

ANALYSING THE PACKER LAYERS OF ROGUE ANTI-VIRUS PROGRAMS

Rachit Mathur, Zheng Zhang
McAfee Inc., 20460 NW Von Neumann Dr.,
Beaverton, OR - 97006, USA

Email {Rachit_Mathur, Zheng_Zhang}@
McAfee.com

ABSTRACT

It is well known that FakeAlert programs have become a real problem to deal with. The major problem for static signature scanners has been their ever-changing layers of decryptors. This paper focuses on the code analysis of the decryptor layers of such programs. It takes a comprehensive look at how the malware family evolved over the past years and the anti-RE tricks they employ to continually evade detection.

INTRODUCTION

The infamous rogue security tool software (aka FakeAlert programs) has been a major problem to deal with. FakeAlert programs purport to be anti-virus software and dupe users into believing that their system is infested with malware even when it may actually be clean. They present fake notifications to the user about malware that does not even exist on the system and use these scare tactics to get the users to pay for a fix for a problem they did not have to begin with. Figure 1 shows the user interfaces of some prevalent FakeAlert malware such as SecurityTool, Internet Security, Home Security and Total Security.



Figure 1: FakeAlert user interfaces.

FakeAlert programs have been around for a few years now and, since they constantly present notifications to users, unlike many other malware, they do not go unnoticed. Even back in 2009, in a poll conducted by *Virus Bulletin* almost 74% voted that they came across FakeAlert [1]. FakeAlert has been an

unusual challenge for the AV industry: the total number of FakeAlert-infected machines globally remains high; new undetected samples are appearing daily in large volumes; generic signatures have proved ineffective with little proactive detection. There are multiple factors that contribute to the pervasive nature of these threats [2–5]:

- To evade detection, the files are mostly encrypted and highly polymorphic.
- Additionally, these notorious money-motivated criminals have various distribution mechanisms for their binaries from spam, to search engine optimizations, to fake video codecs, to social networking sites, to exploits etc.
- Also the virus authors try to make these programs look very legitimate and even try to mimic the look and feel of common AV solutions. It can be very difficult for an average user to spot the difference between a legitimate AV and FakeAlert.

Note that this paper focuses on the analysis of FakeAlert binaries only. We will not delve into other important but unrelated aspects such as the distribution mechanism or law enforcement.

This paper tries to answer two puzzling questions the security industry faces: ‘What is so different in FakeAlert binaries?’ and ‘What can be done to improve the detection effectiveness?’ Software packing and anti-RE tricks are not new. Code and server-side polymorphisms have been seen in other malware as well. The whole AV industry is doing relatively well in coping with other similar threats using unpacking and code emulation. There must be some reasons why the AV industry struggles to contain FakeAlert.

To answer these questions, we analysed FakeAlert samples received from late 2009 up to June 2011. The packing and anti-analysis techniques observed in these samples are documented and categorized.

Almost all FakeAlert samples that have been analysed are packed by some custom and public packers. For example, many of the SecurityTool files have double pack layers – the top layer by custom cryptors and the inner layer by PECompact [6]. As another example, both UPX [7] and custom packers have been used in Internet Security samples.

Public packers and multi-packing are frequently used by other malware and typical AV solutions are capable of handling these problems. The custom cryptors used by FakeAlert have many different strains, are highly polymorphic and constantly mutate. Though code mutation is challenging, the bigger problem is the awareness and the creativity of the malware authors in using anti-analysis tricks to specifically attack AV emulators.

As a result, how to better handle these anti-analysis techniques becomes the biggest obstacle for AV software in proactively detecting these malware.

To understand the FakeAlert challenge from the technical point of view, this paper presents the details of some interesting anti-analysis techniques that the malware uses. These techniques are introduced in the following sections, one category per section. Finally, the conclusion summarizes the paper.

