Summer 8-13-2014

Analysis and Detection of Heap-based Malwares Using Introspection in a Virtualized Environment

Salman Javaid
Computer Sciences, sjavaid@uno.edu

Analysis and Detection of Heap-based Malwares Using Introspection in a Virtualized Environment

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Sciences
with Concentration in
Information Assurance

by

Salman Javaid

B.S. Punjab University Lahore Pakistan, 1999
M.S. National University of Science and Technology Islamabad Pakistan, 2004

August 2014

Funding

This work was supported in part by the NSF grant, CNS #1016807

Acknowledgements

I would like to thank my parents, without whom my life would not be possible. I would like to thank my brother for guiding me throughout my education. I would also like to thank my adviser and my thesis committee :-

- Dr. Golden Richard, III – Adviser
- Dr. Vassil Roussev
- Dr. Irfan Ahmed

And finally, I thank the members of my research group because every graduate student needs to do so.

To my parents.

Contents

List of Tables

List of Figures

Abstract

Malware detection and analysis is a major part of computer security. There is an arm race between security experts and malware developers to develop various techniques to secure computer systems and to find ways to circumvent these security methods. In recent years process heap-based attacks have increased significantly. These attacks exploit the system under attack via the heap, typically by using a heap spraying attack. The main drawback with existing techniques is that they either consume too many resources or are complicated to implement. Our work in this thesis focuses on new methods which offloads process heap analysis for guest Virtual Machines (VM) to the privileged domain using Virtual Machine Introspection (VMI) in a Cloud environment. VMI provides us with a seamless, non-intrusive and invisible (to malwares) way of observing the memory and state of VMs without raising red flags for the malwares.

Key Words: Introspection, Cloud Computing, Malware Analysis, Virtual Machines, XEN, LibVMI, Kernel.

Chapter 1

Introduction

In this chapter we will discuss the motivations behind this research. The primary incentive behind this research was to ameliorate the spread of heap spray which has reached to an epidemic proportion. Heap sprays are mostly used to evade various security mechanisms being deployed out there, especially at operating system (OS) and hardware level. Most exploits use some sort of venerability in the application or the OS to revert or overwrite some pointers. Controlling these overwritten pointers is much difficult now, as many security mechanism like DEP and ASLR are being deployed. Heap spray is being used by malware developer as a landing platform for various vulnerabilities especially in commonly used applications like Web Browsers and PDF viewers to bypass these security measures and increasing the probability of the malwares to work.

In the Section 1.1, we are going to discuss how heap sprays have become an important tool in a malware developer. In Section 1.3, we are going to describe various development libraries and platforms that we have used in our research to develop various tools that detect and analyze heap-based malwares and various other type of kernel level malwares and finally in Section 1.4, we list the various contributions we have made in this thesis.

1.1 Heap Spraying

The goal of any attack is to get the targeted computer to run exploit code supplied by the attacker. To achieve this, two things must happen:

1. The code must end up on the computer.

2. The computer must run that code.

Attackers achieve these goals by using various method and techniques. The earliest type of memory exploit took advantage of buffer-stack overflows. Attackers found ways to overwrite a buffer on the stack and used that vulnerability to change or insert program code to make the program jump to instructions provided by the attacker. Stack-overflow attacks diminished in effectiveness as programming languages evolved to prevent buffer overflows. Memory exploits then focused on heap-based overflows, in which, instead of placing instructions on the stack, attackers found ways to insert them into the program's heap. Nowadays, heap-based exploits are more difficult to achieve. Operating systems such as Windows Vista and on-wards use a technique called Address-based Layout Randomization(ASLR) [65] in which the base address of the code, the heap, and the stack change each time the program runs. This prevents attackers from reliably predicting target addresses for code locations, and if there is one copy of the exploit code in a large heap, it's akin to finding the proverbial needle in a haystack.

1.1.1 Heap Spray Techniques

Heap spraying circumvents these challenges by allocating, or "spraying," multiple copies of exploit code to increase the odds of finding a copy in the heap. The attacker can allocate hundreds of thousands of copies of exploit code into the heap. All that's needed is for one random program jump to land on one copy of such code, and a successful attack begins.

Figure 1.1 illustrates a common method of implementing a heap-spraying attack using JavaScript. Heap spraying requires a memory corruption exploit, as in this example, where

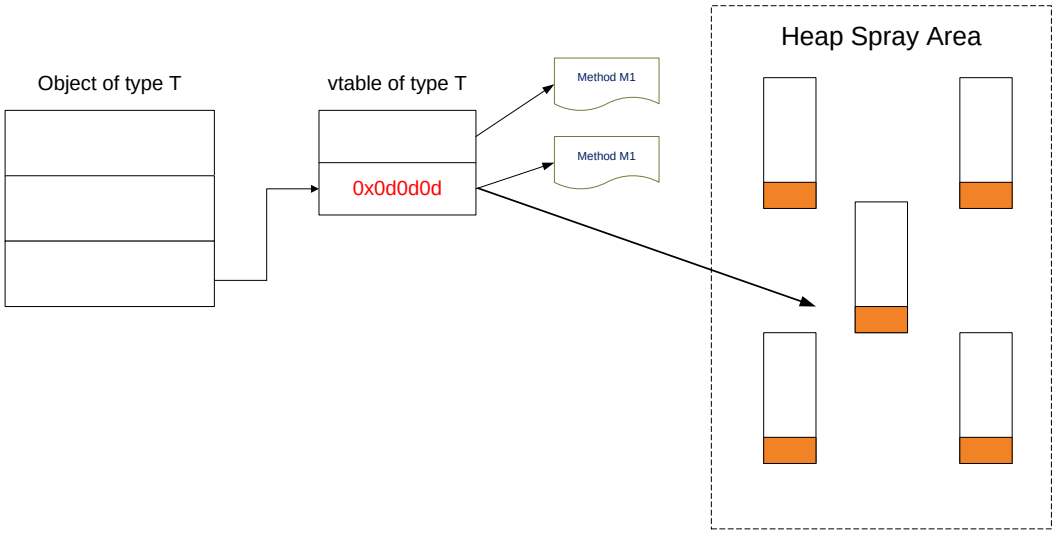


Figure 1.1: Typical Heap Spray

```
<SCRIPT language="text/javascript">

shellcode = unescape("%u4343%u4343% . . . ");

oneblock = unescape("%u0D0D%u0D0D");

var fullblock = oneblock;

while (fullblock.length < 0x40000) {
  fullblock += fullblock;
}

prayContainer = new Array ();

for (i=0; i<1000; i++) {
  sprayContainer [i] = fullblock + shellcode;
}

</SCRIPT>
```

Figure 1.2: Typical Heap Spray Code

an attacker has corrupted a vtable method (M1) pointer to point to an incorrect address of their choosing. At the same time, we assume that the attacker has been able, through entirely legal methods, to allocate objects with contents of their choosing on the heap. Heap spraying relies on then populating the heap with a large number of objects containing the attacker's code, assigning the vtable exploit to jump to an arbitrary address in the heap, and relying on luck that the jump will land inside one of their objects. To increase the likelihood that the attack will succeed, attackers usually structure their objects to contain an initial NOP sled (indicated in white) followed by the code that implements the exploit (commonly referred to as shellcode, indicated with shading). Any jump that lands in the NOP sled will eventually transfer control to the shellcode.

Increasing the size of the NOP sled and the number of sprayed objects increases the probability that the attack will be successful. Heap spraying requires that the attacker control the contents of the heap in the process they are attacking.There are numerous ways to accomplish this goal, including providing data (such as a document or image) that when read into memory creates objects with the desired properties. An easier approach is to take advantage of scripting languages to allocate these objects directly. Browsers are particularly vulnerable to heap spraying because JavaScript embedded in a web page authored by the attacker greatly simplifies such attacks. The example shown in Figure 1.2 shows a basic script how to achieve this. While we are only showing the JavaScript portion of the page, this payload would be typically embedded within an HTML page on the web. Once a victim visits the page, the JavaScript payload is automatically executed. Line 3 allocates the shellcode into a string, while lines 4 to 9 of the JavaScript code are responsible for setting up the spraying NOP sled. Lines 13 to 15 create JavaScript objects each of which is the result of combining the sled with the shellcode. It is quite typical for published exploits to contain a long sled (256 KB in this case). Similarly, to increase the effectiveness of the attack, a large number of JavaScript objects are allocated on the heap, 1,000 in this case.

1.1.2 Just In Time (JIT) Spraying

One of the newest and most popular exploitation techniques to bypass both ASLR and DEP security protections is JIT memory spraying. There are two general reasons why JIT spraying is a very useful exploitation method. Firstly, the code generated by the JIT compiler is stored in memory marked as executable. This should be obvious because otherwise JIT compiler would be unable to work correctly on systems shipped with the DEP feature. Evidently, if the attacker's code is generated by JIT engine it will also reside in the executable area. In other words, DEP is not involved in the protection of code emitted by the JIT compiler. This is a very useful method since the memory was not marked as executable in prior approaches like normal heap spraying. The second reason JIT spraying is powerful is that attacker's code location can be predicted correctly [35] so at this point ASLR is also no longer a big threat for the attacker.

