J Comput Virol (2007) 3:221–236
DOI 10.1007/s11416-007-0046-0

ORIGINAL PAPER

Secure and advanced unpacking using computer emulation

Sébastien Josse

Received: 15 December 2006 / Accepted: 16 March 2007 / Published online: 4 May 2007

Abstract The purpose of this article is firstly to present a secure unpacker which is specifically designed for a security analyst when studying viruses but also any anti-virus scanner. Such a tool is in fact required when assessing security requirements of an anti-virus scanner through a black box approach. During testing of anti-virus software, a security analyst needs to build virus populations required for several penetration tests. Virus unpacking is a first mandatory step before gaining the ability to apply obfuscation transformation or any information extraction algorithm on a viral set. A secure unpacker is also useful when checking security robustness against reverse engineering of any packed or protected security product. Several static and dynamic analysis tools already implement unpacking algorithms, but these often require human intervention and are not well designed to automatically unpack such a dangerous program as a virus. A new algorithm for automatically unpacking encrypted viruses is presented in this paper. Forensics techniques to reconstruct an unpacked executable and advanced heuristics are also presented in order to decrypt more sophisticated self-protected Malwares. We present several detection techniques which are specifically designed to deceive virtual machine monitors and discuss the security of our tool against these low-level viral attacks. Our secure unpacker figures among a set of several tools. We then present in this paper a proof-of-concept human analysis framework which implements most standard components of an anti-virus scanner (real-time scanner, emulator engine) and in addition proposes a reliable system for automatically gaining information about a virus and its interaction with the OS executive (stealth native API hooking), but focuses on human decision as a detection process without the same resource limitation constraint as product oriented anti-virus scanners. This framework is used as a basis/reference for the comparative analysis of security aspects of anti-virus scanners and deals with the robustness of their driver stack and the efficiency of their de-obfuscation and unpacking algorithms.

Keywords Malware analysis · Anti-virus testing · Forensics · Software protection · Fault injection · Human driven analysis

This paper is the extended version presented at the AVAR 2006 Conference [39].
Sébastien Josse is an I.T. consultant at Silicomp-AQL Security Evaluation Lab and also a Ph.D. student EDX Polytechnique Doctoral School within the ESAT Virology and Cryptology Lab in Rennes
e-mail: sebastien.josse@esat.terre.defense.gouv.fr.

S. Josse (✉)
Silicomp-AQL, Security Evaluation Lab, 1 rue de la Châtaigneraie, 51766, Cesson-Sévigné, France
e-mail: Sebastien.Josse@aql.fr

1 Introduction

Anti-virus scanner designers have to face several difficulties: firstly, viral detection is an *undecidable* problem [21]. Thus, the only thing that can be done is to apply state-of-the-art heuristics and expect to have as few errors as possible [32,33]. Secondly, an anti-virus scanner has to be user-friendly. Thus, while analysing a program, it must use as few CPU and RAM resources as possible. Thirdly, an anti-virus scanner is a program, and so has to face the same constraints as a virus program while executing on an operating system: it has to be installed as deep as possible in the operating system, and its installation has to be robust against low-level attacks.

Human analysts do not have the same constraints while analyzing the behaviours of a program. They can run the program in an emulated environment and gather the required information in order to check the security aspects of the targeted program.

They can drive transformations on the targeted program, such as unpacking, in order to obtain an unprotected version of the program, which can then be analysed statically.

The static analysis of the targeted program can involve powerful tools and algorithms, with no limitation on the use of resources, in order to obtain a de-obfuscated variant/ version of a program.

Those two levels of freedom are far less real for an anti-virus scanner: for performance reasons, emulation engines are necessarily much poorer. Thus, viruses can implement detection routines which forbid any efficient emulation process.

Analysis is not driven by a human, and is thus faced with difficulties that can be avoided during a human interactively driven analysis process while disassembling and analyzing a program.

The goal of this paper is to present the general specifications of a secure and advanced analysis framework for virus analysis based on a virtual CPU.

We present the general software architecture of such a tool. In our context, the main security requirement for virus analysis tools is isolation. Security means virus propagation containment.

Another requirement for such a tool is that searched information can be obtained through analysis.

Advanced means stealth (if emulation is detected, the target executable will no longer provide information) and accuracy.

Most security software analysis tools must have a modular architecture. Among the modules/functions that can be implemented by such a tool, we focus on the specifications of an unpacking function.

For self contained purpose, let us recall some definitions and notations:

Packer: a packer is a program that takes an executable, encrypts or compresses it, and forges a new executable made up of an unpacking routine and one or several data blocks. The unpacking routine implements part of a PE loader. At runtime, the original executable is dynamically recovered in memory and then executed.

Unpacker: An unpacker is a program that takes a packed executable, suppresses the loading wrapper, and outputs the embedded executable. The packing routine is specially designed to be resilient to reverse engineering and thus to make it difficult to forge an unpacking routine.

Virtual Machine: when an operating system is virtualized, the ratio of software to hardware execution of CPU instructions can be used to determine if one is dealing with a Complete Software Interpreter Machine (CSIM), a Hybrid VM (HVM), a Type I or II Virtual Machine Monitor (VMM) or a real machine.

A CSIM or emulator uses only software interpretation. All CPU instructions are emulated by a software program. An HVM interprets every privileged CPU instruction via software.

A Type II VMM runs as an application on the host OS. It interprets only sensitive CPU instructions via software. Non-sensitive privileged instructions are executed directly by the CPU.

A type I VMM runs as an OS or kernel. It interprets only sensitive CPU instructions via software. Non-sensitive privileged instructions are executed directly by the CPU. VMware ESX and Xen are type I VMM.

CPU requirements for HVM, Type I and II VMM are analyzed in [37]. The more specific problem of implementing secure VMMs on Intel Pentium architecture is also addressed in [37].

The paper is organised as follow. The Sect. 2 of this paper answers the questions : what is a secure unpacker and why is it useful? The different existing implementations that can be found for such a tool and the constraints related to Malwares unpacking are discussed. Related works are presented in this section. The contribution of this paper, in regard to existing solutions, is also demonstrated in this section. Limitations and usage conditions are given.

The Sect. 3 of this paper presents the specifications of a proof-of-concept human analysis framework, which integrates the secure unpacker and provides reliable information about a targeted program.

2 Secure and advanced unpacking

2.1 What is it?

This tool is designed for security analysts. It covers at least the following range of use:

- *Virus analysis*: In-the-wild viruses are often packed. They also implement software protection mechanisms, which forbid the use of standard development tools such as interactive disassemblers or debuggers.
- *Packed/protected software security assessment*: Other applications implement software protection mechanisms in order to ensure secrecy of critical data or algorithms. Digital Right Management software and other security products (Cloakware, e.g., [26]) need such security functions. In order to check the robustness of their security mechanisms and to recover information on their internals working procedures, unpacking is often the first step in analysis.

- *Anti-Virus product testing*: While assessing the robustness of security functions of an Anti-Virus product, a security analyst often needs to forge a viral population conforming to several criteria before being usable in tests. Among these properties, the first one is that viruses must be decrypted, whenever possible [31].
  Indeed, obfuscation transformations and signature extraction algorithms require unpacking as a first step.
  Use that can be made of a secure unpacking tool in bench tests are described in our previous work [38].
- *Fault injection systems*: Black box testing as described above requires the ability to work with an adequate data set as an input to the target system.
  Grey box testing does the same, but also dynamically applies modifications to the running program itself, not only to its input data.
  With the ability to apply code patching transformations at instruction or basic block level, the unpacking functionality of our analysis framework is a critical component of any fault injection system.

