Virt-ICE: Next-generation Debugger for Malware Analysis

Nguyen Anh Quynh, Kuniyasu Suzaki
National Institute of Advanced Industrial Science and Technology, Japan
Email: (nguyen.anhquynh,k.suzaki)@aist.go.jp

Abstract

Dynamic malware analysis is an important method to analyze malware. The most important tool for dynamic malware analysis is debugger. However, because debuggers are originally built by software developers to debug legitimate software, they have some significant flaws against malware. First of all, malware can easily detect the presence of debugger with various tricks. Another fundamental problem is that because malware run in the same security domain with debugger, they can potentially tamper with the debugger, and prevent it from functioning correctly. Unfortunately, all of the above drawbacks are unfixable in the current architecture.

This research presents a new debugger named *Virt-ICE*, which is designed to address the problems of current malware debuggers. Using virtualization technology, Virt-ICE is invisible to malware, thus renders most available anti-debugging techniques useless. Thanks to the isolation provided by virtual machine, Virt-ICE is out of the reach of malware, and cannot be tampered with. Another advantage of Virt-ICE is that unlike many other popular debuggers, it can deal with ring-0 code, therefore it has no issue handling kernel rootkits. Virt-ICE also offers a novel event-based method to intercept malware execution, which can help to improve the debugging efficiency. Finally, Virt-ICE includes some built-in automatic malware analysis facilities to give the analysts more information on malware, so they can reduce the time on the job by focusing their debugging efforts on important points.

1 Introduction

1.1 Malware Analysis Methods

Understanding what the malware is doing internally is always the headache for security professions. Two main methods are proposed, and each offer unique features.

- **Static analysis**: This method disassembles the malware binary to analyze it, without running it. An advantage of static analysis is that it can inspect all the execution paths of the malware. However, it has some major problems. One is that most malware are packed and using various obfuscated tricks to make the binary code very hard to understand. As a result, the analyst must unpack and de obfuscate the malware before actually diving into analyzing it. This procedure usually takes a lot of time, and requires advanced skills. Besides, some malware activities are only visible at run-time, for example by interacting with environment. Consequently, static analysis cannot give the analyst the full understanding on the malware.

- **Dynamic analysis**: This method observes and analyzes malware by executing them. Dynamic analysis can choose the right time to perform analysis, for example after the malware already unpacked itself. As a result, dynamic analysis suffers less on packing problem. This method can also defeats code obfuscation or polymorphic code by monitoring malware behaviour. However, a major problem of this approach is that we can only analyse the execution path exposed when the malware run, thus might miss other paths.

  In general, dynamic analysis is still a favourite method to inspect malware, because it is much faster and requires less effort to understand malware internals.

While each of these two methods have their benefits and drawbacks, they complement each other and should be combined to simplify the job of analyst. This research tries to address the outstanding problems of dynamic malware analysis tools, focusing on debuggers.

1.2 Problems of Debuggers against Malware

While dynamic malware analysis can rely on monitoring malware behaviour, a lot of details on malware internals can only be revealed thanks to a debugger. In principle, this approach runs malware under the watch of the debugger, and the analyst can put breakpoints or watchpoints at important parts inside the malware to closely inspect it at run-time.

Unfortunately, debuggers suffer many inherent problems when dealing with malware. The main reason of all of these issues is that debuggers are originally designed to debug legitimate software, but not against the malware trying to evade them, or even tamper with them. Some major flaws of debuggers against malware are summarized as follows.

- **Debugger detection**: To debug the malware, debugger must put breakpoints and watchpoints into malware process. Windows provides some subsystems at several layers to support debugging, and also provides some official APIs for debugger to handle debugged events.

  Unfortunately, debugging facilities are not originally designed to be invisible against malware, so it is trivial for the malware to self-detect that it is being debugged. Specifically, malware can use the below methods to detect debuggers ([11], [12]).