Just-In-Time compilation converts code at runtime; typically from bytecode into machine code. By doing this, an interpreted program's performance greatly improves. The JIT spraying method "forces" the JIT compiler to produce a lot of executable pages with embedded attacker's code. In order to write the code to specific location, the JIT compiler must first mark the destination memory as writable. Since multiple generated code chunks may reside on the same memory page, the JIT compiler marks the entire page as RWX (ReadableWritable-Executable). These permissions are necessary because a different chunk of memory residing on the same page may be executed asynchronously (for example by a different thread), resulting in access violation if the requested memory page was not executable at that moment. After the code is written the compiler marks the destination region as RX - readable and executable not writable anymore. In order to force the JIT compiler to generate code that includes shellcode data, attackers must make use of ActionScript operators.

Even though ActionScript consists of multiple operators like: arithmetic, arithmetic compound assignment, bitwise etc. only one appears to be used in the currently known shellcodes. When it comes to ActionScript operators, only XOR appears to produce desirable

results. For example, with the XOR operator, the attacker controls four bytes of every single instruction. In other cases the expression arguments do not provide precise and predictable control over the emitted code blocks. By supplying different arguments to the expression it is possible to change the contents of specified blocks and make them more dependable on attacker's arguments, but the XOR operator appears to be the best option for shellcode usage and that is probably why every known JIT shellcode makes use of this operator. Once the attacker is able to spray controlled executable instructions into the heap, the rest of the exploitation process goes the standard route. The main idea here is to spray the memory with instructions that include attacker's payload and then be able to transfer the execution there (like for example be able to point the instruction pointer (EIP) to the address of xor eax, IMM32 operand).

1.1.3 Heap Feng Shui

The exploitation of heap corruption vulnerabilities on the Windows platform has become increasingly more difficult since the introduction of XP SP2. Heap protection features such as safe unlinking and heap cookies have been successful in stopping most generic heap exploitation techniques. Methods for bypassing the heap protection exist, but they require a great degree of control over the allocation patterns of the vulnerable application.

Heap Feng Shui [68] introduces a new technique for precise manipulation of the browser heap layout using specific sequences of JavaScript allocations. It uses JavaScript library HeapLib with functions for setting up the heap in a controlled state before triggering a heap corruption bug. This allows malware developers to exploit very difficult heap corruption vulnerabilities with great reliability and precision.There are three main components of Internet Explorer that allocate memory typically corrupted by browser heap vulnerabilities.

- The first one is the MSHTML.DLL library, responsible for managing memory for HTML elements on the currently displayed page. It allocates memory during the initial rendering of the page, and during any subsequent DHTML manipulations. The

memory is allocated from the default process heap and is freed when a page is closed or HTML elements are destroyed.

- The second component that manages memory is the JavaScript engine in JSCRIPT.DLL. Memory for new JavaScript objects is allocated from a dedicated JavaScript heap, with the exception of strings, which are allocated from the default process heap. Unreferenced objects are destroyed by the garbage collector, which runs when the total memory consumption or the number of objects exceed a certain threshold. The garbage collector can also be triggered explicitly by calling the CollectGarbage() function.

- The final component in most browser exploits is the ActiveX control that causes heap corruption. Some ActiveX controls use a dedicated heap, but most allocate and corrupt memory on the default process heap.

An important observation is that all three components of Internet Explorer use the same default process heap. This means that allocating and freeing memory with JavaScript changes the layout of the heap used by MSHTML and ActiveX controls, and a heap corruption bug in an ActiveX control can be used to overwrite memory allocated by the other two browser components.

1.2 Operating System based Security

Techniques have been developed to counter against the heap spray attacks. Heap protection features such as safe un-linking and heap cookies have been successful in stopping most generic heap exploitation techniques. Methods for bypassing the heap protection exist [68], but they require a great degree of control over the allocation patterns of the vulnerable application. We are going to discuss some techniques against heap spray in the Chapter 2. Two major security development that are becoming very popular recently are Address-based Layout Randomization (ASLR) and Data Execution Prevention (DEP) [5].

1.2.1 Data Execution Prevention (DEP)

Buffer overflow attacks, in which an attacker forces a program or component to store malicious code in an area of memory not intended for it, are some of the most common exploits seen today. DEP is a Windows feature that enables the system to mark one or more pages of memory as non-executable. Marking memory regions as non-executable means that code cannot be run from that region of memory, which makes it harder for exploits involving buffer overruns to succeed.

DEP was introduced in Windows XP SP2 and has been included in all subsequent releases of Windows desktop and server operating systems [18]. For application compatibility reasons, DEP is "opt-in" in Windows XP, Windows Vista, and Windows 7. DEP protects the operating system and core system files by default, but application developers or IT administrators must specifically configure other programs to take advantage of DEP. DEP is "opt-out" in Windows Server operating systems, meaning that DEP is enabled by default for all programs unless specifically disabled for a program.

1.2.2 Address Space Layout Randomization (ASLR)

In older versions of Windows (Windows XP and below), core processes tended to be loaded into predictable memory locations upon system start up. Some exploits work by targeting memory locations known to be associated with particular processes. ASLR randomizes the memory locations used by system files and other programs, making it much harder for an attacker to correctly guess the location of a given process. The combination of ASLR and DEP creates a fairly formidable barrier for attackers to overcome in order to achieve reliable code execution when exploiting vulnerabilities.

ASLR was introduced in Windows Vista [65] and has been included in all subsequent releases of Windows. As with DEP, ASLR is only enabled by default for core operating system binaries and applications that are explicitly configured to use it via a new linker switch.

Table 1.1: DEP and ASLR Deployment Status (May 2008) Source [59]

Application	DEP (Win 7)	DEP (Win XP)	Full ASLR
Flash Player	N/A	N/A	NO
Sun Java JRE	NO	NO	NO
Adobe Reader	YES	NO	NO
Mozilla Firefox	NO	NO	NO
Apple Quicktime	NO	NO	NO
VLC Media Player	NO	NO	NO
Apple iTunes	NO	NO	NO
Google Chrome	N/A	N/A	N/A
Shockwave Player	N/A	N/A	NO
OpenOffice.org	NO	NO	NO
Google Picasa	NO	NO	NO
Foxit Reader	NO	NO	NO
Opera	NO	NO	NO
Winamp	NO	NO	NO
RealPlayer	NO	NO	NO
Apple Safari	NO	NO	NO

Table 1.2: DEP and ASLR Deployment Status (June 2010) Source [59]

Application	DEP (Win 7)	DEP (Win XP)	Full ASLR
Flash Player	N/A	N/A	YES
Sun Java JRE	NO	NO	NO
Adobe Reader	YES	YES	NO
Mozilla Firefox	YES	YES	NO
Apple Quicktime	NO	NO	NO
VLC Media Player	NO	NO	NO
Apple iTunes	YES	NO	NO
Google Chrome	YES	YES	YES
Shockwave Player	N/A	N/A	NO
OpenOffice.org	NO	NO	NO
Google Picasa	NO	NO	NO
Foxit Reader	NO	NO	NO
Opera	YES	YES	NO
Winamp	NO	NO	NO
RealPlayer	NO	NO	NO
Apple Safari	YES	YES	NO

1.2.3 Status of DEP and ASLR in Third Party Applications

The importance of DEP and ASLR can be judged by the fact that processor makers like Intel [14] and AMD [3] have adopted their processors to support them directly [59]. Various Operating Systems (OS) also adopted them slowly and now newest version of most of the major OS's out there support them in one way or another [59]. Third party applications were rather slow to adopt them. Table 1.1 and Table 1.2 show the pace of adaption of DEP and ASLR from 2008 to 2010 [59]. It shows that most of the major applications out there are still slow to adopt these security methods and still are vulnerable to the various malwares that were hindered by the these security methods. This tell us that it is really important for the security experts to develop various tools that look for malwares that exploit the vulnerabilities of these third party application to keep them from being exploited by malwares.

1.3 Virtual Machine Architecture

1.3.1 Virtual Machine Monitor

A virtual machine monitor (VMM) is a thin layer of software that runs directly on the hardware of a machine. The VMM exports a virtual machine abstraction (VM) that resembles the underlying hardware. This abstraction models the hardware closely enough that software which would run on the underlying hardware can also be run in a virtual machine. VMMs virtualize all hardware resources, allowing multiple virtual machines to transparently multiplex the resources of the physical machine. The operating system running inside of a VM is traditionally referred to as the guest OS, and applications running on the guest OS are similarly referred to as guest applications. Basic architecture of a VMM can be seen in Figure 1.3.