2.2 Related works

Several static and dynamic analysis tools already implement unpacking algorithms, but these often require human intervention and are not well designed to automatically unpack such a dangerous program as a virus.

IDA 4.9 comes with a new plug-in named uunp (universal PE unpacker debugger plug-in module [28,29]) which automates the analysis and unpacking of packed binaries. This plug-in uses the debugger to let the program unpack itself in memory and as soon as the execution reaches the original entry point, it suspends the program. The user may then take a memory snapshot.

The algorithm of this plug-in is:

1. Start the process until the entry point of the packed program is reached.
2. Add a breakpoint at kernel32.GetProcAddress, resume execution and wait until the packer calls GetProcAddress. If the function name passed to GetProcAddress is not in the ignore-list, then switch to trace mode.
3. A call to GetProcAddress() most likely means that the program has been unpacked in memory and is now setting up its import table.
4. Trace the program in single step mode until we jump to the area of the original entry point.
5. As soon as the current instruction pointer belongs to the OEP[1] area, suspend the execution and inform the user.

So, in short, we allow the unpacker to do its job at full speed until it starts to set up the import table. At this moment we switch to single step mode and try to find the original entry point. While this algorithm works with UPX, ASPack, and several other packers, it might fail and execution of the packed program might go out of control. Hence this plug-in must be used with caution.

Ollydbg plug-ins [44,45] (FindCrypt, DeJunk, Ollybone, OllyDump, OllySnake, Polymorphic Breakpoint Manager, PE Dumper, Universal Hooker, Virtual2Physical and OllyScript) are very useful for manually unpacking a protected PE executable.

The plug-in OllyScript allows a security analyst automate Ollydbg by writing scripts in an assembly-like language. This plug-in is particularly useful because when manually unpacking a protected binary, many tasks involve a lot of repetitive work just to get to the OEP within the debugged application. Using this plug-in, this is possible in a single script.

For example, the following script automatically reaches the OEP of an UPX-packed executable:

```
          eob Break
          findop eip, #61#
          bphws $RESULT, "x"
          run
     Break:
          sto
          sto
          bphwc $RESULT
          ret
```

Many OSC[2] scripts have been already written in order to find the OEP of a protected executable, fix the Import Address Table (IAT), remove junk code and find the relocation table and stolen code.

All these dynamic analysis tools must be used with caution, especially when dealing with hostile Malwares. Because the target program is executed on the host system, it could evade the user mode or kernel mode debugger and spread out of control.

As a matter of fact, static disassemblers are easy to fool and debuggers are easy to detect and possibly evade. Execution within a controlled environment like a virtual machine could solve the problems of isolation and stealth. We now go on to present in this section, several approaches based on virtualisation or emulation in order to securely and reliably unpack protected programs.

Several algorithms have already been proposed, based on virtual memory analysis and assumptions made about underlying protection mechanisms. These approaches are close to ours. We can cite many projects which focus on alternative and useful algorithms which make it possible to track the memory of a targeted process and possibly to unpack a

[1] Original Entry Point.

[2] OllyScript plugin's script format.

self-protected program. Among them, we will focus on three interesting approaches:

- *Tainted based memory analysis algorithm*: Argos project [2,3,20] uses tainted based memory analysis [7] in order to detect a hostile (and possibly self-protected) program in memory. Argos is a system emulator designed for use in Honeypots [48]. It uses a modified version of the QEMU [8,51] emulator to track memory.
  Argos' author has extended the QEMU emulator to enable it to detect remote attempts to compromise the emulated guest operating system (such as a Worm shellcode). Using dynamic taint analysis, Argos tracks network data throughout the processor's execution and detects any attempts to use them in a malicious way. When an attack is detected the memory footprint of the attack is logged and the emulator exits.
  Principle of Tainted based memory analysis : data originating from an unsafe source is tagged as tainted (for example, data originating from the network is marked as tainted). Tainted data are tracked during execution (whenever it is copied to memory or register, the new location is also tainted). In order to identify and prevent unsafe usage of tainted data, an alarm is raised (whenever it is used, say, as a jump target).
- *Normalisation algorithm*: WiSA project uses a normalization algorithm [24] in order to unpack viruses before applying signature extraction algorithms and obfuscation transformations. This project also uses a modified version of the QEMU emulator in order to apply the normalization algorithm.
  Their approach is very interesting but is more intrusive/complex than ours. The emulator is modified in order to monitor memory and collect all the memory writes. If an attempt to execute code from a memory area that was previously written to is made by the targeted program, the trigger instruction is captured and the execution is terminated. Using the captured data , an equivalent program is reconstructed. This algorithm is iteratively applied until all nested unpacking code has been removed.
  We now propose in this paper another algorithm for automatically unpacking protected programs. Our algorithm is much easier to implement, in that we do not have to drastically modify the emulator in order to be able to track memory (and thus detect any attempt to execute code from a memory area that was previously written to). Our unpacking engine is, on the contrary, independent of the emulator design. Another difference is that we do not need to restart the process under analysis when the unpacking process requires several steps (for example when the protection implements several ciphering layers).
  For anyone who wants to implement the WiSA project normalization algorithm, this can be done from Argos' modified version of the QEMU emulator which already implements everything required to track memory.
  The WiSA project paper does not explain how to reconstruct the executable after unpacking. We now present in this paper at least one method that can be applied in order to reconstruct an unprotected executable after unpacking, based on Win32 and native API hooking.
- *Forensics algorithms*: the unpacking procedure often covers two problems:
  - firstly, dumping memory
  - secondly, starting from this memory snapshot, finding the areas which belong to the target program and rebuilding the executable.

  When doing a blind memory dump, it is difficult to expect to recover a fully runnable executable.
  However, forensics heuristics can give good results and can be applied to the problem of finding a process in a memory snapshot and reconstructing a binary image of a targeted process.
  Among the most well-known forensics tools, you will find the following interesting: MemParser [9], idetect, ProcEnum, WMFT [14–18], Ramdump, lsproc, lspm, ReadPE [22,23], Dd, md5lib, md5sum, VolumeDump, Wipe, ZlibU, nc, GetOpt [35], Kntlist [36], Dd for Windows [43], Minidumps [46,47], PERL script MemDump and PTFinder [58–66], Mdump [70,71], etc.

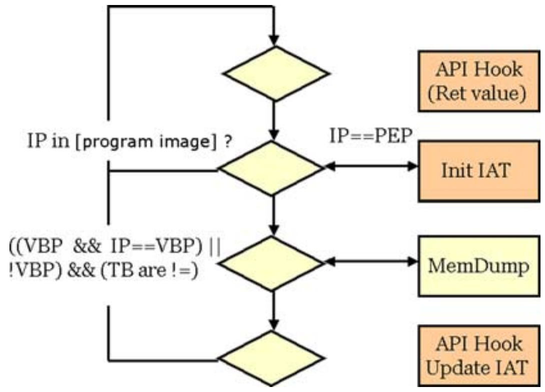


Fig. 1 Unpacking algorithm

2.3 Our approach

Protected executable unpacking

A new algorithm for automatically unpacking a protected executable is described in this section. The underlying idea is a simple integrity check of the executable code of the targeted program (Fig. 1).

```
FOR EACH TB // which is executed in ring3 and belongs to target process execution
            // space, and before the VCPU executes this TB
DO
IF (vcpu.eip == _entry_point) // Entry point of the protection
THEN
    get_IAT(vcpu, PEB)  // Initialisation of the sorted table
FI
_RHOOKS_RING0       // A native API was called. We check NTSTATUS
IF vcpu.eip IN [_image_base, _image_base + _sizeof_image]
THEN
      _RHOOKS_RING3    // A Win32 API function was called. We check return value
      raw_addr=get_raw_addr(vcpu.eip) // Get raw address corresponding to
                                      // current Instruction Pointer
      IF (read_TB(vcpu.eip) != read_File(raw_addr)) // Binary diff
      THEN
       IF ( NOT _done AND NOT _vbp ) OR
            (NOT _done AND _vbp AND vcpu.eip == _vbp)
            // No VBP is set or we are at VBP location
         THEN
            dump_memory(f) // Virtual or physical memory dump
            fix_entry_point(f) // Fix PE header
            fix_section_header(f)
            fix_IAT(f)
            fix_Reloc(f)
            _done=1
         FI
      FI
FI
update_IAT  // (If initialised) the sorted table is updated
_HOOKS  // Win32 or Native API Hook installation
DONE
```

The above algorithm describes the unpacking algorithm. The program is executed in the emulated environment. For each basic block (in fact, for each translation block), a comparison between its value in virtual memory and its value on the host file system is made. As long as the values are identical, nothing is done. As soon as a difference is identified, the current basic block is written into the raw file in place of the old basic block.