  1. **Detect debugger's usage**: Usually debugger uses the service provided by Windows to perform debugging. Malware can easily detect that it is being debugged by using some Windows APIs, such as *IsDebuggerPresent()*, *NtQueryInformationProcess()*, *CheckRemoteDebuggerPresent()* or *OutputDebugStringA()*, or having second process to debug itself, thus effectively prevent analyst to attach his debugger to the malware.
  2. **Detect debugger's impact on malware**: To debug malware, the debuggers use use the facilities provided by hardware architecture: on Intel machine, that is to write *INT3* instruction (opcode *0xCC*) into process's code, or points the hardware breakpoints at the place of interests. However, hardware breakpoint method is severely limited because Intel architecture has only four hardware registers, thus no more than 4 breakpoints can be established at the same time. Besides, it is trivial to detect that hardware breakpoints have been set (and therefore, it is being debugged). Software breakpoint can support unlimited number of breakpoints, but it replaces malware code with breakpoint instruction (opcode 0xCC), thus can also be detected by malware performing self-integrity-check on its code at run-time.
  3. **Detect debugger's presence**: Besides above two methods, malware can easily search for the presence of debuggers in the system. They can either looks for specific windows of debugger, or check for the existence of special devices, registries, or even hidden backdoors, setup by debuggers in system.

- **Tampering with debugger**: To debug the malware, debugger must execute the malware, thus give malware a chance to tamper with debugger when it detects the debugger. To avoid this problem, the analyst must be carefully inspect the malware to nullify its malicious actions. Again, this step might take a considerable time and skills to make sure malware does not cause any damage to the debugger as well as the working environment.

The worst part of our concerns is that there is no way to completely solve the discussed issues with the current design of the debuggers. The fact that the debugger and malware operate in the same security domain also make all the problems unfixable. Consequently, analysts must use many work-around techniques to manually deactivate the anti-debugging tricks in malware, before they can do the real job of inspecting its internals. However, the number of tricks have grown quickly, frequently updated, and become more complicated. Again, to deal with new tricks, analyst must invent novel solutions to defeat them, and once more, we see the same never-end mouse-cat game, as in all other parts of computer security.

The rest of this paper is organized as follows. The next section introduces a novel debugger named *Virt-ICE*, our solution to the discussed problem. Section 3 evaluates our debugger. The related work are summarized in section 4, and we conclude this paper in section 5. In the last page, we attach an appendix presenting few commands of Virt-ICE, so readers can imagine how it works in reality.

2 Solution

Our research proposes a novel approach to solve the discussed problems of malware debuggers. We design and implement an interactive debugger named *Virt-ICE*. This section presents the goals of Virt-ICE, its approach, then focuses on its architecture and design.

2.1 Goals

Guided by the above motivation, Virt-ICE aims to achieve the following goals.

- **Invisible to malware**: By making the debugger invisible to the malware being debugged, we can fix all the problems on detecting debuggers. If malware does not see the debugger, it would go on functioning normally, rather than behave differently or bail out, which usually happens currently. As a result, the analyst does not need to perform manual inspection to disable the anti-debugging procedures any more.

- **Tamper-resistant against malware**: By making Virt-ICE tamper-resistant, we can make sure that it is not affected by malware, and always function reliably and correctly as expected.

- **Ring-0 malware**: Ring-0 malware, such as kernel rootkits, are getting more popular. This kind of malware is significantly harder to deal with (than ring-3 malware) because it runs in the same level with Operating System, thus can use various tricks to defeat a range of analysis tools. Therefore, it is important that Virt-ICE must be able to analyze them, without being tampered by them.

2.2 Approach

Virt-ICE takes advantages of virtualization technology. We execute to-be-inspected malware inside the virtual machine (VM), and Virt-ICE outside of the VM, in the low emulator layer. Figure 1 presents the high level design of Virt-ICE.

Running malware inside VM offers some natural benefits:

- **Sandboxing**: Malware is limited by the sandbox created by VM, and cannot cause problems to the physical system. We can also put network firewall to monitor and nullify its damage to the outside networks.

