Introducing Pathogen: A Real-Time Virtual Machine Introspection Framework

Anthony Roberts, Richard McClatchey, Saad Liaquat, Nigel Edwards, Mike Wray

HP Laboratories
HPL-2013-55

Keyword(s):

Security; Monitoring; Introspection; Malware

Abstract:

In recent years, malware has grown extremely rapidly in complexity and rates of system infection. Current generation anti-virus and anti-malware software provides system protection through the use of locally installed monitoring agents, which are dependent upon vendor generated signature and heuristic based rules. However, because these monitoring agents are installed within the systems they are trying to protect, they themselves are potential targets of attack by malware. Pathogen overcomes this issue by using a real-time system monitoring and analysis framework that utilises Virtual Machine introspection (VMI) to allow the monitoring of a system without the need for any locally installed agents. One of the main research problems in VMI is how to parse and interpret the memory of an executing system from outside of that system. Pathogen's contribution is a lightweight introspection framework that bridges the semantic gap.

External Posting Date: August 21, 2013 [Fulltext] Approved for External Publication
Internal Posting Date: August 21, 2013 [Fulltext]
To be published in CCS 2013: 20th ACM Conference on Computer and Communications Security (ACM CCS)

POSTER: Introducing Pathogen: A Real-Time Virtual Machine Introspection Framework

Anthony Roberts, Richard McClatchey, Saad Liaquat
University of the West of England
Bristol, BS16 1QY, UK
anthony2.roberts@live.uwe.ac.uk
saad2.liaquat@uwe.ac.uk
richard.mcclatchey@uwe.ac.uk

Nigel Edwards, Mike Wray
Hewlett-Packard Laboratories
Long Down Avenue
Bristol, BS34 8QZ, UK
<firstname.lastname>@hp.com

ABSTRACT

In recent years, malware has grown extremely rapidly in complexity and rates of system infection. Current generation anti-virus and anti-malware software provides system protection through the use of locally installed monitoring agents, which are dependent upon vendor generated signature and heuristic based rules. However, because these monitoring agents are installed within the systems they are trying to protect, they themselves are potential targets of attack by malware. *Pathogen* overcomes this issue by using a real-time system monitoring and analysis framework that utilises Virtual Machine introspection (VMI) to allow the monitoring of a system without the need for any locally installed agents. One of the main research problems in VMI is how to parse and interpret the memory of an executing system from outside of that system. *Pathogen's* contribution is a lightweight introspection framework that bridges the semantic gap.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information**]: Security and Protection

Keywords

Security; Monitoring; Introspection; Malware

1. INTRODUCTION

The overall goal of the *Pathogen* project is to develop a system that can be used to monitor multiple virtual machines (VMs) within an organisation's infrastructure. Currently, *Pathogen* is in an early stage of development, and supports the monitoring of a single VM.

Motivations. Current generation security products rely upon the installation of an in-system monitoring agent which performs real-time analysis of file and code execution within the system. However, these monitoring agents are themselves at risk of malware attack, and many pieces of sophisticated malware attempt to identify, and disable any detected security software upon infecting a system [2]. *Pathogen* overcomes this issue by performing all system analysis externally using Virtual Machine Introspection [9] and therefore remains completely transparent to the monitored system.

At present *Pathogen* monitors Microsoft Windows 7 x64 systems, however it is being developed to be as OS agnostic as is possible. This is achieved by performing an offline forensic analysis of the VMs hard disk image, prior to starting the VM. This forensic analysis first identifies the OS version, and then locates core OS files, which are then analysed and used to obtain the correct operating system structure definitions (symbols) directly from Microsoft's Public Symbol Server [6]. This offline forensic analysis is designed to eliminate the so called *"semantic gap"* [7].

The semantic gap refers simply to the fact that when performing external analysis, the view of the monitored system is raw binary data, which must then be reconstructed. In-system monitoring agents are able to draw semantic knowledge directly from the OS's APIs, however, external monitoring agents have no access to these APIs, so must acquire semantic knowledge through other methods. This problem is exacerbated by the need for regular system updates, which change the layout of key OS data structures.

Related Work. To retain access to OS APIs so that semantic knowledge can be obtained, the *Secure In-VM Monitoring (SIM)* [8] project installs an in-system monitoring agent, but protects this agent against malicious modifications by utilising Intel's hardware assisted virtualisation technologies (Intel-VT) [4]. Whilst this approach aims to protect the monitoring agent from tampering, the authors state that it is still potentially possible for malware to detect the presence of the agent, and therefore alter its behaviour.

The *Blacksheep* project [1] attempts to detect malicious actions within systems by monitoring groups of similar (homogeneous) systems, and then comparing memory snapshots taken from each to detect unknown, or unexpected system behaviour. This approach of monitoring multiple similar systems to detect irregularities is unique, however makes assumptions that systems are uninfected prior to beginning analysis.

