loaded. The `l_addr` attribute is the base address at which the library was mapped in. Finally, the `l_next` and `l_prev` attributes are pointers to the next an previous list entries, respectively. By walking the linked list, one can validate whether or not an individual library is valid based on a number of things, such as whether or not it actually exists on disk.

Though this all seems good in theory, there are inherent problems with the fact that one can easily make this detection method implausible. For instance, if an injected library were to remove itself from the `_dl_rtld_map` linked list, one would not be able to detect that it was actually loaded.

With that said, the authors are not aware of a method that can deterministically and reliably detect, either passively or aggressively, that a library has been

injected into a process.

6.1.2 Detecting Function Hooks

One of the critical components to In-Memory library injection is the reliance on being able to hook file operation functions. With that said, if it were possible to detect or prevent a function from being hooked, the problem of library injection, at least In-Memory library injection, could be avoided altogether. At the time of this writing, the authors are not aware of an implemented solution that prevents function hooking. However, a proof of concept capable of doing so is feasible.

One way to prevent the hooking of functions is to make it so an address range that is mapped in from an executable or library as a read-only mapping cannot be converted to read/write. In order to do arbitrary function hooking on Linux, the page of the function's preamble that is being hooked must have its page protection flags modified such that it can be written to. By default, text segments in executables and libraries are mapped in read-only. However, the `mprotect` system call allows for changing the protection flags on the fly for one or more pages. As such, all that is required to be done before modifying the function's preamble to is to use `mprotect` to set the protection flags to `PROT_READ | PROT_WRITE | PROT_EXEC`. If there were a kernel option that prevented the altering of protection from read-only to read-write, the ability to hook functions would be gracefully prevented.

Unfortunately, this comes at the cost of potentially disabling or breaking applications that have valid reasons for hooking functions. For these types of applications an approach can be taken that is similar to PaX[10] in that a binary can have an extra ELF flag set that indicates to the kernel that text segment protection enforcements should be disabled. This would allow for the select few applications that have valid reasons to hook functions the ability to do so without compromising the security of the machine itself. PaX supports some memory protection enforcement, but not for the scenario listed above.

In the event that function hooking cannot be prevented, another method of detection comes in form of comparing libraries in memory with their images on disk. This is useful in that it can easily detect that a function in a given library has been modified from its original form in memory, which by itself is a strong indication that either something malicious is occurring or some other, potentially valid, change has been made. An example of an application that compares ELF images in memory with their respective images on disk is `elfcmp`[12]. Though this can be useful, it is also easily bypassed by having the injected restore the functions to their original form after the library has completed loading.

6.2 Windows

6.2.1 Inspecting Loaded Libraries

The method of enumerating loaded libraries in the context of a given process is a supported feature of the Windows Process API. For this reason it's rather trivial to implement something that checks for libraries that either might be malicious or that do not exist on disk. The API functions and the process used to enumerate loaded libraries, or modules as they are referred to, is shown below[1]:

1. Attach to the process that is to be inspected

   In order to enumerate the modules loaded in the context a given process, one must first attach to it. This is accomplished by making use of the `OpenProcess` function, which takes as an argument a process identifer, an access level, and a boolean parameter that is outside the scope of this paper. When the goal is to enumerate loaded libraries, the `dwDesiredAccess` argument should be set to `(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ)`. Finally, the `dwProcessID` argument should be set to the process identifier of the process that is to be inspected. Upon success, `OpenProcess` will return a valid `HANDLE` which will then be used in subsequent calls. On failure, `NULL` is returned.

2. Enumerate the process' modules

   Once the process has been attached to, the next step involves actually enumerating the list of modules. This is done by making use of the `EnumProcessModules` function which populates an array if `HMODULE`'s, or handles to modules. On success, `TRUE` is returned and the array of `HMODULE`'s will hold one or more handles associated with the modules that have been loaded in the context of the attached process. Also, the `cbNeeded` parameter will be populated with the number of bytes that were actually needed to populate the array, thus indicating how many entries were actually populated. On failure, `FALSE` is returned.

   Once the array has been populated, one can proceed to enumerate it and check the names of the modules by making use of `GetModuleFileNameEx`, which takes as parameters the handle to the attached process and the handle to the module from which the name is being resolved. The buffer used to hold the name of the module is passed in as an output argument. On success, `GetModuleFileNameEx` returns the number of bytes copied into the output buffer. On failure, zero is returned.

The above steps are one way in which loaded libraries in the context of a given process can be enumerated. Another way involves injecting a custom library that

[1] This process is only supported on NT-based versions of Windows.

manually enumerates the loaded module list directly. Like Linux, which exposes the list of load libraries by way of the `_dl_rtld_map` symbol, Windows too has a deterministic location from which libraries can be enumerated. The list of loaded modules is located in the `Process Environment Block`, or PEB. This is an undocumented structure that holds information about the state of the process and can be directly referenced via `fs:[0x30]` on IA-32. Loader information is stored in the `LoaderData` attribute which is of type `PEB_LDR_DATA` and is found in the PEB. The `PEB_LDR_DATA` structure has the following definition as taken from NTInternals *The Undocumented Functions*[8]:

```
typedef struct _PEB_LDR_DATA {
  ULONG       Length;
  BOOLEAN     Initialized;
  PVOID       SsHandle;
  LIST_ENTRY  InLoadOrderModuleList;
  LIST_ENTRY  InMemoryOrderModuleList;
  LIST_ENTRY  InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

The `LIST_ENTRY` structure contains a logical previous and next pointer as used in a doubly linked list. Each entry in the three module linked lists point to a `LDR_MODULE` structure which contains the following information:

