Hypervisor-based Security Architecture to Protect Web Applications

Dilshan Jayarathna **Udaya Tupakula** **Vijay Varadharajan**

Advanced Cyber Security Research Centre
Macquarie University
Sydney, Australia
Email: {`dilshan.jayarathna, udaya.tupakula, vijay.varadharajan`}`@mq.edu.au`

Abstract

Web based applications are very common nowadays where almost every software can be accessible through a web browser in one form or the other. This paper proposes techniques to detect different threats related to web applications by using a hypervisor-based security architecture. The proposed architecture leverages the hypervisor's visibility of the virtual machines' runtime state and traffic flows for securing the web application. The unique feature of the proposed architecture is that it is capable of doing fine granular detection of web application attacks, i.e. to the specific web page level, and protecting the application against zero-day attacks.

Keywords: web server, security attacks, hypervisor, virtual machine introspection.

1 Introduction

Web applications are part of everyone's daily life, where you may be using one to read today's news with rich content or to do online banking with complex functionalities. Similarity, web applications has become a constant target of many malicious attackers trying to exploit their vulnerabilities to obtain sensitive data to benefit financially or to gain access to systems that can be used for other attacks. According to research from the High-Tech Bridge Security Research Lab (HT Bridge 2014), web application vendors are still taking an average of 11 days to release patches for critical security vulnerabilities while the low risk vulnerabilities take an average of 35 days. However, according to WhiteHat security research (Grossman 2013), it took an average of 193 days to apply the patches leased by the vendor to make sure the reported vulnerabilities are no longer exploitable. In addition to web application threats, a study by Symantec (ISTR 2014) reported that there were 23 zero-day vulnerabilities discovered in 2013, which is 61% increase from 2012. Therefore, it has become increasingly important not only to make the web applications more secure, but also to detect attacks even before the vulnerabilities are identified and remediated.

Many security mechanisms have been developed to protect web applications against potential attacks. Some techniques such as Intrusion detection (IDS) and prevention (IPS) systems and Web Application Firewalls (WAFs) are targeted at network level while other techniques applied on the application itself where inputs are validated before the execution and make use of good coding practices. However such techniques are not effective against zero day attacks and fast restoration of the services.

In this paper, we propose hypervisor based techniques to counteract web server attacks. A hypervisor also known as a virtual machine monitor (VMM) (Rosenblum and Garfinkel 2005, Barham et al. 2003) is a software abstraction layer that enables multiple virtualised operating systems to run concurrently on a single physical computer. An instance of a virtualised operating system along with its applications is referred to as a virtual machine (VM). Use of VMs in enterprise, government, and consumer applications is becoming increasingly important due to the diverse security requirements and the different levels of trust associated with different applications, systems and devices where VMs can provide virtual isolation between different environments.

While most of the VMMs provide application security from the network and host perspective, they are lacking in control over the contents specific to the critical applications running on VMs. This paper proposes techniques to protect web applications leveraging the hypervisor visibility of the traffic and runtime state of the web application. Since web servers are used for different applications, we interchange between the terms web server and web application in this paper.

This paper is organised as follows. Section 2 describes the related work in web application and virtualisation security. In Section 3 we focus on some specific attacks related to web applications and develop an attacker model and how it fits into VMM. Section 4 describes our proposed security architecture to counteract the security attacks against the web applications. In Section 4, we will also present how our model deals with the attacks that are considered in the attacker model. Section 5 describes our implementation of the security architecture and presents an analysis of the results. Finally, Section 6 concludes the paper.

2 Related Work

In this section, we describe some of the security tools that are currently available to protect web applications. We will also present different research areas related to the proposed architecture, which include new techniques to counteract web application attacks, virtual machine introspection (VMI) and domain name system (DNS) validation.

IDS and IPS protect web applications from known

---

attacks. A traditional IDS such as Snort (Roesch 1999) examines packets as they enter a network and uses various pieces of information from these packets to determine if a potential threat exists. Generally, this process involves pattern-matching the packets against a known database of threat patterns. Although this functionality is great, it is usually not enough to properly protect a Web server running complex web applications.

WAFs are designed to prevent web applications from attacks that common network firewalls and intrusion detection systems cannot detect. WAF can be configured to drop requests that appear to be HTTP but do not conform to HTTP standards such as RFC 2616 and 1945 or to limit the size of the HTTP body and headers in request. Many WAF products exist in the market today (OWASP WAP 2014) due to increased demand in protecting critical web applications. However such tools are not efficient in zero day attack detection, fine granular isolation of the vulnerable service and fast restoration of the services.

A number of researches published on anomaly-based detection of web attacks. Robertson et al. (2006) proposed an anomaly generalisation technique that automatically translates suspicious web requests into anomaly signatures. These signatures are then used to group recurrent or similar anomalous requests so that an administrator can easily deal with a large number of similar alerts. Sarrouy et al. (2009) approached it differently by examining the data which are sensitive to intrusions and controlling the constraints apply to them by comparing the data flow. Corona and Giacinto (2010) described an approach to detect server-side web attacks by applying a pattern recognition system to detect intrusion attempts on web servers. In another research study, Vigna et al. (2003) proposed a stateful IDS for web servers to perform early detection of malicious activity and possibly prevent more serious damage to the protected site. Ma et al. (2011) proposed a technique to detect malicious websites by applying online learning algorithm to analyse lexical and host-based features of their URLs.

