Andrea Allievi
Security Researcher

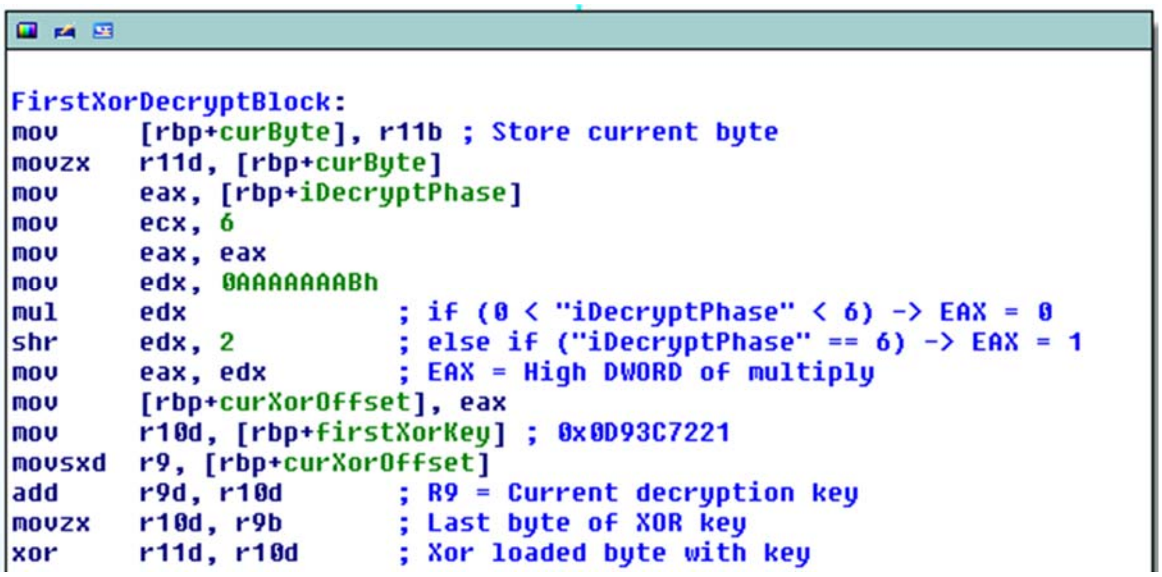
02/01/2014

Expiro: Anatomy of a new 64 bit file Infector

In these last days of December, I have deeply analysed the new Expiro virus, that claims to be the first 64 bit file infector in the wild. This is something new because, till now, all known file infectors target only 32 bit executables. Expiro is instead able to hit 64 bit systems, as well as 32 bit ones.
In this analysis paper we are going to describe the multi-architecture infection process (starting from a 64 bit infected file).

Expiro Infected file – a quick glance

An Expiro forged file does not bring anything new. Expiro adds a section named ".vmpX" (where X is a number from 0 to 9), large about 512 KB, at the end of target PE file, and substitutes about 1248 bytes of code in its entry point. This code has the aim to decrypt Expiro encrypted bytes located in the last section. It stores decrypted bytes in the end of section without exploiting memory allocation APIs. Expiro section indeed has a Virtual size much bigger than its real raw size. The Windows loader, when encounters something like this, creates a section object bigger as PE section virtual size, and maps actual content only in the first part of it. The ending part of section is empty (filled with zeroes), and it is exploited by the virus to write decrypted code. Expiro PE section is divided in three blocks: the **first** block contains original PE entry point code; the **second** contains the current architecture encrypted PE; and finally the **last** block embed another encrypted PE that contains code for the other architecture (x86 versus AMD64). The first decryption algorithm is quite obfuscated but easy to spot out. Entry point infected code knows the real size of Expiro section: it XOR entire section bytes six times with a key (0x21 in our dropper). A clever reader certainly knows that the XOR process repeated a pair number of times yields the same original results. Indeed the last time the Expiro EP code XOR all bytes with the same key increased by one. This fact is showed in the following obfuscated code snippets:



```
FirstXorDecryptBlock:
mov     [rbp+curByte], r11b ; Store current byte
movzx   r11d, [rbp+curByte]
mov     eax, [rbp+iDecryptPhase]
mov     ecx, 6
mov     eax, eax
mov     edx, 0AAAAAAABh
mul     edx                 ; if (0 < "iDecryptPhase" < 6) -> EAX = 0
shr     edx, 2              ; else if ("iDecryptPhase" == 6) -> EAX = 1
mov     eax, edx            ; EAX = High DWORD of multiply
mov     [rbp+curXorOffset], eax
mov     r10d, [rbp+firstXorKey] ; 0x0D93C7221
movsxd  r9, [rbp+curXorOffset]
add     r9d, r10d           ; R9 = Current decryption key
movzx   r10d, r9b           ; Last byte of XOR key
xor     r11d, r10d          ; Xor loaded byte with key
```

Figure 1. Code snipped of obfuscated simple Expiro decryption cycle

Finally, it copies all the decrypted PE sections one by one in the second empty part of target file PE section, performing needed relocations and fixups. Execution control is then returned to "ExpiroMain" function now located in the second part of Expiro infected section. ExpiroMain job is quite simple: first, it searches the Base address of its module (starting from *ExpiroThread* procedure and walking backwards) and of Kernel32 library, gets the infected entry point code size, and then it spawns a new thread (we will call this new thread the main Expiro thread). Finally it copies original Entry point clean bytes stored in the beginning of Expiro section to their real location and jump to original instructions now located in the right place. Infected file execution proceeds as normal. The entire infection logic is executed from now on by spawned thread.

