

```
-----  
| Author: ReWolf |  
| e-mail: rewolf@rewolf.pl |  
| www : http://rewolf.pl |  
-----
```

HACKER CHALLENGE 2007 Phase 3 Report

1. Attack Narrative

Executable is partially encrypted and contains some anti-debug and anti-trace stuff. At first I will describe how to remove encryption and patch executable, so we can debug it easily. Next I will show how to calculate password and where it should be placed. Finally I will describe the way of achieving both objectives.

- Anti-debug

In executable we have several anti-debugging, anti-tracing and anti-patching tricks:

1. **IsDebuggerPresent** - one of the basics checks, it's easy to bypass manually or we can use plugin for **OllyDbg** (plugin called **Olly Advanced**)

```
.text:00402CB3 loc_402CB3: ; CODE XREF: .text:loc_402CA7j  
.text:00402CB3 mov     edi, ds:IsDebuggerPresent  
.text:00402CB9 call    edi ; IsDebuggerPresent  
.text:00402CBB test    eax, eax  
.text:00402CBD jz     short loc_402CC5  
.text:00402CBF push   ebx  
.text:00402CC0 call   _if_debug_exit  
.text:00402CC5  
.text:00402CC5 loc_402CC5: ; CODE XREF: .text:00402CBDj
```

2. **EFLAGS register** - setting **TF** (*trap flag*) to cause exception, under debugger we will not get **single step exception** and function will return wrong value.

```
.text:00401B9D mov     [ebp+var_20], offset sub_401B50  
.text:00401BA4 push   [ebp+var_20]  
.text:00401BA7 push   large dword ptr fs:0  
.text:00401BAE mov     large fs:0, esp  
.text:00401BB5 mov     eax, 1  
.text:00401BBA pushf  
.text:00401BBB xor     dword ptr [esp], 101010100b  
.text:00401BC2 popf  
.text:00401BC3 or     al, al
```

3. **Int 2Dh** - this trick is slightly new. It was described by me ;-) at <http://rootkit.com> (original link: <http://rewolf.pl/int.2d.antidebug.and.code.obfuscation.txt>)

```
.text:00401DAD mov [ebp+var_20], offset sub_401D60
.text:00401DB4 push [ebp+var_20]
.text:00401DB7 push large dword ptr fs:0
.text:00401DBE mov large fs:0, esp
.text:00401DC5 mov ebx, 1
.text:00401DCA int 2Dh ; Internal routine for MSDOS (IRET)
.text:00401DCC nop
.text:00401DCD mov [ebp+var_1C], ebx
```

4. **QueryPerformanceCounter** - it is used to measure execution time between two places in the code
5. **GetTickCount** - similar to point 4. In one case **GetTickCount** is used only if **QueryPerformanceFrequency** return 0. Using performance counters is more precisely than tick counts.
6. **Custom techniques:**

- first customization affects **IsDebuggerPresent** and is used to detect if we have patched **IsDebuggerPresent** to always return 0. It modify **IsDebugged** flag in **PEB**, and checks this modification through call to **IsDebuggerPresent**, because **IsDebuggerPresent** should return this flag.

```
.text:00402EC4 mov eax, large fs:30h
.text:00402ECA mov edx, 0FEh
.text:00402ECF mov [eax+2], dl
```

...

```
.text:00403520 call ds:IsDebuggerPresent
.text:00403526 xor eax, 100h
.text:0040352B mov dword_42EBB8, eax
```

- second customization affects **QueryPerformanceCounter** and **GetTickCount** patching. If we have **GetTickCount** patched to return const value, it will be detected because of **Sleep** function.

```
.text:00403515 push 250 ; dwMilliseconds
.text:0040351A call ds:Sleep
```

...

```
.text:00403530 call ds:GetTickCount
.text:00403536 sub eax, [ebp+78h+var_A8]
.text:00403539 cmp eax, 1250
.text:0040353E jg short loc_403547
.text:00403540 cmp eax, 250
.text:00403545 jge short loc_403550
```

- Removing encryption and anti-debug

For easy debugging we should to remove some anti-debugging tricks (not all). **IsDebuggerPresent** is handled by **OllyAdvanced** plug-in. **Trap flag** based trick is handled by **Olly** with proper exception handling (*Options->Debugging options->Exceptions: check Single-step break*). To bypass **int 2Dh** trick we should patch

function at address **0x401D80** to return always **0**:

```
.text:00401D80 sub_401D80 proc near ; CODE XREF: sub_401C20+1Ap
.text:00401D80 ; sub_402160+6p
.text:00401D80 mov eax, 0
.text:00401D85 retn
.text:00401D85 sub_401D80 endp
```

Finally to bypass execution time checks we should patch **QueryPerformanceFrequency** to always return **0**, so program will use only **GetTickCount**. Basically we can patch also **GetTickCount** that it will always return constant value, but then we should set breakpoint at **0x403530** and correct return value manually to simulate that **Sleep(250)**. Now we can easily debug executable without any „strange“ behaviours.