JUNK API CALLS

Similar to several other advanced threats, junk API calls are widely used by FakeAlert families. Amongst the samples we

have analysed, FakeAlert makes junk API calls for two different purposes:

1. API calls with invalid or ambiguous parameters [7] to detect debuggers and emulators.
2. Long loops of useless API calls to waste computer cycles and to defeat AV emulators.

```

pusha
lea  eax, [esp+28h]
push  eax
push  PAGE_READWRITE
push  MEM_COMMIT | MEM_RESERVE ; flAllocationType
push  9D000h ; dwSize
push  eax ; lpAddress
; VirtualAlloc with invalid lpAddress parameter.
; The return value should be zero and
; the last error code should be 0x000001E7
; ERROR_INVALID_ADDRESS
call  ds:VirtualAlloc
; If eax is not zero, emulator is detected!!!
test  eax, eax
jnz  short DO_RETURN
; Modify the return address based on the
; last error value.
call  ds:GetLastError
shr  eax, 8
pop  ecx
add  [esp+20h], eax
DO_RETURN: ; CODE XREF: CalcNextJump-12j
popa
mov  eax, [esp+0Ch]
mov  ebp, offset dword_404720
ret  0Ch

```

Figure 2: Sample code of junk VirtualAlloc call.

```

pusha
xor  edi, edi
mov  esi, 1001h
; junk loop to call CreateMutexA() 4K times
CM_LOOP_START: ; CODE XREF: sub_405543:loc_405521j
push  0
push  1
push  0
call  ds:CreateMutexA
push  eax
test  eax, eax
cmovz esi, edi
dec  esi
jnz  short CM_LOOP_START
; Compare the difference of the mutex handles
; returned by two consecutive CreateMutexA() calls.
pop  eax
mov  ebx, eax
push  0 ; lpName
push  1 ; bInitialOwner
push  0 ; lpMutexAttributes
call  ds:CreateMutexA
; EBX <== ffffffff /*-4*/
sub  ebx, eax
mov  esi, 1000h
push  eax ; hObject
call  ds:CloseHandle
; Junk loop to call CloseHandle() 4K times.
CH_LOOP_START: ; CODE XREF: sub_405543:loc_405553j
call  ds:CloseHandle
dec  esi
jnz  short CH_LOOP_START
; Change the call return address based on
; the EBX value
add  ebx, 4
add  [esp+20h], ebx
popa
ret  0Ch

```

Figure 3: Sample code of junk API call loops.

The sample code in Figure 2 is a typical stealthy jump routine in the custom packer of SecurityTool, which invokes VirtualAlloc with invalid parameters and modifies the return address based on the return value and the last error code of the VirtualAlloc call.

The code sample in Figure 3 contains two junk loops: the first loop calls CreateMutexA and pushes the created mutex handles into the stack; the second loop calls CloseHandle to release the mutex handle. Because each loop has 4K iterations, it could cause long delays for emulators to create and release these objects and, consequently, to terminate prematurely. In addition, this function also checks the return values of two further CreateMutexA calls and changes the call return value according to the difference.

A very large set of junk APIs have been found in many FakeAlert families, ranging from popular Win32 APIs, e.g. CreateMutexA, VirtualAlloc, ... to some rarely used ones, such as SetProcessPriorityBoost and ICSendMessage.

This poses a big challenge for AV emulators, most of which only support a small set of popular Windows APIs. Moreover, most of the supported APIs lack the rigid parameter validation logic that the real Windows APIs have.

INLINE PATCHING

Inline patching is another anti-emulation technique that was observed in some SecurityTool samples. This technique is designed to evade AV emulation by moving some of the decryption logic into an inline hook to a low level Windows API, which will be triggered by invoking another API of a higher level.

Figure 4 lists the codes that install an inline hook to NtQueryInformationFile API and then trigger the inline hook by issuing a junk GetFileSizeEx call.

