# Scalable Web Object Inspection and Malfease Collection[*]

Charalampos Andrianakis, Paul Seymer, Angelos Stavrou

Center for Secure Information Systems
George Mason University, Fairfax, VA 22030
{candrian, pseymer, astavrou}@gmu.edu

## Abstract

*Internet drive-by downloads attacks are the preferred vehicle to infect desktop computers. In this paper, we propose a new URL analysis framework that combines lightweight virtualization and novel modifications to the WINE engine to detect heap spray attacks against applications. In addition, we are able to extract the attack shellcode used to further download other malicious binaries to the victim machine. Our preliminary results indicate that our system offers a compelling alternative to other process monitoring and virtualization technologies including* `QEMU` *and* `VMware` *since it can scale to thousands of instances per machine.*

## 1 Introduction

Our increasing reliance on the Internet for many facets of our daily lives (*e.g.,* commerce, communication, entertainment, etc.) has inspired several felonious operations (*e.g.,* phishing, spam), and the network has now become an attractive target for a host of illicit activities. While the monetary gains from the myriad of furtive behaviors being perpetrated today are not yet fully understood, it is clear that there is a general shift in tactics of old—wide-scale attacks aimed at overwhelming computing resources are less prevalent than they once were, and instead, traditional scanning attacks are being replaced by other mechanisms. Chief among these is the exploitation of the web, and the services built upon it, to distribute malware.

[*]This work was partly supported by the National Science Foundation through grant CNS-TC-0915291. Opinions, findings, conclusions, and recommendations expressed are work of the authors and do not necessarily reflect the views of the NSF.

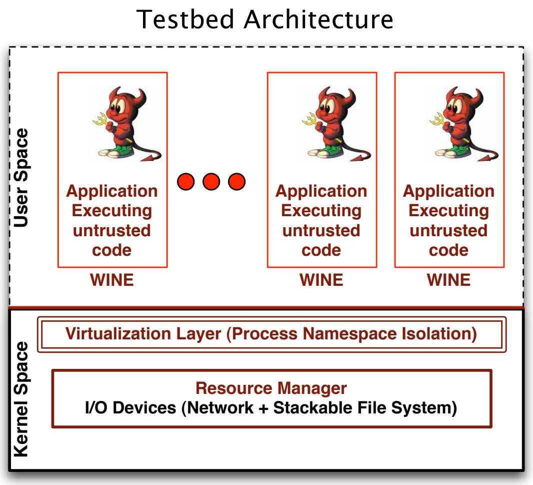
Testbed Architecture
User Space
Application Executing untrusted code
WINE
Application Executing untrusted code
WINE
Application Executing untrusted code
WINE
Kernel Space
Virtualization Layer (Process Namespace Isolation)
Resource Manager
I/O Devices (Network + Stackable File System)

Figure 1: Containers are used to execute multiple URL inspection instances. The resource manager guarantees fairness across physical resources.

Unfortunately, current proposed solutions for both honeypots and binary detection and analysis — including past work [14, 12] — depend heavily on full system virtualization [18, 4, 20, 8, 23] and thus, they do not scale well. These architectures are resource intensive, not allowing us to study the malware infection vector and the corresponding shellcode under different operating conditions. To make matters worse, it is very difficult to analyze the interactions of the malware with the browser or identify all the possible browser infection variants: usually, one has to fully instrument the browser or monitor all operating system activities and then thoroughly examine the produced logs. Each of these solutions comes at a heavy cost in terms of CPU, memory, and storage rendering them unsuitable for the analysis of thousands or even millions of application instances (e.g., per URL inspection).

To transcend the aforementioned limitations, we build a scalable host architecture that harnesses the multi-processor and multi-core capabilities of current commodity machines. Specifically, we envision using a lightweight kernel-level process containment system combined with `Wine` [1]—a Windows API environment on top of X, OpenGL, and Unix—to track malware interactions with the OS (see Figure 1). This contrasts with the use of other process monitoring and virtualization technologies (including `QEMU` and `VMware`) in that it does not impose excessive overhead, while at the same time, provides a robust application driven tracking system that can scale to thousands of instances per machine as shown by our preliminary experimental evaluation in Section 4.