The memory area that is monitored/handled is the whole load-time image of the target program:

```
[image_base,   image_base + virtual_sizeof_image]
```

The same monitoring process/algorithm is applied for each translation block. The protection loader of the packed executable can have several ciphering layers. As soon as the last deciphered basic block has been reached, the only thing to be done is to repair the target executable. The first step is obviously to modify the entry point in the PE header of the target binary.

When the packed executable includes self-generating code, i.e. the original executable implements anti-reverse protections at source level, there is no exact algorithm for obtaining an unpacked version of the original executable, i.e. without its protection wrapper.

In this case, the unpacking process has to be human driven, or heuristics have to be applied by making assumptions about the unpacking routine and the protected executable, in order to be able to distinguish between their respective control-flows.

Because we are not always interested in obtaining a functional unpacked executable, but rather in obtaining a version of the original executable without any ciphered part in order to be able to statically disassemble part of its code or to get information about its disassembled code, we can decide to keep the protection wrapper.

Unprotected executable reconstruction

However, it may be useful after unpacking, to try to recover some of information that has been suppressed or stripped or altered during the protection process.

According to the PE formats specification [41], several part of the PE header are of interest.

Many fields of the PE header can be modified in order to disturb analysis tools [13]. With modifications of ImageBase, LoaderFlags, NumberOfRvaAndSizes, etc., a debugger may well regard the binary as not having a good image and will maybe run the application without breaking at its entry point.

This could be harmful if an analyst wanted to debug a Malware on his computer, because it would get infected.

Values in the PE header can be used as keys to decrypt a few layers of the protection. In this case, modifying these values will result in an unworkable binary.

As a matter of fact, repairing the PE header can prove to be a delicate task.

In order to recover the PE structure of an unprotected executable, several tasks have to be carried out:

- set the original entry point
- consistency check the section header
- rebuild the import address table
- rebuild the relocation table

The method used by our unpacking engine in order to reconstruct the IAT and relocations is based on Win32 and native API hooking.

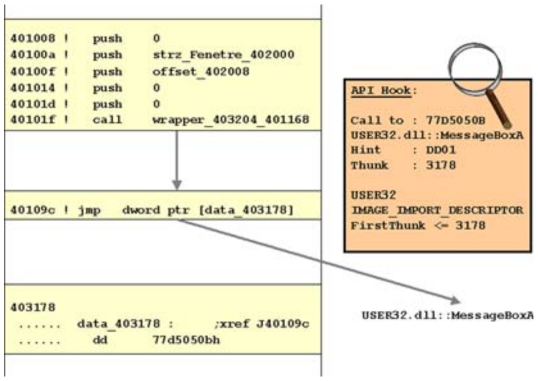
During the unpacking process, all API calls are traced. A sorted table of API calls is initialised at load-time, by walking NT executive structures. Next, after process execution resumes, each API call is traced. This table is updated regularly during the target process execution, and is used to dynamically resolve the API functions' names. Finally, after a dump of the target process memory space has been done, this table is used to fix the IAT in the PE executable.

At this stage of the unprotected executable reconstruction, we also fix the relocations. This information is very useful, before applying obfuscation transformations on binary code (generally, the relocation rebuilding is not required. It concerns more often DLL. We work on this problem only because it is required for several obfuscation engine, such as Zombie's Mistfall engine [73]. The idea is to make a binary diff between several rebased DLL or executable memory dumps).

We use heuristics that are very similar to those that are implemented by MackT [40] and y0da [72] in their PE reconstruction engines. The API import table is sorted in order to bypass some protections which emulate first bytes of API functions and therefore do not jump to the original entry point of API functions. By maintaining a sorted table of API functions, API functions are not indexed by their entry point but by a memory range in memory. We are now able to trace API calls even if their first bytes are stolen.

We trace instruction by instruction until we get an address which may belong to the original IAT (see Fig. 2 page 226).

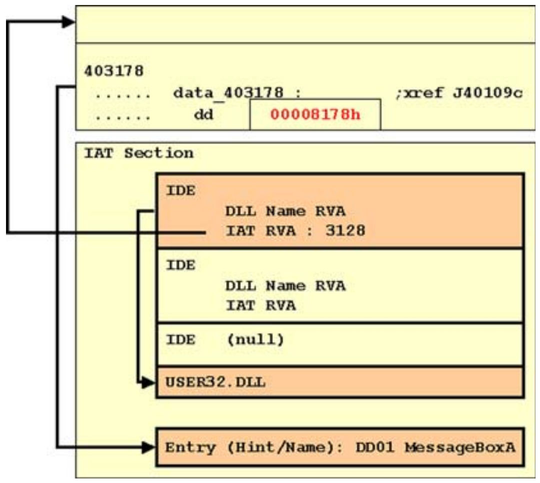
We can compute a provisional value of the first thunk field of the image import descriptor for the User32 DLL. We can observe that the thunks have been initialised by the NT loader with the function virtual address in memory. These values must be replaced with the relative virtual address of the IMAGE_IMPORT_BY_NAME structure in IAT section.



```
401008 !   push    0
40100a !   push    strz_Fenetre_402000
40100f !   push    offset_402008
401014 !   push    0
40101d !   push    0
40101f !   call    wrapper_403204_401168

40109c ! jmp   dword ptr [data_403178]

403178
 ......  data_403178 :      ;xref J40109c
 ......    dd       77d5050bh
```

```
API Hook:

Call to : 77D5050B
USER32.dll::MessageBoxA
Hint     : DD01
Thunk    : 3178

USER32
IMAGE_IMPORT_DESCRIPTOR
FirstThunk <= 3178
```

Fig. 2 Original IAT finding



```
403178
 ......   data_403178 :          ;xref J40109c
 ......      dd       00008178h

IAT Section
   IDE
          DLL Name RVA
          IAT RVA : 3128
   IDE
          DLL Name RVA
          IAT RVA
   IDE    (null)
   USER32.DLL

   Entry (Hint/Name): DD01 MessageBoxA
```

Fig. 3 IAT reconstruction

In order to forge and add a new entry within IAT, we need the following information:

- the name of the function to import.
- Hint. An index into the export name pointer table.

We rebuild a new import directory table by adding a new section to the target executable (see Fig. 3 page 226). The import information begins with the import directory table, which describes the remainder of the import information. The import directory table contains address information that is used to resolve fixup references to the entry points within a DLL image. The import directory table consists of an array of import directory entries, one entry for each DLL to which the image refers. The last directory entry is empty (filled with null values), which indicates the end of the directory table. We can now complete the rebuilding of the IAT.

After the second stage, we have got an unprotected version of the target. Now, the binary can be loaded by the operating system without triggering an error. The resulting binary is a complete runnable executable.