The codes of the installed inline hook are shown in Figure 5, which first change the memory protection flags of the first

```

; Call VirtualProtect to change the protection
; of ntdll!NtQueryInformationFile
lea  ebx, [ebp+41067Eh]
push  ebx ; lpOldProtect
push  PAGE_EXECUTE_READWRITE ; flNewProtect
push  5 ; dwSize
push  eax ; lpAddress = NtQueryInformationFile
mov  eax, [ebp+4105F0h]
call  eax ; VirtualProtect
; esi <== NtQueryInformationFile
pop  esi
; Copy 5 bytes from NtQueryInformationFile
lea  edi, [ebp+4105D7h]
mov  al, [esi]
mov  ebx, [esi+1]
mov  [edi], al
mov  [edi+1], ebx
; Patch NtQueryInformationFile to install the hook
mov  byte ptr [esi], 0E9h
lea  edi, [ebp+41014Ch] ; FAV_Hook
sub  edi, esi
sub  edi, 5
mov  [esi+1], edi
push  esi
; Call GetFileSizeEx to trigger the inline hook
push  ebp ; lpFileSize
push  eax ; hFile
mov  eax, [ebp+4105E8h]
call  eax ; GetFileSizeEx

```

Figure 4: Sample code of inline patching.

```

FAV_Hook:
mov esi, esp
add esi, 44h ; hardcoded stack offset
mov ebp, [esi]
pushaw
; Make the first section read/writable
mov edi, [ebp+410608h]
add edi, 1000h ; edi <== section addr
mov ecx, [ebp+41063Ch] ; ecx <== section size
lea eax, [ebp+41067Eh]
push eax ; lpfOldProtect
push PAGE_READWRITE ; flNewProtect
push ecx ; dwSize
push edi ; lpAddress
mov eax, [ebp+4105F0h]
call eax ; VirtualProtect
popaw
add esp, 48h ; Unwind the stack frame
; Uninstall the inline patch
pop esi ; esi <== NtQueryInformationFile
lea edi, [ebp+4105D7h] ; edi <== stolen bytes
mov al, [edi]
mov [esi], al
mov ebx, [edi+1]
mov [esi+1], ebx

```

Figure 5: Sample code of the inline hook.

section to make it read/writable and then uninstall the hook. If an AV emulator is unable to trigger the hook due to inaccurate API emulation, the emulation could fail due to access violations later on.

Another interesting observation is that the malware is using hard-coded stack offsets, which are highlighted in red in Figure 5, to access the data it pre-pushed into the stack. This means that, to emulate the samples, AV emulators must not only accurately emulate the API call chains but also set up correct stack frames for each API call. This technique has been particularly popular with the variants we saw in 2011.

EXCEPTION CONTEXT MODIFICATION

Many malware families use this technique where:

1. Firstly an exception handler is registered by the malware.

```

_CONTEXT
+0x000 ContextFlags : Uint4B
+0x004 Dr0 : Uint4B
+0x008 Dr1 : Uint4B
+0x00c Dr2 : Uint4B
+0x010 Dr3 : Uint4B
+0x014 Dr6 : Uint4B
+0x018 Dr7 : Uint4B
+0x01c FloatSave : _FLOATING_SAVE_AREA
+0x08c SegGs : Uint4B
+0x090 SegFs : Uint4B
+0x094 SegEs : Uint4B
+0x098 SegDs : Uint4B
+0x09c Edi : Uint4B
+0x0a0 Esi : Uint4B
+0x0a4 Ebx : Uint4B
+0x0a8 Edx : Uint4B
+0x0ac Ecx : Uint4B
+0x0b0 Eax : Uint4B
+0x0b4 Ebp : Uint4B
+0x0b8 Eip : Uint4B
+0x0bc SegCs : Uint4B

```

Figure 6: Context structure snapshot.

2. An exception is intentionally triggered which causes the main thread context to be saved and the malware exception handler to be invoked.
3. The malware exception handler modifies the saved thread context. Specifically, the register values saved within the thread context are modified and return to resume execution for the original thread where the exception was raised.
4. When the main thread resumes execution its registers may have been changed.

