Honeynet Project Scan of the Month 33 - Armored Binary Analysis

Joe Stewart <jstewart@lurhq.com>

3rd December 2004

This month's challenge was to reverse-engineer an unknown binary "0x90.exe" which was "found" on a WinXP system.

1 Q&A

1.1 Identify and explain any techniques in the binary that protect it from being analyzed or reverse engineered.

The binary uses an array of techniques to thwart analysis.

1.1.1 PE Header Manipulation

The PE header has been altered to prevent the executable from loading into OllyDbg. This doesn't prevent it from being loaded into IDA, but can be a major stumbling block for someone who uses OllyDbg as their primary platform. However, the changes to the PE header can be reversed with some basic analysis.

```
00000100   50 45 00 00  4C 01 04 00  PE..L...
00000108   63 31 1C 85  44 61 72 74  c1..Dart
00000110   68 50 45 00  E0 00 8F 81  hPE.....
00000118   0B 01 02 19  00 02 00 00  ........
00000120   00 54 04 00  00 00 00 00  .T......
00000128   00 20 00 00  00 10 00 00  . ......
00000130   00 20 00 00  00 00 DE 00  . ......
00000138   00 10 00 00  00 10 00 00  ........
00000140   01 00 00 00  00 00 00 00  ........
00000148   04 00 00 00  00 00 00 00  ........
00000150   00 90 04 00  00 10 00 00  ........
00000158   00 00 00 00  03 00 00 00  ........
```

```
00000160   00 00 10 00  00 20 00 00  ..... ..
00000168   00 00 10 00  00 10 00 00  ........
00000170   DE FF DB AB  DE DD FF DF  ........
```

At the offsets 0x170 and 0x174 above, we can see some suspicious values filling two DWORDs of the IMAGE_OPTIONAL_HEADER structure[**?**]. The first highlighted DWORD is the LoaderFlags value, and the second is the NumberOfRvaAndSizes value.

According to MSDN, the LoaderFlags value is obsolete. OllyDbg apparently does not use on this field either, so this change is either superfluous, or designed to affect a different debugger. This value can be left as is, or it can be set to zero, as it is in modern PE files.

The NumberofRvaAndSizes value indicates how many directory entries there are of type IMAGE_DATA_DIRECTORY. Microsoft has only defined 15 types of directory entries (plus one reserved) for the PE format; this is where the import/export tables and other loader or debugging information may be found. However, all of these entries are optional, so Microsoft defined the NumberOfRvaAndSizes field to indicate the exact count, so a program would not attempt to look for directory entry that was not present. Surprisingly, the Windows loader does not balk at being given a number of entries like 0xDFFFDDDE - but OllyDbg faithfully attempts to account for the extra directory entries, causing the loading of the EXE in OllyDbg to fail with a “Bad or unknown format” error. If this value is changed to 16 (10 00 00 00) in a hex editor, the file can be loaded in OllyDbg, assuming there are no more PE header tricks.

Ah, but that would be too easy, wouldn’t it? Another technique has been employed to prevent loading in OllyDbg, this time in the section headers:

```
000001F8   43 4F 44 45  00 00 00 00  CODE....
00000200   00 10 00 00  00 10 00 00  ........
00000208   00 10 00 00  00 10 00 00  ........
00000210   00 00 00 00  00 00 00 00  ........
00000218   00 00 00 00  20 00 00 E0  .... ...
00000220   44 41 54 41  00 00 00 00  DATA....
00000228   00 50 04 00  00 20 00 00  .P... ..
00000230   00 50 04 00  00 20 00 00  .P... ..
00000238   00 00 00 00  00 00 00 00  ........
00000240   00 00 00 00  40 00 00 C0  ....@...
00000248   4E 69 63 6F  6C 61 73 42  NicolasB
00000250   00 10 00 00  00 70 04 00  .....p..
00000258   FF AD EF EF  00 70 04 00  .....p..
00000260   00 00 00 00  00 00 00 00  ........
00000268   00 00 00 00  40 00 00 C0  ....@...
00000270   2E 69 64 61  74 61 00 00  .idata..
00000278   00 10 00 00  00 80 04 00  ........
00000280   00 10 00 00  00 70 04 00  .....p..
```

```
00000288   00 00 00 00  00 00 00 00  ........
00000290   00 00 00 00  40 00 00 C0  ....@...
```