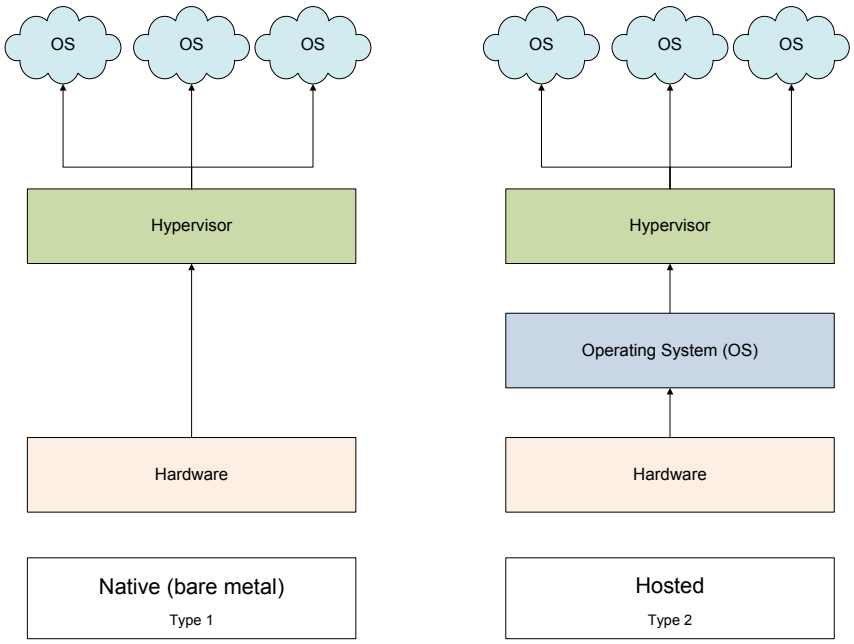


Figure 1.3: Architecture of two basic types of Virtual Machine Monitors

As Figure 1.3 shows, in a virtualized environment, a VMM provides the interface between each VM and the underlying physical hardware. The OS layer between a VMM and the physical hardware is optional, depending on which of the two major types of VM managers you choose. In a Type 1 system the VMM runs directly on the physical hardware, eliminating an abstraction layer and often improving efficiency as a result. Examples of Type 1 systems include VMware ESX [29], XEN [31], and Microsoft Hyper-V [19]. In a type 2 system, the VMM uses an OS as an interface to the physical hardware. Type 2 systems include VMware Workstation [29], the QEMU [26] open source process emulator, KVM [15], Parallels [24], and Virtual PC/Server [28]. Type 2 systems rely on the underlying OS to provide hardware interaction and device drivers, and thus often have a wider range of physical hardware components to interact with.

Traditionally, the VMM is the only privileged code running on the system. It is essentially a small operating system. This style of VMM has been a standard part of mainframe computers for 30 years, and recently has found its way onto commodity x86 PCs. Hosted

VMMs like VMware have emerged that run a VMM concurrently with a commodity "host OS" such as Windows or Linux. In this setting, the virtual machine appears as simply another program running on the host operating system.

Despite a radical difference from the users perspective, traditional and hosted VMMs differ little in implementation. In a hosted architecture the VMM merely leverages a third-party host OS to provide drivers, bootstrapping code, and other functionality common to VMMs and traditional operating systems, instead of being forced to implement all of its functionality from scratch. VMMs have traditionally been used for logical server partitioning, and are supported for a wide range of architectures; for example, the IBM xSeries (x86 servers), pSeries (Unix), zSeries (mainframes), and iSeries (AS/400) all have VMMs available. Recently, as hosted VMMs have appeared on the desktop, they have begun to discover other applications such as cross-platform development and testing.

1.3.2 VMM Implementation

Although the specifics of a VMMs implementation are architecture-dependent, VMMs tend to rely on similar implementation techniques. Among these techniques is configuring the real machine so that virtual machines can safely and directly execute using the machine's CPU and memory. By doing this, VMMs can efficiently run software in the virtual machines at speeds close to that achieved by running them on the bare hardware.

VMMs can also fully isolate the software running in a virtual machine from other virtual machines, and from the virtual machine monitor. A common way to virtualize the CPU is to run the VMM in the most privileged mode of the processor, while running virtual machines in less privileged modes. All traps and interrupts that occur while a virtual machine is running transfer control to the VMM. Attempts by the virtual machines to access privileged operations trap into the VMM; the VMM emulates privileged operations for the VM. In this architecture, the VMM can always control the virtual machine regardless of what the software in the virtual machine does. Memory is commonly virtualized by keeping a virtual MMU

for each virtual machine that reflects the VMs view of its address space. The VMM retains control of the real MMU, and maps each VMs physical memory in such a way that VMs do not share physical memory with each other, or with the VMM. Through this technique the VMM is able to create the illusion that each VM has its own address space that it fully controls. This also allows the VMM to isolate the VMs from one another and prevents them from accessing the memory of the VMM. In addition to virtualizing the CPU and memory, the VMM intercepts all input/output requests from VMs to virtual devices and maps them to the correct physical I/O device. For memory-mapped I/O, the VMM only allows a virtual machine to see and access the particular I/O devices it is permitted to use.

1.3.3 Virtual Machine Introspection

Virtual Machine Introspection (VMI) is a technique for externally monitoring the run time state of a system-level virtual machine. Monitors can be placed in another virtual machine, within the hypervisor, or within any other part of the virtualization architecture. For virtual machine introspection, the run time state can be defined broadly to include processor registers, memory, disk, network, and any other hardware-level events. The introspection library we have used through out this work is the LibVMI [17]. Section 1.3.4 discuss in details the architecture and working of LibVMI library.

1.3.4 LibVMI

We have used libVMI [17] library extensively in our projects. It is an introspection library focused on reading and writing memory from virtual machines (VMs). It is based on the library Xenaccess [58] which was developed as a student project. Initially it only supported Xen [31] but then, support for KVM was added to Xenaccess and Xenaccess was renamed to libVMI. For convenience, LibVMI also provides functions for accessing CPU registers, pausing and un-pausing a VM, printing binary data, and more. LibVMI is designed to work across multiple virtualization platforms (LibVMI currently supports VMs running in either

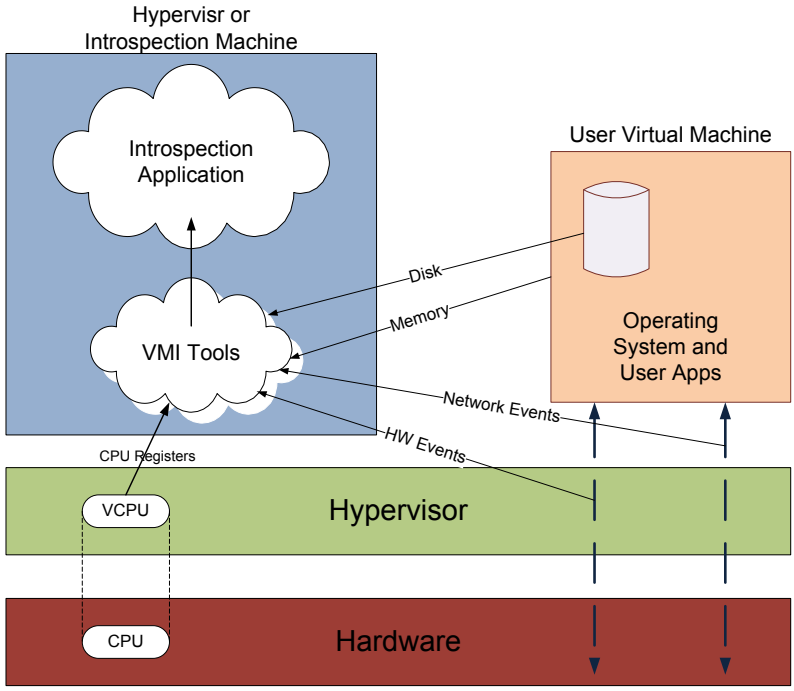


Figure 1.4: Virtual Machine Introspection

Xen or KVM). LibVMI also supports reading physical memory snapshots when saved as a file (like VMware snapshots). It also support memory analysis tools like Volatility [30] to directly access VMs memory but the performance is much slower that native libVMI based programs.

LibVMI architecture is based on six high-level requirements. In a general sense, these requirements can be seen as typical good programming guidelines, or good security guidelines. This is done to leverage known design principles in order to build a robust monitoring architecture. With this in mind following are the six requirements for monitoring VMs.

1. **No superfluous modifications to the VMM:** The VMM should remain as small and simple as possible since it is part of the TCB. If a VMM includes the necessary primitives to support the monitoring architecture, then it should not be modified. If a VMM lacks the necessary primitives, then the modifications made should be what is minimally required to support the monitoring architecture.

2. **No modifications to the VM or the target OS:** Modifications to the target OS (i.e., the OS being monitored) are problematic. The target OS can tamper with this code, and changes to the target OS may require access to the target OS source code, which is not always available. One of the key reasons why virtualization is attractive for monitoring is the isolation between VMs. Placing monitoring code within the same OS that is being monitored bypasses this isolation, negating this key benefit. Therefore, this requirement encourages all monitoring code to remain in an isolated VM unless such a restriction makes it impossible for a monitor to gather the necessary information.

3. **Small performance impact:** An excessive performance impact can render a monitoring architecture worthless. This requirement ensures that the monitoring architecture does not prevent the target OS from performing its intended functions. The performance impact is measured as any reduction in performance of an application caused by the monitoring software. Ideally this impact is both small and consistent, but some initialization costs may be required.

4. **Rapid development of new monitors:** New monitors may be needed to address new types of attacks. Furthermore, it is advantageous to keep the monitor code simple to limit the opportunity for introducing errors into the monitors. The monitoring architecture should provide APIs that are used to develop new monitors. Therefore, satisfaction of this requirement means that the APIs should be designed in a way that simplifies the job of the monitor developer.