## 2 Related Work

Over the past several years, virtual machines have routinely been used as honeypots for detecting attacks (e.g., [2, 9, 10, 22]). Although honeypots have traditionally been used mostly for detecting attacks against servers, the same principles also apply to client honeypots. For example, Moshchuk *et al.* used client-side techniques to study spyware on the web [10]. Their primary focus was not on detecting drive-by downloads, but in finding links to executables labeled spyware by an adware scanner. More germane is the work of Provos *et al.* [13, 11] and Seifert *et al.* [16] which raised awareness of the threat posed by drive-by downloads. These works were focused on explaining how different web page components are used to exploit web browsers, and merely provide a high-level overview of the different exploitation techniques in use today. The focus in this project is to a much more scalable and thus comprehensive analysis of the different aspects of the problem posed by web-based malware. This includes analysis of its exploit vectors, malware analysis, and shellcode collection.

Furthermore, there has also been plethora of work on heap-spray exploit detection and mitigation [15, 7, 6]. However, all this related work focus on techniques to detect the operating system level. Additionally, some work that tries to detect drive-by-download attacks [5] uses anomaly detection with emulation to automatically identify malicious JavaScript code.

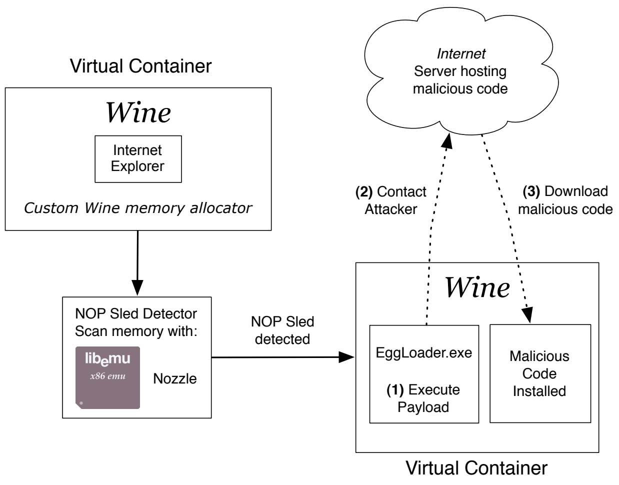

Figure 2: Framework Architecture

## 3 Framework Architecture

Our primary goal is to identify malicious web sites that exploit web browsers by using heap spray [17] techniques. To accomplish this, we have built an active honeynet that can instantiate in parallel a large number of un-patched Microsoft Internet Explorer (IE) instances. We are using container-based operating system virtualization [19] and WINE [1] to execute each application instance in an isolated environment. In our architecture, we use IE as a proof-of-concept. We do so because IE is currently the most widely used browser and as a result the primary target for attackers. In future work, we plan to use other browsers including Mozilla Firefox and Opera among others. The overall framework design has similar design characteristics with previous research [21, 12] on website analysis. However, we developed improvements that enable us to scale to a larger number of instances using the same set of resources. In the following paragraphs, we discuss the details of the system internals including our novel heap spray exploit detection framework inside WINE see Figures 1 and 2.

### 3.1 Virtual Environment Setup

For malware web object detection, we create instances of Virtual Environments (VEs). Each of these VEs have a minimal and stripped version of Debian Linux 5.0 installed. This Debian VE contains the necessary packages to have a functional WINE installation. One issue that cropped up was related to the graphical display interface. Any windows application binary that requires a graphical user interface needs to connect to an X server to display its graphics output. Of course, adding an X server (X.org or XFree86) installation on each VE instance would increase the VM memory and storage footprint and thus the overhead of our system. To address that, we relied on a feature of the X implementation: the capability to redirect running applications to display their output on a remote X server. Therefore, we installed an X server on a separate VE acting as graphical output aggregation point for all other VEs. This X feature helped us conserve state reducing the entire image to approximately 300MB of disk storage. In addition, while a VE instance is idle, the only program being executed is init and sysklogd. This reduces the idle memory usage to less than 3MB per container. Once the basic VE is set up and running we installed an un-patched version of Internet Explorer and took a snapshot of the VE (a procedure also known as CheckPointing). After visiting each URL, we used that snapshot to revert the VE to a known clean state.

## 3.2 Retrieving the URLs

To process a given web object, the system executes IE using Wine and instructs it to visit the object's URL. For each URL we visit, we pause until the content is fully downloaded. To prevent attackers from evading our system using a timer, we wait for the entire process for at least 10 seconds **after all communications** have been completed. At the same time, we monitor every request for memory allocation through Wine. This way we can terminate early if we detect the attack vector. Our empirical results indicate that 10 seconds is an adequate time to retrieve a URL and prevent timing evasion attacks. Moreover, we are not attempting to analyze any malware binaries, only detect attacks to the browser's heap. As a future direction, we plan to explore if longer waiting times (few minutes) have any effect on the detection accuracy.