The highlighed DWORD at offset 0x258 above represents the IMAGE_SECTION_HEADER[**?**] struct member SizeOfRawData of the third section header, “NicolasB”. Older versions would simply try and allocate the full amount of memory requested, leading to resource exhaustion on the analysis machine.The newest version of OllyDbg will complain that the exe “contains too much data”, but will still attempt to load the file without allocating memory for separate sections; instead loading the entire file image as the PE header. It is still possible to run and analyze the file in this state, but probably not desirable. To fix the SizeOfRawData header for the NicolasB section, we can examine the PointerToRawData and SizeOfRawData values for all sections. They are:

Section #	Section Name	PointerToRawData	SizeOfRawData
1	CODE	0x1000	0x1000
2	DATA	0x2000	0x45000
3	NicolasB	0x47000	0xefefadff
4	.idata	0x47000	0x1000

Because the PointerToRawData value for both section 3 and section 4 is the same, we can assert that the proper SizeOfRawData for section 3 should be zero. When this DWORD is edited, the file will now load properly in OllyDbg.

1.1.2 Code Obfuscation

One technique that attempts to foil static analysis is the use of large amounts of useless code to obscure the functional code. When loaded, the first thing we see is largely pointless code:

```
00DE2000  PUSHAD
00DE2001  CALL 0x90-edi.00DE2006
00DE2006  POP EBP
00DE2007  MOV EAX,EBP
00DE2009  SUB EAX,6
00DE200C  SUB EBP,0x90-edi.00DE2006
00DE2012  PUSHAD
00DE2013  MOV AL,BL
00DE2015  PREFIX GS:                       ; Superfluous prefix
00DE2016  PREFIX REP:                      ; Superfluous prefix
00DE2018  MOV ECX,EDX
00DE201A  PREFIX REPNE:                    ; Superfluous prefix
00DE201B  JMP SHORT 0x90-edi.00DE201E
00DE201D  STOS DWORD PTR ES:[EDI]
00DE201E  MOV ESI,ESI
```

```
00DE2020  PREFIX REPNE:                     ; Superfluous prefix
00DE2022  LEA EBX,DWORD PTR DS:[D0E47C6E]
00DE2028  PREFIX GS:                        ; Superfluous prefix
00DE2029  PREFIX REPNE:                     ; Superfluous prefix
00DE202B  MOV AH,51
00DE202E  JMP SHORT 0x90-edi.00DE2031
00DE2030  DAA
00DE2031  MOV AL,CH                         ; Superfluous prefix
00DE2034  PREFIX REP:                       ; Superfluous prefix
```

The presence of the “Superfluous prefix” comments is an indicator that there are useless opcodes in the disassembly. This often is a sign that the code is simply encrypted or compressed, and that the disassembler could not make sense of it. In this case however, the garbage code is executed in many places, but has no effect. Some of the garbage code is simply skipped, as executing it might cause an exception. What we must do then is somehow separate the garbage code from the functional code. Fortunately it has been made relatively easy in this case, as we’ll see below.

This doesn’t mean that the entire section is bogus, however. Note the two PUSHAD instructions above. This instruction takes the values of all of the standard registers and pushes them onto the stack, saving them for later use.

Manually stepping through the file, we see loads of useless register manipulation going on, then finally at virtual address 0xDE2288 we come across something different - a POPAD instruction, followed by some legitimate code:

```
00DE2288  POPAD
00DE2289  PUSHAD
00DE228A  CALL 0x90-edi.00DE22D7
00DE228F  MOV ECX,DWORD PTR SS:[ESP+C]
00DE2293  ADD DWORD PTR DS:[ECX+B8],2
00DE229A  XOR EAX,EAX
00DE229C  MOV DWORD PTR DS:[ECX+4],EAX
00DE229F  MOV DWORD PTR DS:[ECX+8],EAX
00DE22A2  MOV DWORD PTR DS:[ECX+C],EAX
00DE22A5  MOV DWORD PTR DS:[ECX+10],EAX
00DE22A8  MOV DWORD PTR DS:[ECX+14],EAX
00DE22AB  MOV DWORD PTR DS:[ECX+18],155
00DE22B2  MOV EAX,DWORD PTR DS:[ECX+B0]
00DE22B8  PUSH EAX
00DE22B9  CPUID
00DE22BB  RDTSC
00DE22BD  SUB EAX,DWORD PTR SS:[ESP]
00DE22C0  ADD ESP,4
00DE22C3  CMP EAX,0E0000
00DE22C8  JA SHORT 0x90-edi.00DE22CD
```

```
00DE22CA  XOR EAX,EAX
00DE22CC  RETN
00DE22CD  ADD DWORD PTR DS:[ECX+B8],63
00DE22D4  XOR EAX,EAX
00DE22D6  RETN
00DE22D7  XOR EAX,EAX
00DE22D9  PUSH DWORD PTR FS:[EAX]
00DE22DC  MOV DWORD PTR FS:[EAX],ESP
00DE22DF  CPUID
00DE22E1  RDTSC
00DE22E3  XOR EBX,EBX
00DE22E5  POP DWORD PTR DS:[EBX]
00DE22E7  POP DWORD PTR FS:[0]
00DE22ED  ADD ESP,4
```

