J Comput Virol Hack Tech
DOI 10.1007/s11416-017-0291-9

ORIGINAL PAPER

Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in UPA packers

Cătălin Valeriu Liţă[1] · Doina Cosovan[1] · Dragoş Gavriluţ[1]

Received: 28 May 2016 / Accepted: 22 January 2017

Abstract Writing modern day executable packers has turned into a rather profitable business. In many cases, the reason for packing is not protecting genuine applications against piracy or plagiarism, but rather avoiding reverse-engineering and detection of malicious samples. Unlike developers, which show moderate interest for using a packer and lack time and resources for creating one, malware creators show a huge interest and are willing to spend large amounts of money to use this technology (especially if it offers protection against security solutions). This happens mainly because protecting from piracy and plagiarism isn't that profitable as spreading new and undetected malware on as many computers as possible. Consequently, creating a custom packer designed to avoid malware detection has grown into a very profitable business.

However, developing a good packer is not an easy task to accomplish. Novel techniques of achieving anti-static analysis, anti-virtual machine, anti-sandbox, anti-emulation, anti-debugging, anti-patching, and so on, have to be discovered and added regularly. From the malware creator's perspective, this must happen frequently enough so that the updates are issued shortly after malware researchers analyze and bypass the existing mechanisms because, once these techniques are bypassed, the detection rate increases in the case of the malware samples packed with the old version of the packer.

In this paper, we present our findings which resulted from closely monitoring the fight between malware researchers and packer developers during a period of almost two years. We focus on three different packers used for prevalent malware families like Upatre, Gamarue, Hedsen. We named those packers UPA 1, UPA 2, and UPA 3 and we discuss the mechanisms used in them to achieve anti-emulation. Each technique is presented by listing the code and explaining the inner workings in details. In the end, we manage to get a grasp of the current trends in achieving anti-emulation when developing modern packers.

Keywords Malware · Packer · Reverse-engineering · Anti-emulation

✉ Doina Cosovan
doina.cosovan@info.uaic.ro

Cătălin Valeriu Liţă
catalin.lita@info.uaic.ro

Dragoş Gavriluţ
dgavrilut@bitdefender.com

[1] Alexandru Ioan Cuza University, Iasi, Romania

1 Introduction

Malware, just like any other software, started as a plain code program. As this kind of code is really simple to detect, it has become clear that other means for hiding a program's true intentions had to be developed. Thus malware creators started to encrypt the malware body. This lead malware researchers to switch the pattern matching detection signatures from the malware body to the malware decryptor, as the latter was static and written in plain code.

As a consequence, two different techniques emerged in order to protect the malicious code: oligomorphism and polymorphism. While oligomorphic malware uses different decryptors, polymorphic malware uses the same decryptor, changed with the help of various techniques like register swapping, routine permutation, code reordering, equivalent code substitution, and code transposition. The use of these operations can change a piece of code in a way that it becomes unrecognizable even to human eye.

Since it is not feasible to search all decryptors used in the wild in case of oligomorphism nor to generate all possible decryptors in case of polymorphism, the emulators started to be used for malware detection in order to dynamically execute the decryptor and obtain the decrypted virus body. Because the virus body is unchanged or rather similar, the pattern matching signatures can trigger the detection on the content obtained after emulation.

Also, obtaining the original code of the payload is important for reverse engineering the malicious code in order to get an understanding regarding the malware inner workings. This is usually the first step when cleaning an affected computer or taking down a botnet.

As a reaction to the appearance of code emulators, malware creators started to develop anti-emulation mechanisms in order for the sample to be able to realize it is running in an emulator and to stop decrypting the malicious body.

Besides anti-emulation, various anti-virtual machine, anti-sandbox, anti-debugging, and anti-disassembly techniques started to be developed. Some groups specialized particularly in protection against payload extraction, others in various anti-disassembly mechanisms, and so on. Thus building packers became a profitable industry on its own.

Since anti-sandbox, anti-debugging, anti-virtual machine, and anti-disassembly mechanisms have been widely covered in the literature, we focus on less studied anti-emulation techniques. We also cover the context in which these anti-emulation techniques are used and we monitor the changes they undergo as packer creators adapt to advances made in the emulator evolution by the security solutions.

This paper is structured as follows. The Section 2 presents the papers that studied and detailed the anti-sandbox, anti-debugging, anti-virtual machine, and anti-disassembly mechanisms that we will further refer as anti-* techniques. The Sections 3, 4, and 5 cover the evolution of the anti-emulation techniques for the UPA 1, UPA 2, and respectively UPA 3 packers. In the end, in Section 6, we conclude by discussing the difficulty of implementing and bypassing the presented anti-emulation techniques as well as the strategies used by each of the packers in order to achieve their goals.

2 Related work

2.1 Anti-* techniques

Anti-virtual machine, anti-sandbox, anti-debugging, and anti-disassembly techniques have been studied extensively in the literature. A few papers detailing them are presented as follows.

The authors in [1] present an overview of malware protection mechanisms directed against disassemblers, debuggers and virtual machines. 34 anti-debugging, 5 anti-disassembly, 10 obfuscation, and 3 anti-virtual machine techniques are analyzed in detail. Detection mechanisms are proposed accordingly in order to automatically search those specific techniques through a collection of more than 4 million samples and provide statistics. Also, statistics regarding the used packers are provided. Our research focuses on the anti-emulation techniques instead.

In [2], software armoring techniques like packers, runtime obfuscations, virtual machine and debugger detectors are presented with the help of a newly developed platform, called Saffron. It makes use of dynamic instrumentation and page fault assisted debuggers in order to achieve its task.

[3] comprises a study on anti-virtual machine, anti-sandbox, and anti-debugger techniques currently used by malicious samples like Zeus, SpyEye, SpyRat in order to target specific operating systems (Windows XP, Windows Vista, Windows 7), virtual machines (VMWare, Virtual Box), debuggers (SoftICE), and sandbox solutions (Sandboxie, Anubis, GFI CWSandbox, JoeBox, Norman Sandbox). The techniques are discussed in detail and real world case studies are presented.

[4] provides a taxonomy of anti-sandbox techniques: artifact fingerprinting (related processes, environment specific files, registry keys, I/O ports, devices and its attributes), execution environment fingerprinting, timing detection. Also, a system, called TENTACLE, is presented. Its purpose is to automatically discover anti-sandbox techniques. The malicious samples are executed over and over again. Every time, they are expected to take different paths as, at each execution, various sandbox / virtual machine related artifacts are camouflaged. With the help of CEI (Code Execution Integrity), execution branches are detected. In the end, it is supposed to identify unnatural process terminations, which are expected to be anti-sandbox mechanisms.

In [5], the authors consider an unpacker to be one of the following: a memory-dumper, a debugger, an emulator, or an Write-Execute interceptor. The paper covers various ways of tricking each of these perspectives, cumulating more than 30 trick categories. The author continues its research in a series of 9 articles, containing description and countermeasures for anti-dumping, anti-debugging, anti-emulating, anti-disassembly, and other miscellaneous tricks.

[6] contains a survey presenting debugger detection and attacks, software / hardware breakpoint and patching detection, anti-analysis, anti-disassembly, and other tricking mechanisms (process injection, debugger blocker, TLS callbacks, stolen bytes, multi-threaded packers, API redirection, and virtual machines).

[7] covers common unpacking methods in an Anti-Malware engine, ways of tricking various types of unpacking (static unpacking, emulator unpacking, both mixed routine and emulator-based unpacking), and defenses against them.

2.2 Anti-debugging techniques

Some examples of papers that focus on anti-debugging are detailed below.

[8] presents an enormous collection of anti-debugging techniques at different levels (hardware, process-level, system-level, user-interface) regarding different aspects (APIs, execution timing, uncontrolled execution, flags, heap). Both detailed description and code are provided where necessary.

[9] classifies and presents various anti-debugging techniques used on Windows NT-based operating systems, focusing more on changes at the process level.

[10] provides an overview of the debugging mechanism in Windows, a classification of the most prevalent debuggers according to the debugging methods they use, and a description of anti-debugging strategies which makes use of debugging API, special debug structures, and exceptions.

In [11], the authors developed a taxonomy of malware evasion techniques, focusing on anti-virtualization and anti-debugging behavior (hardware, execution environment, application). The prevalence of these techniques is analyzed by executing 6900 distinct malware samples in three different environments: in real systems, in virtual machines, and with a debugger attached. In the end, a mechanism is developed, which protects plain machines by simulating a monitored environment.

While some papers present anti-debugging techniques as actions taken by cyber-criminals in order to harden malware analysis and struggle to find ways of automatically detecting and mitigating them, other papers present the developer's view on the matter. Specifically, these mechanisms are analyzed in the light of helping developers to protect their code, as in [12]. In this paper, a few API-based, registry-based, hardware-based, timing-based, and exception-based anti-debugging techniques are presented along with some ways to automatically detect modified code and directly access process and thread blocks containing details about the running process.

