# Scan of the Month 33

Peter Košinár ⟨ goober@mtos.ksp.sk⟩

December 3, 2004

## Contents

# 1 Introduction

We assume that the reader is familiar with x86 architecture (registers, instruction set, etc.) and basic concepts from Windows (e.g. abbreviations like RVA = Relative Virtual Address). The protected binary was mostly analysed on a Linux system (except for the initial part where we used HIEW and the final part, where we verified our findings).

# 2 Analysis

The executable `0x90.exe` (294.912 bytes, MD5 sum: `7daba3c46a14107fc59e865d654fefe9`) was found on a WinXP system, so it would be reasonable to start with the assumption that it's an executable file in Portable Executable ([PE]) format. In order to verify this assumption, a very nice tool named HIEW ([HIEW]) was used. It has (among many other useful features) ability to display the headers of PE files in a form which is a bit more human-readable than a plain hexdump. Unfortunately, like many other "more-intelligent" tools (including [IDA], [OllyDbg] and even objdump(1)...), it also uses certain values from the headers for displaying more useful information. This works for "regular" binaries, however, the analysed binary was intentionally protected against curious eyes and one part of its protection is intentional modification of certain parts headers in order to prevent such tools from displaying useful output. Thus, the output of such tools might be unreliable and HIEW was chosen as a compromise between "comfort" and "robustness". Naturally, all the information provided by HIEW was independently verified by looking at the plain hexdump.

## 2.1 Old header

PE files begin with a standard MZ header (also called "old" or DOS .EXE header[1]. The MZ header begins with a two-byte signature (either 'MZ' or 'ZM', the later form is quite uncommon). At offset 60 one can find a doubleword value (`e_lfanew`) which represents the offset (in bytes) of the "new" (PE, in this case) header in the file. If the system loader finds that this offset points to a valid PE header, it ignores the remaining bytes of the "old" header and loads the file according to the information from the new header. Otherwise, it loads it as a plain old 16-bit DOS .EXE file (the details of this process aren't important in our case). In our case, `e_lfanew`=0x100 and a valid PE header is present at this offset in the file.

## 2.2 PE header

The structure of PE header is discussed in greater details in [PE], so we'll restrict our attention only to the parts related to the analysed binary. PE header begins with a signature 'PE', followed by many imporant fields[2] we'll examine later. The values of these fields are summarized in table 1. There are few bogus (intentionally obfuscated by the author) values in the header, namely *PointerToSymbolTable*, *NumberOfSymbols* (both of them are unimportant because the image is marked as having no symbols), *LoaderFlags* (most of the bits are unimportant) and, most importantly *NumberOfRvaAndSizes*. This number was (probably) originally meant to be used to denote the number of so-called directories (their purpose will be exaplained later), so some programs (e.g. HIEW) try treat it like that. However, real NT loader ignores this value and always considers the number of directories to be 16[3]. The header is followed by a list of the directories. Directories are tables used for various purposes (e.g. the table of imports, exports, TLS, ...) and they are described by two dwords – RVA[4] and the length of that particular table. In this binary, there is only one such table – namely the table of imports – which is located at RVA 0x48000 and its length is 190 bytes (we'll have a look at it later).

[1] See `_IMAGE_DOS_HEADER` in `winnt.h` in MSVC

[2] See `_IMAGE_NT_HEADERS` in `winnt.h` in MSVC

[3] The constant `IMAGE_NUMBEROF_DIRECTORY_ENTRIES` in `winnt.h`

[4] Relative Virtual Address, the real virtual address can be obtained by adding ImageBase

| | | |
|---|---|---|
| DWORD | Signature | `0x00005045` ('PE') |
| WORD | Machine | `0x14c` (Intel 386) |
| WORD | NumberOfSections | 4 |
| DWORD | TimeDateStamp | `0x851c3163` |
| DWORD | PointerToSymbolTable | **0x74726144** (bogus value "Dart") |
| DWORD | NumberOfSymbols | **0x00455068** (bogus value "hPE") |
| WORD | SizeOfOptionalHeader | 224 |
| WORD | Characteristics | `0x818f` (Little Endian + 32bit + Stripped line numbers, symbols, relocations + Executable) |
| WORD | Magic | `0x10b` (a 32-bit optional header follows) |
| BYTE | MajorLinkerVersion | 2 |
| BYTE | MinorLinkerVersion | 25 |
| DWORD | SizeOfCode | `0x200` |
| DWORD | SizeOfInitializedData | `0x45400` |
| DWORD | SizeOfUninitializedData | 0 |
| DWORD | AddressOfEntryPoint | `0x2000` |
| DWORD | BaseOfCode | `0x1000` |
| DWORD | BaseOfData | `0x2000` |
| DWORD | ImageBase | `0xde0000` |
| DWORD | SectionAlignment | `0x1000` |
| DWORD | FileAlignment | `0x1000` |
| WORD | MajorOperatingSystemVersion | 1 |
| WORD | MinorOperatingSystemVersion | 0 |
| WORD | MajorImageVersion | 0 |
| WORD | MinorImageVersion | 0 |
| WORD | MajorSubsystemVersion | 4 |
| WORD | MinorSubsystemVersion | 0 |
| DWORD | Win32VersionValue | 0 |
| DWORD | SizeOfImage | `0x49000` |
| DWORD | SizeOfHeaders | `0x1000` |
| DWORD | CheckSum | 0 |
| WORD | Subsystem | 3 (console application) |
| WORD | DllCharacteristics | 0 |
| DWORD | SizeOfStackReserve | `0x100000` |
| DWORD | SizeOfStackCommit | `0x2000` |
| DWORD | SizeOfHeapReserve | `0x100000` |
| DWORD | SizeOfHeapCommit | `0x1000` |
| DWORD | LoaderFlags | **0xabdbffde** (bogus) |
| DWORD | NumberOfRvaAndSizes | **0xdfffddde** (bogus) |

Table 1: Contents of PE header

## 2.3 Sections

At the offset *SizeOfOptionalHeader* from the *Magic* value[5] one can find the list of *NumberOfSections*=4 section headers. Each section header describes one section[6], see table 2. The meaning of the fields is quite intuitive – *VirtualAddress* is RVA where that particular section will be loaded (if its *Characteristics* says that it should be loaded), *PointerToRawData* is offset in file where the data of the section are located. The other interesting values are *VirtualSize* and *SizeOfRawData*. First describes the size of the section in memory, the other on in the file. It's important to know that the loader takes smaller of these numbers and only loads that many bytes from the file. Thus, the section "NicolasB", which intentionally contains bogus `SizeOfRawData` is, in fact, uninteresting `:-)`.

| | | | | | |
|---|---|---|---|---|---|
| BYTE[] | Name | "CODE" | "DATA" | "NicolasB" | ".idata" |
| DWORD | VirtualSize | 0x1000 | 0x45000 | 0x1000 | 0x1000 |
| DWORD | VirtualAddress | 0x1000 | 0x2000 | 0x47000 | 0x48000 |
| DWORD | SizeOfRawData | 0x1000 | 0x4500 | **0xefefadff** | 0x1000 |
| DWORD | PointerToRawData | 0x1000 | 0x2000 | 0x47000 | 0x47000 |
| DWORD | PointerToRelocations | 0 | 0 | 0 | 0 |
| DWORD | PointerToLinenumbers | 0 | 0 | 0 | 0 |
| WORD | NumberOfRelocations | 0 | 0 | 0 | 0 |
| WORD | NumberOfLinenumbers | 0 | 0 | 0 | 0 |
| DWORD | Characteristics | 0xe0000020 | 0xc0000040 | 0xc0000040 | 0xc0000040 |

Table 2: List of sections

### 2.3.1 "CODE" section

This section contains just a debug breakpoint ("INT 3"; opcode 0x`cc`) followed by four far jumps which refer to four imported library calls (more details in part 2.3.4). Except for these 25 bytes, this section is filled with zeroes.