The POPAD instruction restores the last set of "good" register values from the stack. When this happens, the program is executing functional code. Therefore we should pay special attention to code that is in-between POPAD/PUSHAD pairs. Also, we should examine code that immediately follows the PUSHAD, since just because the registers were stored does not automatically mean that useless code follows. In the above example, there is an important section of code following the PUSHAD, which leads us to our next analysis evasion technique.

1.1.3 Structured Exception Handler Redirection

Normally exception handlers are used to allow a program to gracefully recover from what would be fatal errors. However, packing programs frequently set up exception handlers and then force an exception in order to make it difficult for static disassemblers to follow the program flow. The 0x90.exe binary goes a step further and actively manipulates the return address of the exception handler to further obfuscate the re-entry point.

This type of redirection may fool static disassemblers, but it doesn't prevent someone from using a tracer to step through the code and log each instruction as it is executed. Tracing gives the reverse-engineer a powerful tool to look past the obfuscation and redirection tricks described in the past two sections. However, the author of 0x90.exe has another trick in store for us in case we decide to go this route.

1.1.4 Trace/Emulation Detection

The code sample above does not run in a linear fashion. Immediately following the initial POPAD/PUSHAD, at 0xDE228A the execution jumps to 0xDE22D7:

```
00DE22D7  XOR EAX,EAX
00DE22D9  PUSH DWORD PTR FS:[EAX]
00DE22DC  MOV DWORD PTR FS:[EAX],ESP
```

This code sets up the exception handler to be 0xDE228F, by virtue of the fact that it was pushed onto the stack during our call to 0xDE22D7. This is followed by two more uncommon instructions:

```
00DE22DF  CPUID
00DE22E1  RDTSC
```

The CPUID instruction is used to return information about the vendor and type of CPU in the system[**?**]. This does not make very much sense by itself until you see the RDTSC command below it. RDTSC returns a 64-bit value in EAX and EDX which show the count of clock cycles elapsed since the system was booted[**?**]. This command is at the core of an anti-tracing technique. Basically the idea is to determine the number of clock cycles used in the CPUID command. If the number is too great, it is an indicator that the program is being traced or possibly emulated. The reason the CPUID appears above the RDTSC instead of the other way around is so that the instruction can be loaded into the CPU's cache, ensuring that latency involved in moving the instruction from memory to the CPU will not trigger false positives on the detection. Normally after saving the result of the RDTSC command, CPUID would be called again, and then RDTSC would be called a final time and the earlier 64-bit result subtracted from the current result to give the total number of clock cycles elapsed.

In this program, however, the CPUID instruction is unneccessary, because immediately following the initial RDTSC setup, we see:

```
00DE22E3  XOR EBX,EBX
00DE22E5  POP DWORD PTR DS:[EBX]
```

Since EBX is zeroed by the XOR instruction, the pointer dereference on the following line will trigger a memory-access violation exception, forcing us to jump to the exception handler. However, the Windows kernel must execute a good deal of code before jumping to our provided exception handler address, and that code may be slightly different on different Windows platforms, so the CPUID instruction count is made insignificant. The only thing it helped in this case was to alert us to the fact that an anti-tracing mechanism was about to be used.

Looking at the exception handler, we see:

```
00DE228F  MOV ECX,DWORD PTR SS:[ESP+C]
00DE2293  ADD DWORD PTR DS:[ECX+B8],2
```

This sets the return from the exception handler to be 0xDE22E7, the instruction immediately following the POP [EBX] that caused the exception. Continuing on:

```
00DE229A  XOR EAX,EAX
00DE229C  MOV DWORD PTR DS:[ECX+4],EAX
```

```
00DE229F  MOV DWORD PTR DS:[ECX+8],EAX
00DE22A2  MOV DWORD PTR DS:[ECX+C],EAX
00DE22A5  MOV DWORD PTR DS:[ECX+10],EAX
00DE22A8  MOV DWORD PTR DS:[ECX+14],EAX
00DE22AB  MOV DWORD PTR DS:[ECX+18],155
```