Another paper aiming at protecting software through anti-debugging techniques is [13]. The proposed solution, SPAD, implements the most popular 13 software-only anti-debugging methods. It was tested and proved successful with 8 widely used debuggers, including user-mode debuggers like HideOD, OllyDbg, StrongOD, Phantom, and Ollyice; kernel debuggers like WinDbg; and system-level debuggers like Syser, SoftICE.

The authors in [14], have also developed an anti-debugging framework, that unlike [13], is based on hardware virtualization technology. It provides solutions against software and hardware breakpoints, but also protects the target process from being accessed by other processes.

2.3 Anti-disassembly techniques

The following papers present an interest in anti-disassembly techniques.

The authors in [15], came up with mechanisms used to thwart the disassembly process, like junk insertion, thwarting linear sweep, thwarting recursive traversal (branch functions, call conversion, opaque predicates, jump table spoofing). These were tested against 2 widely used static disassembly algorithms and they failed to correctly disassemble 65% of the instructions and 85% of the functions.

A technique, based on dynamic code generation at runtime, is proposed in [16].

In [17], authors propose a few techniques regarding binary analysis, based on control flow graph information and statistical methods, in order to improve the disassembly process of malicious samples that make use of anti-disassembly techniques. Interestingly, the authors have observed 4 main wrong assumptions made by most disassemblers, which allows for them to be tricked in incorrectly disassembling samples: valid instructions must not overlap, conditional jumps can either be taken or not taken, an arbitrary amount of junk bytes can be inserted at unreachable locations, and the control flow does not have to continue immediately after a call instruction. They also propose and test solutions for these problems.

2.4 Anti-virtual machine techniques

Research related to anti-virtual machine mechanisms is presented in the following papers.

In [18], the authors discuss a few attacks on virtual machine emulators for VMWare, VirtualPC, Parallels, Bochs, Hydra, QEMU, and Xen. The presented attacks aim at detecting the virtual machine (in order for the malware to stop performing the malicious actions while executed in a virtual environment), and performing denial of service (causing the virtual environment to crash). In the end, a few recommendations are provided in order to protect against some of the presented attacks. The author continues his work in [19], discussing even more detection techniques, attacks, and defense mechanisms for the virtual machine emulators discussed in its previous paper, but also analyzing a few more: Hydra, Sandbox, VirtualBox, CWSandbox.

In [20], the security level of using virtual machines is analyzed. A virtual machine is considered to be "root secure" if no level of privilege within the virtualized guest environment permits interference with the host system. The analysis was performed by analyzing the source code in case of open-source emulators and black-box testing based on fuzz technique for proprietary software products. As tools, they used mainly crashme for stress testing and iofuzz for fuzz testing. In the end, various vulnerabilities have been discovered for QEMU, VMware Workstation and Server, Bochs, Xen, and two other undisclosed virtual machines.

In [21], the author presents some security flaws that are unique to virtual environments and which can be used to exploit any virtualization technology. These include communication between VMs or between VM and host, VM Escape, VM monitoring from the host, VM monitoring from another VM, Denial of Service, Guest-to-Guest attack, external modification of a VM, external modification of the hypervisor.

In [22], a way to detect the presence of a virtual machine using the local data table is described. While [23] illustrates a few virtual machine detection methods and introduces DSD tracer, a malware analysis framework that integrates static and dynamic analysis.

2.4.1 Anti-emulation techniques

Many advances related to the anti-virtual machine, anti-sandbox, anti-debugging, and anti-disassembly techniques have been accomplished and information is readily available. The anti-emulation mechanisms, however, were covered only partly in a small number of papers, presented as follows.

In [24], authors seek to answer the question whether system emulators, which handle instructions in software, are more difficult to detect than traditional virtual machines. In order to find the answer, various mechanisms of detecting system emulators are discussed, which involve analyzing differences in behavior, timing, and hardware specific values.

In [25], an anti-anti-emulation system is proposed. It monitors the changes performed by the emulator and repairs the differences. The anti-emulation checks addressed in this paper involve timing attacks, CPU semantics attacks, and hardware characteristic attacks.

The [26] is an overview of the rogue malware in the past years. Since they also present a few interesting anti-emulation technique used by this malware category, this paper is also relevant to our research.

The paper [7] explains very shortly the implications that arise on executing, in an emulated environment, of samples using modern or undocumented CPU instructions, fake API calls, structured exception handling, and long loops. Various mechanisms of bypassing the conditions used to decide when to stop the emulation process are discussed as well.

The most extensive and detailed explanation of various anti-emulation techniques is presented in [5]. Mechanisms like unimplemented or undocumented instructions, unimplemented or internal APIs, invalid API parameters, software interrupts, time locks, selector verification, memory layout and file format tricks are discussed emphasizing their impact on emulators.

3 UPA 1 packer

UPA 1 is an advanced packer, used by prevalent malware like the Upatre downloader, Gamarue worm, Hedsen spammer and others. We started analyzing this packer's evolution when the first samples were identified, in May 2013, and continued up until the end of 2015. Interestingly, the packer was bundled with various and complex anti-emulation techniques from the beginning, continuing to add even more, month after month. This advertises the packer creators as experts in the field.

During the entire UPA 1 evolution we have been monitoring, the mechanisms used to encrypt the payload remained fairly constant and consisted in a combination of base64, RTL compression, and a simple encryption (for example, the XOR operation using a 1-byte key). In this section, we will present the most important anti-emulation techniques used by this packer to prevent a security solution from detecting its payload.

3.1 Rare instructions

Emulators implement many assembly instructions, but not all of them. Thus, if a packer is using a rare assembly instruction, the emulator will not be able to process it and the payload will not be decrypted. An example of rare instruction usage can be seen in Listing 1.

Listing 1 Using Rare Instructions

```
movq     mm1, qword ptr [ebp+var_4]
movq     qword ptr [esp], mm1
```

3.2 Rare API functions

Similarly, the emulators implement only widely used API functions. Calling obscure or even undocumented API functions cause an abrupt termination of the application when executed in the emulator. UPA 1 uses the not so common CryptStringToBinary API function to decrypt a base64-encrypted buffer.

3.3 PEB structure

The PEB structure [1] is usually parsed by malware creators in order to obtain the image base of loaded libraries or to detect debugger presence by checking the value of the NtGlobalFlag. The second technique is illustrated in Listing 2.

Listing 2 Checking NtGlobalFlag from PEB Structure

```
mov      eax, PEB_pointer
push     eax
```

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx

```
mov     eax, [eax+68h]    ; NtGlobalFlag
and     al, 70h
cmp     al, 70h
jz      short debugger_not_present
```

Although the main purpose of this technique is to detect debugger's presence, it acts as anti-emulation as well because the emulators may not have an exact implementation of the PEB structure.

3.4 TLS callbacks

If the emulator doesn't know that it must run a TLS callback before running the code from the entry point, then the execution might differ from the intended one.

This packer registers the TLS callback TlsCallback_0, which executes the following actions:

- detect debugger presence by checking the NtGlobalFlag value from PEB
- parse LDR_DATA from PEB structure to get the image base of kernel32 and ntdll libraries
- decrypt the name and get the address of the RtlDecompressBuffer function
- decrypt the name and get the address of the ZwUnmapViewOfSection function

The TLS callback computes the addresses of the kernel32 and ntdll libraries and of the RtlDecompressBuffer and ZwUnmapViewOfSection functions, which are used in the code from the entry point. If the TLS callback is not executed before the main function, the sample will try to use those addresses, which are not properly initialized in this case, and will crash.

3.5 Windows API results

Checking the return value of a specific Windows API function, called with specific parameters, is another anti-emulation technique implemented by UPA 1. It is illustrated in Listing 3.

Listing 3 Checking Return Value of API Call

```
no_handle = FindWindowA(&ClassName,
  "nngrohcebymvibaqcwvq dmsbqon")
if(OpenClipboard(no_handle))
{
  unpack_and_execute_payload();
}
```

First, the FindWindowA function is called with a random window name. If it is correctly implemented, then it will fail and will return NULL. Then, its result is passed as a parameter to the OpenClipboard function. The OpenClipboard function succeeds if either the passed parameter is a valid handle or NULL. In case OpenClipboard function succeeds, the sample will continue by unpacking and executing the payload. Otherwise, if the FindWindowA function is incorrectly implemented and returns a value different from 0, which is not a valid handle, then OpenClipboard will fail and the program will finish its execution. The same thing happens if OpenClipboard is incorrectly implemented, failing when receiving NULL as parameter.

3.6 FastPebLockRoutine callback

This trick starts by extracting the operating system version from the PEB structure. If it matches Windows XP or Windows 2000, it sets the FastPebLockRoutine's field from the PEB structure to a pointer to a callback function. By calling RtlAcquirePebLock, it causes the callback function to be executed. In case of newer operating systems, the callback function is called directly, without the entire mechanism of FastPebLockRoutine. The execution flow is illustrated in Listing 4.