### 2.3.2 "DATA" section

Although the section name suggests something else, this section contains the actual code which we are going to analyse, so we'll deal with its contents in more details later.

### 2.3.3 "NicolasB" section

Well, as we already mentioned, this section doesn't seem to contain anything useful `:-)` (to be honest, it overlaps with the ".idata" section in the file, so if we blindly changed its *SizeOfRawData* to 0, the application might be able to detect that this section wasn't loaded and report a tampered executable (or simply die). So, it contains a copy of the ".idata" section, except that the addresses of imported functions aren't replaced by their real addresses in this section. And well, it also contains a message and a signature from the author – at offset 0x`470e0` in the file "`You really thought you would find strings eh?  ;-)`" and at offset 0x47130 "`Scan of the month coded by Nicolas Brulez / Digital River`".

### 2.3.4 ".idata" section

This section contains the table of imported function. This is (in this case) a two-level table where first level is a list of import descriptors[7], each of them describing imports from one dynamic linkable library and second level consists of lists of pointers to blocks describing actually imported

[5] i.e. at offset 0x`1f8` in the file
[6] See `_IMAGE_SECTION_HEADER` in `winnt.h` in MSVC
[7] See `_IMAGE_IMPORT_DESCRIPTOR` in `winnt.h` in MSVC

functions[8]. The information about imports is summarized in table 3. Apparently, the binary probably uses (at least) four Windows API functions – GetCommandLine, GetTickCount, ExitProcess and printf. Of course, it may later load some other functions[9].

It may also be interesting to have a look at values originally stored at places where the addresses of imported functions are going to be stored. These are sometimes set by the compiler to correspond to true addresses of API functions on the system where the file was compiled[10], so they may be useful for tracing the version/kind of OS that was used to build the malicious binary. Unfortunately, we didn't possess enough different variants of `kernel32.dll` so we weren't able to find what kind of system is Nicolas working with `:-)`.

| | | | |
|---|---|---|---|
| DWORD | OriginalFirstThunk | 0x4803c | 0x48044 |
| DWORD | TimeDateStamp | 0 | 0 |
| DWORD | ForwarderChain | 0 | 0 |
| DWORD | Name | 0x4806c ("msvcrt.dll") | 0x48077 ("KERNEL32.dll") |
| DWORD | FirstThunk | 0x48054 | 0x4805c |

| Library | Ordinal | Name | Location VA | Original value |
|---|---|---|---|---|
| msvcrt.dll | 740 | printf | 0xe28054 | 0x77c1186a |
| KERNEL32.dll | 458 | GetTickCount | 0xe2805c | 0x7c8092ac |
| KERNEL32.dll | 258 | GetCommandLine | 0xe28060 | 0x7c812c8d |
| KERNEL32.dll | 175 | ExitProcess | 0xe28064 | 0x7c81caa2 |

Table 3: List of imported functions

#### 2.3.5 Summary

The file headers were altered at (at least) five different places – the important changes are the number of directories (*NumberOfRvaAndSizes*) and *SizeOfRawData* of one section. For example, these changes prevented OllyDbg from loading the file ("Bad of unknown format of 32-bit file ...") and IDA also didn't like the file very much – e.g. my IDA produced "chsize: no space left on device" `:-)`. Even HIEW produced a warning message "Import data No free memory" while reading the file.

Naturally, these modifications could be "fixed" (e.g. by changing *NumberOfRvaAndSizes* to 16 and *SizeOfRawData* of third section to `0x1000`), so the tools wouldn't display warning/error messages. However, such changes might not be "safe" to perform – if the code included checksums or otherwise depended on integrity of the executable, the results of our analysis might be incorrect[11].

If the binary wasn't specifically designed to be hard-to-crack, a good starting point would be setting up a breakpoint at all four API functions imported by the binary and just running it. However, this would not work with this binary[12].

### 2.4 Envelope

So, we are now ready to start playing with the real code of the binary. The code (section "DATA") is loaded at `0xde2000` and this location is also the entrypoint of the program. The first few instructions look quite normally (the disassembly was obtained using `objdump --disassemble-all -M intel 0x90-1.exe` after replacing doubleword at offset 0x10c (*PointerToSymbolTable*) by zero):

[8] See `_IMAGE_THUNK_DATA32` in `winnt.h`; although in this case our naming scheme doesn't follow that file.

[9] Usually, this is accomplished by using LoadLibrary+GetProcAddress API calls or, more paranoidly, by manually searching the address space of loaded libraries for particular exported function.

[10] It may be related to so-called "bound imports", which are not used in this case.

[11] After all, we would be analysing a different executable `:-)`

[12] At least not directly, though there *are* some tricks that can be used for circumventing the breakpoint-detection code, see part 4

```
de2000:  60                  pusha
de2001:  e8 00 00 00 00      call   0xde2006
de2006:  5d                  pop    ebp
de2007:  8b c5               mov    eax,ebp
de2009:  83 e8 06            sub    eax,0x6
de200c:  81 ed 06 20 de 00   sub    ebp,0xde2006
de2012:  60                  pusha
```

This code accomplishes three things – it saves all registers to the stack, sets EAX to point to the entrypoint (0xde2000) and EBP to zero. The call-followed-by-pop technique was very common in old parasitic viruses (which also needed to be position-independent). It is followed by one more innocent instruction – `pusha`. Following this instruction, there is a mess of strangely-looking instructions, many of them prefixed with segment/repeat prefixes (like `gs`, `ss`, `repnz`, ...). Such a mess usually appears in regular (unprotected) binaries as a result of damage in the filesystem `:-)` but in this case, it was intentionally added by the author. So, we'll need to look closer at it.

### 2.4.1 Dummy code

Looking at the first few instructions, it becomes apparent that neither of the instructions does any memory access (neither read, nor write), most of them just move values between registers. They are prefixed with all strange combination of segment-register and/or repeat prefixes. These prefixes are ignored for most instructions, in particular for all instructions used in this part of the program. Following the execution flow further, we arrive at another kind of instruction – unconditional jump, which jump exactly one byte ahead (`eb 01`). This is a common technique for confusing disassemblers – if the byte immediately following the jump (which is jumped over and thus not executed during the real execution) is disassembled and the corresponding instruction is multi-byte, the jump will point into the middle of this instruction and the disassembly may be quite confusing. However, more intelligent disassemblers (like IDA) know that this jump is unconditional, so there is no point in disassembling the bytes following the jump (of course, unless there is a jump which points to them). This technique is sometimes extended a little bit by a "back-forward" jumps (which are sometimes able to confuse even IDA) – for example the following 16-bit code snippet[13]

| | | | |
|---|---|---|---|
| `53` | `push` | `bx` | |
| `bb eb 04` | `mov` | `bx, 0x04eb` | the operand is an unconditional jump |
| `5b` | `pop` | `bx` | |
| `eb fb` | `jmp` | `X` | where X is the second byte of the "mov" instruction |
| `XX` | `XX` | | Some dummy byte, 0x`9a` or 0x`e8` is a good choice |
| `...` | `...` | | following code |

Other common similar construction is "jump-if-x" followed by "jump-if-not-x" pointing to the same location (e.g. `jnz XXX; jz XXX`. Clearly, the effect of this piece of code is the same as the unconditional jump, but most disassemblers doesn't seem to use this fact.

But back to the analysed binary... Following the code further, we'll see just the mess of instructions which do not perform any memory access, until we come to 0x`de2288`, which contains `popa` instruction:

```
de2288:  61  popa
```

Heureka! The whole block between 0x`de2012` and 0x`de2288` did NOTHING[14]! This is a standard, though not very efficient, technique for repelling the analysers based on the assumption that "if there are loads of dummy code, nobody will be patient enough to trace through it" `:-)`. However, this assumption is not completely correct, because with modern tools, such dummy code can be automagically skipped (as we'll mention in part 4).

[13] Borrowed (without permission `:-)` ) from an executable protected by executable protector HackStop

[14] To be exact, it is equivalent to `pusha` followed by `popa`; this sequence isn't equivalent to `nop` because it modifies a few bytes of memory just above the current ESP.