This code zeroes out a structure on the stack except for the last DWORD member, which is set to 0x155. Once past this, we come to the second part of our anti-tracing trick:

```
00DE22B2  MOV EAX,DWORD PTR DS:[ECX+B0]
00DE22B8  PUSH EAX
```

Above we see the lower 32-bit return value of the previous RDTSC instruction is loaded onto the stack. Finally, we have the second set of CPUID/RDTSC instructions:

```
00DE22B9  CPUID
00DE22BB  RDTSC
```

The count is subtracted and removed from the stack:

```
00DE22BD  SUB EAX,DWORD PTR SS:[ESP]
00DE22C0  ADD ESP,4
```

The resulting elapsed clock-cycle count is compared against the value 0xE0000:

```
00DE22C3  CMP EAX,0E0000
```

If the value exceeds that (and it will, if we are slowing down execution through tracing or emulation) then execution jumps to 0xDE22CD:

```
00DE22C8  JA SHORT 0x90-edi.00DE22CD
00DE22CA  XOR EAX,EAX
00DE22CC  RETN
```

Tracing has been detected. As a result, 0x63 is added to the DWORD referenced by ECX+B8, which “should” be our exception handler return address:

```
00DE22CD  ADD DWORD PTR DS:[ECX+B8],63
00DE22D4  XOR EAX,EAX
00DE22D6  RETN
```

This is probably designed to redirect the flow of the program due to our debugging activity. But wait! The CPUID instruction at 0xDE22B9 overwrote the value in ECX and it was never restored. So in most cases, this code will simply cause an unhandled exception to occur and the program will terminate. Either way it means a simple tracer will be thwarted, but since the exception happens right at the end of the anti-tracing check, we can quickly spot what happened. If the program had worked as the author probably intended, we could have been redirected for an unspecified amount of time before finally dumping us from the program; making it harder to tell an anti-tracing mechanism was used, and where it was located. This is a fortunate bug for us.

Using the information we have located in the SEH subroutine, we can see that the critical check is the CMP EAX,0E0000 at 0xDE22C3. If we set a breakpoint on that command or the resulting JA command every time we see it, we can change the program flow or change EAX to pretend like we are running at a normal (untraced) speed.

At some point in the program, the anti-tracing RDTSC checks are no longer employed. When this happens, we can feel free to use OllyDbg's run trace log to help us get past the obfuscation and find the functional code. We do this by selecting “Debug->Trace into” then, when finished, “View->Run trace”, then right-click on the trace and “Log to file”, choosing “Write gathered data”. The run trace output looks like this:

```
00DEF94D Main  CALL KERNEL32.GetCommandLineA
GetCommandLi.. MOV EAX,DWORD PTR DS:[77EE0694] ; EAX=00132268
77E871D1 Main  RETN
00DEF952 Main  PUSHAD
```

Just saw a PUSHAD, so the code that follows is likely garbage. A closer look tells us that it is indeed just simple register-manipulation with no real purpose but to take up space:

```
00DEF953 Main  LEA EDI,DWORD PTR DS:[8BAC3816] ; EDI=8BAC3816
00DEF959 Main  MOV ECX,EDI                     ; ECX=8BAC3816
00DEF95B Main  JMP SHORT 0x90.00DEF95E
00DEF95E Main  LEA ESI,DWORD PTR DS:[76122DC]  ; ESI=076122DC
00DEF964 Main  MOV CH,1C                       ; ECX=8BAC1C16
00DEF967 Main  JMP SHORT 0x90.00DEF96A
00DEF96A Main  MOV EDX,EDX
00DEF96D Main  MOV ESI,ECX                     ; ESI=8BAC1C16
00DEF970 Main  LEA EDX,DWORD PTR SS:[BFCB6F89] ; EDX=BFCB6F89
00DEF977 Main  MOV DH,CH                       ; EDX=BFCB1C89
00DEF979 Main  MOV BH,AH                       ; EBX=00DE223E
00DEF97B Main  MOV BH,0FB                      ; EBX=00DEFB3E
00DEF97D Main  MOV EDX,7F77EAFD                ; EDX=7F77EAFD
00DEF983 Main  MOV ESI,19DC5FE8                ; ESI=19DC5FE8
00DEF989 Main  PREFIX REPNE:
00DEF98D Main  JMP SHORT 0x90.00DEF991
```

So we need to find a way to look past this code until we reach the next POPAD instruction, where the next functional code will be. But finding the POPAD may be a problem due to the use of superfluous prefixes. When OllyDbg encounters a prefix tacked on to a POPAD, the run trace logs the prefix, but not the POPAD, even though both are executed. So the POPAD ends up not being in the run trace file.