Listing 4 Registering FastPebLockRoutine Callback

```
if(is_WinXP() or is_Win2000())
{
  register_callback(
    FastPebLockRoutine,
    callback_function);
  RtlAcquirePebLock();
}
else
{
  callback_function();
}
```

Most probably, the Anti-Virus emulators are running a Windows system matching one of the two specified versions. If the sample is executed on an operating system usually used by the emulators, then, instead of executing the function directly, it is set as callback in the PEB structure. Thus the function is not executed if the emulator doesn't implement this specific callback mechanism.

The callback for the FastPebLockRoutine decrypts the encrypted payload and injects it into a newly created process. Consequently, ignoring the callback means not executing the payload.

3.7 SecureMemoryCache callback

This technique is similar to the previous one. It registers a callback function by calling RtlRegisterSecureMemoryCacheCallback. Next it calls RtlFlushSecureMemoryCache in order to trigger the registered callback function.

An emulator won't execute the callback until it implements this specific mechanism.

3.8 TopLevelExceptionFilter callback

The function to be executed next is registered as the TopLevelExceptionFilter handler by calling the SetUnhandledExceptionFilter function, as illustrated in Listing 5.

Listing 5 Setting UnhandledExceptionFilter Callback

```
SetUnhandledExceptionFilter(
  TopLevelExceptionFilter_handler);

int TopLevelExceptionFilter_handler(
  EXCEPTION_POINTERS *ExceptionInfo)
{
  exception_record =
    ExceptionInfo[0]->ExceptionRecord;
  RtlFlushSecureMemoryCache(
    &memory_cache, memory_length);
  return 1;
}
```

Then, in a newly created thread, an exception is generated by design. The generated exception triggers the TopLevelExceptionFilter_handler function that was previously registered. The callback function calls RtlFlushSecureMemoryCache that triggers another callback function, as described in the previous technique.

3.9 Window creation callback

A WindowClass is registered by calling RegisterClassExA. It has a field called wnd_proc, which can be initialized with the address of a function to be triggered after the CreateWindowExA function is called. Emulators don't usually call the wnd_proc function.

At the beginning of the year 2015, the UPA 1 packer added, for a short period of time, a new anti-emulation technique. Specifically, when calling the window procedure, with the WM_CREATE message, the function expects the EBX register to have the value 0. Based on the EBX register value, it computes the location where the address of an API function, calculated by CRC, will be saved, as presented in Listing 6.

Listing 6 Saving API address

```
mov     [ebx+edx*4], eax
```

If the EBX register has a value different than 0 when the function is called, then the location where the API function address is saved will be dependent on that value. Being saved at a different address, the old value from the intended location will be called later on, causing a crash.

3.10 Big loops

UPA 1 makes use of big loops with millions of iterations. This technique is not new, but still an effective one. It undergoes many changes during the 2015 year.

In March 2015, it just calls a function with ECX containing the number of loops, as illustrated in Listing 7.

Listing 7 Big Loops in March 2015

```
mov     ecx, 6.110.179
call    big_loop_calling_IsDebuggerPresent
```

In June 2015, it uses two big loops, one with an API call and the other without an API call, as can be observed in Listing 8.

Listing 8 Big Loops in June 2015

```
mov     ecx, 6.791.700
call    just_big_loop
mov     ecx, 6.791.701
call    big_loop_calling_IsDebuggerPresent
```

In July 2015, it started to execute sqrt and rol in one of the big loops in order to vary the instructions. This is presented in Listing 9.

Listing 9 Big Loops in July 2015

```
v3 = 199.990;
do
{
  result = sqrt(result);
  a2 = __ROL__(a2, 96);
  --v3;
} while (v3);

mov     ecx, 32.782.721
call    big_loop
```

In August 2015, the number of iterations sqrt is called is significantly increased, as can be observed from Listing 10.

Listing 10 Big Loops in August 2015

```
v5 = 19.999.990;
do
{
  result = sqrt(result);
  --v5;
} while (v5);

mov     ecx, 6.791.700
call    big_loop
```

In September 2015, a test is added after the call of the function containing the big loop: specifically it tests if the ECX register gets to have the value 0 after the execution of the loop. According to Listing 11, this is achieved by adding the value of the ECX register to the address of the next function to be called. In case ECX is zero, the execution will continue successfully. Otherwise, the execution will be unpredictable, executing from the computed address.

This seems to be a defense against emulators trying to bypass the big loops. If an emulator observes the same instructions being executed over and over again, it can draw the conclusion that it is executing a garbage big loop and it can exit the loop early. If this happens, ECX will end up having a value different than zero as this register is used to count the iterations decreasingly.

Listing 11 Big Loops in September 2015

```
mov     ecx, 9.999.999
call    big_loop
lea     eax, sub_4024CF
add     eax, ecx
call    eax
```

In November 2015, the big loop is no longer a function being called, but integrated in the calling code, as illustrated in the Listing 12.

Listing 12 Big Loops in November 2015

```
mov     edx, 998.261
dec     edx
jnz     loc_403D36
```

4 UPA 2

In order to illustrate the evolution of the anti-emulation techniques used by the UPA 2 packer, we monitored the updates it issued during the past year. This packer was mostly used by the Upatre downloader. Its first appearance dates to the beginning of 2015.

4.1 Stack check

Shortly after the process starts its execution, the first anti-emulation technique is put in place. It consists in checking the stack address. Specifically, if the last WORD of the value stored in the ESP register is less than 0xFF00, then the program terminates the execution, as illustrated in Listing 13.

Listing 13 Checking Stack Address

```
mov     eax, 0
add     eax, esp
add     bx, 1
push    0FFh
pop     ecx
rol     ecx, 8
mov     si, ax
cmp     si, cx
ja      continue
mov     eax, offset TerminateThread
push    0
retn
```

The execution is terminated differently in different samples:

– execute an int 3, issuing an exception;
– start executing from address 0, causing access violation;
– jump to an address containing invalid instructions, causing a crash;
– jump to an invalid address, causing an access violation;
– infinitely loop over this check without changing the stack address;
– return;

In the middle of the year, the ESP register was changed with EBP and two months later the change was reverted.

Towards the end of the year, the value was increased to 0xFF01.

Afterwards, the logic was changed so that if the last byte of ESP is smaller than or equal to 2, then the program returns.

And, in the end, this technique was dropped altogether.

4.2 Flag check

Another anti-emulation technique that checks the state of the process at the beginning of execution consists in checking whether the ZF flag is set when the process is started. Listing 14 contains the first 5 instructions executed by the sample.

Listing 14 Checking ZF Flag

```
push    edx
pop     ebx
mov     ecx, edx
mov     ebp, esp
jz      no_anti_1
```

Since push, pop and mov are the only executed instructions and they do not change any flags, we can assume this check allows for an emulator detection. If the ZF flag is set then the size of the stack is no longer checked.

4.3 Registry state after API calls

Another anti-emulation technique consists in checking whether the values of particular registers are being changed after calling specific Windows API functions. This packer implements this method in three different ways.

First, it initializes the ecx register with the value 2, calls the GetACP function, and then expects ecx to be different from zero, as illustrated in Listing 15.

Listing 15 Checking Register Value after API Call

```
mov     ecx, 2
call    ds:GetACP
test    ecx, ecx
jnz     no_anti_1
retn
```

Second, it initializes the ecx register with 0, calls the GetSystemDirectoryA function, and expects ecx to be different from zero (Listing 16).

Listing 16 Checking Register Value after API Call

```
mov     ecx, 0
call    eax ; GetSystemDirectoryA
test    ecx, ecx
jnz     no_anti_3
retn
```

Third, it initializes ecx with 0, calls the function CreateFileA, and expects ecx to have a value different from 0 (Listing 17).

Listing 17 Checking Register Value after API Call

```
xor     ecx, ecx
mov     eax, [eax]
call    eax ; CreateFileA
test    eax, eax
jnz     check_ecx
int     3 ; Trap to Debugger
check_ecx:
test    ecx, ecx
jnz     no_anti
int     3 ; Trap to Debugger
```

In all three cases, if the condition is not met, then int 3 is executed or the program returns.

The GetSystemDirectoryA check is annihilated in one of the next versions. In Listing 18, although the GetSystemDirectoryA check is present, the program executes from no_anti_2 regardless of the check's result.