- **Time saving**: After analyzing malware, we can reverse the VM to the checkpointed-stage, thus clean up the VM to work with another malware. This method saves us a lot of time and effort when we analyze multiple malware.

Operating at the emulator layer, Virt-ICE takes the advantage of the dynamic translation feature of VM to monitor the malware running inside the VM. From there, by instrumenting the VM at instruction-granularity level, we can achieve all the proposed goals above, as follows.

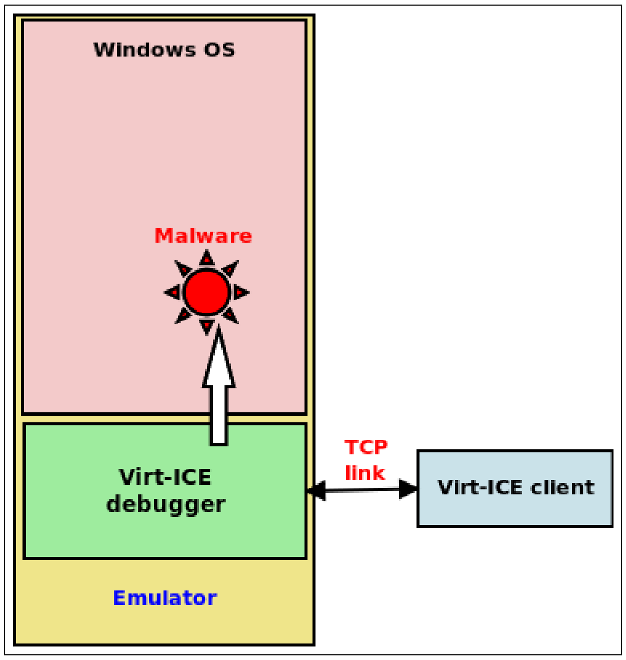

Figure 1: High-level design of Virt-ICE debugger

- **Invisible to malware**: Because Virt-ICE runs outside of the VM, it is not visible to the malware inside the VM, even if the malware runs at kernel level. Besides, because we instrument the dynamic translation code of the VM, but not having to use any software or hardware breakpoints to intercept malware execution, Virt-ICE never modify the malware process or in any place in the VM, as happened with traditional debuggers. Consequently, our debugger becomes transparent, and most anti-debugging techniques fail to work against Virt-ICE.

- **Tamper-resistant against malware**: Naturally, Virt-ICE runs in a different domain of security with the malware inside the VM, so by design, malware cannot attack Virt-ICE.

- **Ring-0 malware**: Dynamic translation allows us to dynamic instrument any place in the VM, from bottom layer (ring-0) to application layer (ring-3) of the VM. As a result, Virt-ICE can debug all kind of malware, no matter it operates in userspace or kernelspace.

It is also noted that using VM to perform malware analysis already become de-facto method in industry, so the fact that Virt-ICE requires VM to work does not cause any negative change to the current procedure of analyst.

2.3 Other Benefits

By instrumenting VM at instruction level, Virt-ICE can also offer debugging events to the analyst, and analyst can achieve a new level of control to the whole VM. As a result, debugging can be done a new way, that is *event-based*. This novel approach offers some interesting features not available to all the other traditional debuggers, such as followings.

- **Before and after action**: It is now possible to set breakpoints or watchpoints so that they are triggered either before or after an instruction is executed, or an area of memory is accessed.

- **Physical-memory access**: It is possible to monitor access to a physical area of memory. This makes the difference, because other debuggers usually only support virtual memory monitoring (again, the reason is because normal debuggers are not originally built to fight malicious tricks). This feature can help to monitor some advanced malware using the dual-mapping evasion techniques [14]

- **Interrupt-based event**: Analyst can choose to intercept interrupt events, so he can know when a particular syscall (using *INT 0x2E* in older version of Windows) is executed. This low-level of information is useful to detect many tricks evading syscall-monitoring methods, employed by various malware.