Our aim is to scalably identify attacks against the browsers that use heap spray techniques. Heap spray attacks allocate thousands of blocks of memory in order to increase their chance of success when they divert the browser execution to an address in the heap. Each allocated memory block is several hundred kilobytes in size. To ferret-out heap spray attacks, we have developed a novel custom memory allocator that detects heap spraying and extracts the offending payload (shellcode) from the heap. An in-depth analysis of our memory allocator will be provided in Section 3.3. Once the attack is detected, Internet Explorer is terminated and the suspicious memory blocks are saved into a file on the host operating system. The host system then copies that file to a secure location for further inspection and uses the snapshot to restore the VE to a clean state. The VE is now ready to inspect the next available URL.

## 3.3 Customized Memory Allocator

Detecting heap spray attacks in windows has been proposed before. However, all previous approaches were using a full windows stack and relied on instrumentation of all heap calls and memory allocations. To avoid excessive processing time and effectively detect heap spray attacks, we have developed a custom memory allocator for Wine. When a request for a large memory block is received, the allocator places the address of that memory block in a linked list. The contents of that address are not inspected at that time because the memory returned by these functions is uninitialized. We have to wait until the memory is initialized to scan the contents. When the next memory request is received, all memory blocks that are already in the linked list are inspected. Our NOP sled detection engine searches that block of memory for NOP sleds. There has been a plethora of research on NOP sled detection [15], libemu [3]. We leverage all these techniques to determine whether that block of memory contains a NOP sled or not. The memory blocks are scanned every time a new address is appended to the linked list. Moreover, when a block is freed, it is removed from the list and not scanned again. If a NOP sled is detected, the block of memory is considered malicious and is stored in a file. When IE is stopped, the host system transfers all these files to a secure location for further analysis.

## 3.4 Payload Execution

The browser visits a malicious website that hosts a web browser exploit. The exploit usually employs

JavaScript to construct the payload and copy it in large chunks of memory (heap spray). Once the vulnerability is exploited, the execution flow control jumps to the nop sled and as a consequence, the shellcode gets executed. Upon gaining control of the system, the shellcode attempts to retrieve other malfease in the form of binaries. To avoid detection, shellcode is usually small, retaining only the necessary functionality to exploit the system using a browser vulnerability. To complete the take over of the system, shellcode usually attempts to reach back and download more binaries that will give complete access to the machine. The payload execution component emulates the part that follows the execution of the shellcode.

When a suspicious memory block is detected it is saved to a temporary location that is accessible only from the host operating system (the VEs don't have access to that location). Usually, this block of memory contains a NOP sled and a shellcode that the attacker was planning to execute at the exploited machine. Since our system successfully blocked the attack and extracted the malicious payload, we can now let the attack continue in a controlled and monitored environment. Our goal is to collect zero day malware binaries by letting the attacker think that he has gained total control of the machine.

To accomplish our goals, we place the malicious payload into a clean VE where we have also installed a 32 bit windows executable, the payload loader. The loader is responsible for reading the malicious data and executing it. Basically, we try to simulate the part of the attack where the exploit gains control of the program execution flow and tries to run the attacker injected shellcode. Before launching the payload loader, we setup a series of sensors that monitor all the activity on the VE and record it in files. The sensors are installed outside the VE, so that the malicious code wont be able to deactivate or bypass them.

Furthermore, we are interested in monitoring all network packets exchanged by this VE. It is very common for attackers to deploy a connect-back shellcode that contacts the command and control server or the malware distribution site and downloads instructions to execute or binaries to install. We want to be able to extract all this information from the communication channel. Moreover, we monitor the Wine virtual filesystem for any changes i.e. newly created files, modified files and folders and also the virtual windows registry (that Wine has built-in) for newly created keys.

Once the sensors are in place, we instruct Wine to execute the payload loader and run the suspicious memory block for about 2 minutes. Previous work [12, 21] has shown that 2 minutes is an adequate amount of time for a malware to take over a system. During that time, the shellcode contacts the malware distribution server and downloads a Trojan downloader. This last piece of code is responsible for downloading more binaries and installing them to the machine. The downloaded binaries are often key-loggers, back-doors or spam agents that try to use the machine or steal information. We allow all the downloaded software to silently execute and take complete control of the system, while we collect every useful bit of information from our logs. After the two minutes period has elapsed, the VE is shutdown and reverted to a known clean state. The VE is ready to be used for another payload execution.