VMITools [10] & *Volatility* [11] are two of the most commonly used forensic tools for performing analysis of a sys-

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2512518.

tem's memory (either via snapshots or through VMI). However, whilst these tools are widely used, they both perform system analysis using hard-coded symbolic information, meaning that the software must be manually updated in the event of an OS update.

Contributions. The contributions made by *Pathogen* are two-fold:
(1) The creation of a lightweight VMI framework. *Pathogen* is currently being developed on Linux based operating systems, and utilises the open-source system emulator *QEMU* with additional current generation virtualisation hardware assistance provided by *KVM*. Introspection is made possible by making minor modifications to both of these systems.
(2) Semantic-gap reconstruction through offline forensic analysis. To perform introspection correctly, it is essential that the correct semantic knowledge be applied for the particular OS version. *Pathogen* ensures this by analysing the VMs hard disk image prior to performing introspection. After analysis has been completed, the Microsoft public symbol server is contacted, and the relevant Program Database File (PDB) is downloaded. PDB files contain specific debugging information for use in the event of a program or system crash, but more importantly they also contain the exact OS structure definitions required for accurate introspection.

2. IMPLEMENTATION

The sequence of events taken by *Pathogen* begins with the initial forensic analysis of the VMs disk image. Once this analysis is complete, the introspection portion of the system is ready to be used. The offline analysis of the VMs hard disk image serves two purposes; first it allows for the location of core OS files and the generation of the correct symbols for use by the introspector. Secondly, during introspection *Pathogen* monitors all VM disk activity (reads/writes), and by using the recorded file offsets can determine which files the OS is currently accessing.

2.1 Emulator Modifications

To perform live introspection, *Pathogen* must first locate within the host system the virtual memory allocated to the VM for use as its own emulated physical memory. During the creation of the VM, QEMU allocates a single region of host memory for use by the guest. This memory region is then made available to *Pathogen* through the use of the Linux *mmap* command, which presents the guests memory as a standard file within the host system.

2.2 Hardware Virtualisation Technologies

Hardware Virtualisation Technologies (HVT) allow VMs to run at speeds similar to physical systems by allowing the VM direct access to the physical CPU within the host system. This is in direct contrast to older generation virtualisation technologies in which a VMs CPU required significant emulation in software. HVT also reduce the interaction between the VM and the hypervisor (KVM). HVT introduce two core concepts that *Pathogen* utilises in order to perform introspection.

1. **VMENTER/VMEXIT:** Certain events that occur within a VM (e.g. page faults) require the intervention of the underlying hypervisor, which is triggered through the vmexit instruction. A vmexit passes control of execution to the hypervisor, which performs the necessary operation and returns control to the VM with the associated vmenter instruction. *Pathogen* makes modifications to KVM's vmexit handler to detect changes to the VMs memory structures.

2. **Virtual Machine Control Structure (VMCS):** A VMCS is a 4KB structure used to maintain transitions between vmexits and vmenters. In the event of a vmexit, both the host's and VM's CPU states are recorded so that they can be restored upon the following vmenter. *Pathogen* utilises existing VMCS management functions present within KVM to extract VM CPU state information, such as the values of key Model Specific Registers (MSR). MSR are a form of control register within the x86 instruction set and are used to store OS configuration data, such as kernel memory locations and flags to enable/disable hardware features.

At present *Pathogen* performs introspection on VMs using Intel-VT HVT, however future implementations for AMD-V HVT [3] are planned.

2.3 Windows 7 x64 Introspection

After generation of the required OS symbols through offline analysis, introspection begins with the location of the Kernel Processor Control Region (KPCR), a structure containing kernel specific data and pointers to additional control structures. Upon creation of the KPCR, the OS stores its physical address within the *GS-MSR* (*FS-MSR* on 32-bit), and KVM is able to detect its creation through a modification made to KVM's vmexit handler, which monitors the *GS-MSR* on every vmexit during the system's early boot process. Once KVM has detected the creation of the KPCR, its physical address is transferred to *Pathogen* through the use of the Linux */proc* file system.
Pathogen must now locate the OS page tables, which are necessary for performing virtual-to-physical address translation. The base address of the OS page tables are stored within the CPU's *CR3* register, and within a secondary kernel structure; the Kernel Processor Region Control Block (KPRCB). Figure 1 shows the relevant parts of the OS structure layouts from KPCR to OS page table address.

```
KPCR{
0x020 CurrentPrcb _KPRCB
  _KPRCB{
  0x040 ProcessorState  _KPROCESSOR_STATE
    _KPROCESSOR_STATE{
    0x000 SpecialRegisters  _KSPECIAL_REGISTERS
      _KSPECIAL_REGISTERS{
      0x010 Cr3 Uint8B
```