There are other approaches to protect web sites from attacks such as injection and XSS attacks. Robertson and Vigna (2009) proposed a strongly-typed web development framework to build robust web applications against XSS and SQL injection attacks where user inputs are passed through specific sanitisation routines to ensure the integrity of the web documents and SQL queries. Samuel et al. (2011) built a reliable context-sensitive auto-sanitisation engine into web template systems based on type qualifiers to provide accurate context-sensitive sanitisation. Another approach is to use taint analysis (Nguyen-tuong et al. 2005, Haldar et al 2005) by using the variables that have been 'tainted' with user controllable input to identify insecure information flows where user inputs are propagated and flow into possible vulnerable functions within web applications. Skrupsky et al. (2013) proposed TamperProof, which is deployed in a trusted environment between client and server and intercepts all communication between them to determine the validation that should to be performed on the server for any given HTTP request. However, this method only protects web applications from parameter tampering attacks. All these approaches attempt to mitigate these attacks by sanitising inputs, but none provides a way of identifying an attack that has not been recognised by the sanitisation process. Our method aim to detect a successful injection attacks that are not detected by the existing sanitisation techniques by analysing the attack payload placed in HTTP response.

As described above, most of the solutions apply protection mechanisms such as WAFs, IDS/IPS and pattern recognition before requests reaching the web application while the others look for anomalies within the web server to detect attacks. All approaches are relevant in the context of incoming attacks and compromise attempts within the web server. In our approach we examine both incoming and outgoing content to detect possible attacks and to verify that the attack actually compromised the application or the server. In addition, we also verify the integrity of the web server by validating associated processes. Our aim is to detect possible threats, which are previously unknown such as zero-day attacks, by applying aggressive detection mechanisms and examining the outcome of the attack. This allows us to deny access to the vulnerable application or the webpage and to build protection mechanism against subsequent attacks. However note that our model can easily block all the known malicious requests before forwarding them to the web server. For example, we can use attack patterns from any of the tools as Snort(Roesch 1999) or Bro (Paxson 1999) to block the malicious requests before forwarding them to the web server. However none of these tools are efficient against the zero-day attacks as the attack detection depends on the known attacks patterns. Hence this is the main focus of our work.

Several techniques were proposed to address web server or application vulnerabilities. Most recently, Chen et al (2014) proposed SafeStack to automatically patch stack-based buffer overflow vulnerabilities by virtualising memory accesses and moving the vulnerable buffer into protected memory regions. While the proposed method is not limited to web servers or applications, it can only address the stack-based buffer overflow attacks. In contrast, our model is capable of identifying any buffer overflow attacks by combining multiple techniques such as application and server availability monitoring, and process validation.

Yang et al. (2014) proposed new technique that can provide collection of attack information using virtualisation technology and stable web service to cope with external attacks. It uses a honeynet system largely composed of a number of dynamically created virtualised honey VMs and virtual server system. However, this system only works for known attacks such as TCP-SYN flooding and ARP spoofing attacks, and does not actively prevent any attacks.

A remote backup and recovery method was proposed by He et al (2013), which focused on web site protection and disaster recovery. It uses a mechanism based on Rsync and FTP protocols to automatically backup web servers and restore them after an adversary attacks a server successfully. However, it is only effective when the attack actually tampered web server files and does not prevent recurrence of the same attacks. In comparison, our model detects a range of attacks and uses VM snapshot to restore web server and blocking the identified attacks in a matter of seconds.

Most of the web applications hosting sensitive data are secured by Secure Socket Layer (SSL) to ensure the confidentiality of the information. However, it is becoming very common most of the websites that run on virtual environments terminate their SSL before the VM hosting the web server, typically on the load balancer, to gain more performance by decreasing the need to encrypt and decrypt traffic at the web server (Windom Sr. 2013) assuming the trust of the virtual environment. Therefore, inspecting the web server traffic just before the VM is not difficult regardless of the web application is using SSL or not.

The method of inspecting a virtual machine from the hypervisor level for the purpose of analysing the software running inside it is called virtual machine introspection (VMI). VMI based architecture for intrusion detection system (IDS) leverages three properties of VMMs, namely isolation, inspection and interposition. VMM enables a virtual machine to be segregated in a way that it cannot access or modify the software running in a separate VM or in the VMM itself. This ensures that the IDS cannot be tampered even if the monitored host is completely subverted. The VMM provides the ability to directly inspect the hardware state of the virtual machine that a monitored host is running on. The VMM also provides the ability to interpose at the architecture interface of the monitored host, which offers even better visibility than standard OS-level mechanisms by enabling monitoring of both hardware and software level events. A good example of VMI IDS is the Livewire prototype.