```
typedef struct _LDR_MODULE {
  LIST_ENTRY     InLoadOrderModuleList;
  LIST_ENTRY     InMemoryOrderModuleList;
  LIST_ENTRY     InInitializationOrderModuleList;
  PVOID          BaseAddress;
  PVOID          EntryPoint;
  ULONG          SizeOfImage;
  UNICODE_STRING FullDllName;
  UNICODE_STRING BaseDllName;
  ULONG          Flags;
  SHORT          LoadCount;
  SHORT          TlsIndex;
  LIST_ENTRY     HashTableEntry;
  ULONG          TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

Enumerating the module lists directly allows one a better glance at the actual state of the loaded libraries in that more information can be gathered vice being limited to just the name of the library. This approach is not foolproof, however. It is possible for the injected library to remove itself from the three linked lists and thus disappear from the record. As such, detecting a malicious library via this method should be seen as inadequate.

6.2.2 Detecting Function Hooks

Function hooking is an integral part to In-Memory library injection. It is what facilitates the ability to emulate file and image mapping operations. As such, developing a method used to detect function hooking can be seen as a means to inherently detect In-Memory library injection.

A plausible approach to detecting function hooking is to write a tool that enumerates the loaded module list in a process and compares the image in memory with the image on disk. Granted, this has the same problems that plague the other detection methods in that a library can remove itself from the lists of loaded modules. This fact alone makes this method of detection inadequate due to the fact that removing a library from said lists is trivial.

An alternative approach was outlined in the Linux section on detecting function hooks. The process involves removing the ability to change the protection flags of a library's text segment such that it cannot be changed from read-only to read-write. This makes it impossible to modify the content of a library's text segment in memory (which is a requirement for function hooking). The main problem with this is that some applications do have valid reasons for hooking underlying API functions, such as needing to add extended functionality or error checking that is not otherwise provided.

Finally, like in Linux, the loaded library's text segments can be compared from memory to disk to see if they match. If they do not, it is a good indicator that something odd or malicious has happened. This method of detection can also be used on Linux. Unfortunately, it's a flawed method. The injected library could manually unhook the functions and thus leave no trace that the hooks had existed.

Chapter 7

Conclusion

Library injection makes it possible for malware developers to write extremely advanced worms and viruses that are capable of executing under the radar of present day virus scanners. In the interest of addressing this issue before it becomes a common problem, this document has detailed how library injection works, what the potential impacts could be, and how it might be detected or prevented from happening. It is the authors' hope that the reader now has a clear understanding of library injection as a whole.

Bibliography

[1] eEye Digital Security. *Blaster Worm Analysis*. http://www.eeye.com/html/Research/Advisories/AL20030811.html; accessed Apr 01, 2004.

[2] The Last Stage Of Delerium. *Win32 Assembly Components*. http://www.lsd-pl.net/documents/winasm-1.0.1.pdf; accessed Nov 27, 2003.

[3] Microsoft Corporation. *Detours*. http://research.microsoft.com/sn/detours/; accessed Apr 03, 2004.

[4] Microsoft Corporation. *The Microsoft Developer Network*. http://msdn.microsoft.com; accessed Apr 04, 2004.

[5] Moore, Brett. *LoadLibrary Shell*. http://www.darklab.org/archive/msg00232.html; accessed Apr 04, 2004.

[6] Moore, David and Colleen Shannon. *The Spread of the Code-Red Worm (CRv2)*. http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml; accessed Apr 01, 2004.

[7] Nebbit, Gary. *Re: Launch an exec / proc from memory*. http://groups.google.com/groups?selm=91a3kr$66q$1@novalfsmtp1.novsvcs.net&output=gplain; accessed Apr 05, 2004.

[8] NTInternals.net. *The Undocumented Functions*. http://undocumented.ntinternals.net/; accessed Apr 03, 2004.

[9] PaX. *Address Space Layout Randomization*. http://pax.grsecurity.net/docs/aslr.txt; accessed Apr 03, 2004.

[10] PaX. *Non-executable Pages*. http://pax.grsecurity.net/docs/noexec.txt; accessed Apr 03, 2004.

[11] Russinovich, Mark. *Inside On-Access Virus Scanners.* http://www.winntmag.com/Articles/Index.cfm?IssueID=42&ArticleID=300; accessed Apr 02, 2004.

[12] skape. *elfcmp.* http://www.hick.org/code/skape/elfcmp; accessed Apr 04, 2004.

[13] skape. *Understanding Windows Shellcode.* http://www.hick.org/code/skape/papers/win32-shellcode.pdf; accessed Apr 02, 2004.

[14] Sysinternals. *Process Explorer.* http://www.sysinternals.com/ntw2k/freeware/procexp.shtml; accessed Apr 01, 2004.

[15] Tool Interface Standards. *Executable and Linkable Format.* http://www.hick.org/~mmiller/elf.txt; accessed Apr 03, 2004.