Expiro main thread

Main thread spawned by Expiro code begins its execution resolving its internal Import address table. IAT resolution exploits a classical *GetProcAddress* / *LoadLibrary* method, obfuscated with the usage of encrypted string for its needed API functions. String encryption algorithm is quite simple:

```
// Decrypt an expiro string
LPSTR DecryptExpiroString(STR_STRUCT * pStrStruct) {
    const LPSTR encMap =                                // Encryption map
      "r@YE<@kPR3z0_hhyLKoLzXTumecicQz>l77";
    DWORD mapSize = strlen(encMap);                    // Encryption map size
    BYTE curKey = pStrStruct->xorKey;                  // Current xor Key

    for (int i = 0; i < (int)pStrStruct->size; i++) {
      CHAR chr = pStrStruct->encStr[i];
      DWORD mapPos = (i + 3) % mapSize;                // Get current position in Map
      chr = (chr ^ curKey) ^ encMap[mapPos];           // Decrypt it ...
      pStrStruct->encStr[i] = chr;                     // and store new value
    }

    return (LPSTR)pStrStruct->encStr;
}
```

where STR_STRUCT represent an encrypted string composed as following: `1 WORD String size + 1 BYTE Xor key + encrypted string`.

Expiro thread figures out many functions addresses of the following modules: Kernel32, Advapi32, Ole32, User32, Sfc, Pstorec, Msvcrt, Crypt32, Oleaut32, Shell32.
"SearchExpiroNames" procedure, as the name implies, searches for some predefined names:

- Terms "SERVICE" or "SYSTEM" included in current user name
- Current user name located in target computer name
- Terms "ervice", "systemprofile" included in one of each environment variable

Whether the routine has found one condition above, it probably means that the infected file is running as a System service, and, if so, another different infection routine is used. We will talk about the latter routine afterwards…
Expiro thread code now tries to open one of Expiro Setup mutex (named "kkq-vx_mtxX", where X is an integer value ranging from 28 to 100). If it succeed, setup procedure is suddenly terminated (indeed another infected file is running and performing infection process). Otherwise, "*AdjustMyPrivilegesAndCreateSecDesc*" routine opens the current process token, gets information, obtains its owner and then creates a global Security descriptor with a null DACL, null SACL and infected process user as owner. It then creates and acquires setup mutex and creates an hidden window (class name "kkq-vx"). Finally, infection process begins…

Service infection process

Infection process starts targeting system services. Before investigate the proper routines, we have to summarize here the memory situation of running infected PE. Its memory layout at this stage is the following:

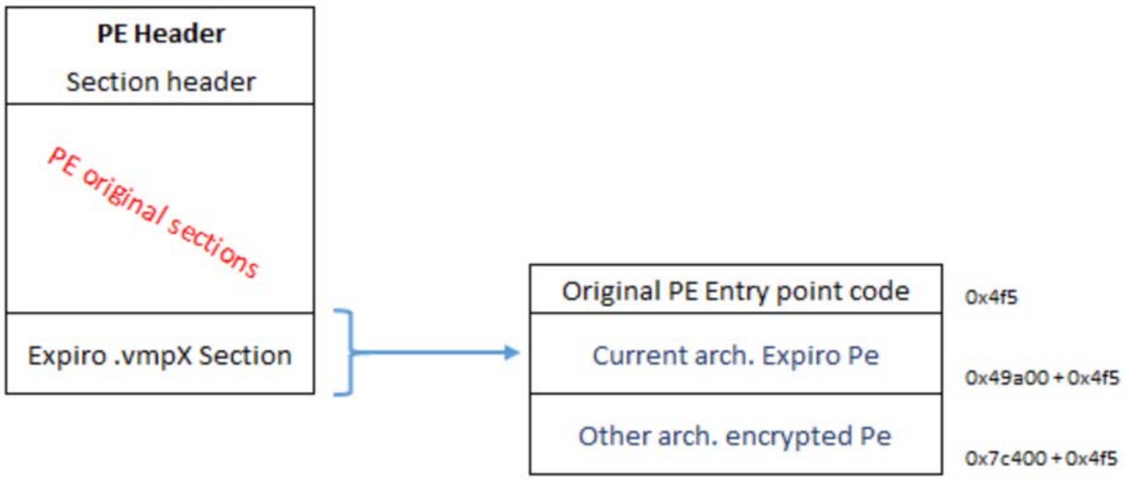


Figure 2. Memory layout of a running Expiro infected process

As the reader can see from the picture above, at this stage there are 2 Expiro modules stored in process decrypted section memory:

- First module is current execution module that is built for current host PE architecture (64 bit in our analysed sample)
- Second module is encrypted and targets executable files built for classical 32 bit architecture.