Several heuristics can be applied in order to distinguish the protection and original program control flows. The corresponding problem is to determine where the original entry point of the protected program is. This problem is probably undecidable. But the correlated uses of our integrity check heuristic with several other classic heuristics (intersection jumps, hooks of load-time API functions like in IDA uunp plug-in, etc.) give good results.

If the use of these heuristics fail, the last resort is a human-driven unpacking process.

When driven by a human analyst, the unpacking procedure is often a two step one:

- First, determine the original entry point, given the information of a first controlled execution of the target executable.
- Next, set a virtual break point to the expected original entry point and run the unpacking engine another time.

2.4 What are the limitations?

Unpacking programs run up against several classic limitations:

- Firstly, it is sometimes difficult to reconstruct an executable even after a successful unpacking procedure has been applied.
- Secondly, more sophisticated protection mechanisms can forbid analysis of a binary and thereby ruin an unpacking procedure [10,31]

Our method can be applied efficiently on simply packed executables, but has to be adapted in order to target programs which implement more sophisticated protection mechanisms, like layered ciphering, environmental key/code generation, advanced emulator-detection routines (Hardware Functionality Scan, for example), τ-obfuscation [10].

These limitations apply to any analysis framework based on PC emulation and will be discussed in the second section of this paper (detection methods of an emulated environment by the protected program will be discussed further in the section entitled: How to increase the security of an emulator?).

3 Toward a reliable human analysis framework

In the first section of this paper, we answered the question : how to automatically gain information about a packed Malware? Several algorithms and heuristics which allow us to decrypt its code have been presented. The purpose of this next section is to answer the following question : how to automatically gain accurate information about a malware using stealthy and secure techniques?

Among the information that can be extracted from an executable, the most important is the nature of its interaction with the operating system. Our approach leads us at the very least to be able to trace both Win32 and native API calls. Being stealthy requires going as deep as possible inside the operating system.

How to stealthily hook Win32 and native API? In order to hook Win32 API, several user mode methods apply. The problem is that whichever the method chosen, user mode API hooking is neither stealthy nor accurate.

In order to hook both Win32 and native API [42], better methods rely on driver installation. Kernel mode API hooking is accurate. It is not absolutely stealthy (cf. [19], last chapter).

Using an emulator is a complementary approach to hook API. It is both stealthy and accurate.

In this section wee will firstly present the methods to hook Win32 and native API calls when the target executable is confined inside a virtual machine. We will then discuss security aspects of malware analysis and the techniques which could apply to increasing the isolation of a test platform. Finally, we present our analysis framework and discuss the advantage of focusing on human decision as a detection process.

3.1 Three ways to inject code (stealthily) into a virtual machine

In this section we present several methods that can be used to instrument the execution space of a targeted process. The use of emulation gives a powerful range of possibilities while observing the execution flow of the process. Hooks can be inserted stealthily and code can be injected by using at least three mechanisms:

- forensics shellcode injection
- manually forged intermediate code injection
- the old way (Kernel mode code patching inside the emulator)

Forensics shellcode

A first method to inject code into a virtual machine consists of writing code directly into virtual memory before redirecting the virtual CPU instruction pointer to this memory area.

Argos [20] implements a forensics shellcode injector [Argos-0.1.4/target-i386/argos-csi.c]. An extract of the Argos shellcode injector's source code is given below:

```
#define WIN32SC_RID_OFF 257
#define WIN32SC_LENGTH 293

static char win32_shellcode[] = "...";

void argos_csi(CPUX86State *env, int type, argos_rtag_t *eiptag) {
    ...
    int rid = rand();
    ...
argos_inject_sc(env, type, rid);
...
}
static void argos_inject_sc(CPUX86State *env, int type, int rid){
    ...
    sclen = WIN32SC_LENGTH;
    scp = win32_shellcode;
    scrid = WIN32SC_RID_OFF;
    ...
    vaddr = env->eip & TARGET_PAGE_MASK;
    if ((paddr = cpu_get_phys_page_debug(env, vaddr)) == -1)
        goto address_lookup;
    while (!ARGOS_MAP_ISTAINTED(argos_memmap, paddr) &&
            (paddr & TARGET_PAGE_MASK) != -1)
        paddr++;
    for (tlen = 0; ARGOS_MAP_ISTAINTED(argos_memmap, paddr);
            paddr++, tlen++)
        ;
    if (tlen >= sclen)
        goto code_inject;
address_lookup:
    vaddr = 0;
    paddr = find_page(env, &vaddr, max_vaddr, PG_USER_MASK | PG_RW_MASK,
            PG_USER_MASK);
    if (paddr == -1)
    {
        printf("[ARGOS] Forensics shellcode will not be injected - "
                "No available page found\n");
        return;
    }

code_inject:
    cpu_physical_memory_rw(paddr + (TARGET_PAGE_SIZE - sclen), scp,
            sclen, 1);
    cpu_physical_memory_rw(paddr + (TARGET_PAGE_SIZE - sclen) + scrid,
            (unsigned char *)&rid, 4, 1);
    env->eip = vaddr + (TARGET_PAGE_SIZE - sclen);
}
```

Forensics shellcode is injected, replacing the malevolent shellcode, to gather information about the attacked process. When an attack is detected, OS-specific forensics shellcode is injected. In other words, the code under attack is exploited as the attack is happening to extract additional information about the attack which is subsequently used in signature generation.

After detecting an attack and logging state, forensics shellcode is placed directly into the process's virtual address space. The location where the code is injected is the last text segment page at the beginning of the address space. Placing the code in the text segment is important to guarantee that it will not be overwritten by the process, since it is read-only. It also increases the probability that any critical process data will not be overwritten. Having the shellcode in place, the EIP register is then pointed to its beginning to commence execution.

As an example, shellcode that extracts the PID of the victim process, and transmits it over a TCP connection has been implemented in the Argos project.

Manually forged intermediate code

Another way to inject code into a virtual machine is to inject it at the level of the virtual CPU. We first present in this section principles of virtual CPU design based on dynamic binary translation [67], and then explain how to use it to inject manually forged VCPU code.

Dynamic binary translation allows sequences of code to be translated into native-CPU code *on-the-fly*. Since the native code is cached or even optimised, it can run significantly faster.

The dynamic translator performs a runtime conversion of the target CPU instructions into the host instruction set. The resulting binary code is stored in a translation cache so that it can be reused. The advantage of this compared to an interpreter is that the target instructions are fetched and decoded only once.

Dynamic translators are usually difficult to port from one host to another because the whole code generator must be rewritten. This represents about the same amount of work as adding a new target to a C compiler. QEMU [8] is much simpler since it simply concatenates pieces of machine code generated off-line by the GNU C Compiler.

The first step is to split each target CPU instruction into fewer simpler instructions called micro operations. Each micro operation is implemented by a small piece of C code. This small C source code is compiled by the GCC to an object file. The micro operations are chosen so that their number is much smaller (typically a few hundred) than all the combinations of instructions and operands of the target CPU. The translation from target CPU instructions to micro operations is done entirely with hand-coded code.

A compile time tool called dyngen uses the object file containing the micro operations as input to generate a dynamic code generator. This dynamic code generator is invoked at runtime to generate a complete host function which concatenates several micro operations.

In order to inject code which triggers a page fault in the guest operating system, the author of TTAnalyze [6] describes a method that can be implemented in the following way in QEMU:

Manually forged intermediate code injection is a very interesting way to stealthily inject code into a virtual machine. But unlike forensic shellcode injection, it is difficult to forge complex functions or to re-use existing instrumentation functions.