Figure 6 shows the layout of the CONTEXT structure, which contains the register snapshot of a running thread and is passed to exception handlers, when exceptions occur. The register values within the structure could be used by packers and malware to detect debuggers and to modify the execution flow. For example, clearing the DRx registers is a popular anti-debugging trick which is heavily used by some protectors and the Swizzor trojan to disable hardware breakpoints.

FakeAlert uses this context modification technique to create loops that waste execution cycles. This is another creative way to attack the emulator using the old time-wasting trick. Since such loops are not typical and are dependent on exception context modification, they are difficult to identify and break during analysis. FakeAlert variants modify general purpose registers (such as eax and ecx) in the context structure and use them as loop counters for a long loop.

Figure 7 shows an example of code of an exception handler from a FakeAlert variant. After registering this handler the malware intentionally triggers an exception. So the context of the main thread is saved on the stack and this handler is invoked. As demonstrated in Figure 7, the EAX register is used as a loop counter here, and until the loop terminating condition is reached this handler just increments the EAX register and tries to resume execution which will again trigger an exception. Only when the counter finally reaches terminating condition, after a large number of exceptions, the exception handler modifies the EIP register of the main thread to a different location so that the main thread can continue execution without raising further exceptions.

This has been another favourite of FakeAlert from the beginning and can be seen even in the most recent variants.

```

pop ecx ; SE handler begin
lea esp, [esp+4]
pop eax
pop edx
inc dword ptr [edx+0B0h] ; Context._EAX
jnz short loc_401039 ; loop check based on eax
; value. Jumps to SE handler
; epilog without modifying
; EIP, so exception will be
; raised again and again until
; this condition is met

push eax
xor eax, eax
xor eax, [edx+0B0h] ; Context._EIP
add eax, 58h
xchg eax, [edx+0B0h] ; To terminate loop, modify
; EIP where execution resumes

pop eax

```

Figure 7: Fake AV loop using thread context modification.

KUSER_SHARED_DATA ACCESS

In its user address space *Windows* maintains a special memory area called the *shared user area* defined by a structure named `KUSER_SHARED_DATA`, which contains many fields that are vital to user-mode *Windows* APIs, including system time related, version related, pointer to system call routine within `ntdll` and pointer system call return routine within `ntdll` [8]. Typical application programs are not supposed to access this structure directly. Instead they should use supported *Windows* APIs.

| _KUSER_SHARED_DATA (0x7ffe0000) | |
|---------------------------------|---------------------------------|
| +0x000 | TickCountLow : 0x62aa |
| +0x004 | TickCountMultiplier : 0xa03afb7 |
| +0x008 | InterruptTime : _KSYSTEM_TIME |
| +0x014 | SystemTime : _KSYSTEM_TIME |
| ... | |
| +0x030 | NtSystemRoot : [260] 0x43 |
| ... | |
| +0x300 | SystemCall : 0x7c90eb8b |
| +0x304 | SystemCallReturn : 0x7c90eb94 |
| ... | |
| +0x320 | TickCountQuad |

Figure 8: `KUSER_SHARED_DATA` fields accessed by fake AV.

Even though *Microsoft* recommends that no assumptions should be made about the location of this structure or its contents, for x86 systems from *XP SP2* onwards, this structure is consistently located at the address `0x7FFE0000`.

On the other hand, because this area is not well documented and platform dependent, the structure might not be accurately emulated by many AV emulators.

FakeAlert authors are taking advantage of this gap by hard-coding access to various fields within this structure in the malicious binaries. These binaries would work on most modern OSs and of course malware authors couldn't worry less about legacy OS support or maintainability. Figure 8 lists the fields commonly accessed by FakeAlert samples.