- **Instruction level**: This feature allows to trigger Virt-ICE when a particular instruction is executed. For instance, recent Windows might use *SYSENTER* instruction to execute syscall, so similarly to above, this is helpful to detect many anti-monitor techniques at syscall level. Another example is that it is now easier to know when a next branch instruction (*JMP*, *CALL* or *RET*) is executed, without having to find and set breakpoint on a particular place on binary code.

- **IO Port-based event**: It is possible to set Virt-ICE to trigger when an access to IO port (read or write) happens. This is useful to monitor some advanced malware manipulating hardware IO ports.

- **Other events** Other interesting events such as changing of *CR3* or ring-level switching are available. The analyst can take advantage of these events to intercept when a particular process is running, or when kernel code of malware is executing. Debugging at ring-0 is therefore becoming much easier.

2.4 Architecture and Design

2.5 Challenges

Virt-ICE is not without some significant problems that we must overcome to realize its design. The most notable challenges are:

1. **Instrumenting VM**: The only emulator that naturally provides instrumentation mechanism is Bochs. We considered Bochs as our emulator, but then rejected it because Bochs has too many shortcomings: it has limited supports to Windows (several Windows OS-es cannot be installed on Bochs due to some reasons), and it is unacceptably slow that it becomes unusable for normal usage. For other emulators, we must build the instrumentation framework ourself, and this is not a trivial task, especially if we want minimize the performance penalty caused by instrumentation.

2. **Performance**: Though performance is not the most concerned issue for malware analyst, it is still always desired to have a good performance system. Instrumentation at the instruction level might degrade the performance of the whole VM to unacceptable degree. It is really tough to design the instrumentation framework and application on top of that to mitigate this problem.

3. **Understanding OS semantics of debugged VM**: From outside, Virt-ICE can access to all the physical memory as well as VM's context like registers and so on. However, these raw data does not reveal OS-semantics information, like which processes are running, or which kernel modules are loaded in the VM at the time. Unfortunately, to be able to debug the malware running inside the VM, Virt-ICE must understand all of these information as well as the malware. On the other words, it must have good visibility on the OS as if it is running inside the VM, even in fact Virt-ICE operates outside. Because our debugger operates in a different context with the VM, we cannot rely on the VM's OS to provide these information for us, but we have to parse the raw data to get the semantic information ourself. This is the classic ”semantic gap” problem in virtualization research [13].

2.6 Implementation

We considered various open source VMs, and choose Qemu version 0.12.4 [9], as the emulator to build Virt-ICE on top of it. Qemu is open source, so we can easily add instrumentation framework to it. Besides, because Qemu's dynamic binary translation is done completely using a software-based JIT compiler, performing expensive action such as instruction tracing does not cause significant performance problem to the VM, if compared with other hardware-based VMs such as Xen [6] or KVM [3]

We solved the three major challenges above with some special designs as follows.

1. **Instrumenting VM**: We build an instrumentation framework named *Kobuta* on top of Qemu. Kobuta instruments the dynamic translator at the right places to put its hooks. The framework provides a range of APIs to outside, so we can build applications on this framework to have access to the instrumentated hooks. The instrumentation is done at instruction level, on special events like interrupts and task switching. Kobuta framework also provides instrumentation to memory access, at both physical and virtual memory level.

   To make it easier for applications built with Kobuta, we make it a shared library, named *libkobuta*, with public APIs provided in a header prototype of C code (*kobuta.h*). Kobuta also provides a whole framework, including a set of Makefile templates, skeleton code and standardized exported functions, to ease the job of building a Kobuta-enable application, called *Kobuta module*.

2. **Performance**: Qemu performs dynamic binary translation using an JIT compiler, thus if done properly, instrumentation does not degrade the VM performance much. Kobuta is carefully designed to mitigate the performance overhead as much as possible. We avoid bottle-neck places, and take advantage of caching mechanism of Qemu. For example, we exploit its *softmmu* mechanism to instrument memory access events: The watchpoints are put deep inside the TLB caching code path, so we can avoid the unnecessary translation from virtual memory to physical memory on every memory access. Execution breakpoint is handled is a similar way.