Finally, we present the old way to inject code reliability into a process. This method applies equally to the context of an analysis framework based on emulation. Even if it is easier for a program to detect the presence of kernel mode API hooks into NT executive internals, the underlying methods are more documented and easier to implement.

Kernel mode code patching inside the emulator

Kernel mode hooking is an interesting third way to inject code into the targeted process. If it is less stealthy than the previously described method, the method is already well documented and has proved its effectiveness against not-too-evolved Malwares, thus in fact most of them.

3.2 Toward a reliable disassembler engine (a way to improve AV pattern search engine?)

The use of emulation is a reliable way of obtaining the disassembled code of a protected program. Most anti-disassembler protections are not accurate against such a dynamic code analysis tool, because the disassembler process is made for each basic block of code. Static analysis tools have to manage the whole executable code as an analysis unit.

Obtaining exact disassembled code of the targeted program execution flow is crucial in order to extract information relative to internals of a virus. It is also important in order to forge a viral set that can be used while black box testing the

```
    uint16_t opc_buf[OPC_BUF_SIZE];      // intermediate code
    uint32_t opparam_buf[OPPARAM_BUF_SIZE]; //
    uint8_t code_buf[4096];              // host code

    int code_size;                       // generated host code size

    if (cpu_memory_rw_debug(env, addr, buf, len, is_write)!=0 && is_write == 0){

        opc_buf[0]=INDEX_op_movl_A0_im;
        opparam_buf[0]=addr;
        opc_buf[1]=INDEX_op_ldsb_user_T0_A0;
        opparam_buf[1]=0;
        opc_buf[2]=INDEX_op_exit_tb;
        opparam_buf[2]=0;
        opc_buf[3]=INDEX_op_end;
        opparam_buf[3]=0;

code_size=dyngen_code(code_buf, NULL, NULL,
                      opc_buf, opparam_buf, NULL);
                     // intermediate code to host code translation

        // TB Execution
        gen_func=(void *)code_buf;
        gen_func();
```

resilience of an Anti-Virus against classic transformations (packing, obfuscation).

Even if we do not always obtain a fully functional version of a virus, after unpacking, we observe that the detection rate of most Anti-Virus programs improves on a viral set when unpacking transformation is applied.

This result is intuitive and confirms the fact that most detection techniques currently implemented by Anti-Virus products can be reduced to pattern matching and that the unpacking process implemented by their emulation engines suffer from limitations.

Using dynamic translation emulation mechanisms inside an Anti-Virus has been studied in [57]. In this study, the author puts forward an algorithm to scan for virus signatures in each translation block during execution. He also defends the possibility of using such an emulator in a real-life Anti-Virus product.

3.3 Why use an emulator as a complementary hooking method?

Using an emulator is a complementary method for understanding the protection mechanisms of a malware. For example, if a user mode API hooker is detected by the targeted process, the security analyst can get useful information from the failure of the analysis : the target implements protection mechanisms at any point in the execution flow, such as integrity control for example.

If using an emulator to trace API call is a powerful method, it suffers from specific limitations, like not being necessarily immune to specific anti-emulation mechanisms. Therefore, a security analyst must have other tools at his disposal in order to bypass all the security mechanisms that a Malware may implement.

3.4 How to increase the security of an emulator?

While analyzing Intel Pentium's ability to support a Secure VMM [37], the authors conclude that current VMMs for the Intel architecture should not be used to enforce critical security policies. Furthermore, it would be unwise to try to implement a high assurance VMM as a type II VMM or CSIM hosted on a generic commercial operating system. Layering a highly secure VMM on top of an operating system that does not meet reference monitor criteria would not provide a high level of security.

Current VMM/CSIMs do not meet high assurance security requirements although some vendors claim security as a feature. For example, some potential problems exists if a VMM is to be used to separate mandatory security levels or networking zones.

In the context of Malware analysis, the objective is to separate and isolate two domains: the controlled guest domain, where the Malware program is executed and isolated and the host system domain. Each of these two domains can involve several VM and physical or virtual network components.

A problem results from resource sharing between virtual machines or between host and guest operating systems. If the host and guest systems have access to the same floppy, USB or CD/DVDROM drive, information can flow from one system to the other.

A similar problem results from support of networking or file sharing. File transfer and information communication are possible in this way between host and guest OS.

Tools that are embedded in the guest OS may cause problems. For example, after installing VMware tools in a guest OS, one feature is the ability to cut and paste between host and guest operating system Desktops.

Many security vulnerabilities emerge due to the lack of assurance available in the underlying host OS. Flaws in host OS design and implementation will render the guest OS vulnerable. To achieve any measure of assurance, a secure host OS is required.

However, VMM vendors try to increase the security level of their products, by creating and enforcing the security functions that are required in order to cover the afore mentioned vulnerabilities (for example, VMware ACE—Assured Computing Environment [68]).

Starting from their security analysis, we have implemented several additional security functions that are mandated in order to increase the security of our analysis framework.

Virtual right management policies

In order to be isolated and secure, an emulator must implement robust device protection, filtering and network quarantine mechanisms.

Network quarantine policy

In order to increase the security of an emulator, a Firewall must be embedded in the virtual machine. A Firewall must be installed on the host system in order to monitor the network communication interfaces between host and guest operating systems.

On-access monitor

An on-access blocker must be installed on the host system in order to prevent an accidental execution and spread of a Malware on the security analyst's Workstation.

Detection of emulated environment by a virus

Because knowing that it is running on a virtual machine is the first step in a viral evasion attack attempt, a secure emulator must simulate as precisely as possible the hardware components and must be resilient against pattern matching recognition algorithms and a hardware functionality scan.

It is very difficult to simulate perfectly the hardware components of an emulated PC.

Other mechanisms can be implemented for a Malware to recognize an emulated environment. Several detection methods are already documented for detecting commercial VMM live VMware or VirtualPC [30,53,74].

Methods can be found to detect CSIM like QEMU, Bochs [12] or Plex86 [50].

New methods have recently been discovered in order to detect VMware, VirtualPC, Bochs, Hydra, Qemu and Xen [34]. According to the author, only CSIM can approach complete transparency. Bochs, Hydra, and QEMU, all suffer from bugs and limitations that allow their detection, but these are problems that are relatively easily fixed.

3.5 Human analysis framework architecture and benchmarks

We describe in this section the design and implementation of our analysis framework and give results obtained by using the current version of this program.

Architecture and implementation

We are currently building a tool (using Lex & Yacc) which, given a set of functions prototypes and structure declarations, can forge the API calls' diverting functions (starting from NtOsKrnl PDB file). This tool can be also used to generate automatically the Hooks code of Detours or any other API Hooking program based on code patching (in user or kernel mode).

For example, given the prototype of a function:

```
NTSTATUS ZwLoadDriver(
    IN PUNICODE_STRING DriverServiceName
);
```

we want to automatically generate the hook code:

```
if (addrMatch("ZwLoadDriver")){
            ulong _addr0;
            wchar_t _arg0[255];
            uint8_t buffer[255];
            readVM(env, (ulong)(vcpu->regs[R_ESP]+4), buffer, 4, 0);
            _addr0=*(DWORD*)buffer;
            readVM(env, (unsigned long int)(_addr0+2*sizeof(USHORT)), buffer, 4, 0);
            _addr0=*(DWORD*)buffer;
            readVM(env, _addr0, (uint8_t *)_arg0, 255, 0);
            fwprintf(logfile, L"[HOOK]  ZwLoadDriver[IN]  ObjectName=%s\n", _{\ul arg}0);
            ZwLoadDriver_WAS_CALLED=1;
    }
```

We have adapted the QEMU emulator in order to implement the core emulation engine of our framework. Other emulators could have been used, such as Bochs, Plex86. We are currently studying the possibility of using a type I VMM like Xen as a core component of our analysis framework (for performance reasons).