5. **Ability to monitor any data on target OS:** Monitors should have a full view into the target OS. The monitoring architecture should not be limited to providing information about a small part of the target OS. For example, an ideal memory monitor should be able to view all memory on the target OS. Likewise, an ideal disk monitor should be able to view all data going to and from the disk device. While this ideal

may not always be possible, the more information a monitor can view, the harder it is for an attacker to evade detection.

6. **Target OS cannot tamper with monitors:** If the target OS can tamper with the monitors, then the possibility exists for malicious code to tamper with the monitors. For this reason, all of the monitors should be isolated or protected from the target OS. This is related to requirement (2), above. However, here it is required that all monitor code, regardless of its location, be protected from attack. If all monitor code is in an isolated VM, then this is not difficult. If some monitor code must be placed outside of the TCB, then additional measures must be taken to protect that code. The extent of these measures will depend on the nature of the code being protected.

This implementation methodology provides an excellent platform for us to develop various malware analysis and detection tools. Although it doesn't fully provide all the aspects required for our research but served as an excellent starting point for us for development of various tools.

1.4 Contributions

This thesis research has been taken from the collaboration which contributed to the fields of malware analysis, computer forensics, Virtual Machine Introspection (VMI) and Cloud Computing.

- We have develop a robust platform Atomizer [45] to analyze the heap of a process running inside a virtual machine in a Cloud environment.
- We developed a library to analyze pages that have been swapped out of the main memory in a VM.

- In our collaboration we developed ModChecker [32] which is a unique and light weight method to verify the integrity of the kernel modules by cross viewing similar virtual machines in a Cloud environment.
- In our collaboration we developed IDTChecker [33] which verify the integrity of important kernel structure like Interrupt Descriptor Table (IDT) to detect the presence of a malware entity.
- Several peer-reviewed research publications have resulted from this collaboration.

1.5 Organization

The organization of the rest of this work is as follows. Chapter 2 describes various publications and works done related to our field of heap-based malware analysis. After discussing some past works we are going to discuss our major work in heap-based malware analysis in the Chapter 3. Chapter 4 provides conclusions and some directions for future work.

1.6 Bibliographic Attributions

This thesis presents the work that was part of the following publication.

- *Atomizer: A Fast, Scalable and Lightweight Heap Analyzer for Virtual Machines in a Cloud Environment, which was presented at the proceedings of 6th Layered Assurance Workshop (LAW'12), In conjunction with the 28th Annual Computer Security Applications Conference(ACSAC'12) on December 2012 at Orlando, Florida*

Chapter 2

Related Work

In this chapter we will describe, in detail, various works done in the field of heap-based malware analysis. Various techniques have been used in many work of literature toward studying and mitigating the effects of the heap based malwares. We will provide a brief introduction of these works in this part of the dissertation. There has been a large volume of work published on heap spray detection. Most of the work focuses on JavaScript analysis and classifying web contents on the basis of dynamic and static features. This chapter covers the existing approaches and tools that are most related to our work. Section 2.1 describe various works in the field of heap-based malware detection and analysis. Section 2.2, Section 2.3 and Section 2.4 describes some major works done by Microsoft [18] in the field of heap-based malware detection and analysis.

2.1 Miscellaneous Work

In this section we are discussing some minor works that are related to heap-based mlaware analysis.

2.1.1 Heap Inspector

LeMasters [51] demonstrates a simple Heap Inspector tool that visualizes heap sprayed NOP sleds by searching for byte patterns that resemble NOP instructions. It does so by injecting a DLL into the process that is being analyzed to create a gateway to the heap object. It further relies on the Windows API to gather the heap data. All this combined not only significantly impacts the performance of the guest system, but also is detectable by malware.

2.1.2 JSAND

Cova *et al.* propose JSAND [37] that provides a framework to emulate JavaScript code and discover the key features that are most commonly found in malware. These features include code obfuscation, environment analysis techniques, and exploitation mechanisms. JSAND uses machine-learning techniques to establish the characteristics of normal JavaScript code and uses these characteristics to detect anomalous code [37]. JSAND is a finger printing technique that can be evaded by a clever malware developer. Techniques like time-based checks and exception handling described in [43] can be used to evade the emulation part of JSAND. Unlike Atomizer, JSAND relies heavily on emulation which can have many limitations [43].

2.1.3 BuBBles

An important property of a heap-spraying attack is that it relies on homogeneity of memory. This means that it expects large parts of memory to contain the same information (i.e., its nop-shellcode). It also relies on the fact that landing anywhere in the nop sled will cause the shellcode to be executed. BuBBle's [40] countermeasure breaks that assumption by introducing diversity on the heap, which makes it much harder to perform a heap-spraying attack. The assumption is broken by inserting special interrupting values in strings at random positions when the string is stored in memory and removing them when the string is used

by the application. These special interrupting values will cause the program to generate an exception when it is executed as an instruction. Because these special values interrupt the strings inside the memory of the application, the attacker can no longer depend on the nop sled or even the shellcode being intact. If these values were placed at fixed locations, the attacker could attempt to bypass the code by inserting jumps over specific possible locations within the code. Such an attack however is unlikely, because the attacker does not know exactly where inside the shellcode control has been transferred. However, to make the attack even harder, the special interrupting values are placed at random locations inside the string. Since an attacker does not know at which locations in the string the special interrupting values are stored he cannot jump over them in his nop-shellcode. This lightweight approach thus makes heap-spraying attacks significantly harder at very low cost.

2.2 Nozzle

Working for Microsoft, Ratanaworabhan *et al.* proposed Nozzle [60] which performs static control flow analysis of parts of the heap, interpreting them as code segments to detect malicious content. Nozzle accomplishes that by intercepting common heap allocating function calls and gathering information about the heap, as well as its content. This technique is browser-specific and could potentially be detected by the attacker. As described by Nozzle, an attacker could time her heap sprays to avoid detection by this mechanism. Scanning heap objects via introspection requires no memory hooks and makes the entire guest operating system oblivious to heap analysis. Nozzle also poses a 10% overhead, which led to the development of an improved version of the programs called Zozzle [38]. The basic architecture of Nozzle is shown in figure 2.1.

Nozzle is a run time monitoring infrastructure that detects attempts by attackers to spray the heap. Nozzle uses lightweight emulation techniques to detect the presence of objects that contain executable code. To reduce false positives, Nozzle uses a notion of

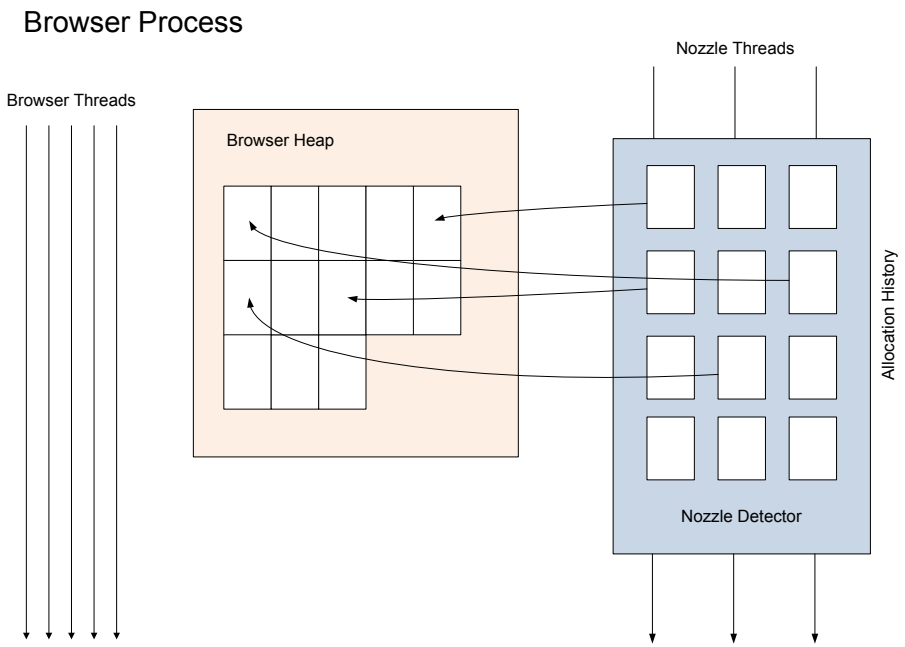


Figure 2.1: Basic Nozzle Architecture

global "heap health". Its lightweight emulator scans heap allocated object data to identify valid $x86$ code sequences, disassembling the code and building a control flow graph. Because the attack jump target cannot be precisely controlled, the emulator follows control flow to identify basic blocks that are likely to be reached through jumps from multiple offsets into the object. Nozzle developed a novel approach to mitigate this problem using global heap health metrics, which effectively distinguishes benign allocation behavior from malicious attacks. Fortunately, an inherent property of heap spraying attacks is the fact such attacks affect the heap globally. Consequently, Nozzle exploits this property to drastically reduce the false positive rate. Nozzle was mostly static method of detecting heap spray attacks. It visited various URLs and tried to detect presence of heap spray based attacks n those URLs. The heavy weight nature of Nozzle made it difficult to be deployed in a running environment.