   In fact, the vanilla Qemu has very acceptable performance, so that it is really practical for the job of malware analyst. We also take advantage of KQemu [8], a popular Qemu accelerator vastly improve the VM speed. The idea is that we use KQemu to complement Kobuta whenever possible: if instrumentation is not required, we disable Kobuta and turn on KQemu to be benefit from its native code execution pace. Vice versa, we dynamically turn off KQemu and switch on Kobuta when instrumentation is requested.

   Since KQemu was dropped from Qemu version 0.12, we had to forward-port it to version 0.12.4, with some minor fixes to make it work again, as well as cooperate with Kobuta.

3. **Understanding OS semantics of debugged VM**: For Virt-ICE to be able to parse the VM's raw data to extract out the semantics information, firstly it must be able to access to the physical memory of the VM and the VM's context. We achieved this by refactoring the Qemu code, combine the related part into libkobuta and exports some APIs for the above functions.

   Once having access to VM's context and physical memory, Virt-ICE accesses to the OS-semantics information thanks to a framework named *EaglEye*. This framework parses the raw memory, and uses the VM's context to extract out all the OS's semantics information such as processes, kernel modules, syscall table, exported DLLs, open registries, ... The implementation of EaglEye relies on various available research about Windows internals by other researchers. EaglEye can locate these OS components from physical memory, then extract out the righ fields in OS structures. Relying on the service provided by EaglEye framework, Virt-ICE can easily access to debugged VM from outside, with good visibility as if it is running inside the VM.

One more challenge we had to deal with is that Qemu is not really designed to be thread-safe: it has some serious bugs on concurrent access on physical memory. This is a problem for our architecture, because Virt-ICE needs to access to VM's memory at the same time with the VM. We fixed the bugs by introducing few patches to handle the concurrency issue.

To take advantage of Kobuta framework, Virt-ICE is built as an Kobuta module on top of the shared library libkobuta. At run-time, on-demand, Virt-ICE is loaded into the Qemu's process. To support loading external code into Qemu, we extended Qemu to handle Kobuta modules, as well as the Qemu monitor interface with a new command *kmodule* to load, unload and reload Kobuta module from external files (The Kobuta modules must be built with the Kobuta framework mentioned above to be accepted with kmodule command). Figure 2 presents the connection between the Qemu emulator, Kobuta, EaglEye and Virt-ICE.

When running inside Qemu, Virt-ICE module opens a TCP channel to outside. The analyst use a *Virt-ICE client* to connect to Virt-ICE module via this channel to interact with the debugger functions provided in Virt-ICE module. Figure 1 presents the link between Virt-ICE module, Virt-ICE client and the debugged VM.

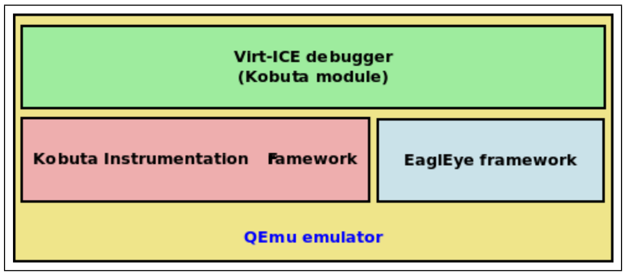

Figure 2: Architecture of Virt-ICE modoule

When get the requests on debugging from Virt-ICE client, Virt-ICE module locates the malware thanks to EaglEye, then set the trigger events with Kobuta. When the desired event is triggered, information is returned to the client. At that time, Virt-ICE module also pauses the VM for the client to come to inspect the VM's internal for information. All the inspected commands are also sent to Virt-ICE module, and Virt-ICE perform the job (mostly with the help of EaglEye) on behalf of the client, then sends back the result to the client. When the client is done with inspection process, it tells Virt-ICE module to resume the VM and let the malware continues to run inside the VM, until the next debugging event is triggered.