The virtual machine embeds a kernel service which communicates through a virtual network interface with the VMM. This communication channel is used to upload the targets binaries into the virtual machine, to start the execution of the main target program and to get information from the kernel which makes it possible to drive the execution of the guest process from the host system.

This information is located in the guest NT executive's tables and structures. This information and the way to get it from kernel mode is well documented in [54,55].

The host system must be protected by a Firewall and by an on-access monitor, in order both to supervise the virtual network communication channel and to prevent the security analyst from making an irrevocable mistake.

The human analysis framework gets information about:

- Win32 and native API calls of the target program
- Sequentially disassembled code of the target program
- Structure of the executable in guest memory (and dynamic comparison with the raw file)

The log file could be given (in the future) in a standardised IDMEF format, in order to facilitate its utilisation by a third party correlation engine or by an Intrusion detection and response system specialized component.

The main program can be used in two console modes:

- Default mode automatically uploads the target executable into the VM, unpacking it and getting required information about its interactions with the guest operating system;
- Interactive mode makes it possible for the security analyst to dynamically drive the execution of the target executable and interact with the VM, by controlling its states.

The interactive mode is mainly used when the target executable forks one of its components as a new process, in order to trace its execution. While entering the interactive mode, the following commands are usable:

```
C:\PARANO> make si
[INFO] System initialisation ...
[INFO] Connection to server (192.168.100.2) ...
P> help
[USAGE] command [args]
      vbp <virtual break point>
      upload <file>
      setoep
      execsuspend <file args>
      setpdb <pid>
      setpeb <pid>
      geteprocess <pid>
      summary
      map <file>
      resumeexec
      unmap
      disconnect
P>
```

The main program can also be used in graphics mode. This mode is useful when the analysis process requires interactions between the user and the target program through a graphical interface.

The same options as in console mode (default or interactive) are available when using the graphical mode.

Benchmarks

In this section we give:

- Several statistical results that encourage the use of an unpacking engine ;
- Several examples of information that can be retrieved by using our human analysis framework.

According to an analysis from AV-Test Team [4,11], over 92% of the Malwares of the WildList 03/2006 are packed. About 30 different packers were used for their tests. They observed that many different packers are used throughout one Malware family to avoid detection. Consequently AV product need to deal with a lot of packers and be prepared to cope with new ones every day. According to their bench tests , nearly all AV products use an unpacking engine, but detection rates are not high enough. Many packers are not detected at all by some AV products. For some AVs, false positive rate is too high: many AV products wrongly flag packed files as certain Malware. AV unpacking engines are not generic enough and have many flaws. When assessing the security of the AV product, this problem must be checked.

A quick analysis with PEiD [49] of the Malware files downloaded from Vx Heavens's virus collection [69], 01/2006 gives the following results:

```
C:\PARANO> make stats
ARM Protector          :      1
ASPack                 :    165
ASProtect              :      4
BJFNT                  :      1
CodeCrypt              :      1
Crunch Bit-Arts        :      4
EXEStealth             :      2
EZIP                   :      2
FSG                    :     42
InstallAnywhere        :      1
InstallShield          :      7
Krypton                :      5
LCC Win32              :     24
MEW                    :      1
Neolite                :     10
PE Crypt               :      2
PE Lock NT             :      1
PE Pack                :     15
PEBundle               :      2
PECompact              :     47
PEDiminisher           :      1
PEncrypt               :      4
PENinja                :     10
PESpin                 :      1
PEtite                 :      6
PEX                    :     12
PKLITE32               :      8
Shrinker               :      2
SoftSentry             :      1
tElock                 :     33
UPX                    :    613
UPXSHiT                :      1
UPX-Scrambler          :      8
WWPack32               :      2
y0da's cryptor         :      8
-------------------------------------
Total                  :   1047
```

These results confirm those obtained by AV-Test Team.

Among information that can be retrieved by using our human analysis framework, you can find information about:

- load-time library functions (IAT)
- dynamically loaded library functions (LoadLibraryA, LoadLibraryExA, LoadLibraryExW, GetProcAddress, etc.)
- modifications made to the registry and interactions with the service manager (ZwOpenKey, ZwSetValueKey, NT Register and services AdvAPI functions, etc.)
- modifications made to the filesystem (ZwCreateFile, NtCreateFile, etc.)
- installation of drivers and interaction between user-land and driver stack (ZwLoadDriver, ZwSetSystemInformation, IFS Filter Drivers FltLib API functions, IOCTL Kernel API functions, SetupAPI Filter drivers installation functions, etc.)
- interaction with the running objects bus through OLE32 API functions calls
- network connections through the Winsock2 API functions, etc.

For example, the use of Super Hidden Files can be easily detected. The following extract of the logs illustrates the use of this method by a virus:

```
[HOOK]    ZwOpenSection[IN]  ObjectName=\NLS\NlsSectionCType
[HOOK]    ZwOpenSection[OUT] NTSTATUS=0000007D
[HOOK]    ZwCreateFile[IN]  ObjectName=Hidden
[HOOK]    ZwCreateFile[IN]  FileAttributes=00000006 |FILE_ATTRIBUTE_SYSTEM
|FILE_ATTRIBUTE_HIDDEN  |
[HOOK]    ZwCreateFile[OUT] NTSTATUS=00000025
```

A Malware can be installed as a device driver and started with ZwLoadDriver or can be loaded directly by ZwSetSystemInformation [42], p.434. The following extract of the logs illustrates the use of this method by a well known rootkit:

```
[HOOK]    FindResourceA[IN]  lpType=MIGBOT
[HOOK]    FindResourceA[IN]  lpName=BINARY
[HOOK]    FindResourceA[OUT] HRSRC=00407048
[HOOK]    ZwCreateFile[IN]  ObjectName=\??\C:\MIGBOT.SYS
[HOOK]    ZwCreateFile[IN]  FileAttributes=00000000 |
[HOOK]    ZwCreateFile[OUT] NTSTATUS=00000025
[HOOK]    GetProcAddress[IN]  hModule=7C910000
[HOOK]    GetProcAddress[IN]  lpProcName=RtlInitUnicodeString
[HOOK]    GetProcAddress[OUT] ProcAddress=7C9112D6
[HOOK]    GetProcAddress[IN]  hModule=7C910000
[HOOK]    GetProcAddress[IN]  lpProcName=ZwSetSystemInformation
[HOOK]    GetProcAddress[OUT] ProcAddress=7C91E729
[HOOK]    ZwSetSystemInformation[IN]
          SystemInformationClass=SystemLoadAndCallImage(38)
[HOOK]    ZwSetSystemInformation[OUT] NTSTATUS=000000F0
```

The following packers have been used in our tests:

- Armadillo
- ASProtect
- PEtite
- UPX
- yC

The following table gives the characteristics of the unpacking procedure:

Packer	Version	Mode	VBP	IAT	Time
Armadillo	4.05	GI	–	–	300
ASPack	2.12	–	R	–	150
ASProtect	1.23 RC4	GI	R	–	200
PECompact	1.56	I	–	–	70
PEtite	2.3	–	R	–	90
PolyCryptPE	2.1	I	–	–	80
Shrinker	3.4	GI	R	–	100
UPX	1.24w	–	–	–	50
Vbox	4.3	I	–	–	80
WWPack32	1.20	–	–	–	60
yC	1.2	I	R	–	70

The five packers were used with the same target program, a Win32 PE executable which displays a small pop-up: Hi!