A Formal model for VMI was described by Pfoh et al. (2009). Another VMI model was described by Payne et al. (2007) have implemented XenAccess, a monitoring library for operating systems running on Xen, which provides virtual memory introspection and virtual disk monitoring capabilities without any modifications to the VMM or to VM being monitored. XenAccess was further developed as LibVMI, which is also compatible with Kernel-based Virtual Machine (KVM). The most recent work by Fu and Lin (2012) described a technique that can automatically generate the VMI tools by monitoring system wide instructions monitoring, which identify the introspection related data and redirect these data accesses to the in-guest kernel memory. Our architecture makes use of the LibVMI for detecting compromise of the web server by analysing the processes running on it.

Domain Name System (DNS) is a system for naming computers and network services that is organized into a hierarchy of domains. DNS is an essential component of the functionality web application of the Internet, where the primary purpose of the DNS is to translate easily memorised/readable Internet or private network domain and host names to IP addresses. DNS services play a vital role when publishing web applications using friendly/readable names as users always remember the website name access the required application. There is previous work such as Seifert et al. (2008) looked at the relationships between DNS and web server to identify malicious web pages. In this paper, we make use of an existing techniques described in Jayarathna et al. (2014) to validate any DNS anomalies of domain names that are included in HTTP requests/responses. This is an additional protection mechanism we use to secure the web application from different kind of threats such as DNS spoofing or cache poisoning attacks.

3 Attacker Model

There are a number of well-known threats against web applications. In this section we consider those attacks that can be addressed by the proposed security architecture to formulate an attacker model.

As per Figure 1, we consider a local area network (LAN) segment with a couple of web servers (server 1 and 2) publishing two web applications (web app 1 and 2) and a number of clients accessing the web application from the same LAN. We also consider a number of external clients including a possible adversary who is trying to attack on of these web applications. However, in this model, we consider any client being the possible attacker. Now let us consider different attacks scenarios we are addressing in this paper.

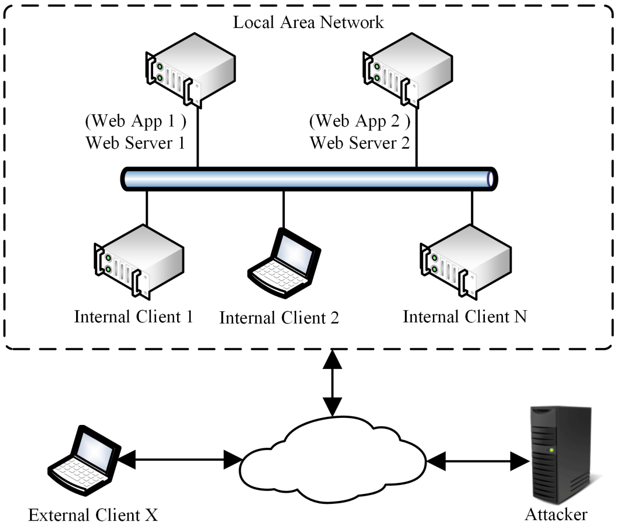


Figure 1: Attacker model

In the first case, we consider unvalidated redirects or forwards. Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorised pages.

In the second case we consider injection attacks against the web server. Attacker injects untrusted data to an interpreter in order to execute unintended syntax/commands or accessing unauthorised data. Commonly used injection attacks are based on SQL queries and OS commands. It is one of the most common web attack mechanisms used by hackers to steal data, which was categorised as the top most dangerous web vulnerability by OWASP in 2013 and 2010, and made the top ten in 2004 and 2007 (OWASP Top 10 2014). In this paper we consider inband injection attacks where data is extracted using the same channel that is used to inject the SQL code or the script. This is the most straightforward attack, in which the retrieved data is presented directly on the application web page.For example, the web application may be vulnerable to certain injection attacks. Known attacks can be detected using signature based tools such as snort. However the attacker can easily alter the inputs to evade by the signature based detection tools. Hence there is need to secure the web applications from zero day attacks.

In the third case we consider buffer overflow attacks against the web server. For example, an adversary could exploit a web application coding bug or operating systems bug to mount a buffer overflow attack and gain control over the server. A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. Sending carefully crafted input to a web application, an attacker can cause the web application to overflow the allocated memory space and execute arbitrary code, possibly taking over the machine. Usually, buffer overflows lead to crashes and consequently causing denial of service.

These types of attacks are becoming increasingly difficult since major web server developers have heightened their security over the years. However the addition of richer web application functionalities introduces the increased risk of untested code to be vulnerable to such attacks. Primary cause of such vulnerabilities is due to delay in applying patches on not only web applications but also the underlying operating system. In the following section we will describe how our model can deal with such attacks.

4 Protecting Web Application Attacks

Now we propose techniques to identify attacks that are considered in the previous section. First we present a high level overview of our model and then describe the architecture components in detail.

4.1 Architecture

Generally, in a standalone computing environment, the operating system has the full control over the execution of all applications and works as an interface between the applications and the hardware. In a virtual environment, VMM controls the access to physical resources while each guest operating system manages its own applications. Our architecture makes use of the VMM capabilities to ensure secure operation of web server.