Listing 18 Checking Register Value after API Call

```
mov     edi, offset ReplaceFileA
mov     ebx, offset SQLPrepare
add     ebx, 1400h
pusha
mov     eax, 0
xor     eax, eax
add     eax, edi
add     eax, 2EAh
sub     esp, 4
mov     [esp+24h+var_24], 190h
push    ebx
sub     eax, 0C7BB6212h
add     eax, 0C7BB6200h
mov     eax, [eax]
mov     ecx, 0
call    eax ; GetSystemDirectoryA
test    ecx, ecx
jnz     $+6 ; jumps to no_anti_2
no_anti_2:
```

This check is reintroduced later, but this time ecx is expected not to have a specific value (0x12FFB0) in order for the program to continue its execution (Listing 19). ecx seems to be initialized with this value by the system before the program starts its execution. So, the actual check consists in ensuring that the GetSystemDirectoryA function changes the value of ecx. Some emulators don't change the value of ecx when executing the GetSystemDirectory function and thus their presence is successfully detected by this technique.

Listing 19 Checking Register Value after API Call

```
push    17Ch
push    ebx
call    GetSystemDirectoryA
cmp     ecx, 12FFB0h
jnz     no_anti_2
int     3 ; Trap to Debugger
```

In the next versions, the same check applies, but the GetSystemDirectoryA function is replaced with the GetWindowsDirectoryA function, as can be observed in Listing 20.

Listing 20 Checking Register Value after API Call

```
mov     [esp+28h+uSize], 17Ch
push    edi
call    GetWindowsDirectoryA
mov     edx, 12FFB0h
cmp     ecx, edx
jnz     no_anti_1
int     3 ; Trap to Debugger
```

4.4 Return values of API calls

Besides checking the values of registers, the actual returned value of specific API calls is checked as well. For example, the UPA 2 packer calls the GetTickCount function and checks its returned value. If the function returns a value smaller than or equal to 1, then the program jumps to a wrong address and crashes, otherwise it continues the execution normally. The code can be observed in Listing 21.

Listing 21 Checking Return Value of API Calls

```
mov     eax, offset GetTickCount
call    dword ptr [eax]
cmp     eax, 1
jle     return_to_wrong_address
```

```
return_to_wrong_address:
add     ebx, 11FAh
push    ebx
retn
```

Since the EBX register hasn't been initialized, the code will continue to execute from an arbitrary address, thus having an unpredictable behavior, but most probably crashing soon.

The use of GetTickCount is explained as follows. The GetTickCount function returns the number of milliseconds that passed from the moment the Operating System was started. That number is stored as a DWORD, therefore if a system runs continuously for a period of almost 50 days, then that number will overflow and restart counting from 0. Assuming the computer is not stopped for a period of one year, this number will be equal to 0 for a maximum of 7 times and each time it will keep the value 0 only for a millisecond. Thus, the probability for GetTickCount to return 0 in a real system is very low.

In some emulators, however, the GetTickCount function was adapted to bypass a well-known anti-emulation technique. It takes advantage by the overhead added by an emulator in order to detect its presence. Specifically, it consists in comparing the difference between the values returned by two consecutive calls to GetTickCount with a threshold, which is chosen so that it is higher than the difference usually obtained on a real system, but lower than the difference usually obtained on emulators. This anti-emulation technique could be bypassed by implementing GetTickCount so that it always returns 0 or 1, which means that the difference between two consecutive calls is always 0 and thus fails to detect the emulator's presence. This adaptation of the GetTickCount function made possible the anti-emulation technique described in this subsection.

4.5 DLL presence and header values

This packer also checks the presence of specific DLL files and the value of certain fields within their headers.

The offset of the PE header [2], the e_lfanew field from IMAGE_DOS_HEADER, must match specific values for the packer to normally continue its execution. For example, e_lfanew must be:

1. 0xE0, 0xF0, or 0xE8 in %system_directory%/mfcsubs.dll
2. 0xE0, 0xE8, or 0x20B in %system_directory%/DuSer.dll

For the second example, however, if the value of e_lfanew doesn't match any value from the list, there is one condition that can still validate the library and let the program continue its execution normally: the value located at the offset 0x120 from the beginning of the header / file must be 0x00042000 (in Windows XP, this value matches and it corresponds to BaseOfData because the address of the PE header is 0xF0) or 0x00024000 (probably BaseOfData in another operating system). In case neither e_lfanew nor BaseOfData field matches the checked values, then the program jumps to an address from the stack and an exception is generated while executing the data from there as code.

In newer versions, the checked field is changed from e_lfanew to SizeOfCode, which is located in the IMAGE_NT _HEADERS. A few examples are illustrated further.

First, in %system_directory%/sbe.dll, if SizeOfCode is smaller than 0x000301FF, then int 3 is executed.

Second, in %system_directory%/qcap.dll, if SizeOfCode is equal to zero then the program starts executing the code without decrypting it first; if SizeOfCode has a value smaller than 0x000201FF then int 3 is executed.

Third, in %system_directory%/catsrv.dll, SizeOfCode must be one of the following values: 0x00000300, 0x00024 005, 0x00024006, or bigger than 0x000201FE, otherwise the program crashes.

In even newer examples, first the presence of the library is checked and only afterwards its corresponding header fields.

In one of the analyzed samples, if %system directory%/catsrv.dll is not present / could not be opened in read mode, then int 3 is executed, as illustrated in Listing 22.

Listing 22 Checking DLL Presence

```
push    edx ; "rb"
push    ebx ; %system_directory%/catsrv.dll
call    dword ptr [eax] ; fopen
test    eax, eax
jnz     no_anti_3
int     3 ; Trap to Debugger
```

If SizeOfCode $\leq$ 0x000206FD, then the code from the compute_wrong_address label is executed, otherwise - the code from the compute_correct_address label (Listing 23).

Listing 23 Checking SizeOfCode Value in Various DLLs

```
mov     eax, ebx
sub     eax, 10h
add     eax, 4Ch
mov     eax, [eax] ; eax = e_lfanew
add     eax, ebx ; eax = PE header address
push    1Eh
pop     esi
sub     esi, 2
mov     eax, [eax+esi] ; eax = SizeOfCode
cmp     eax, 206FDh
nop
jle     compute_wrong_address
```

[2] http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

```
jmp      compute_correct_address
```

The compute_wrong_address function has the code presented in Listing 24.

Listing 24 Computing Incorrectly the Address of the Encrypted Buffer

```
compute_wrong_address:
add      esi, 5FAh
push     esi
retn
```

The code from compute_wrong_address is used by compute_correct_address too, but not before initializing esi with encrypted_buffer_address (Listing 25).

Listing 25 Computing Correctly the Address of the Encrypted Buffer

```
compute_correct_address:
mov      esi, offset unk_404C0A
xor      eax, eax
add      esp, 3Ch
jmp      compute_wrong_address
```

Later, the minimum value of SizeOfCode in %system_directory%/catsrv.dll is increased to 0x000206FF, probably following some updates or new versions of the library.

After some time, the tested library is changed to %system_directory%/gdi32.dll. If its SizeOfCode is smaller than or equal to 0x26000, then it tests the library again (a never ending loop), otherwise it continues with decrypting and executing the payload. The code is presented in Listing 26.

Listing 26 Checking SizeOfCode Value in Various DLLs

```
try_again:
 .........................................
push     IMAGE_DOS_HEADER.e_lfanew
pop      eax
add      eax, edi
mov      eax, [eax] ; eax = e_lfanew
add      eax, edi ; eax = PE header address
push     1Eh
pop      ebx
sub      ebx, 2
mov      eax, [eax+ebx] ; eax = SizeOfCode
cmp      eax, 26000h
jle      try_again
jg       jmp_to_decrypt
int      3 ; Trap to Debugger

jmp_to_decrypt:
jmp      decrypt
```

Afterwards, the %system_directory%/comctl32.dll is searched on the system and validated as follows: if SizeOfCode $\leq$ 0x24000 then it retries the library validation (a never ending loop), otherwise it decrypts and executes the payload.

In the last versions, the packer decrypts and executes its payload only if a specific system library has its SizeOfCode between two specified values. Some examples are:

1. %system_directory%/duser.dll - between 0x20400 and 0x90000
2. %system_directory%/wsecedit.dll - between 0x25200 and 0x90000
3. %system_directory%/wuapi.dll - between 0x21400 and 0x90000

The Listing 27 illustrates this behavior for the first example, the other two being very similar.

Listing 27 Checking SizeOfCode Value in Various DLLs

```
mov      [esp+30h+var_30], 2Fh
inc      [esp+30h+var_30]
inc      [esp+30h+var_30]
pop      eax
add      eax, 0Bh ; eax = 0x3C
push     dword ptr [edi+eax]
pop      eax      ; eax = e_lfanew
add      eax, edi ; eax = PE header address
push     1Eh
pop      esi
sub      esi, 2 ; esi = 0x1C
mov      eax, [eax+esi] ; eax = SizeOfCode
jmp      loc_402316

loc_402316:
mov      edx, 20400h
cmp      eax, edx
jg       loc_40236C
int      3 ; Trap to Debugger

loc_40236C:
cmp      eax, 90000h
ja       call_wrong_address
call     decrypt

call_wrong_address:
add      esi, 0FFFFDEF4h
push     esi
call     [esp+30h+var_30]
```

If SizeOfCode is smaller than the minimum required, then int 3 is executed. If it is bigger than the maximum required then a wrong address is called, resulting in an Access Violation exception being generated.