The characteristics of the unpacking procedure are:

- The name and version of the packer
- Tthe required mode for successfully unpacking the protected executable:
  - (I)nteractive Mode is required for example when the target is made of several binary components
  - (G)raphical mode is often required when the protection interacts graphically with the user
- The status information gives an idea of the difficulty or the completeness of the unpacking procedure:
  - If VBP is checked-(R)equired, that means that it is necessary to put a virtual break point, to indicate the original entry point of the target executable
  - IAT and Reloc fields indicate if information about imported functions and relocations has been fully recovered during the reconstruction process
  - Time needed

4 Conclusion and future works

4.1 What next?

Malwares can be difficult to detect by Anti-Virus products, especially when they operate in the Kernel. Many techniques which are currently used by Anti-Virus or by Forensics tools to detect the presence of a Malware can be implemented in our analysis framework and bring some useful additional information that could help a security analyst make a finer-grained diagnosis:

- File integrity checking: the principle of this approach is to look for an image of the filesystem. A file integrity checker should be run offline against a copy of the drive image, because a Malware that is hiding files by hooking system calls or by using a layered file filter driver will evade this mode of detection. This classic detection method can be applied efficiently within our analysis framework, without the same operational constraints.
  Starting from one or several snapshots of the virtual filesystem, we can easily apply forensic file integrity controls at different stages of a target Malware's life cycle.
- Memory watching: the previous approach will not work if the Malware runs only from memory or if its body is located in the BIOS, for example. The principle of Memory watching is firstly to detect the Malware as it loads into memory. A Malware can use several methods to load itself into memory. By watching the underlying API calls involved , an Anti-Virus can prevent a Malware execution. A lot of equivalent combinations of API calls and critical resources names are possibly involved, and thus matching automatically all the ways a Malware might be loaded in memory is an undecidable problem.
  As it has been demonstrated in the benchmark section of this paper, this detection method is currently implemented by our analysis framework. It focuses on human decision as a behavioural detection process. It can be very useful in order to forge a new Viral signature or to add a rule or a fact in a behavioural intrusion detection system.
- Memory consistency checking: several detection algorithms and heuristics can be implemented in order to identify a hook in memory. Even if there are many places within the operating system and within the processes where a hook can hide, you can define acceptable address ranges of kernel modules in operating system handler tables or check the integrity of the first several bytes of API functions.
  Several methods are described in [19] for identifying a hook in memory.
  These methods can be efficiently implemented in our analysis framework to help the security analyst formulate his diagnosis.
  The following tables and structures must be monitored:
  - Process Import Address Table
  - Driver I/O Request Packet handler
  - System Service Dispatch Table
  - The INT 2E handler in Interrupt Descriptor Table

  The IAT is used when an application uses an API function to import its address.
  IRPs are handled by NT executive and drivers to communicate buffered data or control code to a user-mode program.
  The SSDT is used by NT executive for handling system calls.
  The IDT is used to find interrupt handlers. INT 2E (and SYSENTER) instructions are issued by NtDLL.dll to trap to the Kernel.
  All these tables and structures (and others) can be modified by a Malware to provide stealth or to capture data.
- Memory scanning under Windows NT [56]
- Analysis methods based on execution tracing: Given the fact that hooks cause extra instructions to be executed that would not be called by unhooked functions, the tool PatchFinder [52] can detect API hooks by comparing the number of instructions executed during invocation of several API functions with a clean reference.
  This analysis method could be easily implemented within our framework in a stealthy way.
- Analysis methods based on analysis of hidden files and detection of Registry keys: The tool RootkitRevealer [27] can detect hidden registry entries and hidden files by parsing the registry hives and filesystem at a very low level and without the help of Win32 API, and comparing the result with what is obtained through standard Win32 API calls.
  Such heuristics could be implemented as a plug-in in our analysis framework.
- Analysis methods based on the detection of hidden processes: How to detect DKOM [5]?

4.2 Benefits of a plug-in architecture

Several methods for gaining information about a suspect program have been shown in the previous section. In order to apply them on demand in a program, the best way is to develop them as extensions of the current emulation engine using an adapted API.

We are currently working on a plug-in API which facilitates the validation of binary code analysis methods, like differential binary analysis, for example, that is currently implemented in the IDA Pro framework as a set of static binary analysis algorithms. By using emulation, our dynamic (and interactive) analysis tool could complement the static analysis provided by a tool like IDA Pro.

A plug-in architecture could also be very useful in order to validate learning detection algorithms and the calculation of similarity indices in several statistical detection models.

A lot of statistical models, such as Markov models, have been used in DNA sequence analysis, and can be used in metamorphic virus family analysis and recognition. The implementation of such algorithms on viral sets is technically difficult using a static approach. A dynamic approach, such as the one provided by emulation, is more adapted than a static approach.

Lastly, it might be useful to use existing signature databases, and thus a corresponding pattern matching algorithm (for example, Aho-Corasik pattern matching algorithm [1] implementation of ClamAV [25] and its signature database), at a crucial step in memory, during the execution of a targeted program.

Putting together all this potential use of emulation (operating system internals integrity check, dynamic binary code analysis on a control flow graph level, statistical detection and learning algorithms validation on an intra-procedural level), it is easy to imagine the benefits that could be gained from using plug-in architecture to help the security analyst do his job.