As we will see in a while, Expiro in this way can hit both 32 bit and 64 bit PE files…
Main thread decrypt second in-memory PE, obtain the entry point code size for other architecture modules, and then calls "*InfectServices*" routine.
The latter routine starts the actual system Services infection: it begins verifying the host Operating system ("*ExpiroVerifyOs*" procedure succeeds if host OS platform is Nt and major version is above or equal to Xp). Then it enumerates all installed services exploiting "*EnumServicesStatus*" API and finally calls *InfectService* procedure for each found service.
InfectService routine opens target service, obtains its configuration and analyses its binary path: if service path doesn't end with ".exe" extension the routine exits, otherwise "InfectFileStub" routine is called to perform actual PE file infection. If service is a Dll library living in the Windows service host process, the latter process will be infected (and not the target service library) and a flag is raised. Expiro has an internal list of most popular AVs services. If the infected service is an AV one, it is marked as DISABLED.
Finally, if the target service is not interactive, is marked as interactive. All modified are applied exploiting *ChangeServiceConfig* API.

Expiro detailed file infection process

Expiro actual file infection is launched by a stub routine: "*InfectFileStub*". This procedure analyses path, trimming beginning and end chars if needed, performs some name filtering and then check OS: if the host Operating system is Xp or less, it loads "SFC.dll" library with the aim to deal with system-protected files. "*SfcIsFileProtected*" API is exploited to determine if target file is protected, and, if so, a call to the function exported at ordinal 5 by Sfc library is made. The protection to target file in this way is disabled.
Now execution control is transferred to "***InfectFile***" main infection routine.
The actual infection process is completely implemented in this large function, and is divided in 4 parts:

- Initialization block
- Analysis block
- Pre-infection block
- Infection block

Initialization block

The initialization block begins with the allocation of 2 global buffers (one for 32 bit architecture and one for 64 bit one) of the same size (Expiro 2 Pe modules + 64 Kbytes for padding and Entry Point code). Target file is opened in a clever manner: if "*CreateFile*" API fails, "*ApplyExpiroSdToFile*" function applies the Security descriptor initialized above by main thread, to target file, and then retries to open it. The 2 global buffers are prepared as shown below:

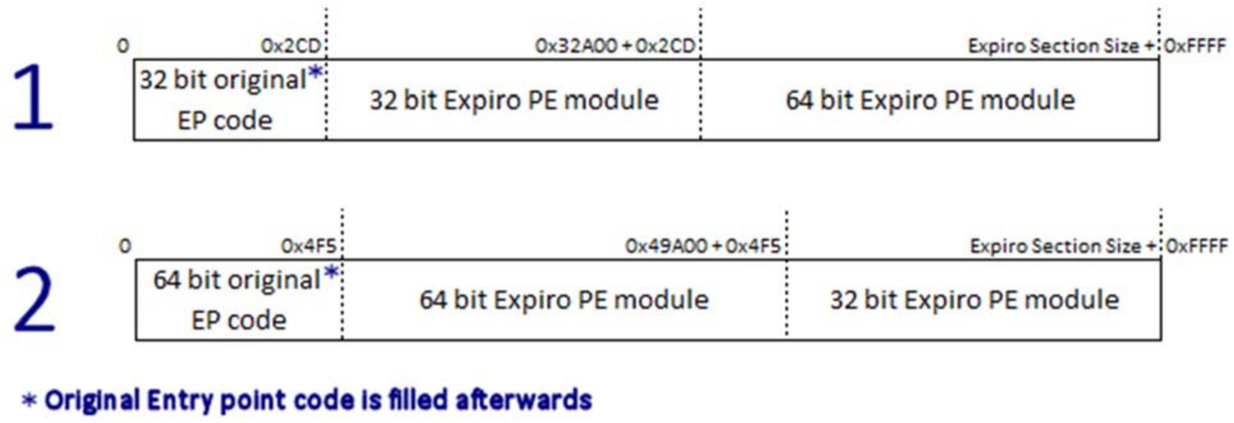


Figure 3. Expiro pre-infection buffers layout

The target file size is obtained, a buffer large as original file size and new Expiro section is allocated. Expiro reads entire file and stores it in the beginning of file buffer content. Now file is pre-analysed:

- If target file architecture is a standard 32 bit, sets first buffer as intermediate, and 0x2CD as Entry point code size
- Otherwise, if target architecture is 64 bit, sets second buffer as intermediate, and 0x4F5 as Entry point code size

Now Expiro code analyses the PE header of target file, with the aim to check whether is not already infected. 3 attributes of Optional header are parsed: If major image version equals to 0xD, major linker version equals to 9 (or 8) and last 4 bits of TimeDateStamp are set to 0, the procedure suddenly exits (the target file is already infected).