Fortunately, the run trace has a feature that helps us out here - the logging of registers. Since a POPAD instruction forces a reload of all registers at once from the stack, we can simply look for a line in our run trace where all registers change at the same time. This tells us a POPAD was executed, even if we can't see it:

```
00DEFC11 Main  PREFIX REPNE:  ; EAX=00132268, ECX= 00E1BC72,
                   EDX=00DE864E, EBX=00DE863E, EBP=00000000,
                   ESI=00E1BA6C, EDI=00DEF952
```

Section 2.1.2 contains a Perl script which can parse OllyDbg run trace files and output the functional code of the program for static analysis.

1.1.5 Import Table JMP Bypassing

A favorite trick of reverse-engineers is setting breakpoints on API calls. In a normal program, most of the work is done by the API, so figuring out the parameters to those calls is essential in understanding the target program. The 0x90.exe program has a few API calls in its import table, so the first reaction of most reverse-engineers might be to set breakpoints on those calls. After the program is loaded, the import addresses are "thunked" into a jump table. Since most of the code is obfuscated and searching for calls to this table yields no results, it is safe to assume that the program must be somehow encrypted as well. It might also seem safe to assume we can just put our breakpoint on the jump table address, catching the calls as they transfer flow to the API via the jump table. We would be wrong.

The jump table looks like this:

```
00DE1001  JMP DWORD PTR DS:[<&msvcrt.printf>]
00DE1007  JMP DWORD PTR DS:[<&KERNEL32.GetTickCount>]
00DE100D  JMP DWORD PTR DS:[<&KERNEL32.GetCommandLineA>]
00DE1013  JMP DWORD PTR DS:[<&KERNEL32.ExitProcess>]
```

The actual opcodes for the first JMP to msvcrt.printf are:

```
FF 25 54 80 E2 00
```

The JMP itself is FF 25, and the remaining four bytes are the location where the loader imported the actual address of msvcrt.printf. Instead of the normal convention of CALL->JMP->API, the author of 0x90.exe uses the jump table address, adds two bytes to skip the JMP to retrieve the pointer address, which he dereferences to find the actual virtual address of the API call, and then calls it directly, rendering any breakpoint on the jump table useless.

“Aha!” you might say, “I can simply skip to the API call’s virtual address myself, and set a breakpoint there!” Well, you might be able to. Unless the author has something else up his sleeve.

1.1.6 API Call Breakpoint Checking

When you set a software breakpoint, what you are really doing is replacing the code in memory with an “INT 3” instruction; x86 opcode 0xCC. The author must have figured you would try to work around the last trick by setting a breakpoint on the API call virtual address, so, not to be outdone, he checks the first four bytes of code at the virtual address to ensure they do not contain the signature 0xCC opcode. An example of this can be seen below. The first instruction loads the pointer to the jump table address for the API call GetCommandLineA:

```
00DE2950 MOV EAX,<JMP.&KERNEL32.GetCommandLineA>
```

Skip the 2-byte JMP prefix to get to the import table pointer:

```
00DE2C2C MOV EAX,DWORD PTR DS:[EAX+2]
```

Dereference the pointer to get the virtual address of the API call and store it in EDI:

```
00DE2EF1 MOV EAX,DWORD PTR DS:[EAX]
00DE2F5B MOV EDI,EAX
```

Prepare to search 4 bytes of memory in REPNE SCAS instruction below:

```
00DE2F5D MOV ECX,4
```

Load 0xCC (INT 3) into EAX, but be sneaky about it:

```
00DE31FA MOV EAX,660
00DE31FF SHR EAX,3
```

Search 4 bytes of memory pointed to by EDI for 0xCC:

```
00DE3202 REPNE SCAS BYTE PTR ES:[EDI]
```

If ECX gets to zero, we didn't find a breakpoint set:

```
00DE3204 TEST ECX,ECX
00DE3206 JE SHORT 0x90.00DE320C
```

Of course, you can feel free to use hardware breakpoints if they are supported on your platform, or simply set your breakpoint 5 bytes into the API call.