References

1. Aho, A.V., Corasik, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM, **18**(6), (1975)
2. Argos project Retrieved from: https://gforce.cs.vu.nl/projects/argos/, http://www.few.vu.nl/argos/ (2006)
3. Argos Howto: Howto: setting up Argos the 0day shellcode catcher. Retrieved from http://www.few.vu.nl/argos/ (2006)
4. AV-Test.org project: Retrieved from http://www.av-test.org/ (2006)
5. Butler, J.: DKOM (Direct Kernel Object Manipulation, slides). Retrieved from: http://www.blackhat.com/presentations/, (2006)
6. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A tool for analyzing malware. In: proceedings of the 15th EICAR Conference, Hamburg, Germany, 29 April–3 May 2006. In: Broucek, V. et al. (ed.) J. Comput. Virol., EICAR 2006 Special Issue, 2006 (2005)
7. Bos, H.: A personal view on the future of Zero-day Worm Containment (slides) (2006)
8. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proceedings of the 2005 USENIX Conference (2005)
9. Betz, C.: MemParser tool. Retrieved from: http://memparser.sourceforge.net/ (2006)
10. Beaucamp, P., Filiol, E.: On the possibility of practically obfuscating programs: towards a unified perspective of code protection. In: Proceedings of the First International Workshop in Theoretical Virology 2006, Nancy, May 2007, In: Bonfante, G., Marion, J.-Y., (eds.) WTCV'06 Special Issue, J. Comput. Virol. **3**(1) 2007 (2006)
11. Brosch, T., Morgenstern, M.: Runtime packers: the hidden problem. Black Hat 2006 Conference (2006)
12. Bochs: Bochs, the open source IA-32 emulation project. Available at: http://bochs.sourceforge.net/ (2007)
13. Brulez, N.: Anti Reverse Engineering Uncovered. Code Breakers Journal. http://www.CodeBreakers-Journal.com. Previously published at the Honeynet Project, Scan of the Month 33 (2005)
14. Burdach, M.: An Introduction to Windows memory forensic. Retrieved from: http://forensic.seccure.net, September 2006 (2005)
15. Burdach, M.: Digital forensics of the physical memory. Retrieved from: http://forensic.seccure.net, September 2006 (2005)
16. Burdach, M.: idetect, ProcEnum, WMFT tools. Retrieved from: http://forensic.seccure.net, September 2006 (2005)
17. Burdach, M.: Digital Investigation. Retrieved from: http://forensic.seccure.net (2006)
18. Burdach, M.: Finding Digital Evidence In Physical Memory (slides). Retrieved from: http://forensic.seccure.net (2006)
19. Butler, J., Hoglund, G.: Rootkits: Subverting the Windows Kernel. Addison Wesley, ISBN 0-321-29431-9 (2006)
20. Bos, H., Portokalidis, G., Slowinska, A.: Argos: An Emulator for Fingerprinting Zero-Day Attacks. In: Proceedings EuroSys (2006)
21. Cohen, F.: Computer viruses, Ph.D. thesis, University of Southern California (1986)
22. Carvey, H.: Reassembling an image file from a memory dump. Retrieved from: http://sourceforge.net/projects/windowsir (2006)
23. Carvey, H.: Ramdump, lsproc, lspm, ReadPE tools. Retrieved from: http://sourceforge.net/projects/windowsir (2006)
24. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware Normalization, Technical Report, University of Wisconsin, Madison, USA (2005)
25. Clam AntiVirus: Available at: http://www.clamav.net/ (2007)
26. Cloakware: Retrieved from: http://www.cloakware.com/ (2007)
27. Cogswell, C., Russinovich, M.: RootkitRevealer. Available at: http://www.sysinternals.com/ (2006)
28. DataRescue: Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables. Retrieved from: http://www.datarescue.com/idabase/, September 2006 (2005)
29. DataRescue: Using the IDA debugger to unpack an hostile PE executable. Retrieved from: http://www.datarescue.com/idabase/ (2006)
30. Elias: Detect if your program is running inside a Virtual Machine. 14 Mars 2005. Retrieved from: http://lgwm.org (Elias homepage), September 2006 (2005)
31. Filiol, F.: Strong cryptography armoured computer viruses forbidding code analysis: the Bradley virus. In: Proceedings of the 14th EICAR Conference, pp. 210–214 (2005)
32. Filiol E.: *Techniques virales avancées*, IRIS Series, Springer Verlag France, January 2007. An English translation is pending (due mid 2007) (2007)
33. Filiol, F., Josse, S.: A statistical model for undecidable viral detection. In: Proceedings of the 16th EICAR Conference, Budapest, Hungary, 5–8 May 2007. To appear in: Broucek, V. (ed.) Eicar 2007 Special Issue, J. Comput. Virol. **3**(2) (2007)
34. Ferrie, P.: Attacks on virtual machine emulators. In: Proceedings of the 2006 AVAR Conference, Auckland, NZ (2006)
35. Garner, G.M.: Forensic Acquisition Utilities: Dd, md5lib, md5sum, VolumeDump, Wipe, ZlibU, nc, GetOpt. Retrieved from: http://users.erols.com/gmgarner/forensics/, (2006)
36. Garner, G.M., Mora, R.: Kntlist tool. Retrieved from: http://www.dfrws.org/2005/challenge/kntlist.html (2006)
37. Irvin, C.E., Robin, J.S.: Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In: Proceedings of Usenix00 Conference (2000)
38. Josse, S.: How to assess the security of your anti-virus? In: Proceedings of the 15th EICAR Conference, Hamburg, Germany, 29 April–3 May 2006. In: Broucek, V. et al. (ed.) J. Comput. Virol. EICAR 2006 Special Issue, **1**(2) (2006)
39. Josse, S.: Secure and advanced unpacking using computer emulation. In: Proceedings of the AVAR 2006 Conference, Auckland, New Zealand (2006)
40. MackT's ImportREC: Available at: http://mackt.cjb.net/ (2006)
41. Microsoft PE-COFF: Microsoft Portable Executable and Common Object File Format Specification, revision 8.0, 2006. Retrieved from http://msdn.microsoft.com/ (2006)
42. Nebbett, G.: Windows NT/2000 Native API Reference. MTP Press (2000)
43. Newbigin, J.: Dd for Windows. Retrieved from: http://uranus.it.swin.edu.au/ jn/linux/rawwrite/dd.htm (2006)
44. Ollydbg: Available at: http://www.ollydbg.de/ (2007)

45. Ollydbg Plugins: Available at: http://www.openrce.org/download/browse/OllydbgPlugins/ (2007)
46. Pennell, A.: Post-Mortem Debugging Your Application with Minidumps and Visual Studio. NET (2002)
47. Pennell, A.: Minidumps tool (2002)
48. Portokalidis, G.: Zero Hour Worm Detection and Containment using Honeypots. Master Thesis, University of Crete (2004)
49. PE iDentifier. Available at: http://peid.tk (2007)
50. Plex86 x86 Virtual Machine Project: Available at: http://plex86.sourceforge.net/ (2007)
51. QEMU Project: Available at: http://fabrice.bellard.free.fr/qemu/ (2006)
52. Rutkowska, J.: Detecting Windows Server Compromises with Patchfinder 2. Retrieved from: http://www.invisiblethings.org/papers/, September 2006 (2004)
53. Rutkowska, J.: Red Pill... or how to detect VMM using (almost) one CPU instruction. Retrieved from: http://www.invisiblethings.org/papers/, September 2006 (2004)
54. Russinovich, M.E., Solomon, D.A.: Inside Microsoft Windows 2000, 3rd edn. Microsoft Press, ISBN 0-7356-1021-5 (2000)
55. Russinovich, M.E., Solomon, D.A.: Microsoft Windows Internals, 4th edn: Microsoft Windows Server 2003, Windows XP, and Windows 2000 (2004)
56. Szor, P.: Memory scanning under Windows NT. In: Proceedings of Virus Bulletin Conference (1999)
57. Stepan, A.E.: Defeating polymorphism: beyond emulation. In: Proceedings of Virus Bulletin Conference (2005)
58. Schuster, A.: Reconstructing a Binary. Part 1, part 2. Retrieved from: http://computer.forensikblog.de/en/2006/04/reconstructing_a_binary.html (2006)
59. Schuster, A.: Tool MemDump.PL (PERL script). Retrieved from: http://computer.forensikblog.de/ (2006)
60. Schuster, A.: Tool PTFinder.PL (Find Processes and Threads in a Microsoft Windows memory dump, PERL script). Retrieved from: http://computer.forensikblog.de/en/topics/windows/memory_analysis/ (2006)
61. Schuster, A.: Improving list-walkers. Retrieved from: http://computer.forensikblog.de/ (2006)
62. Schuster, A.: Acquisition: dd. Retrieved from: http://computer.forensikblog.de/ (2006)
63. Schuster, A.: Adapting PTfinder to other Versions of Microsoft Windows. Retrieved from: http://computer.forensikblog.de/ (2006)
64. Schuster, A.: Converting Virtual into Physical Addresses. Retrieved from: http://computer.forensikblog.de/ (2006)
65. Schuster, A.: Searching for Processes and Threads. Retrieved from: http://computer.forensikblog.de/ (2006)
66. Schuster, A.: More on Processes and Threads. Retrieved from: http://computer.forensikblog.de/ (2006)
67. Tröger, J.: Specification-Driven Dynamic Binary Translation. Ph.D. Thesis from Queensland University of Technology, Brisbane, Australia (2004)
68. VMware ACE: Available at: http://www.vmware.com/products/ace/ (2007)
69. VX Heavens Virus Collection: Retrieved from http://vx.netlux.org/ (2006)
70. Weariless: Performing a hex dump of another process's memory. Retrieved from: http://www.codeproject.com/, September 2006 (2003)
71. Weariless: MDump tool. Retrieved from: http://www.codeproject.com/, September 2006 (2003)
72. y0da's LordPE: Available at: http://y0da.cjb.net (2007)
73. Z0mbie: Automated reverse engineering: Mistfall engine. Retrieved from: http://vx.netlux.org/, September 2006 (2000)
74. Z0mbie: VMWare has you. Retrieved from: http://vx.netlux.org/, September 2006 (2001)