Figure 1: KPRCB structure leading to page table address

Windows 7 x64 uses a standard 4-level page table hierarchy [5], and once located all virtual-to-physical address translations are performed by the *Pathogen* function *walkPageTables*.
To determine the correct offsets of elements within structures, *Pathogen* must first generate an in-memory hashmap

of the necessary structure; for example, to generate the structure of the KPCR, the symbol information obtained from the PDB is parsed, and each element's offset, name, and type is recorded. To display a list of all currently active processes, the structure KDDEBUGGER_DATA64 (KDBG) which is used for OS kernel debugging is located. Prior to the release of Windows Vista, an element within the KPCR named *KdVersionBlock* stored a pointer to the KDBG, however this field is now nullified by the OS during the boot process. The KDBG is stored in a memory location relative to the start of the KPCR, and by performing a memory search through the two pages prior to the start of the KPCR, the KDBG can be located by its header which contains the fixed string 'KDBG'. Upon location of the KDBG, the element 'PsActiveProcessHead' is parsed to locate an OS maintained doubly-linked list of active processes.
Each active process within the OS is referenced through an EPROCESS structure, which contains data relating to the processes current system activity, such as the number of active threads, current memory utilisation, and a unique numeric identifier (PID). As each EPROCESS structure is within a doubly linked list, each contains a link to the next (flink), and previous (blink) process in the list, *Pathogen* is able to parse each active process by following each flink within the list. A partial example of this is shown in figure 2. In addition, *Pathogen* also monitors the currently executing thread, and by comparing what is currently executing with the data obtained from the EPROCESS linked list can identify potentially hidden processes.

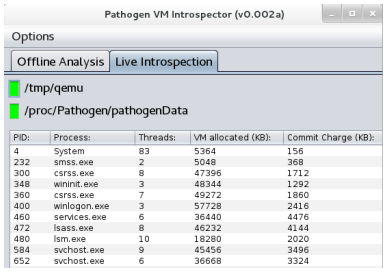

Figure 2: Pathogen showing active processes

3. EXAMPLE MALWARE ANALYSIS

To demonstrate *Pathogen's* current capabilities, a VM running Windows 7 x64 was created and infected with a variant of the *sirefef* malware (TrojanDropperWin32Sirefef.B). After infection, *Pathogen's* log files were examined to identify any symptoms of malicious behavior.
Following execution of the malware, *Pathogen* detected the creation of the files *'msimg32.dll'* & *'InstallFlashPlayer.exe'*, which were followed by a User Access Control (UAC) prompt to allow the execution of the Adobe Flash player installer. Upon confirmation of the UAC prompt, the Adobe Flash player installer was able to execute with administrator level privileges.
To deliver its malicious payload, *sirefef* takes advantage of the way in which Windows locates and loads dynamic library files (DLLs). The legitimate *msimg32.dll* is located in the Windows\System32 directory, however on loading an executable, Windows will first check to see if any DLLs required exist within the same directory as the executing file, and if so, will load the local DLL, rather than performing an additional search. This results in the legitimate Adobe Flash player installer self-injecting the malicious payload contained within the local copy of *msimg32.dll*. In addition, *Pathogen* also detected the creation of the file *'consrv.dll'* which is reported by Microsoft's Security Protection Center to be associated with *sirefef* malware families.

4. CONCLUSIONS

The purpose of this poster is to highlight *Pathogen's* current capabilities in performing VMI whilst narrowing the semantic gap through the use of offline forensic analysis. Current VMI solutions perform introspection using hard coded symbolic definitions which must be updated manually. This issue is further compounded by the frequency of OS updates which alter structure definitions. Now that a reliable VMI framework has been established, the next stage of *Pathogen's* development is to introduce malware detection techniques, along with additional live disk and network monitoring.

5. REFERENCES

[1] A. Bianchi et al. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security. P.341-352.*, 2012.

[2] Microsoft Protection Center. Sirefef malware definition. http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2FSirefef, 2013.

[3] AMD Corporation. AMD Virtualization (AMD-V) Technology. http://sites.amd.com/uk/business/it-solutions/virtualization/Pages/amd-v.aspx, 2013.

[4] Intel Corporation. Hardware-Assisted Virtualization Technology. http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html, 2013.

[5] Intel Corporation. IA-32e Paging. Intel Developer's Manual. Volume 3. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2013.

[6] Microsoft Corporation. Microsoft Public Symbol Server. http://support.microsoft.com/kb/311503, 2013.

[7] Simon. Crosby. Mind the Gap!" The Limitations of VM Introspection. http://blogs.bromium.com/2012/10/01/mind-the-gap-the-limitations-of-vm-introspection/, 2012.

[8] M. Sharif et al. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and Communications Security, P.447-487*, 2009.

[9] T.Garfinkel and M.Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2003.

[10] VMI Tools. VMI memory analysis framework. https://code.google.com/p/vmitools/, 2013.

[11] Volatility. Volatile memory analysis framework. https://www.volatilesystems.com/default/volatility, 2013.