### 2.4.2 Dummy exception blocks

Now, we are looking at another `pusha`, this time at 0x`de2289`. However, this time, it's not followed by messy-looking instructions, the code looks quite "normally":

```
de2289:  60                     pusha
de228a:  e8 48 00 00 00         call         0xde22d7
de228f:  8b 4c 24 0c            mov          ecx,DWORD PTR [esp+12]
de2293:  83 81 b8 00 00 00 02   add          DWORD PTR [ecx+184],0x2
de229a:  33 c0                  xor          eax,eax
de229c:  89 41 04               mov          DWORD PTR [ecx+4],eax
de229f:  89 41 08               mov          DWORD PTR [ecx+8],eax
de22a2:  89 41 0c               mov          DWORD PTR [ecx+12],eax
de22a5:  89 41 10               mov          DWORD PTR [ecx+16],eax
de22a8:  89 41 14               mov          DWORD PTR [ecx+20],eax
de22ab:  c7 41 18 55 01 00 00   mov          DWORD PTR [ecx+24],0x155
de22b2:  8b 81 b0 00 00 00      mov          eax,DWORD PTR [ecx+176]
de22b8:  50                     push         eax
de22b9:  0f a2                  cpuid
de22bb:  0f 31                  rdtsc
de22bd:  2b 04 24               sub          eax,DWORD PTR [esp]
de22c0:  83 c4 04               add          esp,0x4
de22c3:  3d 00 00 0e 00         cmp          eax,0xe0000
de22c8:  77 03                  ja           0xde22cd
de22ca:  33 c0                  xor          eax,eax
de22cc:  c3                     ret
de22cd:  83 81 b8 00 00 00 63   add          DWORD PTR [ecx+184],0x63
de22d4:  33 c0                  xor          eax,eax
de22d6:  c3                     ret
de22d7:  33 c0                  xor          eax,eax
de22d9:  64 ff 30               push         fs:DWORD PTR [eax]
de22dc:  64 89 20               mov          fs:DWORD PTR [eax],esp
de22df:  0f a2                  cpuid
de22e1:  0f 31                  rdtsc
de22e3:  33 db                  xor          ebx,ebx
de22e5:  8f 03                  pop          DWORD PTR [ebx]
```

```
de22e7:  64 67 8f 06 00 00      addr16 pop fs:[0]
de22ed:  83 c4 04               add          esp,0x4
de22f0:  61                     popa
```

Now, let's try to understand its purpose. . . First, it saves all registers to the stack, the execution is transferred (via call, thus saving the address of next instruction to the stack) to 0x`0xde22d7`, where it saves current content of FS:[0] to the stack and replaces it with current value of ESP. Then, it performs `cpuid` followed by `rdtsc`. The later instruction retrives the value of 64-bit counter named TSC which increases every clockcycle and stores it in EDX:EAX pair. Finally, the code attempts to `pop` the contents of DS:[0] from the stack.

However, there is just a little catch – there is no accessible memory at that address. Thus, the CPU will raise an exception which will be processed by the OS and so-called "application exception handler" will get called. How does it work? We won't describe all the gory details of exception handling under NT-based OS, just the way it's used in this particular executable[15].

Segment register FS points to something called TEB[16], whose first entry is a doubleword pointing to the last entry of linked list of exception handlers. If an exception occurs during the execution of the application, actual state of registers is saved on the stack and the first handler

[15] Curious reader may find them at `http://?/`

[16] Thread Environment Block, detailed structure can be found at `_NT_TIB` in `winnt.h` in MSVC

in this list is called. It can then take appropriate action (like, informing the user that something unexpected happend, or silently fix the problem, or cause the application to die, etc.). Finally, it tells the system if it was able to process the exception. If not, next handler in the list is called, and so on[17]. Applications can easily register their own exception handlers by pointing that doubleword to a block consisting of two doubleword-sized pointers – one points to the actual exception handler and the other is a pointer to the tail of the list. It's quite common to store these blocks on stack and this is precisely what this application does. The head of the list is at `FS:[0]`. The stack layout at the time the exception occurs is quite simple:

| | | |
|---|---|---|
| ESP, FS:[0] ⟶ | old contents of FS:[0] | saved by `push fs:DWORD PTR [eax]` |
| | 0x`de228f` | saved by `call` |
| | old contents of registers | saved by `pusha` |

Thus, when the exception occurs, the code at 0x`0xde228f` will get called. Of course, the stack layout at that time will be different – among many other things, it'll contain the saved values of all registers, some additional information about the exception, etc. Once again, although the exact details are necessary for full understanding of what really happens[18], we'll talk only about the parts relevant to this particular case.

At ESP+12, there is a pointer to the saved context, which is loaded into ECX register. It points to a large structure, which contains, all the saved registers. At offset 184 from the beginning is the value of EIP which points to the place where the exception occured. So, the first instruction just shifts EIP to point to the instruction following the one which (intentionally) caused the exception. The next few instructions cleans up the contents of debug registers DR0-DR3, DR6, DR7 in order to eliminate any hardware breakpoints. Then, original value of EAX is loaded (i.e. which was the value returned by `rdtsc` before the exception occured), the `cpuid`/`rdtsc` combo is executed once again and the old value of EAX is subtracted from the new. If the result is not too big (i.e. not greater than 0x`e0000`), the handler returns, otherwise it shifts the saved EIP once again, this time by a larger amount.

Under normal execution, the value in EAX will be smaller than 0x`e0000`, so the execution will never change the value of EIP. However, if the code is executed under debugger, there will be some additional overhead by the debugger which may cause this value to overstep the threshold and the execution of the application will be redirected to some crazy place where it'll probably die (or cause an almost-endless loop of exceptions).

So, what the code essentially does – it moves EIP to point to the next instruction, clears the debugging registers and verifies, whether it is executing at reasonable speed. Let's have a look at what happens once it returns from the exception handler. The execution will continue at 0x`de22e7`, where it restores the original value of FS:[0], removes the address of exception handler from the stack and finally restores all the registers which were saved by `pusha`. Thus, once again, this code does almost NOTHING `:-)`.

### 2.4.3 The first version DummyKiller

Looking at the following instructions, it becomes apparent that there will be many other occurances of these two anti-curious-eyes tricks, so the time to develop our first AntiAntiDebug tool has come. The approach demonstrated here is based on using a disassembler to skip these two known types of dummy code and displaying only the interesting portions of the file (for alternative approach, see section 4). For this purpose, the BFD library ([BFD]) was used and a small disassembler/deprotector was born.

Essentially, it is a finite state automaton (with one counter) which disassembles every instruction, checks if it is a popa/pusha/unconditional jump and changes it state accordingly. The counter is used to keep track of the number of `pusha`-s found. If this number becomes equal to 2 (because we need to ignore the `pusha` at the very beginning of the program), the code is assumed to be a part of a dummy block. If it matches a known pattern (the dummy exception block described in

[17] Again, exact defails can be found elsewhere.

[18] Look for `_EXCEPTION_POINTERS`, `_EXCEPTION_RECORD` and `_CONTEXT` (for x86) in `winnt.h` in MSVC.

part 2.4.2 ), it is skipped at once, otherwise, it is disassembled instruction-by-instruction and each instruction is checked for memory-access[19] and aborts in case it detects it. Unconditional jumps are also processed, in order to avoid disassembling the instructions which are never going to be executed. If the instructions aren't part of the dummy block, they are disassembled and displayed. The program which performs this "de-dummyfication" is in `gen1.c` file. WARNING: Do not try to read or understand the program. It may cause serious damage to your mental and/or physical health `:-)`

First part of program's output (after removing the lines added by the program to show the presence of dummy code and rewriting the value on first line to hexadecimal) is in table 4.

We already know the purpose of the first block of the code, the second (one-line) block is also quite simple – it just stores the value of EAX (which holds the address where the code begins – namely 0x`de2000`) to doubleword at 0x`e26441`. This is followed by three very similar blocks.