Another interesting aspect to note here is the use of unnecessary spaces at the end of library names or duplicated slashes between the system directory and the library names, when loading them. An example of such a string is “%system_directory%//wuapi.dll”. The Operating Systems can deal with these cases, although it is not documented. Because of the missing documentation, emulators don't usually treat these cases.

5 UPA 3

UPA 3 is a custom packer used exclusively for the Upatre downloader. The first samples were spotted in the wild in November 2013. Unlike the UPA 1 packer, which started with many different anti-* techniques, the UPA 3 packer didn't have neither many nor advanced anti-* techniques in the beginning. Next, we will present the mechanisms it uses.

5.1 Return values of API calls

It checks whether RegisterClassA returns a non zero value and whether acmStreamConvert returns the value 5 for specific values of the parameters, as illustrated in Listing 28.

Listing 28 Checking Return Value of API Call

```
call    ds:RegisterClassA
sub     esp, 4
call    loc_401000

loc_401000:
and     eax, eax
jz      exit___loc_4010E9
call    sub_40221E

sub_40221E:
xor     edi, edi
push    0                ; fdwConvert
push    0                ; pash
push    has              ; has
call    ds:acmStreamConvert
cmp     eax, 5
jz      short loc_402236
retn
```

5.2 Window messages

UPA 3 contains interesting and complex anti-emulation techniques involving window messages, which were introduced a few days after the first samples were found.

In the first example, the packer creates a window with a wnd_proc callback function waiting for window messages. When wnd_proc receives a WM_CREATE message, it creates a button and sends a BM_CLICK message to the newly created button, as presented in Listing 29.

Listing 29 Handling WM_CREATE in Window Callback

```
if(msg == WM_CREATE)
{
  hWnd_button = CreateWindowExA(
    0, "button", "prerequ",
    1342177281, 5, 45, 160, 35,
    hWndParent, 0, hInstance, 0);
  PostMessageA(hWnd_button,
    BM_CLICK, 0, 0);
}
```

The BM_CLICK message generates a WM_COMMAND message, which is received by the wnd_proc function and handled in Listing 30.

Listing 30 Handling WM_COMMAND in Window Callback

```
if(Msg == WM_COMMAND)
{
  if((HWND)lParam == hWnd_button)
    return decrypt_next_layer();
}
```

This technique is more complicated because the emulator must know how to handle window messages and how to send the WM_COMMAND message in case of a MB_CLICK message with the lParam containing the button handle.

In the second example, the DialogBoxParamW function is used to create a window with a callback function, as presented in Listing 31.

Listing 31 Creating Window with Callback Function

```
DialogBoxParamW(hInstance,
  (LPCWSTR)0x3E8, 0, DialogFunc, 0);
```

The callback function (DialogFunc) handles window messages. In case of a WM_INITDIALOG message, it retrieves a dialog item having the ID equal to 1001 and sends a WM_SETFONT message to it, as in Listing 32.

Listing 32 Handling WM_INITDIALOG in Window Callback

```
if ( msg == WM_INITDIALOG )
{
  hWnd = GetDlgItem(hDlg, 1001);
  SendMessageW(hWnd,
    WM_SETFONT, ::wParam, 1);
}
```

When called, the GetDlgItem function generates a WM _COMMAND message. The DialogFunc callback receives this message in Listing 33.

Listing 33 Handling WM_COMMAND in Window Callback

```
if ( msg == WM_COMMAND )
{
  GetSystemTime(&SystemTime);
  result = decrypt_next_layer();
}
```

This WM_COMMAND message is treated in more details in Listing 34, which presents the corresponding assembly code.

Listing 34 Handling WM_COMMAND in Window Callback

```
mov     eax, offset SystemTime
push    eax
push    eax ; lpSystemTime
call    ds:GetSystemTime
pop     edx
xor     eax, eax
mov     ax, [edx+2] ; wMonth
add     eax, offset unk_404087
push    ds:GetModuleHandleA
push    offset sub_401912
call    decrypt_next_layer

decrypt_next_layer:
mov     esi, offset word_402676
mov     edi, eax
mov     eax, 4
mov     ecx, 0Fh
call    decrypt_buffer
push    edi
```

The current month, retrieved from the time structure returned by the GetSystemTime function, is used to compute where to decrypt the current buffer. The example from Listing 35 emphasizes this behavior.

Listing 35 Computing the Address for Decrypted Buffer

```
mov     ax, [edx+2] ; wMonth
add     eax, offset unk_404087
mov     edi, eax
```

This mechanism ensures the sample executes properly only in a specific month of the year. This behavior is understandable for a downloader like Upatre because it uses the same version of the packer for a few days only. Once the security vendors add detection for this version of the packer, Upatre will start using another version of the packer in order to evade detection.

From the emulator's point of view, however, this is a problem because it must detect the samples anytime, not just in a specific month of the year. This presents difficulties when debugging as well because the malware analyst must pay attention and make sure that the date of the system is the expected one.

In 2014, the mechanism had undergone some changes. The code can be analyzed in Listing 36.

Listing 36 Window Callback in 2014

```
if(msg == WM_INITDIALOG)
{
  hWnd = GetDlgItem(hWnd, 503);
  SendMessageA(
    hWnd, WM_SETFONT, wParam, 1);
}
else
{
  while(msg != WM_SETFONT); //infinite loop
  GetSystemTime(&SystemTime);
  result = decrypt_next_layer();
}
```

Among the changes are the use of another value for the dialog item ID used by the GetDlgItem function, and the infinite loop executed when receiving a message different from WM_INITDIALOG and WM_SETFONT.

In some of the next packer iterations, it combines the use of the BM_CLICK message as in the first example, handled by a callback function registered with the help of the DialogBoxParamW function as in the second example. The relevant code is illustrated in Listing 37.

Listing 37 Handling BM_CLICK in a callback registered with DialogBoxParamW

```
DialogBoxParamA(hInstance,
  (LPCSTR)0x335, 0, DialogFunc, 0);

unsigned int DialogFunc(
 HWND hDlg, int msg,
  unsigned int wParam, HWND lParam)
{
  if(msg == WM_INITDIALOG)
  {
    hDlgItem_99 = GetDlgItem(hDlg, 99);
    SendMessageW(
      (HWND)hDlgItem_99,
      BM_CLICK, 0, 0);
  }
  if(msg == WM_COMMAND)
  {
    if(wParam == 99)
      return decrypt_next_layer();
  }
}
```

In later packer versions, the things get even more complicated as the previous techniques are combined into a single one and more window messages have to be generated and processed. The code form Listing 38 helps understanding the message flow and taken actions.

Listing 38 Combining Techniques

```
entry_point:
  DialogBoxParamA(hInstance,
    (LPCSTR)0x1F4, 0, DialogFunc, 0);

unsigned int DialogFunc(
  HWND hDlg, int msg,
  unsigned int wParam, HWND lParam)
{
  if(msg == WM_INITDIALOG)
  {
    hDlgItem_501 = GetDlgItem(hDlg, 501);
    hDlgItem_504 = GetDlgItem(hDlg, 504);
  }
  if(msg == WM_COMMAND)
  {
    if(lParam != hDlgItem_504)
      return PostMessageA(
        hDlgItem_504, BM_CLICK, 0, 0);
    if(!(wParam >> 16))
    {
      decrypt_next_layer();
    }
  }
}
```

At entry point, a DialogFunc callback is registered for the newly created window. Immediately after the window is created, the callback function receives the WM_INITDIALOG message and thus calls the GetDlgItem function with the dialog item ID 501 and then 504, both generating a WM_COMMAND message.

At receiving the WM_COMMAND message, the DialogFunc posts a BM_CLICK message with wParam equal to 0. When the BM_CLICK message gets processed, a WM_COMMAND message with wParam equal to 0 is generated, triggering the execution of the next decryption layer.

Later on, an even more advanced technique is put in place. By sending the message EM_GETLINECOUNT to hEdit_2, the return value will be 4 because it's name has 4 lines. This number is later used to compute the address from which the execution is about to be continued. Listing 39 presents this technique.