Analysis Block

Analysis on Pe header proceeds: Expiro code obtains the target file Import address table RVA, entry point address and some other data. A basilar check is made: **if the IAT is located between Entry point and new EP infected code block** (size 0x2CD or 0x4F5 bytes) the procedure exits. The same is done for Entry point code size: if there is less bytes available than the calculated infected EP code size, the procedure exits. Expiro takes care of an eventual Bound Directory and calculates the new PE header size with the following formula:

```
newPeHdr = BoundImportDir.Size + IMAGE_FIRST_SECTION(pNtHdr) +
           ((numSect+1) * sizeof(IMAGE_SECTION_HEADER))
```

As the reader certainly already knows, bound import directory usually stores its data immediately after section header (see http://msdn.microsoft.com/en-us/magazine/cc301808.aspx, "Binding" section). A good file infector **MUST** take this fact in consideration.
Infection code now analyses each target PE sections. For each section:

1. Calculates END Raw address (Raw Address + Raw Size) and END RVA (Section RVA + Virtual Size)
2. Based on previous results, calculates the biggest section end RVA / end Raw Address

The debug directory and the Security directory are now processed: for each debug entry, Expiro checks its pointer to RAW data. If Raw data is located after last section, Expiro accounts it and stores the debug entry data size in a local variable. If the security directory is present and NOT located at the end of PE, exits from procedure; otherwise it sets security directory size and pointer to 0. In this way the digital signature (now made invalid) is trimmed down. Last steps of the analysis block are related to IAT: if IAT doesn't include "Kernel32.dll" module, the procedure stops. Finally, IMAGE_FILE_RELOCS_STRIPPED flag is added to PE Characteristics.

Pre-Infection block

The author of the analysis calls this block in this way because Expiro file infector modifies target file headers to correctly deal with new added Section.
First, if Bound Import directory is present, its new location is calculated (indeed a new Section header has to be added), its data moved to the right position, and its corresponding RVA and size attributes updated in PE header. A new Section header is inserted in the PE header: Expiro calculates the aligned size of original target file and set resulting value as the new RAW address of section; then compiles each field of section header (SizeOfRawData = Expiro section aligned size; VirtualAddress = last section end VA, Section characteristics = Read, write and execute) and copies section name (".vmpX"). Finally, it updates 2 fields of Nt headers: *NumberOfSection* and *SizeOfImage*. As the reader already knows, the "*SizeOfImage*" field is a multiple of *SectionAligment* and indicates the size of loaded executable in virtual memory. Indeed it is filled with the following formula: `Expiro Section RVA + Section aligned Virtual Size`.
At this stage Expiro code calculates new encryption values (they are different from originals). For both PEs located in intermediate buffer:

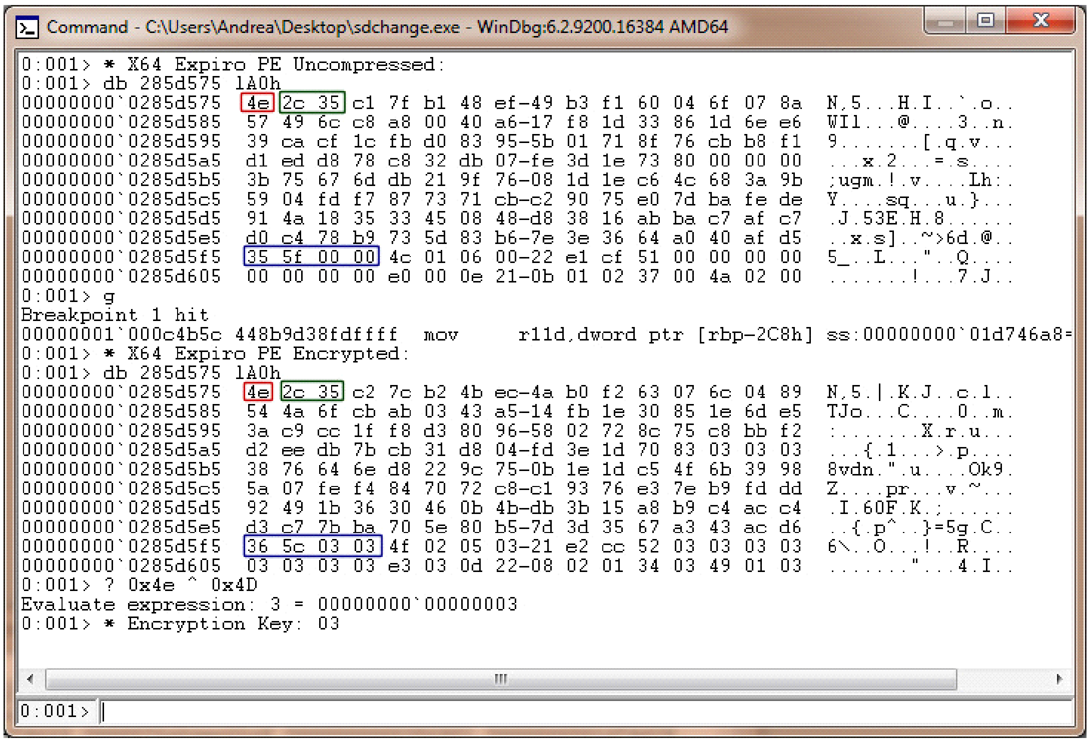
- The Pe module size is shifted right by 12 bit. The resulting byte is placed in the second byte of PE.
- The remaining 12 Less significant bit are shifted right by 9 position and placed in third byte of PE
- Gets a random value exploiting "rand()" API, divides it by 0xFD and gets remainder; adds 1 and sets it as a new XOR Encryption key for current module; xor it with 0x4D value (hex representation of 'M' character), and stores in the first byte of PE.
- Stores original PE entry point virtual address at offset 0x41 of current PE
- Fill Dos Header with random values (from offset 0x03 to 0x3C). Do the same for all padding bytes between Dos Header and Nt headers (including 2 bytes of Nt Headers signature, and excluding the DWORD value located at offset 0x41). In this way Expiro would like to try to bypass some AV protections.