Let's analyse first of them! 0x`de100d` points to the far jump instruction in "CODE" section (see part 2.3.1) which jumps to to the GetCommandLine API function. Then, it takes its argument (i.e. the address of memory location, where the actual address of the API function is stored, in this case 0x`e28060`) and dereferences it (thus obtaining the true address of the API function). This value is then stored in EDI register, ECX is filled with 4, EAX with 0x`cc` (which is, not very surprisingly, the opcode of debug breakpoint (`int 3`)). Finally, ECX bytes beginning at EDI are scanned for value contained in AL. If it is not found (i.e. no breakpoint found on that API call), the execution follows normally, otherwise it jumps to a random location (obtained by reading the actual value of TSC `:-)`). This is the reason why putting a breakpoint at the API function wouldn't work (as we mentioned in part 2.3.5). On the other hand, if we put breakpoint not to the first instruction, rather to a later one (which is more than 4 bytes from the beginning of that API handler), we would have passed this check `:-)`. The other two blocks check printf and ExitProcess.

### 2.4.4 Encryption

Now, we are coming to something more interesting. The aforementioned code is followed by the code in table 5 (again, the non-hexadecimal values were manually replaced by their hexadecimal equivalents and adding a few lines which weren't on the path followed by the automated tool but which are nevertheless relevant).

This code looks like a loop (in fact, a "for"-cycle), where EDI is the control variable of the cycle. Its value is incremented by 1 in every pass of the loop and once it reaches 4, execution is transferred to 0x`de5423`. In every iteration, the code loads ESI and EBX registers from tables stored at 0x`e26419` and 0x`e2642d` (EDI-th doubleword in the table). Both values are adjusted[20] by adding base address of the code (0x`de2000`). The value of EBX is then pushed to the stack. The next `call` followed by `pop` sets EBX to the address of instruction immediately following the `call` (in this case, 0x`de540a`). Finally, the execution is transferred to the code at location pointed to by ESI.

Naturally, we'll need to verify whether this code returns back to this "loop" (otherwise, it wouln't be a loop `:-)`, just something loop-like looking and attempting to fool us). So, let's examine the contents at 0x`e26419` and 0x`e2642d`.

| `[e26419]` | | `[e2642d]` | |
|---|---|---|---|
| Original | Adjusted | Original | Adjusted |
| **0xdead** | | `0x31000` | |
| `0004440b` | `00e2640b` | `fece5a48` | `ffac7a48` |
| `000443f6` | `00e263f6` | `fe686eda` | `ff468eda` |
| `000443fe` | `00e263fe` | `ff2d68f4` | `000b88f4` |
| `0004435b` | `00e2635b` | `ff63ff58` | `00421f58` |

[19] The check is incredibly dumb, it just looks for the presence of '[' in the disassembled string; the exception is `lea` instruction, which is permitted to use it

[20] This description doesn't follow the chronological order!

| | | | |
|---|---|---|---|
| de2000: | 60 | pusha | |
| de2001: | e8 00 00 00 00 | call | 0xde2006 |
| de2006: | 5d | pop | ebp |
| de2007: | 8b c5 | mov | eax,ebp |
| de2009: | 83 e8 06 | sub | eax,0x6 |
| de200c: | 81 ed 06 20 de 00 | sub | ebp,0xde2006 |
| de2603: | 89 85 41 64 e2 00 | mov | DWORD PTR [ebp+0xe26441],eax |
| de2950: | b8 0d 10 de 00 | mov | eax,0xde100d |
| de2c2c: | 8b 40 02 | mov | eax,DWORD PTR [eax+2] |
| de2ef1: | 8b 00 | mov | eax,DWORD PTR [eax] |
| de2f5b: | 8b f8 | mov | edi,eax |
| de2f5d: | b9 04 00 00 00 | mov | ecx,0x4 |
| de31fa: | b8 60 06 00 00 | mov | eax,0x660 |
| de31ff: | c1 e8 03 | shr | eax,0x3 |
| de3202: | f2 ae | repnz | scas al,es:[edi] |
| de3204: | 85 c9 | test | ecx,ecx |
| de3206: | 74 04 | je | 0xde320c |
| de3208: | 0f 31 | rdtsc | |
| de320a: | 50 | push | eax |
| de320b: | c3 | ret | |
| de3274: | b8 01 10 de 00 | mov | eax,0xde1001 |
| de3558: | 8b 40 02 | mov | eax,DWORD PTR [eax+2] |
| de355b: | 8b 00 | mov | eax,DWORD PTR [eax] |
| de37f1: | 8b f8 | mov | edi,eax |
| de3a90: | b9 04 00 00 00 | mov | ecx,0x4 |
| de3a95: | b8 60 06 00 00 | mov | eax,0x660 |
| de3a9a: | c1 e8 03 | shr | eax,0x3 |
| de3d2f: | f2 ae | repnz | scas al,es:[edi] |
| de3d31: | 85 c9 | test | ecx,ecx |
| de3d33: | 74 04 | je | 0xde3d39 |
| de3d35: | 0f 31 | rdtsc | |
| de3d37: | 50 | push | eax |
| de3d38: | c3 | ret | |
| de4048: | b8 13 10 de 00 | mov | eax,0xde1013 |
| de430b: | 8b 40 02 | mov | eax,DWORD PTR [eax+2] |
| de430e: | 8b 00 | mov | eax,DWORD PTR [eax] |
| de45cf: | 8b f8 | mov | edi,eax |
| de45d1: | b9 04 00 00 00 | mov | ecx,0x4 |
| de45d6: | b8 60 06 00 00 | mov | eax,0x660 |
| de45db: | c1 e8 03 | shr | eax,0x3 |
| de4884: | f2 ae | repnz | scas al,es:[edi] |
| de4886: | 85 c9 | test | ecx,ecx |
| de4888: | 74 04 | je | 0xde488e |
| de488a: | 0f 31 | rdtsc | |
| de488c: | 50 | push | eax |
| de488d: | c3 | ret | |

Table 4: First part of automatically de-dummyfied code

| | | | |
|---|---|---|---|
| de517d: | 33 ff | xor | edi,edi |
| de517f: | 47 | inc | edi |
| de5180: | 8b b4 bd 19 64 e2 00 | mov | esi,DWORD PTR [ebp+edi+0xe26419] |
| de5187: | 8b 9c bd 2d 64 e2 00 | mov | ebx,DWORD PTR [ebp+edi+0xe2642d] |
| de53fe: | 03 9d 41 64 e2 00 | add | ebx,DWORD PTR [ebp+0xe26441] |
| de5404: | 53 | push | ebx |
| de5405: | e8 16 00 00 00 | call | 0xde5420 |
| *de540a:* | eb 01 | jmp | 0xde540d |
| de540c: | e8 | dummy byte | |
| de540d: | 03 b5 41 64 e2 00 | add | esi,DWORD PTR [ebp+0xe26441] |
| de5413: | ff e6 | jmp | esi |
| *de5415*: | 83 ff 04 | cmp | edi,0x4 |
| *de5418*: | 0f 85 61 fd ff ff | jne | 0xde517f |
| *de541e*: | eb 03 | jmp | 0xde5423 |
| de5420: | 5b | pop | ebx |
| de5421: | eb ea | jmp | 0xde540d |

Table 5: Second part of automatically de-dummyfied code

The values in second column look like valid addresses, so our belief in the conjectured functionality of the code in table 5 is strengthened. Let's disassemble the code at the referenced locations!

```
e2640b:  8d 85 23 54 de 00     lea   eax,[ebp+0xde5423]
e26411:  81 04 24 cd d9 31 01  add   DWORD PTR [esp],0x131d9cd
e26418:  c3                    ret
```