Listing 39 Computing Next Execution Address

```
entry_point:
  CreateWindowExA(0, "contents",
    0, 0, 3300, 1400, 756,
    500, 0, 0, hInstance, 0);

LRESULT wnd_proc(
  HWND hWndParent, UINT Msg,
  WPARAM wParam, LPARAM lParam)
{
  if(Msg == WM_CREATE)
  {
    window_name = "sabcdeaaaa\r\n";
    strcat(window_name, "ebubu\r\n");
    strcat(window_name, "iccasc\r\n");

    hEdit_1 = CreateWindowExA(
      0, "edit", 0, 0x50010000,
      25, 155, 240, 30, hWndParent,
      0, hInstance, 0);

    hEdit_2 = CreateWindowExA(
      0, "edit", window_name,
      0x50010004u, 25, 155, 240, 30,
      hWndParent, 0, hInstance, 0);

    SendMessageA(hEdit_1, WM_SETTEXT,
      0, (LPARAM)"instability");
    SendMessageA(hEdit_2, WM_SETFONT,
      (WPARAM)window_name, 1);
  }
  if(Msg == WM_COMMAND)
  {
    if((HWND)lParam == hEdit_1
      && wParam == 0x4000000)
    {
      nr_lines = SendMessageA(hEdit_2,
        EM_GETLINECOUNT, 0, 0);
      ((void(*)(void))((char*)loc_401273
        + 16 * nr_lines + 1))();
    }
  }
}
```

The assembly code that computes the final address is presented in the Listing 40, the resulted value being 0x401274 + 0x40 = 0x4012B4.

Listing 40 Computing Next Execution Address

```
cmp     eax, hEdit_1
jnz     loc_4012AE
mov     eax, [ebp+wParam]
cmp     eax, 4000000h
jnz     loc_4012AE
push    0 ; lParam
push    0 ; wParam
push    0BAh ; Msg      ; EM_GETLINECOUNT
push    hWnd ; hWnd
call    SendMessageA
shl     eax, 4
mov     ecx, eax
add     ecx, (offset loc_401273+1)
call    ecx
xor     eax, eax
pop     ebp
retn    10h
```

During the next evolution step, the wnd_proc function is changed completely as can be observed from the Listing 41.

Listing 41 Using WM_PARENTNOTIFY Message

```
int starts_with_value_3 = 3;

void wnd_proc(
  HWND hWnd, UINT Msg,
  WPARAM wParam, LPARAM lParam)
{
  switch(Msg)
  {
    case WM_INITDIALOG:
      LoadLibraryA("Riched32.dll");
      create_3_windows();
      break;
    case WM_QUIT:
      PostQuitMessage(0);
      break;
    case WM_PARENTNOTIFY:
      if(starts_with_value_3 == 1)
        decrypt_next_layer(
          starts_with_value_3 - 1);
      else
        --starts_with_value_3;
      break;
    default:
      DefWindowProcA(hWnd,
        Msg, wParam, lParam);
      break;
  }
}
```

At dialog initialization, the callback calls the CreateWindowExA function three times: two times with the “edit” type and once with the “RichEdit” type. Afterwards, it waits three WM_PARENTNOTIFY messages before executing further from the loc_401D3E location, illustrated in Listing 42.

Listing 42 Using EM_LINEFROMCHAR Message

```
loc_401D3E:
mov     ecx, 0CAh
push    0 ;lParam
push    1Dh ;wPram
dec     ecx
push    ecx ;Message => EM_LINEFROMCHAR
push    hRichEdit_1 ;hWnd
call    ds:SendMessageA
mov     ecx, 401C4Eh
push    eax
test    eax, eax
jnz     short loc_401D93
```

The next step consists in sending the EM_LINEFROMCHAR message (the code 0xC9) to hRichEdit_1, one of the three windows created previously inside the create_3_windows function. The creation of the hRichEdit_1 window is illustrated in the Listing 43.

Listing 43 Window Creation

```
window_name = "gitydon\r\n";
strcat(window_name, "abloom\r\n");
strcat(window_name, "ability\r\n\r\n");
strcat(window_name, "aberration\r\n");
strcat(window_name, "abbey\r\n");
strcat(window_name, "kibitz\r\n");
strcat(window_name, "increase\r\n");
strcat(window_name, "evangelist\r");
strcat(window_name, "kicky\r\n");
hRichEdit_1 = (int)CreateWindowExA(
  0, "RichEdit", window_name,
  0x50810004u, 26, 186, 242,
  32, v2, 0, hInstance, 0);
```

Then it checks the value returned for the EM_LINEFROMCHAR message to be different than zero. It returns the value 3 in OllyDbg in this particular case because the character at position 0x1D is on the third line.

In newer samples, the check uses the exact value returned by the EM_LINEFROMCHAR message.

In even newer samples, the technique is changed completely again. The Listing 44 details it.

Listing 44 Window Procedure Evolution

```
if(Msg == WM_CREATE)
{
  hRichEdit = CreateWindowExA(0,
    "RichEdit", "␣␣␣", 0x40000004, 40,
    40, 160, 28, hWnd, 0, hInstance, 0);
  SendMessageA(hWnd,
    WM_COMMAND, 0x68u, 105);
  SendMessageA(hRichEdit,
    EM_FINDWORDBREAK, 0, 15);
  for(i = 0;i < 10000;++i)
    Sleep(100);
  result = 1;
}

if(Msg == WM_COMMAND)
  if(wParam == 0x68)
  {
    SendMessageA(hRichEdit,
      EM_SETWORDBREAKPROC, 0,
      (LPARAM)wordbreak_callback);
    return DefWindowProcA(
      hWnd, Msg, wParam, lParam);
  }

int wordbreak_callback(
  int a1, int a2, int a3, int a4)
```

```
{
  char_position = SendMessageA(
    hWnd, EM_POSFROMCHAR, 0, 0);
  DestroyWindow(hWnd);
  decrypt_next_layer(char_position);
  return 10;
}
```

At window creation, the window procedure callback sends two messages: a WM_COMMAND message with wParam equal to 0x68 and a EM_FINDWORDBREAK message. After sending these messages, it just sleeps 1000 seconds.

The first message sent by the window procedure callback - the WM_COMMAND message - is received and processed by the same window procedure callback. In order to make sure it's the same message it sent itself earlier, it checks if the wParam parameter equals 0x68. If it is, then the callback sends an EM_SETWORDBREAKPROC message, thus registering a word break callback: wordbreak_callback.

The second message sent by the window procedure callback - the EM_FINDWORDBREAK message - triggers the wordbreak_callback.

The wordbreak_callback function sends an EM_POSFROMCHAR message, that returns a char_position value equal to 0x7fff in ollyDbg. This value is used to compute the location from where to execute further.

In the next iterations, the sleeps are removed and the EM_CHARFROMPOS message is used instead of the EM_POSFROMCHAR message.

Another change consists in making sure some files don't exist before sending the initial WM_COMMAND. This is performed as illustrated in Listing 45, by checking the return value of the CreateFile function.

Listing 45 Checking File Non-Existence

```
if(CreateFileA("etings.txt",
  GENERIC_READ, 0, 0, OPEN_EXISTING,
  0, 0) == HANDLE-1)
{
  SendMessageA(hWnd,
    WM_COMMAND, 0x68, 100);
}
```

At the beginning of the year 2015, it returned to basics and started to use simple techniques similar to the ones used in the 2013 year. An example is illustrated in Listing 46. The original code is quite obfuscated with various SendMessage calls, but the listing presents a simplified version containing only the relevant commands.

Listing 46 Returning to Simple Tricks

```
if ( msg == WM_INITDIALOG )
{
  dlg_item = GetDlgItem(hDlg, 0x71);
  ::wParam = 0x65;
  SendMessageW(dlg_item, BM_CLICK, 0, 0);
}

if ( msg == WM_COMMAND )
{
  if ( wParam == 0x71 )
  {
    SendMessageA(hDlg,
      WM_COMMAND, ::wParam, 0);
  }

  if ( wParam == 0x65 )
  {
    ((void (__thiscall *)(WPARAM))
      decrypt_next_layer)(::wParam);
  }
}
```

The GetDlgItem function, called with the id equal to 0x71, is used to generate a WM_COMMAND. When dealing with this WM_COMMAND, it sends another WM_COMMAND message with wParam equal to 0x65. Finally, when receiving the WM_COMMAND with wParam equal to 0x65, it starts decrypting the next layer.

Listing 47 presents another interesting mechanism. It sends the TB_INSERTBUTTONW (0x443) message. At the first execution of the callback, a value is set to 0x10FF while at the second execution - that specific value is returned. The importance of this value consists in the fact that the address from which to execute further on is computed with its help.

Listing 47 Using TB_INSERTBUTTONW Message

```
if(Msg == WM_CREATE)
{
  SendMessageA(hWnd, 0x467, 0, 0);
  return 0;
}

if(Msg == 0x467)
{
  SendMessageA(::hWnd,
    TB_INSERTBUTTONW, 0, 0x10FFu);
  for(i = 0x48Du; i < 0x4C9u; ++i)
    SendMessageA(hWnd, i, i, 0);
}

if(Msg == 0x496)
{
  v8 = SendMessageA(::hWnd,
    TB_INSERTBUTTONW, 0, 0x307Fu);
  decrypt_next_layer(v8);
}
```

On the next iteration of this trick, the value for TB_INSER TBUTTONW was changed and a check was added in order to make sure that the file “Desktopini” doesn’t exist. It is illustrated in Listing 48.