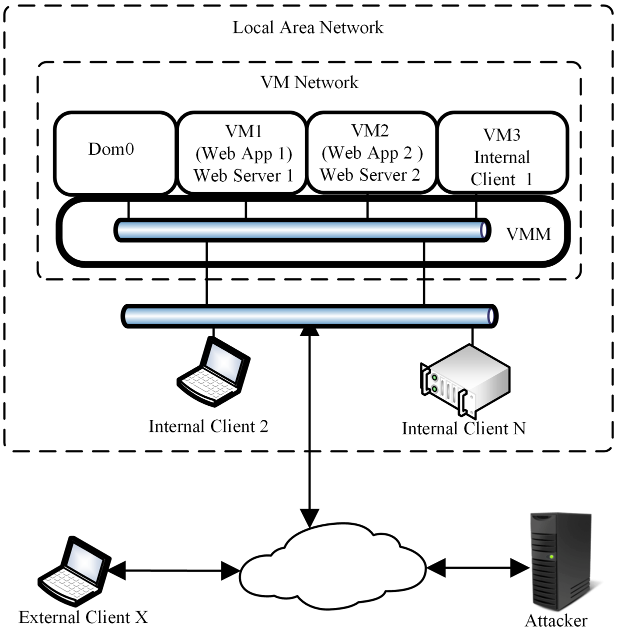


Figure 2: Attacker model mapped into a virtual environment

Figure 2 shows how we can apply the attack model into a virtualised environment in order to identify web application attacks. For simple presentation, we consider that the web server (VM1) and one of the web application clients (VM3) reside on the same VMM. However note that operation remains same when the web application client (e.g. VM3) in a different VMM or outside VMM similar to client 2. The only difference is between two models is that the virtualised environment will have its own LAN for VMs and internal clients will be on a different LAN segment. Therefore anything outside the VM LAN can be considered as external.

The proposed architecture leverages the hypervisor level visibility to monitor traffic flows in and out of the virtual machines. As shown in Figure 3, our architecture consists of three logical components, which are strategically placed in Dom0 for detection of attacks at fine granular level and fast restoration of the attacked services.

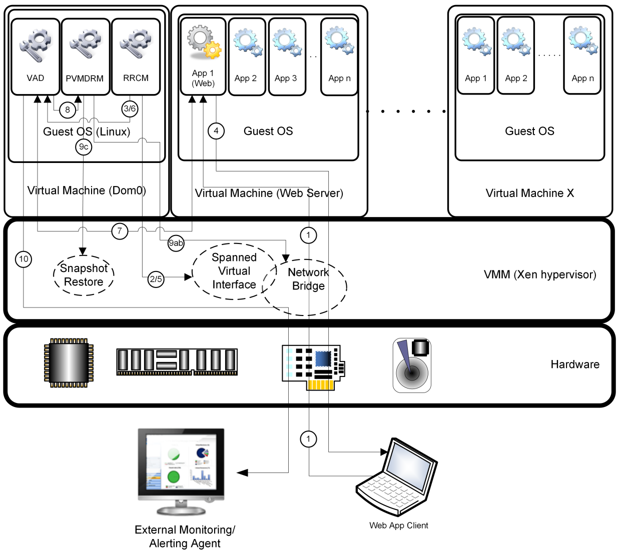


Figure 3: Architecture overview and operation

4.2 Logical components

In this Section we will describe the important logical components of our architecture.

4.2.1 Request/Response Capture Mechanism (RRCM)

RRCM consists of IDS component, which inspects all traffic flows using to and from virtual machines by simply sniffing the spanned virtual interface, which is mirroring the traffic of all the virtual bridge interfaces of the hypervisor. This reduces any performance impact of applying IDS alert rules directly into the active bridge interface. The second component of the RRCM is the Logger, which records content of the client request along with the time stamp, source IP address, domain name and the accessed web page in relation to the hosted web application for all alerts picked up by the IDS. Logger also captures the content of the corresponding responses, which are sent to the validation component for verification. We discard these captures after 24 hours to reduce the disk utilisation. But captures can be archived for longer period time if further analysis is required. Collected information is used for exploit diagnosis and deny further access to similar attacks.

4.2.2 Validation and Attack Detection (VAD)

VAD can detect and categorise the attacks by validating the runtime state and communication of the web application. Let us consider how VAD can detect the attacks on the web application.

VAD matches requests and corresponding responses sent by RRCM, and checks the content of captures to verify that the suspected attacks actually compromised the web application. If the content of the responses contains the proof of a compromise, the VAD flags the corresponding page as vulnerable and sends the information to PVMDRM (Protection and VM Data Restore Mechanism) component of the architecture. If VAD detected that the exploitation made a permanent change to the content website serving by inspecting the subsequent responses, VAD will also sends a restore request so that the changes done by the malicious requests are rolled back.