Executable contains three encrypted blocks at:

- **00401EF0-00401FD0**
- **00402D50-00402E30**
- **00402F20-00403000**

I have decrypted it during debug session and dumped decrypted regions to file (**LordPE**, *dump region*). Next I have put that decrypted regions to executable. Already we have to patch decryption routine to avoid processing already decrypted regions. It can be done in two ways (In my solution I've lesser version). We can patch function at address **00401C20** that it will return immediately without doing anything or we can „nop“ three times calls to this function (**00402C0D**, **00402CD4**, **00402D18**). The last thing to fix is memory **checksums**. There are four places in the main function where we have to patch **checksum** calculations (probably it is one **checksum** function but **inlined**):

- **00402C4F**
- **00402C9E**
- **0040350C**
- **0040358B**

Four addresses above points to *conditional jumps* (**jz**). To fix checksum error we should change that jumps to unconditional (**jmp**).

- *calculating password*

Ok, time to calculate password (the most horrible thing in this phase). In phase 1 we have keyfile called *password.txt* and everything was simple. This time password is embedded in *data.txt* file. Basically password checking routine starts at **00402D54**. Function **sub_40A170** reads password from the first line of *data.txt* file. Password have **12 characters** and it is passed to algorithm that calculate **4 characters** from that 12 chars:

```

char password[12];
char passSum[4];

passSum[0] = password[0] + password[4] + password[8];
passSum[1] = password[1] + password[5] + password[9];
passSum[2] = password[2] + password[6] + password[10];
passSum[3] = password[3] + password[7] + password[11];

```

If **password** is correct, **passSum** is equal to „4242“. There is many correct passwords, for example:

```
zzzzzzzz@>@>
```

- reverse engineer the mathematical formula (objective 1)

Location of code that generates value **-59.0079**, I have found in similar way like in phase 1. So I've set breakpoints on all „user“ functions containing **FPU** operations. This time it was six functions:

```

- 00401E20
- 00401EF0
- 00402040
- 00402160
- 00402280
- 00402380

```

This breakpoints were hit few times during processing first two sets of data from *data.txt*, but only two are hit when third set is processed: **00401EF0**, **00401E20**. Function at **00401E20** is unimportant, so we have only one function responsible for searched **mathematical formula**:

```

.text:00401EF0 sub_401EF0 proc near ; DATA XREF: _main+203o
.text:00401EF0 var_C = qword ptr -0Ch
.text:00401EF0 var_4 = dword ptr -4
.text:00401EF0
.text:00401EF0 push ebp
.text:00401EF1 mov ebp, esp
.text:00401EF3 sub esp, 0Ch
.text:00401EF6 push esi
.text:00401EF7 mov esi, ecx
.text:00401EF9 jmp short loc_401F00
.text:00401EFB ; -----
.text:00401EFB mov eax, 0FFFFFF0h
.text:00401F00
.text:00401F00 loc_401F00: ; CODE XREF: sub_401EF0+9j
.text:00401F00 xor eax, eax
.text:00401F02 mov dword ptr [ebp+var_C], eax
.text:00401F05 mov dword ptr [ebp+var_C+4], eax
.text:00401F08 call sub_401B70 ;
.text:00401F0D test eax, eax ; this is anti-debug function
.text:00401F0F jz short loc_401F18 ; (setting trap flag)
.text:00401F11 push 0FFFFFFFFh ; uExitCode
.text:00401F13 call $LN26 ; if debugged then exit
.text:00401F18
.text:00401F18 loc_401F18: ; CODE XREF: sub_401EF0+1Fj