This looks interesting! The first instruction sets EAX to 0x`de5423` (which is exactly the location, where the code in table 5 would jump after the loop, what a coincidence!), the second does something strange to the value stored on stack and then returns. What was the value stored on the stack? It was 0x`ffac7a48` and after adding 0x`131d9cd`, the result is 0x`de5415`, again a value we expected! So, let's have a look at the next piece of code!

```
e263f6:  81 04 24 3b c5 97 01  add   DWORD PTR [esp],0x197c53b
e263fd:  c3                    ret
```

This piece is even more trivial than the previous – it just returns to 0x`de5415`. So, what about the third?

```
e263fe:  b9 38 0f 04 00        mov   ecx,0x40f38
e26403:  81 04 24 21 cb d2 00  add   DWORD PTR [esp],0xd2cb21
e2640a:  c3                    ret
```

This one loads ECX with the value 0x`40f38` and again, returns to 0x`de5415`. What about the final one?

```
e2635b:  30 08                  xor         BYTE PTR [eax],cl
e2635d:  40                     inc         eax
e2635e:  49                     dec         ecx
e2635f:  85 c9                  test        ecx,ecx
e26361:  75 f8                  jne         0xe2635b

e26363:  8d 85 1e 54 de 00      lea         eax,[ebp+0xde541e]
e26369:  80 38 cc               cmp         BYTE PTR [eax],0xcc
e2636c:  75 04                  jne         0xe26372
e2636e:  0f 31                  rdtsc
e26370:  50                     push        eax
e26371:  c3                     ret

e26372:  e8 5c 00 00 00         call        0xe263d3
e26377:  c7                     dummy byte
e26378:  8b 7c 24 0c            mov         edi,DWORD PTR [esp+12]
e2637c:  83 87 b8 00 00 00 02   add         DWORD PTR [edi+184],0x2
e26383:  33 c0                  xor         eax,eax
e26385:  8d 7f 04               lea         edi,[edi+4]
e26388:  ab                     stos        es:[edi],eax
e26389:  ab                     stos        es:[edi],eax
e2638a:  ab                     stos        es:[edi],eax
e2638b:  ab                     stos        es:[edi],eax
e2638c:  ab                     stos        es:[edi],eax
e2638d:  66 b8 aa 01            mov         ax,0x1aa
e26391:  34 ff                  xor         al,0xff
e26393:  ab                     stos        es:[edi],eax
e26394:  8b 87 a8 00 00 00      mov         eax,DWORD PTR [edi+168]
e2639a:  81 40 28 f0 a3 87 01   add         DWORD PTR [eax+40],0x187a3f0
e263a1:  8b 87 94 00 00 00      mov         eax,DWORD PTR [edi+148]
e263a7:  50                     push        eax
e263a8:  0f a2                  cpuid
e263aa:  0f 31                  rdtsc
e263ac:  2b 04 24               sub         eax,DWORD PTR [esp]
e263af:  83 c4 04               add         esp,0x4
e263b2:  3d 00 00 0e 00         cmp         eax,0xe0000
e263b7:  77 10                  ja          0xe263c9
e263b9:  8b 87 a8 00 00 00      mov         eax,DWORD PTR [edi+168]
e263bf:  81 68 28 33 6f eb 00   sub         DWORD PTR [eax+40],0xeb6f33
e263c6:  2b c0                  sub         eax,eax
e263c8:  c3                     ret
e263c9:  83 87 9c 00 00 00 32   add         DWORD PTR [edi+156],0x32
e263d0:  2b c0                  sub         eax,eax
e263d2:  c3                     ret
e263d3:  ff 04 24               inc         DWORD PTR [esp]
e263d6:  64 67 ff 36 00 00      addr16      push fs:[0]
e263dc:  64 67 89 26 00 00      addr16      mov fs:[0],esp
e263e2:  60                     pusha
e263e3:  0f a2                  cpuid
e263e5:  0f 31                  rdtsc
e263e7:  33 db                  xor         ebx,ebx
e263e9:  89 1b                  mov         DWORD PTR [ebx],ebx
e263eb:  61                     popa
e263ec:  64 67 8f 06 00 00      addr16      pop fs:[0]
e263f2:  83 c4 04               add         esp,0x4
e263f5:  c3                     ret
```

Whew! What a long code :-) The first part is a simple decryption loop – it xor-s ECX bytes, beginning at EAX, with a repeating key. Second part is also quite simple – it just checks for the presence of a breakpoint at 0xde541e, which is exactly the place where someone would put a breakpoint if (s)he wanted to stop right after the EDI-loop (table 5). The last part is again something exception-related. This time, however, there's something new – the exception handler doesn't follow directly after the `call`, rather there is one dummy byte (the address pushed by the `call` is incremented at 0xe263d3). And, the handler also contains code, which modifies the value on the top of the stack (at 0xe2639a and 0xe263bf). Again, this code returns to 0xde5415.

### 2.4.5 The second version DummyKiller

After performing the decryption semi-manually, it becomes apparent that the code at 0xde5423 is again filled with dummy pieces of code interspersed with exception blocks, just like the outermost layer of the envelope. Thus, it'll probably be useful to add the ability of decryption to our automated tool. The amended version is in `gen2.c` file[21]. So, after unpacking quite many layers of the protection, we finally arrive to something resembling a real code at location 0xde8653.

## 2.5 Main code

Again, the main code is, just like the envelope, intermixed with dummy code, in order to make the analysis more difficult. However, our automated tool is already able to get rid of a few forms of such dummy code, so this is not a big problem. Thus let's have a look at the de-dummyfied code:

```
de8653:  68 3d ba e1 00                 push  0xe1ba3d
de86c0:  68 2f 87 de 00                 push  0xde872f
de872d:  eb 1f                          jmp   0x00de874e
de874e:  81 34 24 30 58 41 48           xor   DWORD PTR [esp],0x48415830 HAX0
de89fc:  8f 05 73 bc e1 00              pop   ds:0xe1bc73
de8ccc:  c7 05 36 87 de 00 45 76 69 6c  mov   ds:0xde8736,0x6c697645 Evil
de8f79:  c7 05 3a 87 de 00 20 48 61 73  mov   ds:0xde873a,0x73614820 ␣Has
de920e:  c7 05 3e 87 de 00 20 4e 6f 20  mov   ds:0xde873e,0x206f4e20 ␣No␣
de94aa:  c7 05 42 87 de 00 42 6f 75 6e  mov   ds:0xde8742,0x6e756f42 Boun
de977c:  c7 05 46 87 de 00 64 61 72 69  mov   ds:0xde8746,0x69726164 dari
de99e9:  c7 05 4a 87 de 00 65 73 20 21  mov   ds:0xde874a,0x21207365 es␣!
de9c9a:  8b 34 24                       mov   esi,DWORD PTR [esp]
de9c9d:  58                             pop   eax
```

Apparently, this is some kind of initialization routine. The doublewords at 0xde8736—0xde874a are filled with a message from the author: "Evil Has No Boundaries !" and the value on the stack is xor-ed by "HAX0" [22]. Except for this simple activity, the code initializes both ESI and EAX to point to 0xe1ba3d and [0xe1bc73] to be equal to 0xDE872F xor 0x48415830.

### 2.5.1 Exception-al "goto"

Now we arrive to the nicest part of the code, which is shown in table 6.