1.1.7 Recursive Code Decryption

It has been said that "in order to understand recursion, you must first understand recursion". In this case, if you were to attempt to trace through the code, stopping at each CMP EAX,0E0000 you would notice that with each iteration a preceeding chunk of code was being decrypted, and flow was eventually being passed to that decrypted code section, which then begins to decrypt a preceeding section of code and so forth. So you can't simply set a breakpoint on every CMP EAX,0E0000, because most of them have not been decrypted yet. The act of tracing one loop at a time is too tedious - it could take hours of mind-numbing work, all of which could be lost if the wrong button is pressed. To solve this problem and automate the process of fixing the anti-trace checks and proceeding through the code decryption subroutines, we turn to SHaG's scripting plugin for OllyDbg known as OllyScript[**?**]. With the simple script shown in section 2, the bulk of the work in unpacking the code can be automated.

1.1.8 Others?

There may be more anti-debugging tricks hidden in the binary. However, the ones listed above are the only ones I had to overcome in my analysis - meaning other tricks are either ineffective, or out of the scope of my reverse-engineering technique. That is not to say that the binary has been cracked at this point. Read on.

1.2 Something uncommon has been used to protect the code from beeing reverse engineered, can you identificate what it is and how it works?

There are a lot of uncommon tricks employed as described above. I would say that the trace/emulation detection in section 1.1.4 is probably the most uncommon technique.

1.3 Provide a means to "quickly" analyse this uncommon feature. Which tools are the most suited for analysing such binaries, and why?

The means is provided by the script in section 2.1.1. Countering this kind of technique requires an interactive disassembler that can be scripted. Most people assume this means IDA, but as demonstrated by the supplied script, OllyDbg can be extended in the same manner.

1.4 Identify the purpose (fictitious or not) of the binary.

The binary is a fictional exploit, protected from casual use by means of a password. When run with an incorrect or missing password, the output of the program is simply:

```
Please Authenticate!
```

When the correct password is supplied, the output is:

```
Welcome...
Exploit for it doesn’t matter 1.x Courtesy of Nicolas Brulez
```

1.5 What is the binary waiting from the user? Please detail how you found it.

The binary is waiting for the correct password to be given. Because of the way the password check is done, actually several password variations are possible. However, the closest one I could find to a word in any language (accounting for ’leetspeak) is:

```
1D3ND@dO
```

The password is evaluated in different steps. First, the first four characters are put through some addition/substraction operations, then compared to a value stored in memory. The simplest method to reverse-engineer this is to try a set of repeating characters, such as “AAAA”, then when the evaluation is made, determine the difference between the permutation result and what the program expects it to be. Then, for each character, add or subtract the difference to get the correct ASCII values. This is illustrated below:

Get the first four characters of our commandline argument as a DWORD:

```
00E0F030  MOV EAX,DWORD PTR DS:[EAX]          ; EAX=41414141
00E0F032  MOVZX EDI,BYTE PTR DS:[ESI+3]       ; EDI=00000002
...
00E0F2E0  MOV DWORD PTR DS:[EDI*4+DE8736],EAX
```

After a large block of junk code, we come to the first permutation. In the section below, ESI points to a "key" in a section of the program that has been put through several permutations itself, as to avoid storing a static number. This is not significant enough to show here, since the run trace makes it all for naught anyway. Suffice it to say that the author of the program is a fan of the band Slayer, and the key is derived from permutations on song titles[**?**].

The code below takes our pattern of "AAAA" (0x41414141) and adds to it 0x1D9BDC45 (derived from the Slayer-based key), for a result of 0x5EDD1D86:

```
00E189C0  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=41414141
...
00E18C5F  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=1D9BDC45
...
00E18F2C  ADD EBX,EDI                          ; EBX=5EDD1D86
...
00E19502  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

On to permutation #2. Our previous result now has key value 0xAD45DFE2 subtracted from it, resulting in 0xB1973DA4:

```
00E1A852  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=5EDD1D86
...
00E1AB03  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=AD45DFE2
...
00E1ADD3  SUB EBX,EDI                          ; EBX=B1973DA4
...
00E1B05F  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Permutation #3. 0x68656c6c ("hell") is added to our previous result, for a total of 0x19FCAA10:

```
00E189C0  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=B1973DA4
...
00E18C5F  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=68656C6C
...
00E18F2C  ADD EBX,EDI                          ; EBX=19FCAA10
...
00E19502  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Permutation #4. 0x41776169 ("Awai") is subtracted from our previous answer, resulting in 0xD88548A7:

```
00E1A852  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=19FCAA10
...
00E1AB03  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=41776169
...
00E1ADD3  SUB EBX,EDI                          ; EBX=D88548A7
...
00E1B05F  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Permutation #5. 0x69747320 (“its “) is added to our previous total, resulting in 0x41F9BBC7:

```
00E189C0  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=D88548A7
...
00E18C5F  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=69747320
....
00E18F2C  ADD EBX,EDI                          ; EBX=41F9BBC7
...
00E19502  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Permutation #6. 0x64726976 (“driv”) is added to our previous total, resulting in 0xA66C253D:

```
00E189C0  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=41F9BBC7
...
00E18C5F  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=64726976
...
00E18F2C  ADD EBX,EDI                          ; EBX=A66C253D
...
00E19502  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Permutation #7. 0x6E757473 (“nuts”) is subtracted from our previous answer, resulting in 0x37F6B0CA:

```
00E1A852  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=A66C253D
...
00E1AB03  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=6E757473
...
00E1ADD3  SUB EBX,EDI                          ; EBX=37F6B0CA
...
00E1B05F  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Permutation #8. 0x65683F21 (“eh?!”) is subtracted from our previous answer, resulting in 0xD28E71A9:

```
00E1A852  MOV EBX,DWORD PTR DS:[EAX*4+DE8736]  ; EBX=37F6B0CA
...
00E1AB03  MOV EDI,DWORD PTR DS:[ESI+3]         ; EDI=65683F21
...
00E1ADD3  SUB EBX,EDI                          ; EBX=D28E71A9
...
00E1B05F  MOV DWORD PTR DS:[EAX*4+DE8736],EBX
```

Retrieve value of EBX and push it onto the stack:

```
00E17BF4  MOV EAX,DWORD PTR DS:[EAX*4+DE8736]  ; EAX=D28E71A9
...
00E17EB7  PUSH EAX
```

Get final key permutation and push it onto the stack:

```
00E17BF4  MOV EAX,DWORD PTR DS:[EAX*4+DE8736]  ; EAX=DF807499
...
00E17EB7  PUSH EAX
```

Compare our result (0xD28E71A9) against key permutation (0xDF807499):

```
00DFF2A3  MOV EAX,DWORD PTR SS:[ESP]           ; EAX=DF807499
...
00DFF51B  CMP EAX,DWORD PTR SS:[ESP+4]
00DFF51F  JNZ 0x90.00E005E5
```

Because all the permutations above are simple addition/subtraction, we can derive the original DWORD the program is looking for by subtracting the difference between each byte of the final comparison values:

```
0xDF - 0xD2 = 13
0x80 - 0x8E = -14
0x74 - 0x71 = 3
0x99 - 0xA9 = -16
```

Adding the results above to the ASCII value of “A” gives us:

```
0x41 + 13 = 0x4E (N)
0x41 - 14 = 0x33 (3)
0x41 + 3  = 0x44 (D)
0x41 - 16 = 0x31 (1)
```

Or, in little-endian, “1D3N”.

The second DWORD of the password is evaluated using permutations as well, but there are important differences. Rather than detail the corresponding ASM code here, I will summarize it, since it could get rather long and hard to follow, plus this isn’t so much of an armoring technique as it is an anti-cracking technique. It could also be arranged in a myriad of ways, making this pretty much a “one-off” technique, that is not guaranteed to ever be used in the exact same way again.

Part of the DWORD is used as an XOR decryption key, to decode the text that will be printed to the user only when the password is valid, ensuring that the string is not detectable in the binary otherwise. Also, the odd and even bytes of the DWORD must equal a certain value when totalled. Finally, an adjustment is made to the SEH return handler based on an offset lookup into a set of overlapping tables of pointers. If you select the wrong table and pointer by having the wrong two bytes in your DWORD, the program flow will jump to a “bad” address at a later time. In this way an immediate “yes/no” check is avoided, because such a check makes it easy for a reverse-engineer to crack a program simply by changing the evaluation result. In the method employed by 0x90.exe’s author, you must choose the right path from many wrong paths.

I found the correct path through intelligent brute-force. Knowing the relationships between the bytes in the final DWORD gave me a subset of memory addresses to choose from. I made a list of valid pointers to try (pointers that seemed to be SEH returns). I then extrapolated the possible ranges of bytes in the DWORD that could land me at one of those addresses. Running this list of arguments using a batch script (see script in section 2.1.3), I was able to find the possible passwords which printed the correct output message. Since the tables of pointers overlap, it is possble that different table+offset pairs can point to the critical “right” path, leading to multiple correct passwords. From the list of passwords I found that worked, “1D3ND@dO” seemed to be the only one that made any sense at all, being “leetspeak” for the Portuguese word “idendado”. This may or may not be the password the author intended, but it works just the same.

1.6 Bonus Question: What techniques or methods can you think of that would make the binary harder to reverse engineer?

This binary was crackable due to use of string evaluations via permutations that could be easily reversed. If the working code of the exploit had been encrypted by a strong symmetric cipher, it would have been impossible to use standard R-E to determine the password. At that point we are only left with brute-force decryption.

2 Scripts and Tools Used

2.1 Scripts

2.1.1 Un-Brulez