The last step concludes Pre-Infection block.

Infection block

Infection block starts with the filling of the intermediate buffer allocated in the Initialization block. Target original entry point code is copied in the beginning of intermediate buffer. The latter buffer is now completely formed (but not encrypted), and is directly copied in target file mapped memory (new Expiro section RAW address). All needed data from Expiro current architecture PE sections are obtained, then the code starts to encrypt both Expiro PEs located in target file mapped memory: with the values obtained from Pre-Infection block, it xor PE bytes starting from forth to last. Each Expiro PE has its own calculated encryption key stored xored (with 0x4D) in the first byte. Target file Pe header is again modified:

- "SizeOfStackReserve", "SizeOfHeapReserve" values are increased (by Expiro PE values)
- "MajorImageVersion" is set to 0xD, "MajorLinkerVersion" to 9, "MinorImageVersion" to 0x1C, and last 4 bits of "TimeDateStamp" are zeroed out. This fact represent the Expiro signature (target file will not be re-infected)

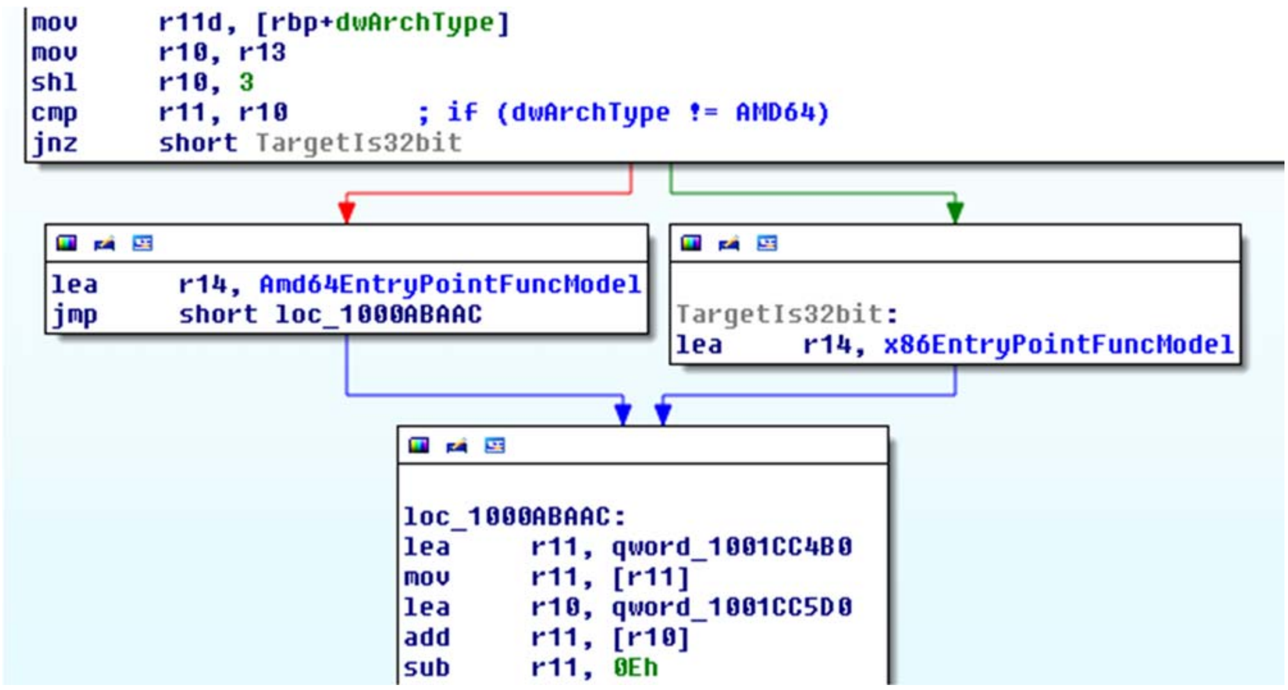

Command - C:\Users\Andrea\Desktop\sdchange.exe - WinDbg:6.2.9200.16384 AMD64

```
0:001> * X64 Expiro PE Uncompressed:
0:001> db 285d575 1A0h
00000000`0285d575  4e 2c 35 c1 7f b1 48 ef-49 b3 f1 60 04 6f 07 8a  N,5...H.I..`.o..
00000000`0285d585  57 49 6c c8 a8 00 40 a6-17 f8 1d 33 86 1d 6e e6  WIl...@....3..n.
00000000`0285d595  39 ca cf 1c fb d0 83 95-5b 01 71 8f 76 cb b8 f1  9.......[.q.v...
00000000`0285d5a5  d1 ed d8 78 c8 32 db 07-fe 3d 1e 73 80 00 00 00  ...x.2...=.s....
00000000`0285d5b5  3b 75 67 6d db 21 9f 76-08 1d 1e c6 4c 68 3a 9b  ;ugm.!.v....Lh:.
00000000`0285d5c5  59 04 fd f7 87 73 71 cb-c2 90 75 e0 7d ba fe de  Y....sq...u.}...
00000000`0285d5d5  91 4a 18 35 33 45 08 48-d8 38 16 ab ba c7 af c7  .J.53E.H.8......
00000000`0285d5e5  d0 c4 78 b9 73 5d 83 b6-7e 3e 36 64 a0 40 af d5  ..x.s]..~>6d.@..
00000000`0285d5f5  35 5f 00 00 4c 01 06 00-22 e1 cf 51 00 00 00 00  5_..L...."..Q....
00000000`0285d605  00 00 00 00 e0 00 0e 21-0b 01 02 37 00 4a 02 00  .......!...7.J..
0:001> g
Breakpoint 1 hit
00000001`000c4b5c 448b9d38fdffff  mov     r11d,dword ptr [rbp-2C8h] ss:00000000`01d746a8=
0:001> * X64 Expiro PE Encrypted:
0:001> db 285d575 1A0h
00000000`0285d575  4e 2c 35 c2 7c b2 4b ec-4a b0 f2 63 07 6c 04 89  N,5.|.K.J..c.l..
00000000`0285d585  54 4a 6f cb ab 03 43 a5-14 fb 1e 30 85 1e 6d e5  TJo...C....0..m.
00000000`0285d595  3a c9 cc 1f f8 d3 80 96-58 02 72 8c 75 c8 bb f2  :........X.r.u...
00000000`0285d5a5  d2 ee db 7b cb 31 d8 04-fd 3e 1d 70 83 03 03 03  ...{.1...>.p....
00000000`0285d5b5  38 76 64 6e d8 22 9c 75-0b 1e 1d c5 4f 6b 39 98  8vdn.".u....Ok9.
00000000`0285d5c5  5a 07 fe f4 84 70 72 c8-c1 93 76 e3 7e b9 fd dd  Z....pr...v.~...
