Atomizer: Fast, Scalable and Lightweight Heap Analyzer for Virtual Machines in a Cloud Environment

Salman Javaid
University of New Orleans
New Orleans, LA, USA
sjavaid@uno.edu

Aleksandar Zoranic
University of New Orleans
New Orleans, LA, USA
azoranic@uno.edu

Irfan Ahmed
University of New Orleans
New Orleans, LA, USA
irfan.ahmed@uno.edu

Golden G. Richard III
University of New Orleans
New Orleans, LA, USA
golden@cs.uno.edu

ABSTRACT

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. A large number of malicious files and URLs offering dangerous contents are potentially encountered every day, both by client-side and server-side applications. Static and dynamic methods have been proposed to detect heap-based attacks in the literature, using various methodologies like NOZZLE. The main drawback with existing techniques is that they either consume too many resources or are complicated to implement. In this paper we propose Atomizer, which offloads process heap analysis for guest VMs to the privileged domain using Virtual Machine Introspection (VMI). Atomizer APIs can be used to implement various heap analyzing algorithms on processes running inside a VM. A simple heap-spray detection algorithm using Atomizer was implemented to determine the effectiveness of Atomizer. Use of Atomizer cannot be detected by in-guest VM malware, has minimal impact on the cloud server, is very effective in detecting heap spraying malwares, and is simple to deploy. Our architecture is particularly applicable to cloud environments where virtualization is used to host guest VMs.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

General Terms

Security, Measurement, Experimentation

Keywords

Xen, Introspection, Cloud Computing

1. INTRODUCTION

Recent advances in operating system-based security mechanisms like Address Space Layout Randomization (ASLR) [22] and Data Execution Prevention (DEP) [3], as well as compiler-based security mechanisms like stack protection [13] have made it more difficult for malware developers to inject code to exploit computer systems. This is why more and more hackers have turned to heap-based techniques such as heap spray attacks [6] and just-in-time (JIT) spraying [11] to bypass these security mechanisms. These techniques rely heavily on the heap of a process to bypass protections and execute the exploit. This makes monitoring the activities that occur in the heap extremely important. Unfortunately, monitoring heap space is a resource intensive task and many heap monitoring tools do not examine the heap contents for performance reasons, instead relying on hooking or diverting system calls to detect malware in the heap.

In this paper we present 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 [19], that's why in this paper we are mainly focused on heap spray attacks, although Atomizer can 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 [23] 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 [15].

Techniques like JIT spraying [11] 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) [20] 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.

The rest of the paper is organized as follows: Section 2 summarizes related work. Section 3 presents Atomizer and details its architectural assumptions. Section 4 outlines the implementation and Section 5 the evaluation. Section 6 concludes the paper.

2. RELATED WORK

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 section covers the existing approaches and tools that are most related to Atomizer.

Cova *et al.* propose JSAND [12] 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 [12]. 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 [17] can be used to evade the emulation part of JSAND. Unlike Atomizer, JSAND relies heavily on emulation which can have many limitations [17].

Ratanaworabhan *et al.* propose NOZZLE [19] which performs static control flow analysis of parts of the heap, interpreting them as code segments to detect malicious content. NOZZLE accomplishes this 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 has led to the development of an improved version of the programs called ZOZZLE [14].

LeMasters [18] 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.

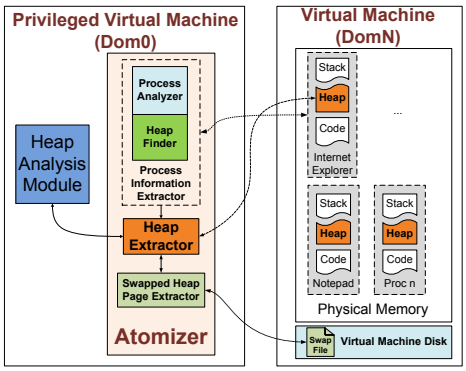


Figure 1: Atomizer Architecture

3. ATOMIZER ARCHITECTURE

The Atomizer infrastructure consists mainly of three components, Process Information Extractor, Heap Extractor and Swapped Heap Page Extractor as shown in Figure 1.