Figure 9 shows an instruction that accesses the `SystemCallReturn` pointer which normally points to the `ntdll.KiFastSystemCallRet` function. Some samples access this directly like below and others calculate the address dynamically. An emulator would need to have this address accessible and populated to perform further analysis on the malware.

```
ADD ECX,DWORD PTR CS:[7FFE0304] ;
ntdll.KiFastSystemCallRet
```

Figure 9: `SystemCallReturn` direct access.

As another example, the code snippet in Figure 10 first verifies that the `SystemCall` pointer is non-zero. Thereafter it jumps to the location pointed to by the `SystemCallReturn` pointer. As shown in Figure 10, the address `0x7ffe0304` is supposed to contain a pointer to the `ntdll.KiFastSystemCallRet` function, which is just a return (`0xc3`) instruction. So this field can also be used by push-return based obfuscated calls.

Here the malware is not only verifying the values of shared user area fields but also transferring execution control using its pointers to system DLL areas. An emulator would need to support both behaviours to correctly tackle such techniques.

```
cmp     dword ptr ds:7FFE0300h, 0 ; SystemCall pointer points
;                               to ntdll.KiFastSystemCall
jz      short nullsub_1
jmp     dword ptr ds:7FFE0304h ; SystemCallReturn pointer points
;                               to ntdll.KiFastSystemCallRet
endp
```

Figure 10: `KUSER_SHARED_DATA` access code sample.

Similarly, some other samples directly access the `NtSystemRoot` field (address `0x7ffe0030`) to verify the string values and to use the string as decryption keys.

Figure 11 lists another sample that directly accesses the timestamp fields of the `KUSER_SHARED_DATA` structure. The sample accesses the timestamps within a decryption loop, which will continue to loop until the timestamps match with a random key that decrypts the data pointed by `EDI`.

```
push     7FFDFFF8h
LOOP_START:
clc
mov     eax, [esp] ; eax <= 7ffdfdf8
push     ecx
pop      ecx
; access KUSER_SHARED_DATA
; [eax+328] = 0x7ffe0320 TickCountQuad
mov     ecx, [eax+328h]
; [eax+8] = 0x7ffe0000 TickCountLow
add     ecx, [eax+8]
shr     ecx, 2
; edi points to an encrypted data area
mov     eax, [edi]
movsx   ecx, cl
xor     eax, ebx
xor     eax, ecx
xor     al, 4Dh
jnz     short LOOP_START
add     esp, 4
```

Figure 11: Sample codes to access `KUSER_SHARED_DATA` timestamps.

This adds extra requirements to the emulator of not only statically emulating this structure but also constantly updating the time field.

Similar to junk API calls, techniques to access shared user area fields have been very effective for FakeAlert to break emulation and to evade detection. Its usage has been observed in samples from early 2010 even till recently in June 2011.

Though not all FakeAlert samples use this technique, at a given time some samples would be using this in one form or another.

INT 2C CHECK

Some SecurityTool samples use the `INT 2C` instruction as an anti-analysis technique. Apart from returning an error code in the `eax` register, the `INT 2C` instruction also causes the `edx` register to be set to the address of the instruction next to `INT 2C` [9]. Single stepping in a debugger may not yield the

```
int     2Ch ; Internal routine for MSDOS (IRET)
; Has a side-effect of setting edx
; to address of next instruction.
add     eax, 4
inc     esi
lea     ebx, [ebp+arg_40FCFF] ; load address saved earlier
add     edx, 3 ; add 3 to address of the instruction
; just after int 2Ch
mov     eax, 17h
cmp     edx, ebx ; Verify that edx was modified properly
; due to int 2Ch instruction
jz      short loc_1444379
```

Figure 12: `INT 2C` snippet.

correct result. Also an emulator needs to be aware of this behaviour.

Figure 12 shows a snippet from a SecurityTool variant that verifies the resultant edx value as an anti-debugging/anti-emulation trick. It has been used by variants since April 2011 and is still being used by the current variants in June 2011.