The first part of this code can be symbolically rewritten as `EDI=0xe1b991[*(byte*)ESI]` followed by `EAX=*(byte*)(ESI+1)`. The second parts is once again an exception handler, just like the ones we have already seen in part 2.4.2 (e.g. DR cleaning). However, there is an important difference between these "new" handlers and those old ones. The old handlers returned to the same place[23], whereas the new handlers intentionally return somewhere else. The new location is

[21] This one is even more sloppy about doing necessary checks; it was written in haste and for only one purpose – unpacking this particular executable and nothing more

[22] Or 0XAH, whichever you prefer. :-)

[23] well, almost; up to two skipped bytes

| | | | |
|---|---|---|---|
| de9c9e: | 0f b6 06 | movzx | eax,BYTE PTR [esi] |
| de9f5d: | 8b 3c 85 91 b9 e1 00 | mov | edi,DWORD PTR [eax+0xe1b991] |
| dea213: | 0f b6 46 01 | movzx | eax,BYTE PTR [esi+1] |
| dea4b0: | 60 | pusha | |
| dea4b1: | e8 34 00 00 00 | call | 0x00dea4ea |
| dea4b6: | 8b 4c 24 0c | mov | ecx,DWORD PTR [esp+12] |
| dea4ba: | 33 c0 | xor | eax,eax |
| dea4bc: | 89 41 04 | mov | DWORD PTR [ecx+4],eax |
| dea4bf: | 89 41 08 | mov | DWORD PTR [ecx+8],eax |
| dea4c2: | 89 41 0c | mov | DWORD PTR [ecx+12],eax |
| dea4c5: | 89 41 10 | mov | DWORD PTR [ecx+16],eax |
| dea4c8: | 89 41 14 | mov | DWORD PTR [ecx+20],eax |
| dea4cb: | c7 41 18 55 01 00 00 | mov | DWORD PTR [ecx+24],0x155 |
| dea4d2: | 8b 81 b0 00 00 00 | mov | eax,DWORD PTR [ecx+176] |
| dea4d8: | 8b b9 9c 00 00 00 | mov | edi,DWORD PTR [ecx+156] |
| dea4de: | 8b 04 87 | mov | eax,DWORD PTR [edi+eax] |
| dea4e1: | 89 81 b8 00 00 00 | mov | DWORD PTR [ecx+184],eax |
| dea4e7: | 33 c0 | xor | eax,eax |
| dea4e9: | c3 | ret | |
| dea4ea: | 64 67 ff 36 00 00 | addr16 | push fs:[0] |
| dea4f0: | 64 67 89 26 00 00 | addr16 | mov fs:[0],esp |
| dea4f6: | 33 db | xor | ebx,ebx |
| dea4f8: | 8f 03 | pop | DWORD PTR [ebx] |
| ......: | 64 67 8f 06 00 00 | addr16 | pop fs:[0] |
| ......: | 83 c4 04 | add | esp,0x4 |
| ......: | 61 | popa | |

Table 6: Goto code

determined by the contents of EAX (which is stored at `[ECX+176]`) and EDI (stored at `[ECX+156]`) registers at the time when the exception occured. Specifically, the new EIP will be equal to `EDI[EAX]`. Finally, the last piece of code will restore the stack and registers once the exception handler returns. However, this "last piece" is not necessarily the one which follows in memory after the exception handler ! Instead, it's at the place where the handler returns. Thus, all "subroutines" that are jumped-to in this way (using this "exception-al goto") need to begin with such short prologue.

Conclusion: This whole code is just an obfuscated way for jumping to a new location. The new location will be `0xe1b991[*(byte*)ESI][*(byte*)(ESI+1)]`.

# 3 Reconstructed code

## 3.1 Third version of DummyKiller

After adding the functionality of replacing the "exception-al goto" by an equivalent piece of code which doesn't use exceptions (and naturally, replacing the prologue of functions) it became apparent that the code works like this:

1. The program "simulates" a CPU which has a very limited set of "commands" ("instructions" would be more appropriate but it would be too easy to confuse with instructions of real CPU). This CPU has a set of 6 registers which we'll call $R_0$–$R_5$ which are stored at 0x`de8736`. The last register ($R_5$) also serves one other purpose – it roughly corresponds to the Z(ero) flag in EFLAGS.
2. ESI is "instruction pointer" – it points to actually processed in instruction in the program. Initially, it points to 0x`e1ba3d`.
3. The simulated CPU also has a stack of (unlimited) length and random-access memory.
4. At 0xe1b991 is a table of "basic commands" indexed by pairs of bytes (first byte selects one subtable, second byte picks certain entry from that subtable). Thus, another useful feature was added to DummyKiller – it's called on each function separately, in order to clean up as much dummy code as possible. The resulting program is in `gen3.c`[24].

## 3.2 Simulated commands

Commands will be described as n-tuples of bytes. First two bytes are always the opcode, the meaning of other bytes varies from command to command. Notational convention: [ABCCCCDD] means that there are four different fields in this instruction – A, B (both one byte long) and dword CCCC followed by word DD. These symbolic names are usually used in the desctiption the that particular instruction. [12ABC] denotes that the instruction starts with bytes 1, 2 followed by any three bytes.

| | |
|---|---|
| [00AAAA] | PUSH Imm32 |
| Pushes (AAAA xor 0x`37195411`) to the stack. | |
| [01AAAA] | PUSH Imm32 |
| Pushes (AAAA + 0x`add01337`) to the stack. | |
| [02A] | PUSH Reg |
| Pushes register (A xor 0x`47`) to the stack. | |
| [03A] | POP Reg |
| Pops register (A xor 0x`66`) from the stack. | |
| [04A] | AdjustESP |
| Removes (A xor 0x`45`) bytes from stack. | |

[24] Boasting: after running this program, a new executable `0x90-2.exe` will be created, which should be functionally equivalent to the original, and moreover, it'll be Win98 compatible which the original wasn't `:-)`

| [10(?)] | Does not work(?) |
|---|---|
| Attempts to add two topmost values on the stack, removes them from the stack and pushes back the result. Uses self-modifying code and shares big part of code with following two commands. Due to some strange stack manipulations performed by the common part, this function does not seem to work. | |

| [11(?)] | Does not work(?) |
|---|---|
| Attempts to xor two topmost values on the stack, removes them from the stack and pushes the result. Uses self-modifying code and shares big part of code with previous and next commands. Due to some strange stack manipulations performed by the common part, this function does not seem to work. In fact, the pointer to the handler in the table of commands is one byte off `:-)` | |

| [12(?)] | Does not work(?) |
|---|---|
| Attempts to subtract two topmost values on the stack, removes them from the stack and pushes the result. Uses self-modifying code and shares big part of code with previous two commands. Due to some strange stack manipulations performed by the common part, this function does not seem to work. | |

| [13ABC] | XOR Mem, Reg |
|---|---|
| Depending on the value of B, xor-s the byte(0)/word(1)/doubleword(2) at location pointed to by register $R_C$ by lowest byte/word/doubleword of register $R_A$. | |

| [14ABBBB] | ADD Reg32, Imm32 |
|---|---|
| Adds BBBB to $R_{B-3}$. If the result is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [15ABBBB] | SUB Reg32, Imm32 |
|---|---|
| Subtracts BBBB to $R_{B-2}$. If the result is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [16ABBBB] | AND Reg32, Imm32 |
|---|---|
| And-s $R_{B-5}$ with BBBB. If the result is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [17ABBBB] | OR Reg32, Imm32 |
|---|---|
| Or-s $R_{B-4}$ with BBBB. If the result is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [18ABBBB] | XOR Reg32, Imm32 |
|---|---|
| Xor-s $R_B$ with BBBB. If the result is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [19ABBBB] | ADD Reg32, Reg32 |
|---|---|
| Adds contents of $R_{B-3}$ to contents of $R_{A-1}$. If the result is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [1aABBBB] | CMP Reg32, Reg32 |
|---|---|
| Compares contents of $R_{B-1}$ to contents of $R_{A-2}$. If the result is non-zero (i.e. inequality), $R_5$ is set to 1, otherwise to 0. | |

| [20] | Finish |
|---|---|
| This function jumps to the address pointed to by doubleword at 0x`e1bc73` xor-ed with 'HAX0'. As we know from part 2.5, the value stored at this location is 0x`de872f` xor-ed with 'HAX0'. Therefore, these two xor-s cancel out and this function jumps to 0x`de872f`. At that location, there is a short routine which just calls ExitProcess API function with argument 0. The pedantic reader probably noticed that we have no guarantee yet, that the contents of 0x`e1bc73` won't change during the execution. That's true; but as we will see later, this "conjecture" about the behaviour of this function is actually correct. | |