The first component, Process Information Extractor, is responsible for determining the basic attributes of the process we are monitoring. 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.

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.

4. IMPLEMENTATION

We developed a proof of concept prototype of Atomizer on XEN [9] that had Microsoft Windows XP (Service Pack 2) VMs running. We used the libVMI library [8] to introspect the heap of web browsers (Internet Explorer or Firefox) running on Windows XP virtual machines. We also used libguestfs [7] to access the page file of memory pages that have been swapped to the hard drive.

4.1 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 [21]. 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
```

4.2 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 [21]. To find the VAD tree of a process being monitored we use the process environment block structure (PEB) information received from Process Information Extractor. Using this information, we 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 we access pages that are swapped out are provided in Section 4.3. 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. Our 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 4.4).

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
```

4.3 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 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. Our 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 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 [7] 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.

4.4 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 [10] and ECL-Polynop [16] 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 our implementation, we 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 our 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.

5. EVALUATION

5.1 Experimental Settings

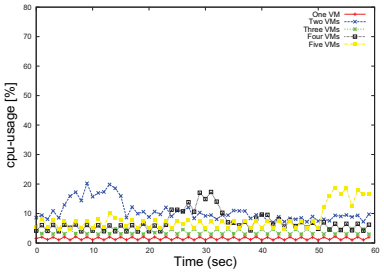
For this experiment, we built a simple cloud environment. This test bed featured a Quad Core i7 (2.67 GHz * 8) server with HyperThreading enabled and 18 GB of RAM. This server had a 64-bit privileged virtual machine (`Dom0`) running Fedora 16 (kernel 3.3.2-6) along with Xen 4.1.2 [9]. We instantiated 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 used the introspection library for VMI (libvmi-0.6) [8] and libguestfs [7] in all the experiments.

5.2 Malware Detection

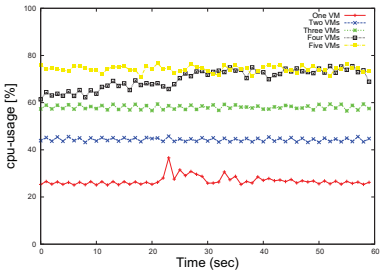
To test the effectiveness of Atomizer we tested it against various types of heap sprays commonly found on the web, as well as some custom made heap spray programs.

The first set of experiments involved the Skypher heap spray generator [5], 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 [4]. 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.

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. Our 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



(a) 1–5 Idle VMs running



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

Figure 2: CPU Performance (CPU usage in *Dom*0)

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 [16]. NOP sleds that might include unknown NOP replacements may not be detected by our implementation but newly discovered NOP replacements may potentially be added seamlessly to our hash table. Our implementation successfully detected the polymorphic NOP sleds without any false positives.

The Atomizer was tested against a state of the art exploit, Heap Feng Shui [23]. 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.

5.3 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 our 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. Figure2(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 our 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 2(b) shows the effect of Atomizer on the CPU. The Figure 2 shows the cumulative CPU usage of all the Virtual CPU's (VCPU) on the server (in our case we had eight VCPUs). This shows the overall effect of all the applications, including Atomizer and VMs on the CPU usage. Xentop [9] (the tool used by us 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.

6. CONCLUSION

In this paper we 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. Expermental 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 of our method, via a multi-threaded implementation of Atomizer.

Acknowledgment

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

7. REFERENCES

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

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

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

[6] Internet explorer iframe src&name parameter bof remote compromise. `http://skypher.com/wiki/index.php/Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php`.

[7] Libguestfs. `http://libguestfs.org/`.

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

[9] XEN. `http://www.xen.org/`.

[10] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. 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.

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

[12] M. Cova, C. Kruegel, and G. 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.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows:attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.

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

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

[16] Y. Gushin. http://www.ecl-labs.org/papers/ecl-poly.txt.

[17] F. 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`.

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

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

[20] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[21] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Microsoft Windows internals*. Microsoft Press, fifth edition, 2009.

[22] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, Washington, DC, US, October 2004.

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