VM INSTRUCTIONS

Virtualization instructions such as VMCall and VMLaunch are privileged and related to hardware-based virtualization support. Typically, these instructions would trigger an exception when called within a user-mode program. Because the virtualization instructions are relatively new and rarely used, they might not be recognized by some debuggers and not correctly emulated by many AV emulators.

The malware authors exploit this vulnerability and use these instructions in the user-mode programs. The malware first sets up an exception handler and then transfers control to the exception handlers by executing one such instruction. On the other hand, for AV emulators that don't support these VM instructions, the exceptions might not be triggered. Or even when they do, the exception codes might not be the correct value that the malware is anticipating.

This was observed in SecurityTool variants that appeared in late 2009 and early 2010.

PEB ACCESS

Directly accessing the Process Environment Block (PEB) structure is another favourite technique that many FakeAlert families use. Figure 13 shows a snapshot of a PEB structure. In particular the Ldr (at offset 0xc) and ProcessHeaps (offset 0x90) are frequently checked by most FakeAlert variants.

```

_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged         : UChar
+0x003 SpareBool             : UChar
+0x004 Mutant                 : Ptr32 Void
+0x008 ImageBaseAddress      : Ptr32 Void
+0x00c Ldr                   : Ptr32 _PEB_LDR_DATA
.....
+0x088 NumberOfHeaps         : Uint4B
+0x08c MaximumNumberOfHeaps  : Uint4B
+0x090 ProcessHeaps          : Ptr32 Ptr32 Void

```

Figure 13: PEB relevant fields.

The PEB ProcessHeaps field contains a pointer pointing to an array of active heap pointers, which in turn points to areas of process heap structures defined in Figure 14. The first item in the array always points to the default heap for the process.

```

_HEAP
+0x000 Entry          : _HEAP_ENTRY
+0x008 Signature     : Uint4B
+0x00c Flags         : Uint4B
+0x010 ForceFlags    : Uint4B
....

```

Figure 14: HEAP structure.

Figure 15 shows an example snippet of FakeAlert code that accesses the ProcessHeaps and checks that the signature value

```

mov     eax, [eax+90h] ;get ProcessHeaps from PEB
mov     eax, [eax]    ;get default heap
mov     eax, [eax+8]  ;get signature
test    eax, 1       ;check value
jnz     loc_409D98    ;signature value to get zero
xor     eax, 0EEFFEEFFh ;xor with expected
                                ;signature value to get zero
xor     eax, ecx ;assign ecx to eax
jmp     dword ptr [eax+ebp] ;jumps to correct
                                ;location if signature matched

```

Figure 15: Sample Fake AV code for PEB ProcessHeaps.

equals 0xEEFFEEFF. The target of the final JMP instruction depends on the correctness of the heap signature value.

Another field widely accessed by FakeAlert is the Ldr field in PEB (offset 0xc) which points to the Loader Data structure (PEB_LDR_DATA) shown in Figure 16. This structure contains pointers to the loaded DLL modules, ordered by the order in which these modules are loaded, initialized or located in memory. These lists are populated and maintained by the OS, which contains information about modules such as current exe image, kernel32, ntdll etc. They also contain information like module names, image base addresses, etc. [10].

```

_PEB_LDR_DATA
+0x000 Length          : Uint4B
+0x004 Initialized     : UChar
+0x008 SsHandle        : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress : Ptr32 Void

```

Figure 16: PEB Loader Data structure.

FakeAlert variants access the list using PEB and check for values of fields such as module name before proceeding to fully decrypt.