VAD also monitors the runtime state of the virtual web server and detect the compromise of the web servers. If the attacker exploits vulnerabilities in the web application or the operating systems in the virtual machine and install rootkits, such attacks are also detected by the VAD component. At regular intervals VAD checks what processes are running on the monitored web server. VAD keeps a track of processes running on the web server and uses this as reference to monitor the web server during runtime. Recorded process details are User ID (UID), Process ID (PID) and process running command (CMD). VAD component make use of the VMI interface to directly extract the runtime information from the monitored web server. If there is any variation from the default process list then the web server is considered to be compromised. Note that it is not an easy task for the attackers to alter the process list obtained by the VAD since it is running in the VMM. Once a compromise is detected, VAD sends a restore request to PVMDRM with the time stamp of initial connection of the attack obtained from the access logs recorded by the RRCM to perform a complete restore of the server.

VAD keeps a track of the usage of critical processes to identify possible buffer overflow attacks by analysing the process/memory utilisation trends and then map the time to the access logs recorded by the packet captures. A dynamic instrumentation tool Luk et al. (2005) is used to monitor all the program instructions to check their memory accesses.

VAD also checks the heartbeat of the server and the availability of published applications to identify a possible compromise and crash of the web applications.

4.2.3 Protection and VM Data Restore Mechanism (PVMDRM)

The PVMDRM takes a snapshots of the VM (without memory) at five-minute intervals and delete them automatically when the age of the snap shot is greater than 60 minutes except the each snapshot taken at first on every hour. It also deletes hourly snapshots after 24 hours keeping the first snapshot of the day for 30 days. Therefore the maximum number of snapshots it may keep at a given time is 64 (12/hour + 23 hours + 29 days). In this paper we mainly focus on securing the web application with static web pages. Once a positive compromise is detected for the attacks considered in this paper, we were able to perform an automatic restore using a clean snapshot (out of 12 snapshots taken within an hour before the detection). For example, in case of SQL injection we were able to restore the services with the snapshot that was taken during the last 5 minute interval. Hourly and daily snapshots were kept in case a manual restore in required.

Using the information fed by VAD component, PVMDRM identifies a snapshot with timestamp closest but before the attack's timestamp and triggers a restore process to revert back the VM to its last known good configuration.

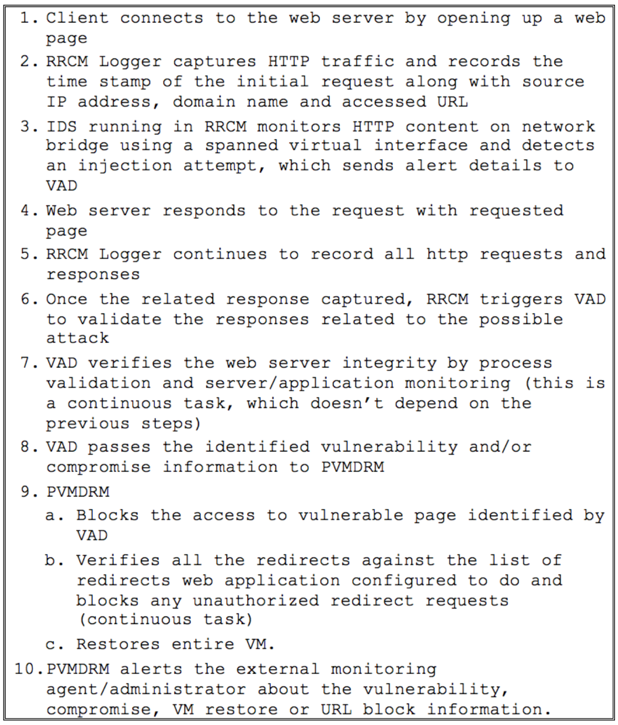
If the information received from VAD indicates that application is vulnerable to detected attacks, protection mechanism applies fine granular deny rules directly into the VMMs bridge interface to block the vulnerable pages. The vulnerable parts of the applications have to be patched before making them available to the users. As already stated in Section 1, it takes considerable time before the patches are released by the vendors. Hence our model is able to prevent access to the vulnerable part of the application at fine granular level and make the unaffected services available to its users.

PVMDRM also has a collection of whitelisted URLs which web server or application is configured to redirect or forward requests. An IPS rule checks for all redirects and drops connections to any URL not whitelisted.

Any of the above action triggers an alert to an external monitoring agent, which can be a third party security operations centre or a web application developer, who can verify the alert and take remediate action to rectify the problem. Once the problems are rectified, the drop rules can be removed to have the page available again. In case of a false-positive identification of a legitimate redirect is being blocked but not in the predefined list, it can be whitelisted to remove the block automatically

4.3 Operation

Let us discuss a simple walk through of the operation of our model by considering the architecture overview diagram in Figure 3. Steps on the following Figure 4 should be read in reference to the numbered arrows on the Figure 3.



1. Client connects to the web server by opening up a web page
2. RRCM Logger captures HTTP traffic and records the time stamp of the initial request along with source IP address, domain name and accessed URL
3. IDS running in RRCM monitors HTTP content on network bridge using a spanned virtual interface and detects an injection attempt, which sends alert details to VAD
4. Web server responds to the request with requested page
5. RRCM Logger continues to record all http requests and responses
6. Once the related response captured, RRCM triggers VAD to validate the responses related to the possible attack
7. VAD verifies the web server integrity by process validation and server/application monitoring (this is a continuous task, which doesn't depend on the previous steps)
8. VAD passes the identified vulnerability and/or compromise information to PVMDRM
9. PVMDRM
    a. Blocks the access to vulnerable page identified by VAD
    b. Verifies all the redirects against the list of redirects web application configured to do and blocks any unauthorized redirect requests (continuous task)
    c. Restores entire VM.
10. PVMDRM alerts the external monitoring agent/administrator about the vulnerability, compromise, VM restore or URL block information.