| [21AAAA] | API |
|---|---|
| Calls API function whose handler is at location AAAA+0x`fea731de`. Returned value is saved to $R_0$, if the return value is non-zero, $R_5$ is set to 1, otherwise to 0. | |

| [22AAAAB] | MOV Reg32, Imm32 |
|---|---|
| Sets $R_B$ to AAAA+0x`aefde04`. | |

| [23A] | DEC Reg32 |
|---|---|
| Decrements $R_A$, if it becomes zero, sets $R_5$ to 0, otherwise to 1. | |

| [24A] | INC Reg32 |
|---|---|
| Increments $R_A$, if it becomes zero, sets $R_5$ to 0, otherwise to 1. | |

| [25A] | ZERO Reg32 |
|---|---|
| Sets $R_A$ and $R_5$ to zero. | |

| [26ABB] | MemChr |
|---|---|
| Searches BB bytes of memory starting at location pointed to by $R_0$ for byte with value A. If found, $R_0$ will be set to point to it, otherwise $R_5$ is set to 0. Interestingly enough, $R_5$ is NOT set to 1 if the byte was found. | |

| [27] | INT 3 |
|---|---|
| Calls INT 3. | |

| [28AB] | Mov Reg32, Mem32 |
|---|---|
| Register $R_B$ will be set to the value of doubleword at location $R_A$. | |

| [29A] | BSWAP |
|---|---|
| Real register EAX will be set to the bswapped (endian-reversed) contents of register $R_A$. | |

| [2aAB] | Mov Reg8, Mem32 |
|---|---|
| Register $R_B$ will be set to the value of byte at location $R_A$. | |

| [2bAB] | Mov Reg16, Mem32 |
|---|---|
| Register $R_B$ will be set to the value of word at location $R_A$. | |

| [30AAAA] | StackCmpJe |
|---|---|
| Compares two topmost doublewords on the stack, if they are equal, instrution pointer ESI is set to AAAA-0x`31337`, otherwise its unchanged. Those two doublewords are then removed from the stack. | |

| [40AAAA] | JMP Immed32 |
|---|---|
| Sets instruction pointer ESI to AAAA. | |

| [41AAAA] | JNZ Immed32 |
|---|---|
| Sets instruction pointer ESI to AAAA+0x`e1ba3e` if $R_5$ is non-zero . The "mysterious" added number is just the address of the beginning of the program, plus 1. | |

| [42AAAA] | JZ Immed32 |
|---|---|
| Sets instruction pointer ESI to AAAA+0x`e1ba41` if $R_5$ is zero. The "mysterious" added number is just the address of the beginning of the program, plus 4. | |

| | |
|---|---|
| [50AAAA] | CALL Immed32 |
| Saves actual instruction pointer (increased by the length of current command) and proceeds to execute at AAAA. | |

| | |
|---|---|
| [51] | RET |
| Restores instruction pointer from stack; used for returning from a call. | |

### 3.3 Final version of DummyKiller

Now, we are (almost) ready to analyse the code. The final part – a disassembler for the simulated CPU was added to the our little proggie and the program was disassembled. Almost `:-(`. Although the first few instructions looked OK, the disassembling stopped very soon because the rest of the code was encrypted using a simple XOR cipher executed on the simulated CPU (the addresses are relative to the beginning of the code):

```
000: R3 = 020e
007: PUSH e1ba65
00d: POP R0
010: R2 = 0053
017: XOR [R0], BYTE(R2)
01c: R0++
01f: R3--
022: JNZ 0017
```

This loop xor-s 0x`20e` bytes beginning at 0x`e1ba65` (which corresponds to the relative address 0028) by 0x`53`. Okay, after adding one trivial cycle to the DummyKiller, we were finally able to look at the code.

| | | |
|---|---|---|
| 028: | `R1 = 5cc80e31` | |
| 02f: | `API3` | GetCommandLine |
| 035: | `MemChr(R0, '\x20', 0x255)` | Maybe this should have been 255 decimal? |
| 03a: | `JZ 0132` | |
| 040: | `R2 = [R0]` | |
| 044: | `R2 += 1d9bdc45` | |
| 04b: | `R1 += 74519745` | |
| 052: | `R2 -= ad45dfe2` | |
| 059: | `R1 += deadbeef` | |
| 060: | `R2 += 68656c6c` | hell |
| 067: | `R1 -= 17854165` | |
| 06e: | `R2 -= 41776169` | Awai |
| 075: | `R1 += 73686f77` | show |
| 07c: | `R2 += 69747320` | its␣ |
| 083: | `R1 -= 206e6f20` | ␣no␣ |
| 08a: | `R2 += 64726976` | driv |
| 091: | `R1 += 6d657263` | merc |
| 098: | `R2 -= 6e757473` | nuts |
| 09f: | `R1 -= 79212121` | y!!! |
| 0a6: | `R2 -= 65683f21` | eh?! |
| 0ad: | `R2 &= dfffffff` | |
| 0b4: | `PUSH R2` | |
| 0b7: | `PUSH R1` | |
| 0ba: | `StackCmpJe 00c9` | |
| 0c0: | `R3 = 0` | |
| 0c3: | `JZ 0132` | |
| ... | ... | ... |
| 132: | `PUSH 0000` | |
| 138: | `PUSH e1bc13` | reference to strings at relative offset 01d6: "Please Authenticate\n" |
| 13e: | `API0` | printf |
| 144: | `ESP += 8` | |
| 147: | `Finish` | |

This part of code is quite simple – it calls GetCommandLine(), finds the first space in it, and loads the first doubleword into $R_2$. Then, it performs a few mysterious calculations with registers $R_1$ and $R_2$ (some of the used constants correspond to readable text – e.g. "show no mercy!!!", etc.). The results of these calculations are then compared and if they are not equal, the execution continues at 0132. There, a call to printf() is made, which displays the string "Please Authenticate\n" and the execution is terminated by a call to ExitProcess(0). Thus, we can calculate the value which must be present in the first doubleword of the command line if we don't want the program to terminate with a request for authentication. There are two possible values – '1D3N' or '1D3n' (because of line 0ad).

| | | |
|---|---|---|
| 0c9: | R0++ | |
| 0cc: | R0 += 0002 | |
| 0d3: | R0++ | |
| 0d6: | R1 = Word [R0] | $R_1$ now contains fifth and sixth byte of commandline |
| 0da: | PUSH R1 | |
| 0dd: | POP R2 | $R_2$ also contains fifth an sixth byte of the commandline |
| 0e0: | PUSH R0 | |
| 0e3: | PUSH d8360d | |
| 0e9: | POP R0 | |
| 0ec: | R0 += 98548 | |
| 0f3: | R0-- | $R_0$=0xe1bb54 |
| 0f6: | XOR [R0], WORD(R2) | Word at 0xe1bb54 is xor-ed by fifth and sixth byte of the commandline |
| 0fb: | POP R0 | |
| 0fe: | R2 = Byte [R0] | $R_2$=commandline[5] (counting from 1) |
| 102: | R0 += 0002 | |
| 109: | R1 = Byte [R0] | $R_1$=commandline[7] |
| 10d: | R2 += R1 | $R_2$ = commandline[5]+commandline[7] |
| 111: | Call 0149 (absolute e1bb86) | offset corresponding to line 117 is saved on stack |
| ... | ... | ... |
| 149: | R1 = 004c | |
| 150: | R1++ | |
| 153: | R1++ | |
| 156: | R1 += 0005 | |
| 15d: | R1-- | |
| 160: | R1 -= 0004 | |
| 167: | R2 -= 005a | |
| 16e: | R1 ?= R2 | |
| 172: | JNZ 0132 | |

This part of the code performs several interesting actions. First, it xor-s a word somewhere in the region which was decrypted by the first xor-loop. Thus, the code (running on the simulated CPU) is even self-modifying! Then, the fifth and seventh byte of the commandline are added and the resulting value must be equal to 0xa8 (again, there is some add/subtract magic performed at lines 149-167). Okay.