2.3 Zozzle

Zozzle [38] was the next iteration after Nozzle in the work being done by Microsoft with regards to heap spray attacks. Zozzle is a low-overhead solution for detecting and preventing JavaScript malware that is fast enough to be deployed in the browser. Its approach uses Bayesian classification of hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that are highly predictive of malware. Zozzle is able to detect JavaScript malware through mostly static code analysis effectively. Zozzle has an extremely low false positive rate of 0.0003, which is less than one in quarter million. Despite this high accuracy, the Zozzle classifier is very fast, with a throughput at over 1 MB of JavaScript code per second [38].

Zozzle is designed to perform static analysis of JavaScript code on a given site and quickly determine whether the code is malicious and includes an exploit. In order to be effective, the tool must be trained to recognize the elements that are common to malicious JavaScript, and the researchers behind it stress that it works best on de-obfuscated code. The Zozzle was trained by crawling millions of Web sites and using Nozzle, to process the URLs and see whether malware was present.

Zozzle is specifically designed to detect and defend against heap-spraying exploits launched by malicious JavaScript found on Web sites. In many cases these days, that kind of exploit is hosted on a legitimate site that's been compromised and is being used as part of a drive-by download attack. Often, the code is hosted on a specific page for a day or even a few hours and then is taken down, either by the attacker or the site owner. it's been shown in [38] this, along with the multiple layers of obfuscation that attacker use to cloak JavaScript exploits, can make it difficult for automated tools to identify such malware with a high degree of accuracy.

Once Nozzle has labeled JavaScript contexts, it needs to extract features from them that are predictive of malicious or benign intent. For Zozzle, Nozzle creates features based on the hierarchical structure of the JavaScript abstract syntax tree (AST). Specifically, a feature

consists of two parts: a context in which it appears (such as a loop, conditional, try/catch block, etc.) and the text (or some substring) of the AST node. For a given JavaScript context, it only track whether a feature appears or not, and not the number of occurrences. To efficiently extract features from the AST, it traverses the tree from the root, pushing AST contexts onto a stack as it descends and popping them as it ascends. These features are then used in Zozzle to identify suspicious activity.

Zoozle was created to remove the main drawback in the Nozzle, that was it's heavy weight nature. Although Zozzle was faster than Nozzle but it still lack various feature [49] that allowed various malware to pass by it.

2.4 Rozzle

In recent years, attacks that exploit vulnerabilities in browsers and their associated plugins have increased significantly. These attacks are often written in JavaScript and literally millions of URLs contain such malicious content.

While static and runtime methods for malware detection been proposed in the literature [60] [38] [40], both on the client side, for just-in-time in-browser detection, as well as offline, crawler-based malware discovery, these approaches encounter the same fundamental limitation. Web-based malware tends to be environment-specific, targeting a particular browser, often attacking specific versions of installed plugins. This targeting occurs because the malware exploits vulnerabilities in specific plugins and fail otherwise. As a result, a fundamental limitation for detecting a piece of malware is that malware is triggered infrequently, only showing itself when the right environment is present. In fact, we observe that using current fingerprinting techniques, just about any piece of existing malware may be made virtually undetectable with the current generation of malware scanners.

Rozzle [49] is a JavaScript multi-execution virtual machine, a way to explore multiple execution paths within a single execution so that environment-specific malware will reveal

itself. Using large-scale experiments, it is shown that Rozzle increases the detection rate for offline runtime detection by almost seven times. In addition, Rozzle triples the effectiveness of online runtime detection. It is also shown that Rozzle incurs virtually no runtime overhead and allows to replace multiple VMs running different browser configurations with a single Rozzle-enabled browser, reducing the hardware requirements, network bandwidth, and power consumption.

Chapter 3

Heap-base Malware Detection

In this chapter we are going to discuss in detail our major contribution. We developed a robust tool to detect heap-based malwares using introspection in virtualized environment like Clouds. Section 3.1 discusses the Atomizer [45] which is a novel technique developed to detect malicious activities in the heap of a process.

3.1 Introduction to Atomizer

Atomizer which browses through the heaps of processes running inside VMs and looks for heap-based activities like heap spray. Atomizer runs on a privileged VM and uses VMI to access the heaps of the processes inside virtual machines running on a cloud server. Currently, only the heaps that are allocated by the operating system for the processes can be accessed by Atomizer. Application-generated heaps (e.g., those created by the Java Virtual Machine (JVM)) can also be examined by Atomizer, through available Atomizer APIs. Atomizer requires no changes to be made to virtual machine manager(VMM), VMs and the programs being monitored.

Atomizer's architecture makes it difficult for any malware inside the VM to detect and disable it. Atomizer also monitors all the pages of the heap that have been swapped out of main memory. Atomizer is designed to be modular, so that new features can easily be

added. The most popular type of heap-based attacks are heap sprays [60], that's why in this chapter we are mainly focused on heap spray attacks, although Atomizer can also be used for detecting any heap-based attacks.

Heap Spraying techniques involve instructing client side languages such as JavaScript to allocate large blocks of heap memory (50-200MB) containing malicious shellcode and NOP sleds that "slide" into the shellcode. The final piece of the puzzle relies on overwriting a function pointer to point to a random location within the large NOP sled heap object [2], which typically requires a separate exploit. Large blocks of heap spray can easily be detected by simple scans, however, newer and less intrusive heap spray attacks that more accurately manipulate the layout of the application heap layout increase reliability and precision, without the need for large blocks of heap spray. Techniques like Feng Shui Heap Spray [68] defragments and makes holes in the heap object to insure that the function pointer is readily available and positioned properly for smashing with a heap overflow [39].

Techniques like JIT spraying [35] have been developed to bypass both ASLR and DEP. JIT spraying utilizes knowledge about a JIT compiler's architecture to spray the heap with executable code that can then be compiled by the JIT compiler. The JIT heap spray is constructed large enough to overwhelm and bypass ASLR. JIT spray uses return oriented programming (ROP) [61] gadgets to mark the heap pages as executable so that the contents of the pages can be executed. JIT spray uses a leaked pointer, which is essentially a random heap address, to jump to that location and follows the NOP sled down to the JIT shell code that is executed to exploit the system. Modern exploitation techniques, such JIT spraying, makes it even more important to look for malicious activities inside the heap of a process.

3.2 Atomizer Architecture

The Atomizer infrastructure consists mainly of three components, Process Information Extractor, Heap Extractor and Swapped Heap Page Extractor as shown in Figure 3.1. We

assume a typical cloud computing environment consisting of hardware, a VMM (or hypervisor such as XEN [31]), guest VMs and a privileged VM (where Atomizer gains access the heap memory content using VM introspection). The reason behind the development of a real time out-of-VM heap spray analyzer as a proof of concept that offloading heap spray detection to the hypervisor has insignificant impact on the guest OS.

The first component, Process Information Extractor, is responsible for determining the basic attributes of the process being monitored. These basic attributes of the process include the internal operating system structures that provide us with valuable information about the location of the process in physical memory. The location of the heap is extracted in the second step called Heap Finder. There might be various heaps in the process and the location of all of them are passed on to the next component, Heap Extractor.