## 4 Experimental Evaluation

To conduct our experiments we used two servers with the same hardware specifications. Specifically, the server models were DELL PowerEdge R710 with quad core intel x64 3.2Ghz CPU, 72GB of RAM memory and Seagate Baracuda 3.3TB for disk storage. In terms of software, we installed the 64 bit version of CentOS 5.3 on both servers. On top of that we deployed OpenVZ on one server and VMware Server 2.02 and QEMU 0.9.0-4 on the other. Note that neither was run concurrently during the experiments. For the OpenVZ server we installed the 32 bit version of Debian 5.0 on each container. We removed unnecessary software like sendmail, apache and mysql and created a minimal environment. On each virtual container we deployed our custom version of Wine 1.1.41 and then used winetricks to install Internet Explorer 6.0 with Service Pack 1. On the QEMU and VMWare Server, we created virtual machines with Windows XP 32-bit SP3 and Internet Explorer 6.0.

### 4.1 Experimental Results

In our first experiments, we measured the effectiveness of our heap-spray detection mechanism by ac-

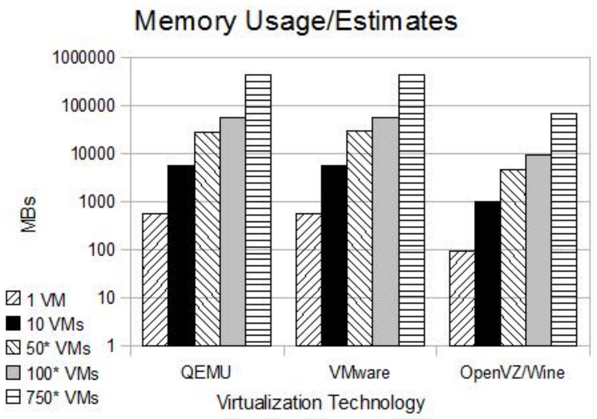

Figure 3: Memory Usage

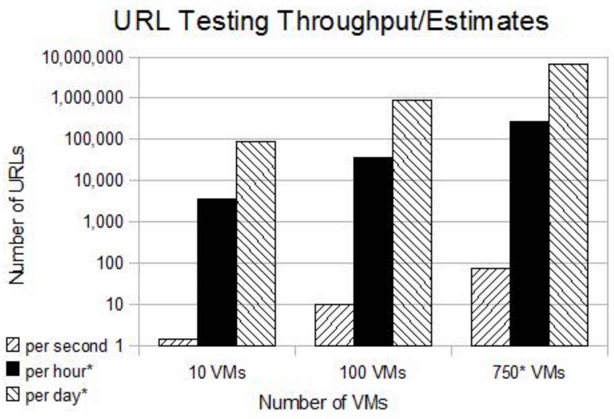

Figure 4: Experiment Throughput

cessing URLs that were hosting known heap-spray exploits. Second, we used our system on malicious pages that are hosted on the Internet and infect thousands of computers daily. Third, we quantified the performance overhead of our approached in comparison to other similar systems.

## 4.2 Heap Spray detection

To validate the effectiveness of our heap spray detector, we employed two different sources to gather a list of browser exploits that use heap-spray attacks. One source is milw0rm a popular exploit repository available on the Internet. Another source was metasploit, a framework that can produce exploits for a large number of vulnerabilities. In total, we collected 53 different classes of working exploits based on heap-spray attacks. All of those 53 exploits where successfully detected by our framework and the offending payload (shellcode) was successfully extracted.

In addition, we exposed our framework to 6,122 URLs provided by Google Inc. that were known to contain different attacks. These URLs contain attacks that target not only browsers but also plugins including Flash, ActiveX and PDF, that go beyond heap spray attack. These are not currently supported by our framework. Due to this limitation, we were only able to detect 274 URLs that used heap spray attacks. Using manual inspection, we concluded that the majority of the URLs were targeting other browser components that we plan to support on a future release as here we present a proof-of-concept prototype.