00000000`0285d5d5  92 49 1b 36 30 46 0b 4b-db 3b 15 a8 b9 c4 ac c4  .I.60F.K.;......
00000000`0285d5e5  d3 c7 7b ba 70 5e 80 b5-7d 3d 35 67 a3 43 ac d6  ..{.p^..}=5g.C..
00000000`0285d5f5  36 5c 03 03 4f 02 05 03-21 e2 cc 52 03 03 03 03  6\..O...!..R....
00000000`0285d605  03 03 03 03 e3 03 0d 22-08 02 01 34 03 49 01 03  ......."...4.I..
0:001> ? 0x4e ^ 0x4D
Evaluate expression: 3 = 00000000`00000003
0:001> * Encryption Key: 03

0:001>
```

*Figure 4. Dump of X64 Expiro PE found in memory before and after encryption. Noteworthy is the encryption key (4E ^ 4D = **3**), PE size [(0x2C << C) + (0x35 << 9) = **32A00**], and Nt Header with signature stripped*

At this stage, Expiro generates Polymorphic entry point code: "*GenerateEpCode*" procedure is called with original EP address as parameter. This function has 2 procedure addresses hardcoded used as model: one for 32 bit code, and one for 64 bit code. Unfortunately **I have not found any polymorphic engine** inside all analysed Expiro samples. I am sure that malware authors has left their polymorphic engine inside their organization. By the way, we are going to give a short sight at different polymorphic code types afterwards….



```
mov     r11d, [rbp+dwArchType]
mov     r10, r13
shl     r10, 3
cmp     r11, r10          ; if (dwArchType != AMD64)
jnz     short TargetIs32bit
```

```
lea     r14, Amd64EntryPointFuncModel
jmp     short loc_1000ABAAC
```

```
TargetIs32bit:
lea     r14, x86EntryPointFuncModel
```

```
loc_1000ABAAC:
lea     r11, qword_1001CC4B0
mov     r11, [r11]
lea     r10, qword_1001CC5D0
add     r11, [r10]
sub     r11, 0Eh
```

Figure 6. A snapshot of Expiro "GenerateEpCode" routine

The job of "*GenerateEpCode*" is to determine the entry point code function model (based on target file architecture), to compile it with right relocated addresses (and right decryption key), and then it has to copy the new generated code in the target file entry point address. The function exits and return the new EP code size. Expiro exploits a big data structure to pass all needed information to the "*GenerateEpCode*" routine:

```
int GenerateEpCode(LPBYTE targetPtr, EXPIRO_STRUCT * pExpStruct, DWORD ExpMainRva,
   DWORD dwOrgEpRva, DWORD archType)
```

At this stage the target file is quite ready to be written. The last thing remained to write is the right offset for the JUMP to "*ExpiroMain*" procedure located in the end of infected Ep Code. Expiro calculates offset and writes it in the right position (pattern "`mov rax, <ExpiroMain> - push rax - retn`"). Finally it creates a target file with the same name of the original one but with ".vir" extension and writes the new memory mapped infected PE. "*CopyFile*" Windows API is used to actual overwrite original file, and, if it succeed, the ".vir" file is deleted. The target file is now infected, procedure exits.