```

```

.text:00401F18 mov ecx, [esi+0D0h] ; \
.text:00401F1E sub ecx, [esi+0C4h] ; \
.text:00401F24 mov eax, [esi+0C0h] ; \
.text:00401F2A add ecx, [esi+0B8h] ; | this calculates var1 in my
.text:00401F30 lea edx, [eax+eax*2] ; | formula:
.text:00401F33 lea eax, [edx+ecx*2] ; | var1 = (3 * p3) +
.text:00401F36 sub eax, [esi+0CCh] ; | (2 * (p1 - p2 + p4)) -
.text:00401F3C mov ecx, [esi+34h] ; / p5 - p7 + p8
.text:00401F3F sub eax, [esi+0BCh] ; /
.text:00401F45 add eax, [esi+0C8h] ; /
.text:00401F4B cmp ecx, 1 ; \
.text:00401F4E mov [ebp+var_4], eax ; > for third set ecx = 1
.text:00401F51 jnz short loc_401F80 ; /
.text:00401F53 fild [ebp+var_4] ; so we are here
.text:00401F56 mov ecx, eax ; \
.text:00401F58 imul ecx, eax ; \
.text:00401F5B fmul ds:dbl_4284C0 ; | this calculates var2 in my
.text:00401F61 fsubr ds:dbl_4284B8 ; | formula:
.text:00401F67 mov [ebp+var_4], ecx ; | var2 = (var1 * var1 * g3) +
.text:00401F6A fild [ebp+var_4] ; | (g2 - var1 * g1) - (p9 * g4)
.text:00401F6D fmul ds:dbl_4284B0 ; |
.text:00401F73 faddp st(1), st ; |
.text:00401F75 fild dword ptr [esi+30h] ; |
.text:00401F78 fmul ds:dbl_4284A8 ; /
.text:00401F7E jmp short loc_401FB0 ; /
.text:00401F80 ;

-----
.text:00401F80
.text:00401F80 loc_401F80: ; \
.text:00401F80 cmp ecx, 2 ; \
.text:00401F83 jnz short loc_401FB5 ; \
.text:00401F85 fild [ebp+var_4] ; |
.text:00401F88 mov edx, eax ; | this code is not executed
.text:00401F8A imul edx, eax ; | during objective 1
.text:00401F8D fmul ds:dbl_4284A0 ; |
.text:00401F93 fsubr ds:dbl_428498 ; |
.text:00401F99 mov [ebp+var_4], edx ; |
.text:00401F9C fild [ebp+var_4] ; |
.text:00401F9F fmul ds:dbl_428490 ; |
.text:00401FA5 faddp st(1), st ; /
.text:00401FA7 fild dword ptr [esi+30h] ; /
.text:00401FAA fmul ds:dbl_428488 ; /
.text:00401FB0
.text:00401FB0 loc_401FB0: ; CODE XREF: sub_401EF0+8Ej
.text:00401FB0 fsubp st(1), st ; \ end of var2 calculations
.text:00401FB2 fstp [ebp+var_C] ; /
.text:00401FB5
.text:00401FB5 loc_401FB5: ; CODE XREF: sub_401EF0+93j
.text:00401FB5 fild dword_42EBB8 ; \
.text:00401FBB mov ecx, esi ; \
.text:00401FBD fdiv [ebp+var_C] ; | final calculations:
.text:00401FC0 fadd dbl_42EBB0 ; / result = (g5 / var2) + g6 - g7;
.text:00401FC6 fsub ds:dbl_428440 ; /
.text:00401FCC fstp qword ptr [esi+98h] ; -> store -59.0079
.text:00401FD2 call sub_401E20
.text:00401FD7 pop esi
.text:00401FD8 mov esp, ebp
.text:00401FDA pop ebp
.text:00401FDB retn
.text:00401FDB sub_401EF0 endp ; sp-analysis failed

```

So final formula looks like this:

var1 = (3 * p3) + (2 * (p1 - p2 + p4)) - p5 - p7 + p8;

```
var2 = (var1 * var1 * g3) + (g2 - var1 * g1) - (p9 * g4);
result = (g5 / var2) + g6 - g7;
```

Unfortunately I have two „typo“ in my submission ;/ About first I've written e-mail but didn't get any reply, and second I've found during writing this report. In my original submission I've missed square of **var1**, and to the final result I've added **g7** instead of subtract.

- patch executable to extend some functionality (objective 2)

This was the easiest part of challenge. I've found upper limit in data section:

```
.rdata:004284E8 db 4284E8 dq 1.9999999e2 ; DATA XREF: _main+50Ar
```

It is referenced from main function:

```
.text:00402F6A fld ds:dbl_4284E8 ; loading upper limit
.text:00402F70 add esp, 4
.text:00402F73 fcomp [ebp+78h+var_30] ; compare
.text:00402F76 fnstsw ax
.text:00402F78 test ah, 5
.text:00402F7B jp short loc_402F8B ; if above then set
.text:00402F7D mov dword ptr [ebp+78h+var_30], 0EB074A77h ; limit to upper limit
.text:00402F84 mov dword ptr [ebp+78h+var_30+4], 4068FFFFh
.text:00402F8B loc_402F8B:
.text:00402F8B push 38h ; size t
```

To remove input limit we should change that *conditional jump (jp)* at **00402F7B** to the *unconditional jump (jmp)*.

2. Time to break

- removing encryption

about **1 hour** (during this stage I've also partially removed checksum checking and anti-debug stuff)

- searching password

2,5 hours (I think the hardest part in this phase)

- reverse engineer the mathematical formula (objective 1)

1,5 hour (searching and reversing, not so hard)

- patch executable to extend some functionality (objective 2)

30 minutes (it was easy)

Developed tools : none

Internet research: none

Overall time : 5,5 hours