Listing 48 Checking Existence of Desktopini File

```
lpFileName = "Desktopini";
if(CreateFileA(lpFileName, 0x80000000u,
  1u, 0, 3u, 0x80u, 0) == (HANDLE) − 1)
{
  SendMessageA(hWnd, 0x46Cu, 0, 0);
}
```

In the newer samples, the message used to keep the value for address computation was changed from TB_INSERTBU TTONW (0x443) to EM_SETEVENT- MASK (0x445), as can be observed in Listing 49.

Listing 49 Using EM_SETEVENTMASK Message

```
if ( Msg == WM_CREATE )
{
  ::hWnd = CreateWindowExA(0, "richedit",
    &byte_405133, 0x40000004u, 4, 94,
    600, 300, hWnd, 0, hInstance, 0);
  SendMessageA(::hWnd, EM_SETEVENTMASK,
    0, 0x101u);
  for(i = 1173; (signed int)i < 1225; ++i)
    SendMessageA(hWnd, i, i, 0);
}

if ( Msg == 1174 )
{
  v9 = SendMessageA(::hWnd,
    EM_SETEVENTMASK, 0, 0);
  decrypt_next_layer(v9);
}
```

In the middle of 2015, the UPA 3 packer switched to a new mechanism, illustrated in Listing 50.

Listing 50 Using SetFocus Message and EditControl IDs

```
if(Msg == WM_CREATE)
{
  hEdit_1 = CreateWindowExA(
    0, "EDIT", &lpWindowName,
    0x40000000u, 300, 35, 300, 30,
    hWnd, (HMENU)4, hInstance, 0);
  hRich_edit_1 = CreateWindowExA(
    0, "richedit", &lpWindowName,
    0x40000004u, 5, 95, 600, 300,
    hWnd, 0, hInstance, 0);
  SetFocus(hEdit_1);
}

if(Msg == WM_COMMAND)
{
  if(wParam == 4)
  {
    sub_403610(hWnd, 4);
    return 0;
  }
}

int sub_403610(HWND hWnd, int a2)
{
  UINT v2;
  SendMessageA(hRich_edit_1, 0x445u,
    0, a2 + 1537);
  v2 = 1172;
  do
  {
    SendMessageA(hWnd, v2, v2, 0);
    ++v2;
  } while ((int)v2 < 1226);
}

if(Msg == 1173)
{
  wanted_value = SendMessageA(
    hRich_edit_1, 0x445u, 0, 0);
}

if(Msg == 1174)
{
  DestroyWindow(hWnd);
  decrypt_next_layer(wanted_value);
}
```

Let’s dissect the content of the Listing 50 step by step. In the beginning, in the call to CreateWindowExA, from Listing 51, the tenth parameter (which has the value 4 in this case) is the identifier of this edit control. This identifier is used later, when a WM_COMMAND message, generated by SetFocus (hEdit_1), is received. Next, it uses the previous technique with the EM_SET EVENTMASK (0x445) message.

Listing 51 Creating EditControl with Specific ID

```
hEdit_1 = CreateWindowExA(0, "EDIT",
  &byte_405907, 0x40000000u, 300, 35,
  300, 30, hWnd, (HMENU)4, hInstance, 0);
```

This technique is changed soon by generating the WM_ COMMAND in another form. If previously the SetFocus function was used, now the SendMessageA(hEdit_1, WM_SETFOCUS, 0, 0) is executed. The callback for the WM_COMMAND generated by WM_SET- FOCUS is different as well. It initializes wanted_value with the high word of the wParam dword (wParam $\gg$ 16), which corresponds to the control-defined notification code. If it is not equal with the id of hEdit_1, it defaults to 0x100. After the WM_SETFOCUS message is processed, wanted_value contains the value 0x100. The next step consists in calling the CreateWindowExA function twice, as illustrated in Listing 52.

Listing 52 Calling CreateWindowExA with Specific IDs

```
CreateWindowExA(0, "edit", byte_406871,
  0x40000000u, 300, 5, 10, 32, hWnd,
  (HMENU)3, hInstance, 0);
CreateWindowExA( 0, "button", "Ok",
  0x40000000u, 505, 405, 100, 31, hWnd,
  (HMENU)7, hInstance, 0);
```

Each CreateWindowExA generates a WM_PARENTNOTIFY message with the identifier of the child window, HIWORD(wParam), being equal to the previously mentioned identifier: 3 for the first CreateWindowExA and 7 for the second one. When receiving the WM_PARENTNOTIFY message, the callback will add the identifiers to wanted_value, resulting the value 0x100 + 0x3 + 0x7 = 0x10A. In the end, the value from EM_SETEVENTMASK and a constant are added to wanted_value, the final result being passed to the decrypt_next_layer function. The piece of code achieving this in presented Listing 53.

Listing 53 Computing Needed Value

```
if(Msg == WM_CREATE)
{
  hEdit_1 = CreateWindowExA(0, "edit",
    0, 0x40000000u, 300, 60, 300, 32,
    hWnd, (HMENU)5, hInstance, 0);
  wanted_value = 0x96u;
  SendMessageA(hWnd, 0x478u, 0, 0);

  hRichEdit = CreateWindowExA(
    0, aRichedit, &lpWindowName,
    0x40000004u, 5, 105, 520, 330,
    hWnd, (HMENU)4, hInstance, 0);
}

if(Msg == 0x478)
{
  for(i = 0x490u; i < 0x4C6u; ++i)
  {
    PostMessageA(hWnd, i, 0, i + 1);
  }
  SendMessageA(hEdit_1, WM_SETFOCUS, 0, 0);
}

if(Msg == WM_PARENTNOTIFY)
{
  wanted_value += (wParam >> 16);
  if((wParam >> 16) == 3)
  SendMessageA(hRichEdit,
    0x443u, 0, 0x10E8u);
}

if(Msg == WM_COMMAND)
{
  wanted_value = wParam >> 16;
}

if(Msg == 0x49C)
{
  v9 = SendMessageA(hRichEdit,
    0x443u, 0, 0x3F0u);
  v8 = wanted_value;
  for(j = 128; j > 0; --j)
    v8 += j;
  wanted_value = v9 + v8;
}

if(Msg == 0x4A2)
{
  decrypt_next_layer(wanted_value);
}
```

With the passing of time the code gets simplified a little. For example, in August 2015, it looks like in the Listing 54. It begins with a trick from the previously described mechanism: CreateWindowExA generates a WM_PARENTNOTIFY with the identifier of the child window, HIWORD(wParam), being equal to the identifier. This is executed multiple times. The newly added part consists in using the LB_INIT- STORAGE message to return a value. The code is illustrated in Listing 54.

Listing 54 Using LB_INITSTORAGE Message

```
wanted_value += SendMessageA(hWnd,
  LB_INITSTORAGE, 0x7D00u, 0xD07u);
```

In this cases, the returned value is 0x7D00. The wanted_value is computed and then used in the function that decrypts the next layer.

The interesting part here is that not sending the WM_PAINT message causes a jump to the decryption function, whose execution causes a crash because the wanted_value variable is not properly initialized. The Listing 55 illustrates this mechanism.

Listing 55 Execution Flow for the Last Variants of the UPA3 Packer

```
if(Msg == WM_CREATE)
{
  wanted_value = 0;
  CreateWindowExA(,,,,,,,,,4,,);
  CreateWindowExA(,,,,,,,,,C,,);
  CreateWindowExA(,,,,,,,,,0,,);
```

```
    CreateWindowExA( , , , , , , , , ,0 , ,);
    CreateWindowExA( , , , , , , , , ,0 , ,);
    CreateWindowExA( , , , , , , , , ,5 , ,);
    CreateWindowExA( , , , , , , , , ,6 , ,);
    CreateWindowExA( , , , , , , , , ,7 , ,);
    CreateWindowExA( , , , , , , , , ,8 , ,);
    CreateWindowExA( , , , , , , , , ,9 , ,);
    CreateWindowExA( , , , , , , , , ,1 , ,);
    CreateWindowExA( , , , , , , , , ,2 , ,);
    CreateWindowExA( , , , , , , , , ,3 , ,);
    CreateWindowExA( , , , , , , , , ,A, ,);
    CreateWindowExA( , , , , , , , , ,B, ,);
}

if(msg == WM_PARENTNOTIFY)
{
    wanted_value += wParam >> 16;
    if((wParam >> 16) == 5)
    {
        wanted_value += SendMessageA(hWnd,
          LB_INITSTORAGE, 0x7D00u, 0xD07u);
    }
    else
    {
        if((wParam >> 16) == 9)
        PostMessageA(hWndParent, 0x78u,
          0xFFFFu, 0);
    }
}

if(msg == WM_PAINT)
{
    BeginPaint(hWndParent, &Paint);
    goto LABEL_18;
}

if(msg == 0x78)
{
   LABEL_18:
        decrypt_next_layer(wanted_value);
}
```

This mechanism was kept as part of the packer until the end of year, only small changes being applied here and there: the numbers used for the identifiers, the value used for the LB_INITSTORAGE message.