Polymorphic code – a quick sight about different types of polymorphism and obfuscation

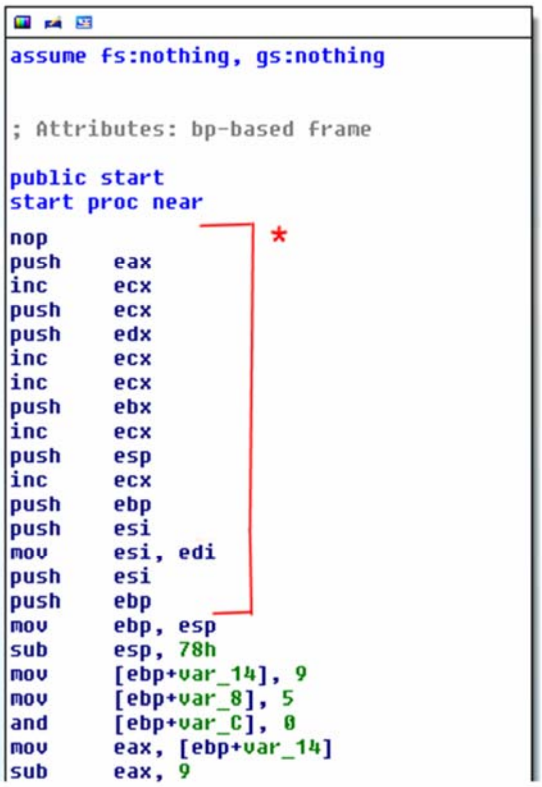
As already explained, no clues of polymorphic engine has been found (this is different from what we have seen in the olds Virut droppers). By the way, Expiro exploits a lot of polymorphic code to try to evade AV detections. Furthermore, Expiro code is quite difficult to understand because It uses some obfuscation techniques. The obfuscation is almost done using math. The typical pattern found is something like:

```
            lea     r11, g_qwEight              ; 8
            mov     r11, [r11]
            lea     r10, g_qwFive               ; 5
            mov     r10, [r10]
            lea     r11, [r11+r10+2Fh]          ; R11 = 0x8 + 0x5 + 0x2F = 0x3c
            mov     r11d, [r11+r12]             ; r11 = Nt Header Rva
```

This code snipped is used to load Nt header RVA in R11 register. This can be accomplished with only a single assembly instruction: `mov r11D, dword ptr [r12+3Ch]`. The usage of 6 instructions that load pointers of some static global variables, resolve them, and perform some math, render code understanding a bit tricky. By the way this kind of obfuscation is quite easy to understand, even because the pattern is highly repeated:

```
            mov     rax, 1CEh
            lea     r9, g_qwSix
            mov     r10, ds:0[r9]         ; r10 = 6
            cqo
            idiv    r10                   ; RAX = 'M'
```

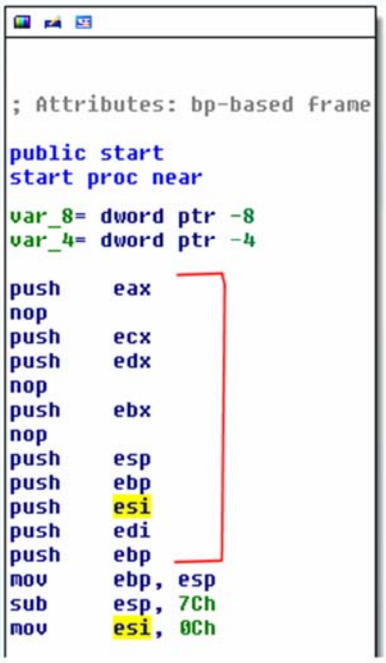
The assembly code above is another sample of easily obfuscated code. To move 'M' char to EAX register Expiro exploits an "integer divide" math operation and 5 instructions.
In the same way, Polymorfism is quite simple but effective. 32 bit Expiro infected entry point code, for example, exploits a common pattern in its beginning:



```
assume fs:nothing, gs:nothing

; Attributes: bp-based frame

public start
start proc near
nop
push    eax
inc     ecx
push    ecx
push    edx
inc     ecx
inc     ecx
push    ebx
inc     ecx
push    esp
inc     ecx
push    ebp
push    esi
mov     esi, edi
push    esi
push    ebp
mov     ebp, esp
sub     esp, 78h
mov     [ebp+var_14], 9
mov     [ebp+var_8], 5
and     [ebp+var_C], 0
mov     eax, [ebp+var_14]
sub     eax, 9
```



```
; Attributes: bp-based frame

public start
start proc near

var_8= dword ptr -8
var_4= dword ptr -4

push    eax
nop
push    ecx
push    edx
nop
push    ebx
nop
push    esp
push    ebp
push    esi
push    edi
push    ebp
mov     ebp, esp
sub     esp, 7Ch
mov     esi, 0Ch
```