Figure 4: Architecture in operation

4.4 Attack Scenarios

Now we consider the attack scenarios described in the attacker model.

In the first scenario, the web application redirects the client to an external URL. PVMDRM checks the redirected URL against the whitelisted URLs it has on record and identifies that the redirected URL is not whitelisted which is resulted in blocking the request going to the redirected URL. For example, the main website of a University is set up in a virtual environment and have redirects configured to go to its enrolment application or online learning applications hosted in a different server possibly hosted by a third part using an external service provider. When we have the proposed architecture applied to this virtual environment, IPS component verifies that the redirect to other application is whitelisted and allows the client to receive the redirect request. However, it blocks the URL redirect request that it identifies as not whitelisted and notifies the website developer via email. This URL may be for a phishing site or a social networking site. In the case of phishing site is being blocked, the architecture protects the user from going to a phishing site even without the knowledge of it is malicious. For the latter, a developer has added the redirect link of the University's social media site page upon a request from the marketing department, but forgot to update the whitelist. Simply updating the whitelist allows the redirect.

In the second scenario, a particular web page was identified as vulnerable to an injection attack and blocked. For example, an attacker sends an inband injection attack against the simple web-based DNS lookup application hosted on a server protected by the proposed technique. Analysis of response message identified that the response has the entire listing of the root file systems of the server. By matching the logged access requests, it identified and blocked the URL of the vulnerable web application by adding new IPS rule. Once the problems are addressed and verified, the IDS rule is removed to allow the application accessible again. Hence our model can raise alerts for the attacks that are not identified earlier.

In the third scenario, a web application is vulnerable to a buffer overflow attack, which compromises the server by escalating privileges to gain root access to the server. This attack is identified, the VM is restored and the vulnerable application is blocked. For, example, an account firm's website protected by the proposed architecture has an online application that can be used for calculating tax. An adversary mounted a buffer overflow attack against this application. The capture mechanism logged the connection to application, but IDS did not detect any anomalies. However, the VAD component identified that the calculator application is not responsive for a short period of time and the process validation detected that there are a couple of processes running as root, which was not in the original web server configuration. This generated a positive compromise alert and triggered a VM restore. The logs are analysed to determine the time the attack traffic was received. The VM is restored with the snapshot of known good state that was taken before receiving the attack traffic. Simultaneously, protection mechanism in PVMDRM applied deny rules to block further access to the calculator application until the problem is rectified and notified. This attack may have not been previously identified by any protection system due to the stealth nature of its behaviour.

Techniques applied in scenarios two and three can be categorised as capable of being detecting zero-day attacks. In all scenarios, the block can be temporary or permanent depending on the security level set.

5 Implementation and Analysis

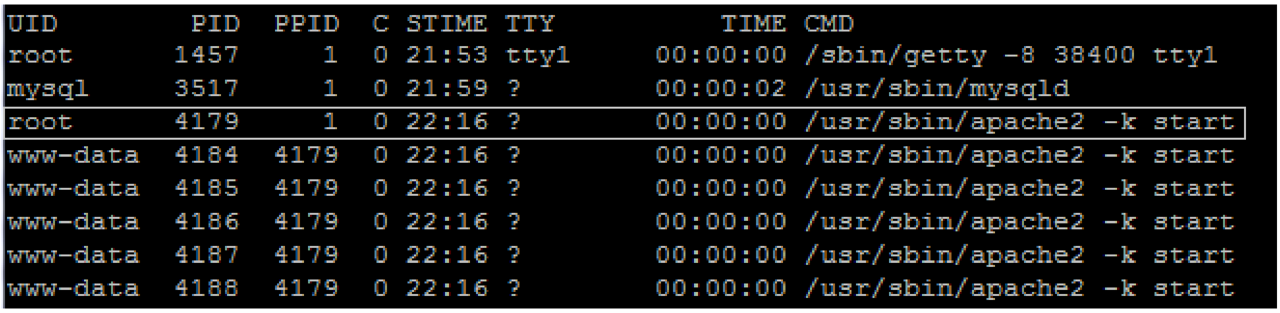
We have implemented our model using Xen VMM and web server running as virtual machines on the VMM. Xen VMM started off as a research project at the University of Cambridge and it was first introduced by Barham et al. (2003) as a high-performance resource-managed virtual machine monitor, which enables applications such as distributed web services, server consolidation and secure computing platforms. Xen is a native (or hypervisor-based) VMM where it runs directly on the hardware as lowest and most privileged layer. In Xen terminology domain" refer to a running virtual machine within which a guest OS executes. domain 0" (Dom0) boots with the hypervisor and works as the control interface with special management privileges which has direct access to underlying physical hardware. All other virtual machines are called domain U" (domU) in Xen terminology. Initially, on x86 architecture, Xen kernel code runs in Ring 0, while the hosted domains run in Ring 1 or Ring 3. Running operating system in Ring 1 or 3 instead of usual Ring 0 required operating system to be modified in order to suit the new privilege levels. This means only paravirtualized (i.e. modified operating systems) guests were able run on Xen. But Xen version 3.0 and above can use unmodified guest operating systems (e.g. Microsoft Windows XP) for hardware-assisted virtual machines (HVM) with supporting underlying hardware (e.g. Intel VT and AMD Pacifica).