| | | |
|---|---|---|
| 178: | R0-- | |
| 17b: | R2 = Byte [R0] | $R_2$=commandline[6] |
| 17f: | R0 += 0002 | |
| 186: | R1 = Byte [R0] | $R_1$=commandline[8] |
| 18a: | R2 += R1 | |
| 18e: | R2++ | |
| 191: | R2 -= 004e | $R_2$=commandline[6]+commandline[8]-0x4d |
| 198: | PUSH e0dd64 | |
| 19e: | POP R0 | |
| 1a1: | R0 += deac | |
| 1a8: | R0++ | $R_0$ = 0xe1bc11; correponds to relative offset 1d3 |
| 1ab: | XOR [R0], BYTE(R2) | |
| 1b0: | R3 = 0049 | |
| 1b7: | PUSH e1bc2a | |
| 1bd: | POP R0 | |
| 1c0: | R2++ | |
| 1c3: | XOR [R0], BYTE(R2) | `Another xor-loop!` |
| 1c8: | R0++ | |
| 1cb: | R3-- | |
| 1ce: | JNZ 01c3 | |

This part is the most interesting – again, it performs some numerical woodoo and xor-s another byte in the code (this time, the byte at location 1d3 is xor-ed by (commandline[6]+commandline[8]-0x4d)). Finally, it xor-s 73 bytes beginning from 0xe1bc2a with the value of (commandline[6]+commandline[8]-0x4c).

Following this part, the code is once again uncomprehensible. After all, its first byte is xor-ed by a constant depending on the input given on commandline, so why should it be comprehensible without correct input? `:-)` Thus, to be able to analyse the code further, we'll need to find the right value...

How long can the next command(s) be? We already know that at the offset 1d6, there is the string "Please ..." and it's highly unlikely that it'll be a part of the code. Therefore, we have just two bytes for the command which should get us to some other place. How many two-byte commands do we have? The answer is simple – just 3. Moreover, we know that the second byte of the command is 0x01, because only the first byte is xor-ed. Thus, the only possible command is "RET" (opcode 0x05, 0x01). In other words, (commandline[6]+commandline[8]-0x4d) = 0x05 xor 0x47 (the original value of the xor-ed byte). Therefore, (commandline[6]+commandline[8]) = 0x8f. Moreover, we also know the constant the data are xor-ed by 0x43. After performing the xor, something interesting appears, so apparently, we are on the right track. Let's analyse the remaining part of the code!

Once the "RET" command is performed, we are back in the main code – namely, at line 117. Unfortunately, the command on this line is garbled as well (because of the xor performed at line 0f6). Let's repeat the analysis we performed in previous paragraph but this time backwards. We've already analysed the code at offset 132, so we'll try to go back from this location and use the fact that opcodes are very small numbers and that commands are at most 7 bytes long. The last four bytes of this region are way too big, so if there is a command, then it must start at offset 12c. Looking at the opcode, it would need to be an unconditional jump to address 0xf2f6dcd7+0xdeadead=0xe1bb84, which sounds quite reasonable. Proceeding in this manner, we'll obtain following piece of code:

| | | |
|---|---|---|
| 11a: | PUSH R3 | |
| 11d: | PUSH e1bc2a | reference to the interesting data we mentioned before |
| 123: | API0 | printf |
| 129: | ESP += 8 | |
| 12c: | JMP 0147 | |

A logical choice for the first command would be "$R_3$=0" (i.e. opcode 0x02, 0x05)[25]. So, we'll assume that this is the intended command and using therefore, we're able to calculate fifth-to-eighth bytes of the commandline – "EGcH".

*Alternatively, we could have brute-forced the few valid opcodes which could have been a part of the first command and see, if the resulting code looks reasonable. But I hate brute-force* `:-)`.

### 3.4 Conclusion

If the binary is executed with a commandline "1D3NEGcH" (and with some others as well), it'll present a message: "`Welcome...\nExploit for it doesn't matter 1.x Courtesy of Nicolas Brulez`".

## 4 Alternative methods

As you have probably noticed, the described method is rather slow and clumsy. There are some faster methods – for example, if owned the IDA disassembler, we could use the its integrated IDC language to write a script which would remove the dummy code and some other irrelevant pieces. On the other hand, it wouldn't be as much fun as analysing a Windows binary on Linux `:-)`.

If we were on Windows machine, there would be a very easy way for passing through the outer envelope – we would just modify the API functions imported by this binary from kernel32.dll/msvcrt.dll in such way that the execution would be stopped once that particular API gets called (of course, not by placing a breakpoint at the beginning of the API function `:-)`; this would be detected by the binary), e.g. by replacing the first instruction by a jump to our piece of code in some unused portion of the library's data space.

If we were brave enough, we could also write our own loader of Windows libraries for Linux (which is quite easy to do), load the library, set all of its memory pages to be unreadable/unwritable and execute it. Once it would perform any memory access, we could decide whether it was an important instruction or a part of the dummy code (e.g. the leading pusha). In the first case, it would be reported, in second it would be silently executed. Naturally, this approach would also require emulation of exceptions (which is not very difficult as we don't need full emulation, just the parts used by this binary) and some API's (again, not a very difficult task for a binary which calls 3 API's alltogether). This way, we would be able to see the interesting parts of the code without intervening blocks of dummy code.

## 5 Answers

1. - PE headers modification
   - Many Dummy code
   - Many exception handlers which measured the elapsed time
   - Multiple layers of encryption
   - Simulated CPU
   - Exceptions used as "goto"'s.
   - Self-modifying simulated(!) code
   - Code execution depending on the input
2. I'm not sure which method of protection does the author refer to but probably it was the simulated CPU. If so, the description can be found in previous paragraphs.

[25] Although it seems unnecessary because $R_3$ was zero when the xor-loop finished

3. A tool was developed, which effectively removes the protection and translates the simulated code into more readable form (on the other hand, it was constructed in rather ad-hoc manner, so it's very far from being a true "unpacker" for this protection).

4. I personally would prefer using IDA (unfortunately, I don't own a copy of this great tool), where one could write an IDC script which would skip the dummy code, perform the decryption and many additional things (or even write a disassembler module for the simulated CPU `:-)` ). From other tools, OllyDbg would probably work as well, because of its plugins architecture (again, one could write a plugin for skipping the dummy pieces of code).

5. Well, as the message says, it's an exploit for ... ah, it doesn't matter :-) If this was a real malicious binary, it could have been e.g. exploit for some vulnerability or, it could be a binary left on a compromised system, in order to attract attention of the forensic analyst and distract him from more important stuff.

6. The binary expects authentication string on the commandline. Only first 8 bytes of the commandline are important. The details are described in previous paragraphs, it was found that e.g. the string "1D3NEGcH" works.

Bonus There are many methods for making the binary harder to analyse. For example, one could use more "intelligent" dummy code (look for the description of Level3 virus from old DOS times and/or for mutation libraries like MtE, or newer KME, etc.), online decryption/encryption of executed pieces of the code (i.e. only the currently executing piece of code would be visible and other pieces would be decrypted by a checksum of the remaining ones), checksumming of parts of the code and using these checksums for decisions, longer pieces of executed code when the debugger/tracer is detected (i.e. do not die immediately, perform some decryption, encryption and then jump to some strange place), etc. I think this should be enough, otherwise Nicolas' Armadillo protector might soon become too complex to remove :-).

## References


[HIEW] Hacker's View, used version 6.11 (the last freeware version)
`http://www.serje.net/sen/`

[PE] Portable Executable
`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp`

[IDA] Interactive Disassembler, `http://www.datarescure.com/`

[OllyDbg] OllyDbg, used version 1.10
`http://home.t-online.de/home/Ollydbg/`

[BFD] Binary File Descriptor library
`http://www.gnu.org/software/binutils/manual/bfd-2.9.1/bfd.html`.


# 6 Final words

Sorry for the poor English and even worse code `:-)`.