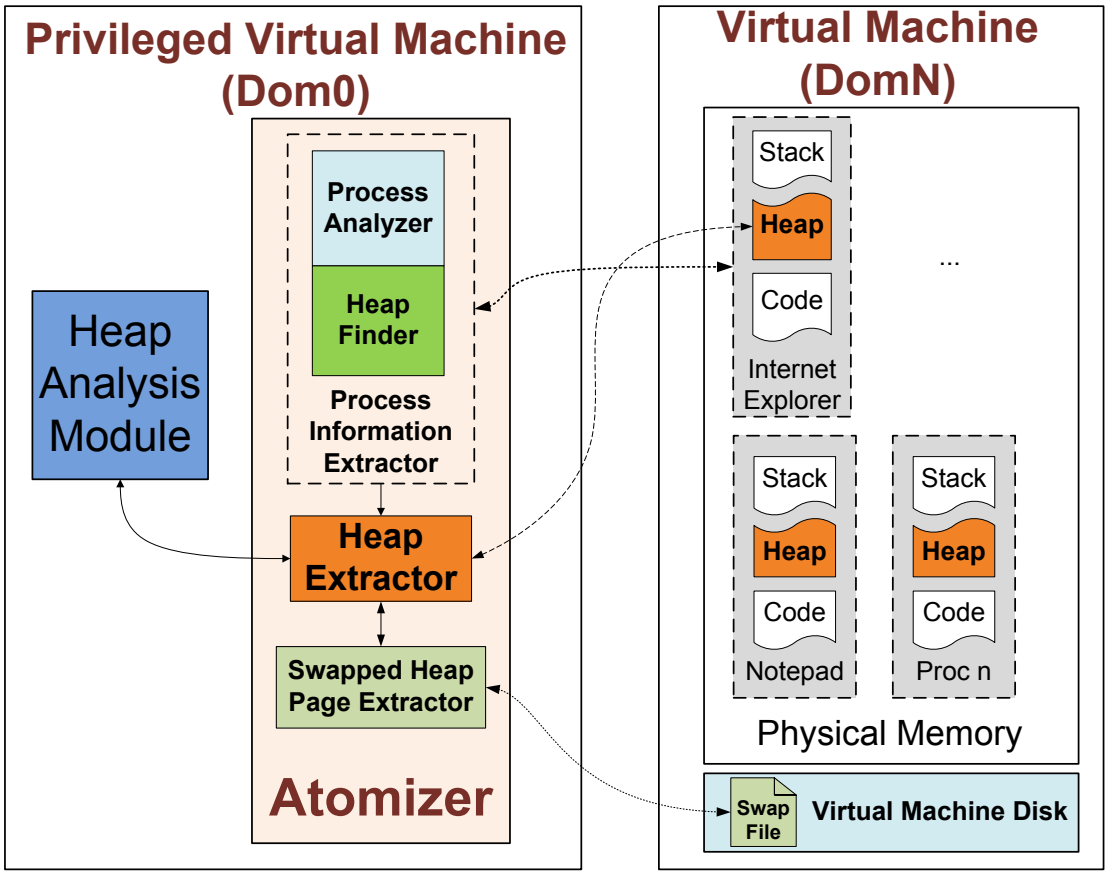


Figure 3.1: Atomizer Architecture

The Heap Extractor component receives the physical memory locations of the heaps from the first component of the Atomizer and extracts all the heap pages of the process. In a typical system some heap pages may be swapped out of physical memory for efficiency, so the Atomizer architecture also has the capability to access pages that are currently swapped out. When the Heap Extractor module encounters a swapped heap page, it uses the Swapped Heap Page Module to access those pages. The Swapped Heap Page Extractor component of the Atomizer uses information about the guest OS' swapping architecture to access swapped pages.

The Atomizer is designed to work in a cloud environment to monitor multiple VMs running in the environment using a single privileged system. The VMI interface provides access to the memory of all the VMs running on a single physical server. The Atomizer can browse through the heaps of all the processes running in these VMs with low performance impact, i.e., without using many of resources of the cloud server. The Atomizer has a modular design which allows adding support for various operating systems. The Heap Analysis Module part is also designed in such a way that adding additional modules to enhance heap analysis is very straightforward. In particular, the Heap Extractor module provides an application programmable interface (API), allowing new heap analysis techniques to use the Heap Extractor API's to access heap memory. These API's can be used to detect any type of heap-based attacks.

3.3 Atomizer Implementation

We developed a proof of concept prototype of Atomizer on XEN [31] that had Microsoft Windows XP (Service Pack 2) VMs running. We used the libVMI library [17] to introspect the heap of web browsers (Internet Explorer or Firefox) running on Windows XP virtual machines. We also used libguestfs [16] to access the page file of memory pages that have been swapped to the hard drive.

3.3.1 Process Information Extraction

Windows operating systems maintain information about executing processes in Process Environment Block (PEB) structures. In this part of the implementation, Atomizer looks for memory address of the PEB structure of the process. It goes through known memory addresses to locate the PEB structure using appropriate PEB signatures [62]. Once the location of the PEB structure is found the information regarding the process is extracted from it. That information includes the location of the heaps, number of heaps available and maximum number of heaps allowed by the OS. This information is then forwarded to the next component of the Atomizer.

Algorithm 1 Heap Memory Browsing using VAD tree

```
for i = 0x7FFD0000 to 0x7FFDF000 do
  if ((5 == i + 0xa4)&&(1 == i + 0xa8)) then
    PEB = i
    break;
  end if
end for
HEAPNUM := PEB + 0x88
HEAPADDRESS := PEB + 0x090
heapCounter := 0
while heapCounter < HEAPNUM do
  HEAPNODE := HEAPADDRESS +(4 * heapCounter)
  segmentCounter := 0
  while segmentCounter < 64 do
    HEAPSEGMENT := HEAPNODE + 0x58 + (4 * segmentCounter)
    HEAPENTRY := HEAPSEGMENT + 0x20
    while (HEAPENTRY + 0x005) ≠ 0 do
      HEAPSIZE := HEAPENTRY
      READ_MEMORY(HEAPENTRY, HEAPSIZE)
      HEAPENTRY := HEAPENTRY + (HEAPSIZE * 8)
    end while
    segmentCounter++
  end while
  heapCounter++
end while
```

3.3.2 Heap Extractor

The Heap Extractor component of Atomizer provides the main facility for browsing through the heap of the process using the virtual address descriptors (VAD) obtained via introspection. The memory manager in Windows maintains a set of VADs that describes the status of the process's address space [62]. To find the VAD tree of a process being monitored, it uses the process environment block structure (PEB) information received from Process Information Extractor. Using this information, it browse through all the heap nodes, heap segments, and heap entries in various heaps of a process. Algorithm 1 describes the algorithm used to browse through the entire heap. If a heap page entry is in the VAD tree and not in physical memory, than there is a possibility that the heap page has been swapped out of memory. As modern heap sprays do not require large amounts of NOP sleds, it is important that those heap pages are also examined. Details of how it access pages that are swapped out are provided in Section 3.3.3. Heap extractor can access the page memory both page by page and byte by byte. This facility has been provided to facilitate various algorithms that might perform better with either of these memory access methods. This implementation uses the page by page access method to extract one page at a time from the heap and send it to the Heap Analysis Module (discussed in Section 3.3.4).

3.3.3 Swapped Heap Page Extractor

When the Atomizer needs to access a swapped out page, it follows a procedure similar to that of the guest OS. We can describe this method as a two-step process where in the *first step*, the Atomizer retrieves the page file number and the page offset and in the *second step*, it uses this information to obtain the value of the virtual address from the page file.

Atomizer prototype is based on Windows XP (SP2), with Physical Address Extension (PAE) support enabled, for implementation. The PAE gives us four levels of virtual address translation where a (32-bit) virtual address (in PAE) contains the pointers to the page directory pointer table, page directory table, page table and page byte offset. Unlike the

Algorithm 2 Simple NOP Sled Detection

```
NOPZ ← HASH-TABLE of NOPs/NOP-replacements
LIMIT ← 150
BUFFER ← Memory buffer from Heap size = SIZE
SKIP ← 1
index := 0
nops := 0
skipped := 0
while index < SIZE do
  if NOPZ[ BUFFER[index++] ] then
    nops++
  else if skipped < SKIP then
    skipped++
  else
    nops := 0
  end if
  if nops == LIMIT then
    NOP sled detected
  end if
end while
```

page directory pointer table, the page directory and page table both have 64 bit entries. PAE supports two page sizes i.e. 4K and 2M (also referred as large page). Depending on the page size (4K and 2M), the page file number and page offset are recorded into the page table or the page directory respectively. If the 7th bit of page directory entry is valid, it means that the entry is pointing to a large 2M page instead of a page table. In order to ensure that an entry in the page directory or page table contains the offset of a page in page file, its 0th (valid bit), 10th (prototype bit) and 11th (transition bit) bits must be zero. Moreover, the offset is present in the higher 32 bits (from 32 to 51 bits) of the 64-bit entry and the page number is present in the lower 32-bit of the entry. In the second step, Atomizer uses the libguestfs [16] library to access the page file of the guest VMs from Dom0 and reads the corresponding page using the page offset information. Atomizer then uses the page byte offset present in the virtual address to access the value of the virtual address. The page byte offset is the offset of the value in the page pointed to by the virtual address.

3.3.4 Heap Analysis Module

The heap memory received from the heap extractor can be analyzed in the Heap Analysis Module. Various algorithms may be used here to determine whether the heap of the process contains malicious contents or not. Some techniques like STRIDE [34] and ECL-Polynop [42] use sequential analysis of network packets to detect NOP sleds in network traffic, rather than in the process heap itself. Using the same model in this implementation, I implemented a simple Polymorphic NOP detection algorithm to detect the presence of NOP sleds in the heap using sequential analysis of the heap data. This implementation uses the Atomizer architecture to sequentially go through the heap memory and compares each byte of data with a hash table of NOPs and NOP replacements. The algorithm keeps tracks of sequences of NOPs/NOP replacements found and if the length of these sequences are beyond a certain limit it raises a flag. The sequence limit used in our experiments was 150 NOPs. This limit was selected because it gave no false positives in our experiments. This simple NOP sled detection algorithm is depicted in the Algorithm 2. This module demonstrates that Atomizer architecture provides a simple way of implementing these kinds of algorithms in the heap analysis module to detect NOP sleds. The presence of NOP sleds typically means that malicious content is present in the heap.

3.4 Atomizer Evaluation

3.4.1 Experimental Settings

For this experiment, we build a simple cloud environment. This test bed featured a Quad Core i7 (2.67 GHz * 8) server with HyperThreading enabled and 18 GB of RAM. This server had a 64-bit privileged virtual machine (`Dom0`) running Fedora 16 (kernel 3.3.2-6) along with Xen 4.1.2 [31]. We instantiated five VM clones (`DomU: Dom1-Dom5`) in Xen from a single 32

bit Window XP (SP2) installation to make sure that all VMs are identical. We also used the introspection library for VMI (libvmi-0.6) [17] and libguestfs [16] in all the experiments.

3.4.2 Malware Detection

To test the effectiveness of Atomizer was tested against various types of heap sprays commonly found on the web, as well as some custom made heap spray programs.