DLL IMAGE CHECK

To emulate files that directly access DLL images, most AV emulators load faked DLL files into the emulator memory.

```

; eax <== kernel32 pe header
mov     eax, [ebp-4]
; Check the kernel32 TimeDateStamp stamp field.
; If matched, go to a wrong branch.
mov     edx, [eax+IMAGE_NT_HEADERS.FileHeader.
TimeDateStamp]
sub     edx, 12345678h
jnz     loc_409D98 ; taken if not detected
; Get here, if the timestamp is faked

```

Figure 17: Sample code of DLL checking.

```

; Find the address of ntdll!RtlCreateAcl
push   72B01D88h
call   FindAddrByHash
; If the first 4 bytes of RtlCreateAcl equal to
; 0x0cc2c033 (xor eax, eax; ret 0c),
; take the jump and eventually terminate
cmp    dword ptr [eax], 0CC2C033h
jz     loc_40AB30 ; taken, if detected

```

Figure 18: Sample code of API checking.

For simplicity, these faked DLL files could be very different to those found in real *Windows* systems.

A FakeAlert family, Home Security 2012, is exploiting this difference by adding specific checks to the DLL PE structures and the API entry bytes. For example, the code in Figure 17 checks the TimeDateStamp field of the Kernel32 DLL. Besides the TimeDateStamp field, the malware also checks for the section number and the relocation and resource directories of the kernel32 DLL.

If any faked values are matched, the sample will execute into a wrong branch and eventually exit.

Figure 18 is another sample of code that XP Home Security uses to check the entry point of the ntdll!RtlCreateAcl API. If it matches with the pattern (0x0cc2c033), the malware will take a jump and eventually crash. In addition to RtlCreateAcl, many other NTDLL APIs (e.g. sqrt, RtlCopyString ...) are also checked. The API list is very random and can be changed frequently.

CONCLUSIONS

This paper has highlighted some interesting anti-analysis techniques found in the FakeAlert malware seen over the last 18 months. Though most of these are not unique to FakeAlert, FakeAlert is using these techniques in creative ways to specifically attack AV emulation, which is one of the most effective assets for AV scanners. To be effective in this battle against these sophisticated threats, emulation technology needs constant improvements such as more accurate API emulation, better support for system memory structures and newer instruction sets.

Numerous anti-analysis techniques in many sub-strains together with fast-paced updates make FakeAlert unique compared with other threats.

REFERENCES

- [1] Have you ever come across a fake anti-virus product? Virus Bulletin. September 2009. <http://www.virusbtn.com/news/polls/index?id=39>.
- [2] Bestuzhev, D. What next for rogue AVs? Virus Bulletin Magazine. March 2011. <http://www.virusbtn.com/virusbulletin/archive/2011/03/vb201103-comment>.
- [3] TrendLabs. Unmasking FAKEAV. 2010.
- [4] McAfeeLabs. Combating FakeAlerts. 22 June 2011. https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/23000/PD23178/en_US/McAfee_Labs_Threat_Advisory-Combating_FakeAlerts.pdf.
- [5] Ször, P. The 'Art' of Fake Anti-Virus Software. McAfee Labs Blog. 31 May 2011. <http://blogs.mcafee.com/mcafee-labs/the-art-of-fake-anti-virus-software>.
- [6] Technology, Bitsum. PECompact Executable Compressor. <http://www.bitsum.com/pecompact.php>.
- [7] UPX. <http://upx.sourceforge.net/>.
- [8] Schmidt, A. Getting OS Information – The KUSER_SHARED_DATA Structure. Windows Research Kernel. 9 August 2007. http://www.dcl.hpi.uni-potsdam.de/research/WRK/2007/08/getting-os-information-the-kuser_shared_data-structure/.
- [9] zairon. Beware of Int 2c instruction. <http://zairon.wordpress.com/2007/12/19/beware-of-int-2c-instruction/>.
- [10] Pravat D.; Hewardt, M. Advanced Windows Debugging: Memory Corruption Part II –Heaps. informIT. 9 November 2007. <http://www.informit.com/articles/article.aspx?p=1081496>.
- [11] Ferrie, P. Anti-Unpacker Tricks. Amsterdam: CARO Workshop, 2008.
- [12] dzzie. Understanding the PEB Loader Data Structure. Sandsprite. http://sandsprite.com/CodeStuff/Understanding_the_Peb_Loader_Data_List.html.