```
# Un-Brulez
# (c)2004 Joe Stewart <jstewart@lurhq.com>
# For use in OllyDbg with OllyScript plugin
#
# Run exe until last RDTSC check has passed, then stop
#
var compare
var search
eoe handle_exception
run
#
handle_exception:
cmp eip,DE2000
jb in_code
mov search, eip
sub search, 3C
log "Searching for CMP EAX, 0E0000"
findop search, #3D00000E00#
mov compare,$RESULT
cmp compare,0
je cant_find
bp compare
eob handle_cmp
eoe handle_exception
esti
run
#
handle_cmp:
cmp eip,DE2000
jb in_code
cmp eip,DE86FF
je stop_now
cmp eip,E27000
ja in_system
bc eip
mov eax,1
eoe handle_exception
run
#
cant_find:
```

```
msg "Can’t find CMP EAX,0E0000 (are we done?)"
ret
#
in_code:
msg "EIP is below data segment (are we done?)"
ret
#
stop_now:
mov eax,1
bp DEF94C
ret
#
in_system:
msg "Reached system dll breakpoint"
ret
```

2.1.2 parsetrace.pl

```
#!/usr/bin/perl
#
# parsetrace.pl
# (c)2004 Joe Stewart <jstewart@lurhq.com>
#
# Usage: cat runtrace.txt | ./parsetrace.pl > out.txt
#
my $goodcode = 0;
while (<STDIN>) {
  chomp(my $line = $_);
  $line =~ s/\r//;
  if ($line =~ /EAX=.* ECX=.*, EDX=.*, \
EBX=.*, EBP=.*, ESI=.*, EDI=.*/) {
    $goodcode = 1;
    $line =~ s/.*\;/\t\t\t\t\t\t    ;/;
  } elsif ($line =~ /POPAD/){
    $goodcode = 1;
    $line =~ s/.*\;/\t\t\t\t\t\t    ;/;
  }
  $goodcode = 0 if $line =~ /PUSHAD/;
  $line =~ s/ Main     /  /;
  print "$line\n" if $goodcode;
}
```

2.1.3 bruteforce.pl

```
#!/usr/bin/perl
```

```
#
# bruteforce.pl
# (c)2004 Joe Stewart <jstewart@lurhq.com>
#
# Generate a list of possible passwords to try
# that would actually reach a valid SEH return ptr
# Save output as .bat file and execute on target
#
for my $g (5,4,3,2,1,0) {
  for (0..255) {
    my $b = 0x42;
    my $x = $b ^ $_;
    if ($g == 5) {
      p("B",chr($_),"f") if $x < 2;
    }
    if ($g == 4) {
      p("C",chr($_),"e") if $x < 5;
    }
    if ($g == 3) {
      p("D",chr($_),"d") if $x < 6;
    }
    if ($g == 2) {
      p("E",chr($_),"c") if $x < 18;
    }
    if ($g == 1) {
      p("F",chr($_),"b") if $x < 32;
    }
    if ($g == 0) {
      p("G",chr($_),"a") if $x < 37;
    }
  }
}
sub p {
  my ($a,$b,$c) = @_;
  $d = chr(0x8f - ord($b));
  print "0x90.exe 1D3N$a$b$c$d\n";
}
```

2.2 Tools

- OllyDbg
- OllyScript
- Perl
- StudPE

References

[1] http://msdn.microsoft.com/library/en-us/debug/base/image_optional_header_str.asp

[2] http://msdn.microsoft.com/library/en-us/debug/base/image_section_header_str.asp

[3] http://www.paradicesoftware.com/specs/cpuid/index.htm

[4] http://www.df.lth.se/~john_e/gems/gem0029.html

[5] http://ollyscript.apsvans.com/

[6] http://www.darkelucidation.com/slayer_discography.php