3 Evaluations

We evaluated Virt-ICE against various malware employing anti-debugging techniques presented in [11] and [12]. The result shows that Virt-ICE is invisible to all of them, except for the *execution timing* trick, using *external timer*.

Basically, the execution timing technique relies on the fact that when analyst debugs malware, it takes time to analyze indivisual instructions. This delay can be measured using several time sources, like followings.

- **Local timer**: malware can use *RDTSC* instruction, or APIs functions such as *GetTickCount()*, *timeGetTime()*. However, while this trick is effective against all the traditional debuggers, it fails on Virt-ICE. The reason is that when analyst examines the malware inside the VM - typically after a debugging event happens - Virt-ICE pauses the whole VM, thus effectively stops all the local clocks. The analyst can take as much time to inspect the malware as he wants to. After that, Virt-ICE resumes the VM - and also the local clocks, to let the malware continues to run. The malware - and even the VM - is not aware of the suspend with the system, thus cannot measure the delay caused by the debugger.
- **External timer**: malware can use external clock, such as Network Time service (using NTP protocol [5]) to measure the delay, without relying on system timer. Unfortunately, all the debuggers, including Virt-ICE, suffer against this trick. The analyst must manually disable the time checking procedure in malware code before continuing his work.

This research proposes to run debugged malware inside VM to inspect it. Unfortunately, it is known that some malware can detect the VM environment, then either refuse to run, or behave differently [10]. Though the amount of these malware are only around 3% [7], this is still a headache for analyst.

While we are well aware of this problem, we consider it the issue of all debuggers, not only of Virt-ICE. Indeed, it has become a common practice for malware analyst to analyze malware inside VM, and all debuggers suffer when having to deal with anti-virtualization malware. The analyst must either manually nullify the VM checking procedure of malware, or choose alternative VM that is known not vulnerable to the malware.

However, notice that the visibility of VM and visibility of debugger are completely different problems, and should be solved separately by another research. The fact is that while malware can detect the VM, it cannot detect the presence of the Virt-ICE hiding behind the VM. Bottom line, this issue makes no difference to the transparency of Virt-ICE.

4 Related Works

All the debuggers that are commonly used to debug malware such as IDAPro, OllyDbg, Immunity Debugger, Windbg and SoftICE suffer from the problems we discussed in section 1 of this paper: they can be easily detected, and tampered with by the malware. IDAPro, Immunity Debugger and OllyDbg can only analyze ring-3 malwere, while Windbg and SoftICE can be used to inspect kernel level malware.

Most of these debuggers have some methods, provided in the shape of plugins or scripts, to defeat anti-debugging. But as discussed earlier, this approach might need manual inspection, and it still requires some special efforts to deal with upcoming anti-debugging tricks.

Some debuggers try to fix a part of the visible problem of debugger with new approaches. One example is Obsidian [4], which avoids to use the OS service for debugging: it does not use INT3 instruction, but to write a loop-equivalent instruction into the debugged process to set breakpoint. Unfortunately, while this trick defeats the detection method (1), Obsidian still badly suffers from methods (2) and (3), presented in section 1.

Readers might wonder if it is possible to solve the problem by using the remote-debugging features of

Windbg. Indeed, Windbg and Visual SoftICE allows to perform debugging on one machine from another machine, and the connection between two machines can be established using serial port, firewire port or network protocol. In this case, the debugger can avoid the tampering problem, because it cannot be attacked by the malware through the connection between two machines. Unfortunately, this scheme still requires an agent running inside the machine being debugged, or need special bootup configuration, thus clearly disclose the presence of debugger.

Another choice to fix the malware debugger problem is to use the built-in debuggers available in VM such as Bochs, Qemu and VMWare. These VMs provide some primitive debugging features from emulator layers, and can be used to inspect the VM at run-time. The problem is that these debuggers only offer very simple, inconvenient facilities, and only provide raw information, without any OS-semantic at all. Moreover, they do not offer advanced features such as event-based triggers like Virt-ICE.