Although currently limited in scope, our framework scales very well when compared to Qemu and VMware and we were able to scan more URLs with fewer containers. Indeed, we can process approximately 36,000 URLs in a day with only 10 virtual containers. In addition, we can further increase the overall throughput when we raised the virtual machines to 100 VMs. We illustrate this on figure 4. Note that we used a logarithmic scale. The results indicate that we can process the same number of URLs in as little as an hour. Also, we can estimate that the maximum amount of virtual containers that we could possibly install on a single server is approximately 750. Should this prove to be feasible, we would be able to scan approximately 6.5 million URLs a day.

## 4.3 Scalability and Performance overhead

To get a rough estimate for the capacity of our servers, in terms of memory and disc space, we used Unix measuring utilities including time and top. We conducted these experiments for multiples of 10 VMs across all three virtualization platforms. Then we used this information to extrapolate what we estimate is the memory consumption for 50, 100, and 750 VMs. The results are shown in figure 3, using a logarithmic scale. Memory usage per VMware virtual machine during our experiments was between 532MB and 578MB, averaging at 565MB. QEMU behaved similarly with and average of 549MB per virtual machine. The OpenVZ/Wine platform used, on average, 91MB of memory for a single VM. Memory consumption scaled to 10 VMs well for all platforms. As we can see in figure 3, the OpenVZ/Wine platform demonstrated a footprint much smaller then QEMU and VMware.

We also recorded approximate sizes for each platform's VMs physical disk footprint, along with its corresponding snapshot. Although disk space is of low cost, and as a result is of less importance in capacity planning, the comparison between each platform helps to illustrate the strengths of OpenVZ/Wine. QEMU and VMware averaged around 1.5-1.6GB on disk, which the OpenVZ/Wine platform consumed only 300MBs. We were also able to make comparisons between VMware and the OpenVZ/Wine platform in terms of the time it takes to stop a running VM (say, at the end of an experiment) and restart with a clean snapshot (ready for the next URL). VMware demonstrated a 0.91 to 1.01 second (0.96 average) time in reverting to a restored snapshot. Performing this operation from the CLI also required that the VM be started, which took between 1.52 and 1.66 seconds (1.57 average). This resulted in a total time to replace a running VM with a clean snapshot of between 2.43 and 2.67 seconds (2.53 average). The OpenVZ/Wine platform required and average of 0.52 seconds to stop a VM, and 0.21 seconds to restore to a saved snapshot (ready for the next URL), resulting in a total experiment restart time of approximately 0.73 seconds. Barring any optimization in either platform, OpenVZ outperforms VMware in terms of experiment throughput.

## 5 Conclusions

Our results, while preliminary, show a significant scalability benefit when deploying our framework over traditional virtualization technologies. More experimentation will be conducted in the near future to push the performance limits of our framework, and adapt it to become more robust and more extensible. As part of our future work, we plan to explore the limitations of heap spray memory detection.

## References

[1] B. Amstadt and M. Johnson. Wine. *Linux Journal*, 1994(4es):3, 1994.

[2] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. August 2005.

[3] P. Baecher and M. Koetter. libemu. http://libemu.carnivore.it.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *WWW 2010*, Raleigh, NC, April 2010.

[6] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. *iNetSec 2009–Open Research Problems in Network Security*, pages 52–62, 2009.

[7] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA '09*, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, Chicago, IL, June 2005.

[9] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. August 2007.

[10] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware in the web. In *Proceedings of Network and Distributed Systems Security Symposium*, 2006.

[11] M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *LEET*, April 2008.

[12] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[13] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of HotBots'07*, April 2007.

[14] M. A. Rajab, J. Zarfoss, F. Monrose, A. Terzis, and N. Provos. A multifaceted approach to understanding the botnet phenomenon. In *ACM TISSEC*, pages 41–52. ACM, October 2006.

[15] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, 2009.

[16] C. Seifert, R. Steenson, T. Holz, Y. Bing, and M. A. Davis. Know Your Enemy: Malicious Web Servers. `http://www.honeynet.org/papers/mws/`, August 2007.

[17] A. Sotirov. Heap feng shui in JavaScript. *Proceedings of Blackhat Europe*, 2007.

[18] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[19] Virtuozo. OpenVZ. http://wiki.openvz.org.

[20] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *5th Symposium on Code Generation and Optimization*, pages 209–217, San Jose, CA, March 2007.

[21] Y. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys. In *NDSS 2006*, pages 35–49. Citeseer, 2006.

[22] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys. In *NDSS '06*, pages 35–49, 2006.

[23] J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux J.*, 2008(166):1, 2008.