5.1 Design Choices and Implementation

Components RRCM and VMDRM are placed in Dom0 of the hypervisor as RRCM needs to be in the data path of the bridge interface using the VMMs backend driver domain and PVMDRM require restoration and prevention abilities which can trigger commands available within the hypervisor. VAD component which is placed inside the Dom0 is used for process validation using LibVMI, and attack detection by validating the request and response messages.

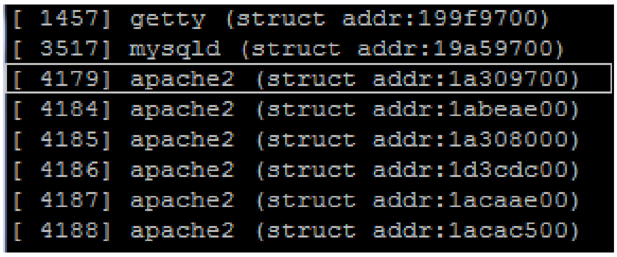
RRCM captures all packets related to web application communication between DomUs, the Dom0 and the physical network interface. We achieve this by spanning the Dom0's bridge interface (xenbr0), which is essentially the virtual switch of the hypervisor, to a separate virtual interface of Dom0 running in promiscuous mode to get an exact copy of the traffic flows in and out of the virtual machines.

We have analysed the default processes for Apache web server and MySQL running on a Linux server with Ubuntu 3.5.0-23 kernel. There are total of 63 processes in the clean state installation of Apache, PHP and MySQL on Linux. Figure 5 shows partial list of the processes running in the web server virtual machine and the highlighted process apache2 runs under root UID, which is located in specified path (/usr/sbin/apache2) is responsible for handing the client requests. We excluded the number of Apache child process running under www-data UID into the total count as primary process is responsible for launching child processes which listen to new connections and serve them when they arrive. Therefore the number of these child processes can change depending on the number of client connections. However, we keep a track of their Parent Process ID (PPID) to ensure that they are spawned from the same primary process. Figure 6 shows the partial list of processes of the web server and their corresponding address location obtained by the VAD component.



```
UID        PID  PPID  C STIME TTY          TIME CMD
root      1457     1  0 21:53 tty1     00:00:00 /sbin/getty -8 38400 tty1
mysql     3517     1  0 21:59 ?        00:00:02 /usr/sbin/mysqld
root      4179     1  0 22:16 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  4184  4179  0 22:16 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  4185  4179  0 22:16 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  4186  4179  0 22:16 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  4187  4179  0 22:16 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  4188  4179  0 22:16 ?        00:00:00 /usr/sbin/apache2 -k start
```

Figure 5: VM Processes list



```
[ 1457] getty (struct addr:199f9700)
[ 3517] mysqld (struct addr:19a59700)
[ 4179] apache2 (struct addr:1a309700)
[ 4184] apache2 (struct addr:1abeae00)
[ 4185] apache2 (struct addr:1a308000)
[ 4186] apache2 (struct addr:1d3cdc00)
[ 4187] apache2 (struct addr:1acaae00)
[ 4188] apache2 (struct addr:1acac500)
```

Figure 6: Runtime process validation using VMI

We have used Snort as the IDS/IPS components and SNMP to monitor the server heartbeat and web application availability. Pin tool by Luk et al. (2005) was used as the dynamic instrumentation tool to monitor all the program instructions to check their memory accesses. We used Metasploit to generate a number of injections attacks to verify the detection of known attacks.

5.2 Algorithms

This section describes algorithms used for various procedures of the three key components of the architecture.

5.2.1 RRCM Procedures

Algorithm 1 depicts the requests and responses capture process. It takes the packets from the spanned virtual interface and log access request information while detecting various attacks.

Algorithm 1 capRR (request/response capture in RRCM)

```
Input: spanned virtual interface packets
Output: None
webserver = web_server_IP
src_IP_list = empty list
alert_list = empty list
for all packets do
  if destination = webserver then
    add src_IP to src_IP_list
    log src_IP, rtime, page
    for all IDS_rules do
      if IDS_rule_match = true then
        log src_IP, domain_name, rtime, page, alert_type
        add src_IP to alert_list
        req_cap = capture upto next 1514 bytes
      end if
    end for
    for all src_IP in src_IP_list do
      if src_IP in alert_list then
        res_cap =capture upto next 1514 bytes
        call valReq (src_IP, req_cap, rtime, res_cap, page, alert_type)
      end if
    end for
  end if
end for
```

5.2.2 VAD Procedures

Algorithm 2 shows high-level steps of validating response in relation to the attack source, content of the request, timestamp of the initial request, accessed page and detected attack type provided as inputs by inspecting the content of the response.