Simple Heap Spray

The first set of experiments involved the Skypher heap spray generator [9], an example of a simple heap spraying technique. Different variants of Skypher were tested on both Internet Explorer and Mozilla Firefox. Atomizer effectively detected the heap spray with no false positives. Atomizer was also tested against another known heap spray attack, Aurora [8]. This types of heap spray demonstrated some simple obfuscation techniques to hide the payload. The payload in Aurora was encrypted using the JavaScript libraries and decrypted at run time. The Atomizer detected the heap spray without any false positives.

Polymorphic NOP Sled Detection

To test the effectiveness of Atomizer against polymorphic NOP sled detection, in the second set of experiments, we created a small application in C that sprayed various polymorphic NOP sleds along with a dummy shell code in the heap. Most of the NOP sled obfuscation tools like ADMutate [1] are designed to evade Intrusion Detection Systems(IDS) by encrypting the packets which contain the heap spray code until it is executed. As we are monitoring the heap, the NOP sled cannot be encrypted in the heap making it the best place for detecting it. Polymorphic NOP sleds use NOPs and NOP replacements to simulate the behavior of random data in the heap. This implementation uses a hash table of up to 118 known NOP replacements (which includes both one and two bytes NOP replacements) to detect the NOP sleds. This hash table represents the most commonly used NOP replacements used by roots

kits like ADMutate [1], which contains the biggest public list of NOP replacements [42]. NOP sleds that might include unknown NOP replacements may not be detected by this implementation but newly discovered NOP replacements may potentially be added seamlessly to our hash table. This implementation successfully detected the polymorphic NOP sleds without any false positives.

Heap Feng Sui

The Atomizer was tested against a state of the art exploit, Heap Feng Shui [68]. Heap Feng Shui is a deterministic heap spray that reduces the amount of heap spraying required for the exploit. Heap Feng Shui uses the `HeapLib` (a JavaScript heap manipulation library) to defragment the heap so that it can align the heap nodes and consequently requires a smaller size of heap spray to execute the exploit. The Atomizer was able to detect this exploit which shows its effectiveness against state of the art exploits.

3.4.3 Experimental Performance Analysis

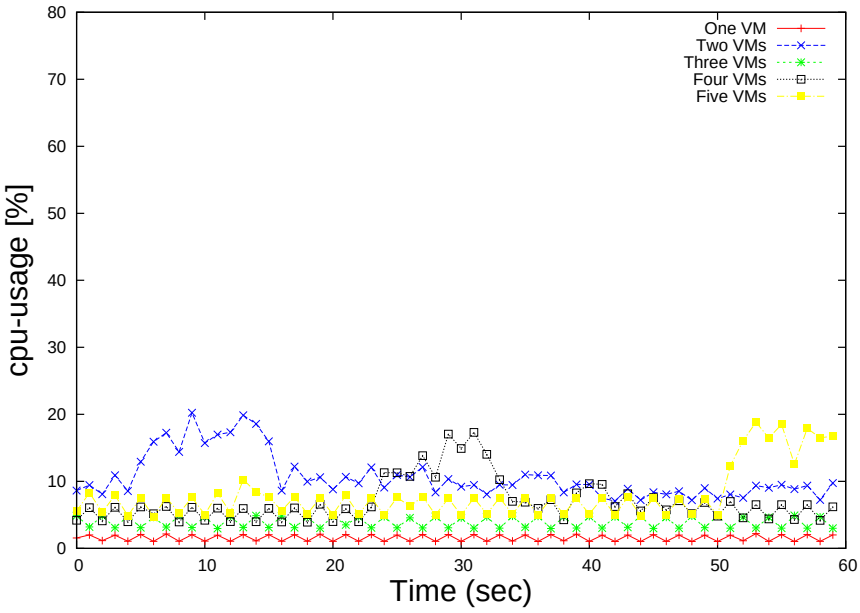
An experimental performance analysis of the Atomizer with respect to CPU resource utilization was done to determine the effect of heap spray detection on the cloud environment. The main purpose of these experiments was to determine the CPU resources used by Atomizer. For performance evaluation purposes in these experiments, the Atomizer was allowed to continue scanning the heap even after the heap spray block was found. For better performance the detector should be stopped when the first sign of malicious activity is raised.

In these experiments, the Atomizer was run on VMs with a simple workload (like a web browser running a YouTube video). This workload also represents the application being monitored by Atomizer during the experiment. The CPU usage was monitored while the systems ran. This was done to set a baseline that shows CPU usage in the normal usage of the VMs. Figure3.2(a) shows the CPU baseline for the VMs.

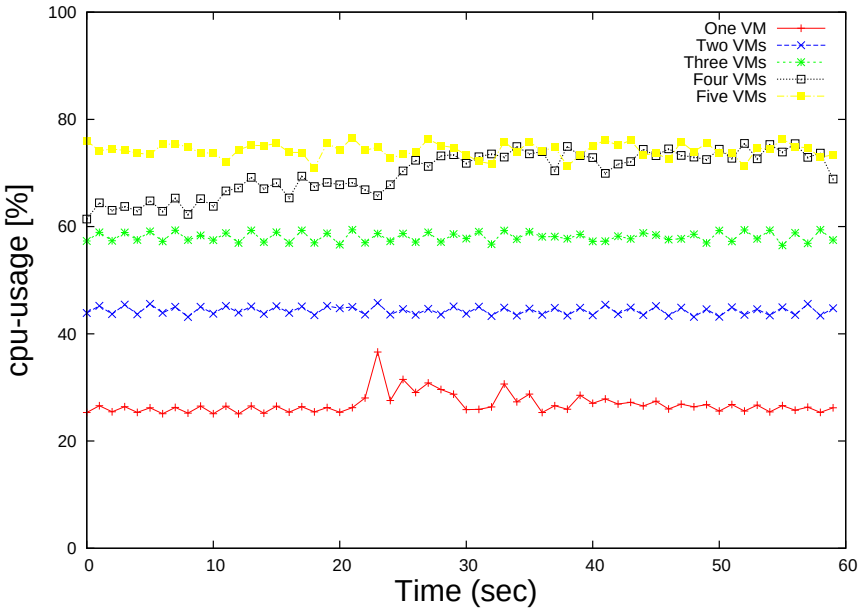
The Atomizer was run as a multi-process application to monitor all the virtual machines running on the server. The current multi-process implementation of Atomizer works around a lack of thread safety in the libVMI library, an issue that we are currently working to address. The Figure 3.2(b) shows the effect of Atomizer on the CPU. The Figure 3.2 shows the cumulative CPU usage of all the Virtual CPU's (VCPU) on the server (in this case there were eight VCPUs). This shows the overall effect of all the applications, including Atomizer and VMs on the CPU usage. Xentop [31] (the tool used in this experiment to measure the CPU usage) adds up the percentage of all the VCPUs available to the system. The rise in the CPU usage is due to the processing needed by the light load running on the VMs plus the effect of Atomizer. The multi-process nature of the Atomizer also exerts more load on CPU. This limitation can be removed by implementing a multi-threaded version of Atomizer, a project that we have currently underway. The average CPU load increased incrementally, that shows that Atomizer has acceptable performance and is scalable.

3.4.4 Conclusion

Atomizer presented a novel and scalable method to analyze the heap memory of the processes running inside a set of VMs, via VM introspection. Atomizer can be easily extended by implementing new detection methods for any type of heap-based attacks. Experimental results show that Atomizer successfully detects various heap spray attacks and randomly generated polymorphic NOP sled samples with no false positives. Further work is required to improve the performance via a multi-threaded implementation of Atomizer.



(a) 1–5 Idle VMs running



(b) 1–5 VMs with light load & Atomizer

Figure 3.2: CPU Performance (CPU usage in *Dom*0)

Chapter 4

Conclusion

In this thesis we have discussed various tools developed by us that can enhance our ability to detect and analyze various heap-based malware and kernel level exploits. Our tools provide robust ways of detecting the malwares in Cloud based environment. This ability to detect malware at a cloud level is unique to our tools and distinguishes them from all other tools out there. The reason behind this assumption is that they are specially designed to work in an environment where multitudes of VMs are running at the same time. The similarity between the VMs created by the notion that it is simple to deploy similar VM at the same time give us a unique opportunity to observe the changes that can be created by malware entity in that environment. We use this opportunity to investigate these changes using our tools. These changes can be sometimes quite striking like a nop-sled in the heap or can be subtle like a small change in the heap like Heap Fang Shui [68]. These changes can be difficult to find because they are cleverly crafted to be hidden from the naked eye by hiding them behind the regular working of the Operating System or the Application they are targeting. Our tool Atomizer [45]is designed specifically to look through the regular working environment of the heap of process or kernel memory space (that also includes kernel heap) to look for signs of these malware using introspection. The regular working working environment of any system can be really chaotic and our tools are designed to look for certain aspect of

the malwares that are only specific to them. The large scale deployment of our tools is also a unique feature. As they are designed to monitor a Cloud environment they can be used to observe large number of VMs at the same time without putting unnecessary load on the Cloud Servers and with out making huge changes to the code. This provides a great feature for our tools to be deployed quite easily in an environment like Clouds, which was designed to use the physical power of the machines to the maximum by deploying large number of VMs at the same time. This also provided us with certain challenges that are not available in the non-virtualized environment. We had to look at the memory of the VMs as a whole and use various signatures to figure out where different applications were located (running in the memory) and then look for the heap and other memory area and structures in those applications. Inside the VMs this thing can bee done using system libraries directly which is considered as their weakness because the malware can detect the presence of these monitoring tools and possibly disable them, which is not possible in our scenario. This also give us more control over the analysis, that was not available inside the VMs.