* = Expiro common startup code pattern

As the reader can see, NOP instructions are substituted by INC instructions in this sample BUT 9 registers are pushed on the stack in both code snippets. After the common pattern, real polymorphic code begins. As I have already stated, Expiro engine employs polymorphic techniques to bypass AVs detection. We will make here some examples. Assume that we would like to set EAX register to 0, Expiro does this in at least 3 different ways:

```
1. mov esi, 0Ch
... (other instructions)
mov eax, esi
sub eax, 0Ch                  ; EAX = 0

2. mov [ebp+var_14], 9
mov [ebp+var_8], 5
and [ebp+var_C], 0
... (other instructions)
mov eax, [ebp+var_14]
sub eax, 9                    ; EAX = 0

3. mov [ebp+var_8], 10h
mov [ebp+var_C], 4
... (a lot of other instructions)
mov eax, [ebp+var_C]
mov ecx, [ebp+var_8]
inc ecx
div ecx                       ; EAX = 0
```

In my analysis, I haven't seen any classical polymorphism patterns, like, for this example, "`xor eax, eax – sub eax, eax`" or something classical like these. The polymorphism exploited by Expiro hinders the static identification of its infected files. An AV company should use an x86-64 emulator to correctly clean and identify an Expiro forged PE file. This is actually **not True**. A careful reader has already identified some clear weak points in infection algorithm that let even a static software to be able to correctly clean an infected file. Am I right? (if this is not the case and if the reader is interested just mail me at address aall86@gmail.com).

Next steps after service infection

Expiro file infector, after it has completed service infection, returns execution control to main thread. Main thread immediately calls "*InfectProgramsMenuAndDesktop*" routine. This, as the name implies, exploits the "*SHGetSpecialFolderPath*" API to obtain Start menu and Desktop directory paths. Then it calls "*InfectDirectory*" function to enumerate all files in the target directory and infects all *lnk*, *exe*, *scr* file types. File and directory enumeration is performed using the classical *FindFirstFile / FindNextFile* APIs. Infection process for "*scr*" files is the same what we have already seen for *exe* files. Windows Link files instead are parsed by "*InfectWindowsLink*". The latter Expiro procedure manually parses target link file (here are windows Link specs: http://msdn.microsoft.com/en-us/library/dd871305.aspx). If some conditions are met (HeaderSize equals to 0x4c, LinkFlags includes *HasLinkInfo* flag, and others), link target is resolved and "*InfectFileStub*" routine is used to perform actual infection.
Execution control is again returned to Expiro Main thread to perform the 4 last things.

1. Builds configuration files strings and creates 3 threads that performs the following jobs: system volume ID string building (the Volume where Windows System directory resides), creation and management of configuration files; temporary internet files analysis, active windows enumeration and analysis
2. All system drives infection. Expiro spawn another thread that begin its execution at "*InfectDrives*" routine. This procedure builds a memory map of each drive letter found in the system, and infects all files located in a target volume only if the drive is fixed, removable or represent a remote connection (this is a very infective threat). "*InfectDirectory*" routine is again exploited to perform actual infection in each volume. "*InfectDrives*" function is called in an endless loop every 120 seconds. In this way even a just-connected USB stick could be detected and infected by Expiro.
3. Steal user personal certificates
4. Finally start Expiro window message pump

Unfortunately, I haven't had much time to deep investigate these 4 last characteristics. I personally think that they can be interesting. If a curious reader would like to deepen how they are implemented, please absolutely let me know…. (send me a mail at aall86@gmail.com)

Conclusions

In this brief analysis paper we have investigated about the first "multi architecture" file infector in the wild. We have seen that, although Expiro employs some clever ideas to do its job, there are a lot of characteristics that are very weak, in particular:

- Absence of the real polymorphic engine in infected files
- Simple XOR encryption of its internal section
- Usage of simple Win32 APIs to perform actual infection process (like *Process32First / Process32Next* toolhelp functions or Service control manager APIs)
- Some noticeable patterns that let polymorphism and obfuscation useless. Even a static remover can be used to correctly hit and destroy infection part of files.

By the way keep in mind that this threat is **very effective**. As we have seen previously, It can even infect files that reside in removable or network drives. This can be a serious problem for corporate users that have active network shares. Even an USB key can be used as virus spreading vector.

We will see in these months if a new improvement version of this threat is released, perhaps with some weak points modified.

Special thanks always goes to *Kernelmode.info* folks, who provide me fresh droppers for the analysis. And, of course, thanks even to Expiro authors, who yield me hours of fun in reversing and understanding their creature… ☺ ☺ ☺

Andrea Allievi
Last revision: 14/01/2014