Other projects such as Anubis [1] and BitBlaze [2] also use Qemu to analyze malware. However, their work focus on automatic malware analysis, but not on problems of interactive debuggers like Virt-ICE.

5 Conclusions

Virt-ICE is the next-generation debugger, built to fix some outstanding problems of current malware debuggers: it is invisible against malware, and cannot be attacked by malware being debugged. Virt-ICE provides rich feature sets not available in other debuggers, such as event-based trigger and API monitoring ability. All of these advanced choices vastly ease the jobs of malware analyst, and give them new methods to make their job done more efficiently.

References

[1] Anubis: analyzing unknown binaries. `http://anubis.iseclab.org`.

[2] Bitblaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu`.

[3] Linux kernel virtual machine. `http://wiki.qemu.org/KQemu/TechDoc`.

[4] Obsidian: non-intrusive debugger. `http://deneke.biz/obsidian`.

[5] Rfc 958 - network time protocol. `www.faqs.org/rfcs/rfc958.html`.

[6] Xen virtual machine monitor. `http://www.xen.org`.

[7] U. Bayer, I. Habibi, and D. Balzarotti. A view on current malware behaviors. In *2nd Usenix workshop on large-scale exploits and emergent threats - LEET' 09*, 2009.

[8] F. Bellard. Qemu accelerator module. `http://wiki.qemu.org/KQemu/TechDoc`.

[9] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, 2005.

[10] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *DSN-DCCS*, 2008.

[11] N. Falliere. Windows anti-debug reference. `http://www.symantec.com/connect/articles/windows-anti-debug-reference`.

[12] P. Ferrie. Anti-unpacker series. `http://pferrie.tripod.com`.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[14] skape. Using dual-mappings to evade automated unpackers. `http://uninformed.org/?v=10&a=1`.

Appendix

Virt-ICE client provides a console interface to user. This section presents some Virt-ICE commands. (Lines with # are comments).

```
# From Qemu's monitor, load Virt-ICE module into Qemu
monitor> kmodule load /opt/kobuta/virt-ice.km
# From the host, run Virt-ICE client to connect to Virt-ICE module
%> virt-ice
# In Virt-ICE client, list all the running processes inside the Windows VM
vice> ps
# List all kernel modules currently loaded in the Windows VM
vice> kmodules
# List all DLL files open by process with pid 134
vice> dlls -p 134
# List all registries open by process with pid 134
vice> registry -p 134
# Stop the VM when malware.exe is loaded into memory, before it is executed
vice> db -S -p malware.exe
# Set breakpoint at WriteProcessMemory() function
vice> db -p malware.exe -s WriteProcessMemory
# Set watchpoint at a particular address
vice> db -p malware.exe -s 0x788120d0
# Resume the VM to let the malware to continue to run.
# The VM will pause and inform Virt-ICE client when it is hit a breakpoint,
# or a watchpoint
vice> db -r
# Disassemble a range of memory in malwre.exe process
vice> disasm -p malware.exe -s 0x7468fc00
# View a range of memory (256 bytes) in hexa and ascii mode
vice> view -p malware.exe -s 0x7468fc00 -c 256
# Search for a string (with regular expression format) in memory of malware
vice> search -p malware.exe -s 0x7468fc00 -c 0x2000 -a "??SICE"
# Get the strings of the whole memory of malware.exe, and pipe out
# to less to review the result
vice> strings -p malware.exe | less
# Dump a range of memory in malware.exe process
vice> dump -p malware.exe -s 0x7881024 -e 0x7985c00 -f
# Monitor and output all the calls to Windows registry API, without interrupting
# the VM. The list of registry APIs is put in file api.reg
vice> db -M api.reg
# Run malware.exe in single-step mode
vice> db -S
```

Figure 3: Sample commands of Virt-ICE debugger