Atomizer provides us with the capability to look through the heap of any type of process running inside the virtual machines. This is like a library which can be used to create tools, which are trying to detect malicious activity inside the heap. The dynamic nature of the heap is quite challenging for any tools to detect and analyze the heap directly. Most tools rely on indirect methods for the detection of heap-based malware. This can reduce the reliability of these tools to certain extant and makes it possible for malwares to detect the presence of a detection mechanism. Atomizer not only analyzes the whole heap but also checks for malicious activity in the heap pages that have been swapped out of the main memory. These features are unique to our tool and keep the false positive to the minimum.

This work provides us with various important tools. During this research, we discovered newer arenas that we could not cover in this work and are left as future work. The most important thing that we observed was that, there were various applications that were using custom heap managing libraries on top of OS provided heap mangers. These custom heap

managing libraries are being used to either provide more control over heap management (for performance purposes) or to provide extra security on the heap. One study on Adobe PDF Reader custom heap manager [52] suggests that custom heap management in these application can open new doors for malware developers to take advantage of weakness in them. More study is required to detect malicious activities in custom heap managers and heaps created by them.

Bibliography

[1] Admutate. http://www.pestpatrol.com/zks/pestinfo/a/admmutate.asp.

[2] Advanced heap spraying technique. http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

[3] Amd. http://www.amd.com/.

[4] Cff explorer. http://www.ntcore.com/exsuite.php.

[5] Data execution prevention. http://technet.microsoft.com/en-us/library/cc738483.aspx.

[6] Digital signatures for kernel modules. http://msdn.microsoft.com/en-us/library/bb530195.aspx.

[7] Hacking with linux kernel module. http://newdata.box.sk/raven/lkm.html.

[8] Heap spray exploit tutorial: Internet explorer use after free aurora vulnerability. http://grey-corner.blogspot.com/2010/01/heap-spray-exploit-tutorial-internet.html.

[9] Heap spray generator. http://skypher.com/SkyLined/heap_spray/small_heap_spray_generator.html.

[10] Heavyload. http://www.jam-software.com/heavyload/.

[11] Hooking the kernel directly. http://www.codeproject.com/Articles/13677/\Hooking-the-kernel-directly.

[12] Idtguard. http://www.msuiche.net/2006/12/10/idtguard-v01-december-2005-build/.

[13] Inline hooking in windows. www.exploit-db.com/download_pdf/17802/.

[14] Intel. http://www.intel.com/.

[15] Kvm. http://www.linux-kvm.org/page/Main_Page/.

[16] Libguestfs. http://libguestfs.org/.

[17] Libvmi. http://code.google.com/p/vmitools/.

[18] Microsoft. http://www.microsoft.com/.

[19] Microsoft hyper-v. http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx.

[20] Ollydbg. http://www.ollydbg.de/.

[21] Opdis. http://mkfs.github.com/content/opdis/.

[22] Openssl. http://www.openssl.org/.

[23] Osr driver loader. http://www.osronline.com/article.cfm?article=157.

[24] Parallels. http://www.parallels.com/.

[25] Peering inside the pe: A tour of the win32 portable executable file format. http://msdn.microsoft.com/en-us/library/ms809762.aspx.

[26] Qemu. http://wiki.qemu.org/Main_Page.

[27] Sql slammer. http://www.sans.org/security-resources/malwarefaq/ms-sql-exploit.php.

[28] Virtual pc. http://support.microsoft.com/kb/958559.

[29] Vmware. http://www.vmware.com.

[30] Volatility. http://code.google.com/p/volatility/.

[31] Xen. http://www.xen.org/.

[32] Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, and Golden G. Richard III. Modchecker: Kernel module integrity checking in the cloud environment. In *ACM symposium on Applied computing*, 4th International Workshop on Security in Cloud Computing (CloudSec '12), September 2012.

[33] Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden G. Richard III, and Vassil Roussev. Idtchecker: Rule-based integrity checking of interrupt descriptor tables in cloud environments. In *Proceedings of the 9th IFIP WG 11.9*, International Conference on Digital Forensics, Orlando, Florida, January 2013.

[34] Periklis Akritidis, Evangelos P. Markatos, Michalis Polychronakis, and Kostas G. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *20th IFIP International Information Security Conference (IFIP/SEC)*, Makuhari-Messe, Chiba, Japan, May 2005.

[35] Dion Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. In *BLACK HAT DC*, Arlington, VA, US, January 2010.

[36] R. Buyyaand, J. Broberg, and A. Goscinski. *CLOUD COMPUTING: Principles and Paradigms*. John Wiley and Sons, Inc., Hoboken, New Jersey, 2011.

[37] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *International World Wide Web Conference*, Raleigh, North Carolina, US, April 2010.

[38] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Low-overhead mostly static javascript malware detection. In *USENIX Security Symposium*, San Francisco, California, US, August 2011.

[39] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploitswith javascript. In *WOOT: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*, San Diego, California, US, 2008.

[40] Francesco Gadaleta, Yves Younan, and Wouter Joosen. Bubble: a javascript engine level countermeasure against heap-spraying attacks. In *Proceedings of the Second international conference on Engineering Secure Software and Systems*, ESSoS'10, pages 1–17, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[42] Yuri Gushin. http://www.ecl-labs.org/papers/ecl-poly.txt.

[43] Fraser Howard. Malware with your mocha? obfuscation and anti-emulation tricks in malicious javascript, September 2010. `http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf`.

[44] G. Hoglund J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, Boston, 2005.

[45] Salman Javaid, Irfan Ahmed, Aleksandar Zoranic, and Golden G. Richard III. Atomizer: A fast, scalable and lightweight heap analyzer for virtual machines in a cloud environment. In *Proceedings of the 6th Layered Assurance Workshop (LAW'12)*, in conjunction with the 28th Annual Computer Security Applications Conference (ACSAC'12), Orlando, Florida, December 2012.

[46] Kad. Handling interrupt descriptor table for fun and profit. `http://www.phrack.org/issues.html?issue=59&id=4&mode=txt`.

[47] Kimmo Kasslin. Kernel malware: The attack from within. In *9th Annual Association of anti-Virus Asia Researchers Conference (AVAR)*, Auckland, New Zealand, December 2006.

[48] E. Kirda, S. Jha, and D. Balzarotti. Toward revealing kernel malware behavior in virtual execution environments. *Lecture Notes in Computer Science*, 5758:304–325, 2009.

[49] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ROZZLE: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, San Francisco, California, US, May 2012.

[50] Greg Kroah-Hartman. Signed kernel modules. *Linux Journal*, 117:48–53, 2004.

[51] Aaron LeMaster. Heap spray detection with heap inspector. In *Blackhat USA*, Las Vegas, Nevada, US, 2011.

[52] Haifei Li and Guillaume Lovet. Adobe reader's custom memory management: a heap of trouble. `http://www.fortiguard.com/paper/Adobe-Reader-s-Custom-Memory-Management-a-Heap-of-Trouble`.

[53] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *ACM workshop on Scalable trusted computing, STC '07*, pages 21–29, New York, NY, USA, 2007.

[54] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, pages 327–340, Lisbon, Portugal, March 2007.

[55] P. Mell and T. Grance. The NIST definition of cloud computing. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, 2007.

[56] mxatone and ivanlef0u. Stealth hooking : Another way to subvert the windows kernel. `http://www.phrack.org/issues.html?issue=65&id=4`.

[57] Walter Oney. *Programming the Microsoft Windows Driver*. Microsoft Press, New York, second edition edition, 2002.

[58] B.D. Payne, M.D.P. de Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, 2007.

[59] Alin Rad Pop. Dep/aslr implementation progress in popular third-party windows applications anti-emulation tricks in malicious javascript, June 2010. `http://secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf`.

[60] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, San Francisco, California, US, August 2009.

[61] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[62] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Microsoft Windows internals*. Microsoft Press, fifth edition, 2009.

[63] Joanna Rutkowska. System virginity verifier defining the roadmap for malware detection on windows system. In *Hack in the Box*, 2005.

[64] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Twentieth ACM Symposium on Operating Systems Principles*, pages 1–16, 2005.

[65] Hovav Shacham, Matthew Page, Ben Pfaff, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, Washington, DC, US, October 2004.

[66] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 8th Edition*. John Wiley and Sons, Inc., Hoboken, New Jersey, 2009.

[67] Skywing Skape. Bypassing patchguard on windows x64. `http://uninformed.org/?v=3&a=3&t=sumry`.

[68] Alexander Sotirov. Heap fang shui in javascript. In *BLACK HAT Europe*, Amsterdam , Netherlands, March 2007.

Vita

Salman Javaid was born in Rawalpindi, Pakistan and received his B.S in Mathematics with a minor in Physics from Punjab University Lahore and then proceed to receive his Post-graduate diploma in Information technology from Govt. Asgharmall College Rawalpindi. He then received his M.S in Information Technology from National University of Science and Technology(NUST), one of the top University in Pakistan. His research interests are in the area of digital live forensics, particularly malware analysis using introspection, and security and privacy in cloud environment, and network pen-testing.