6 Conclusions

6.1 Anti-emulation techniques

The presented anti-emulation techniques differ when it comes to the amount of work required by the emulators to patch them and by the malware creators to bypass these patches.

The following anti-emulation techniques require a high effort in order to be fixed in the emulators, but once it's done, the packer creators cannot change them easily to evade emulators again. After a period of time it will be of no use to keep them in the packer code because they can easily trigger detection and not stop the emulators any more.

1. TLS callbacks
2. FastPebLockRoutine callbacks
3. SecureMemoryCache callbacks
4. TopLevelExceptionFilter callbacks
5. window creation callbacks
6. message handling

The medium difficulty for fixing the emulators comes with the medium difficulty for the packer creators to change some characteristics of the mechanism in order to bypass the patches issued by the emulators. The following techniques fall within this category:

1. PEB structure fields
2. library header values
3. rare instructions
4. window creation

For example, imagine an emulator that can be bypassed by searching a not implemented field within the PEB structure. In this case, the emulator can be fixed to properly implement that field. Unfortunately, packer developers can find another PEB structure component which is improperly implemented or not implemented at all to use against the newly updated emulator.

Similarly, adding a rare instruction to the set of instructions the emulator implements solves the issue only temporarily until the packer creators find another rare instruction that is not implemented in the same emulator.

The techniques listed below are part of the category according to which both patching the emulators and bypassing the patch are easily performed.

1. rare API functions
2. return values of API functions
3. register values changed by API function
4. rare library presence

Fixing and bypassing the use of rare API function calls is even easier. The packer just calls an API without checking the return value as in the previous technique. To fix this, the emulator can implement a function with that name by simply returning the value 0, while the packer creators just change the function.

While most anti-emulator techniques requires the same effort (high, medium or low) for both emulator and packer writers, there are some which don't meet this requirement: the usage of big loops and the usage of time / logic bombs. Both are hard to fix in an emulator and easy to implement in a packer

Regarding the usage of big loops, an emulator can't afford to execute a sample for a long period of time, which is required to bypass a big loop. Even more, an emulator is slower on execution than a real system. At the same time, packer creators will not have to do a complicated work to change a big loop so that it is no longer detected and bypassed by the emulator.

Similarly, adding a time / logic bomb is easily achieved, while finding and bypassing it - hard.

6.2 UPA packers strategies

The UPA 1 packer started with multiple and various complex anti-emulation techniques, encompassing rare instructions and functions, API result checks, PEB checks, big loops, as well as various callbacks for FastPebLockRoutine, SecureMemoryCache, TopLevelExceptionFilter, TLS, and window creation. As a response to emulators implementing the missing instructions, API functions, and callback mechanisms, it started to remove them one by one, remaining, in the end, mainly with big loops, containing simple but varying API calls or / and instructions.

The UPA 2 packer performed, in its beginnings, some checks on the state of the stack and on the values of the flags, but understood fairly fast that these are not reliable ways for emulator detection. Maturing, it started to rely heavily on checking that the values of registers are changed accordingly after specific API calls and that specific libraries are present on the system and have correct values for their specific header fields. Although these mechanisms are easily fixed in emulators, packer writers can also easily change the checked API function or register in case of the first technique and the library or header field in case of the second technique. The only solution to fix this problem generically would be implementing correctly all the libraries and API functions in the emulator, which is not feasible.

The UPA 3 packer individualizes by using almost exclusively mechanisms involving window creation and message handling. It started with rather simple techniques, complicating them with a fast pace. In the end, however, it ended up returning to the simple ones, but obfuscating the code implementing them so that detection is harder to add on the anti-emulation technique itself.

The simplicity, from the very beginning, of the UPA 2 packer as well as the simplification of the execution flow we witnessed by monitoring the evolution of the other two, lead us to the idea that packer creators are trying to reach an equilibrium between the complexity of the behavior and the simplicity of the aspect of the developed anti-emulation techniques. In this way, two goals are achieved.

First, if anti-emulation techniques are hidden in plain sight by having the looks of simple pieces of code achieving simple tasks makes it harder for the malware researchers to spot and bypass them.

Second, by ensuring the emulation techniques are composed of operations found in many genuine use cases, the malware creators prevent the malware researchers to add detection to the anti-emulation technique itself as this would cause false positives. Many genuine applications use big loops with common API calls and instructions (UPA 1), library checks (UPA 2), window creation and message handling (UPA 3).

References

1. Branco, R.R., Barbosa, G.N., Neto, P.N.: Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Blackhat, Las Vegas (2012)
2. Quist, D., Smith, V.: Covert debugging circumventing software armoring techniques. Black Hat Briefings, Las Vegas (2007)
3. Issa, A.: Anti-virtual machines and emulations. J. Comput. Virol. **8**(4), 141–149 (2012). doi:10.1007/s11416-012-0165-0
4. Chubachi, Y., Aiko, K.: Tentacle: Environment-sensitive malware palpation
5. Ferrie, P.: Anti-unpacker tricks–part one. Virus Bull. **4** (2008). http://www.virusbtn.com/pdf/magazine/2008/200812.pdf
6. Yason, M.V.: The art of unpacking (2007). Retrieved 12 Feb 2008
7. Tan, X.: Anti-unpacker tricks in malicious code. In: Proceedings of 10th Annual AVAR International Conference (2007)
8. Ferrie, P.: The ultimate anti-debugging reference, p 14. Tech. rep. (2011)
9. Falliere, N.: Windows anti-debug reference (2007). Retrieved 1 Oct 2007
10. Gao, S., Lin, Q., Xia, M., Yu, M., Qi, Z., Guan, H.: Debugging classification and anti-debugging strategies. In: Fourth International Conference on Machine Vision (ICMV 11), pp. 83503C–83503C. International Society for Optics and Photonics (2011)
11. Chen, X., Andersen, J., Mao, Z. M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24–27, 2008, Anchorage, Alaska, USA, pp. 177–186 (2008)
12. Shields, T.: Anti-debugging–a developers view. Veracode Inc., USA (2010)
13. Qi, Z., Li, B., Lin, Q., Yu, M., Xia, Mingyuan, Guan, Haibing: SPAD: software protection through anti-debugging using hardware-assisted virtualization. J. Inf. Sci. Eng. **28**(5), 813–827 (2012)
14. Yi, T., Zong, A., Yu, M., Gao, S., Lin, Q., Yu, P., Ren, Z., Qi, Z.: Anti-debugging framework based on hardware virtualization technology. In: ICRCCS'09 International Conference on Research Challenges in Computer Science, IEEE, pp. 218–220 (2009)
15. Linn, C., Debray, S.K.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, ACM, Washington, DC, October 27–30, 2003, pp. 290–299

16. Aycock, J., deGraaf, R., Jacobson Jr., M.: Anti-disassembly using cryptographic hash functions. J. Comput. Virol. **2**(1), 79–85 (2006)
17. Krügel, C., Robertson, W.K., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of the 13th USENIX Security Symposium, August 9–13 2004, San Diego, CA, USA, pp. 255–270 (2004)
18. Ferrie, P.: Attacks on virtual machine emulators. Symantec Adv. Threat Res. (2008)
19. Ferrie, P: Attacks on more virtual machine emulators. Symantec Technol. Exch. **55** (2007)
20. Ormandy, T.: An empirical study into the security exposure to hosts of hostile virtualized environments. 2007. Ce court article de recherche analyse la sécurité de quelques solutions de virtualisation, dont certaines traitées dans mon mémoire. Lauteur analyse la robustesse et la résilience des applications testées (2007)
21. Reuben, J.S.: A survey on virtual machine security, vol. 2, p 36. Helsinki University of Technology. http://www.tml.tkk.fi/Publications/C/25/papers/Reuben_final.pdf (2007)
22. Danny, Q., Smith, V.: Detecting the presence of virtual machines using the local data table. Offens. Comput. (2006)
23. Lau, B., Svajcer, V.: Measuring virtual machine detection in malware using DSD tracer. J. Comput. Virol. **6**(3), 181–195 (2010)
24. Raffetseder, T., Krügel, C., Kirda, E.: Detecting system emulators. In: Information Security, 10th International Conference, ISC 2007, Valparaíso, Chile, October 9–12, pp. 1–18 (2007)
25. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating emulation-resistant malware. In: Proceedings of the 1st ACM workshop on Virtual machine security, pp. 11–22. ACM (2009)
26. ODea, H.: The Modern Roguemalware with a Face. In: Proceedings of the Virus Bulletin Conference (2009)