Algorithm 2 valReq (response validation in VAD)

```
Input: spanned virtual interface packets
Output: src_IP, req_cap, rtime, res_cap, page, alert_type
if res_cap has content matching the alert_type then
  if page content changed then
    call restoreVM (rtime)
  end if
  call blockPage (src_IP, page)
end if
```

Algorithm 3 explains the process validation procedure where it obtains a list running processes using VMI and compares against the previously recorded list of processes.

Algorithm 3 valProc (process validation in VAD)

```
Input: None
Output: None
vmi_plist = process list obtained using VMI
vm_plist = process list obtained from clean VM
for every x minutes do
    vmi_plist = current process list obtained using VMI
    if vmi_plist doesn't match with vm_list then
        page = getPage (dtime) // page retrieval from logs
        call blockPage (page)
        call restoreVM (rtime)
    end if
end for
```

Algorithm 4 describes the process of the performing web server and application health checks, which also responsible of sending alerts.

Algorithm 4 checkHealth (application and server health check function)

```
Input: None
Output: None
for every x minutes do
    poll webserver
    if webserver is unresponsive then
        record unresponsive time
    end if
    poll application y
    if y is unresponsive then
        record unresponsive time against y
    end if
    if unresponsive time >set threshold then
        alertMsg(null, null, 5)
    end if
end for
```

5.2.3 PVMDRM Procedures

Algorithm 5 shows the steps of identifying and restoring the correct VM snapshot closest before the attack time taking the initial request time of the attack as the input. It also alerts the external recipient of the action it has taken.

Algorithm 5 restoreVM (VM data restore function in PVMDRM)

```
Input: rtime
Output: None
restoreSnapT = snapshot_t [0]; //first snapshot time
for all all snapshots snapshot_t[x] before rtime do
    if restoreSnapT before rtime then
        restoreSnapT = snapshot_t[x]
        increment x by one
    end if
end for
if restoreSnapT after rtime then
    alertMsg(null, rtime, 4)
end if
call restore (restoreSnapT); //hypervisor function
alertMsg(null, restoreSnapT, 3)
```

Algorithm 6 takes the web page/URL as input and creates a dynamic IPS rule to block it and sends alerts about the block.

Algorithm 6 blockPage (page block in PVMDRM)

```
Input: page
Output: None
apply IPS rule to block page
alertMsg(page, null, 1)
```

Algorithm 7 below used for checking redirects against a list of whitelisted URLs, which calls the Algorithm 7 to block any unidentified URL redirects and send relevant alert.

Algorithm 7 redirectCheck (URL redirect check in PVMDRM)

```
Input: None
Output: None
RDwhiteList = list of URLs whitelisted to redirect
dURL = detected redirect URL
block = true
for each URL in RDwhiteList do
    if vURL = dURL then
        block = false
    end if
end for
if block = true then
    apply IPS rule to block dURL
    alertMsg(dURL, null, 2)
end if
```

Algorithm 8 takes the inputs blocked page, restore time and alert type to formulate a correct message relevant to the action taken at the function calling it and sends alerts.

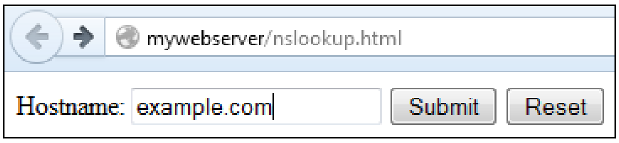
Algorithm 8 alertMsg (alert fuction in PVMDRM)

```
Input: page, restoreSnapT, type
Output: None
alertRecList = all alert recipients
for each recipient in alertRecList do
    switch (type)
    case 1:
        send "page has been blocked" recipient
    case 2:
        send Unauthorised redirect to page has been blocked recipient
    case 3:
        send VM was restored to restoreSnapT to recipient
    case 4:
        send No snapshot found before rtime to recipient
    default:
        send server/application is unresponsive
    end switch
end for
```

5.3 Example Scenario

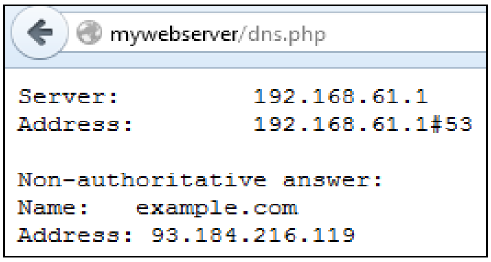
This section describes the second attack scenario described in the Section 4.4 with detection steps as a practical example. The hostname `mywebserver` on following four figures is a host entry on the client machine pointed to a web server with IP address 192.168.61.17 running on a virtual infrastructure described at the beginning of Section 4.

First, let us consider legitimate operation for this scenario. Figure 7 below shows a simple DNS lookup application running on the web server using PHP as the scripting language. Figure 8 shows the intended output for DNS resolution of example.com.



Hostname: example.com

Figure 7: Simple web-based DNS lookup application

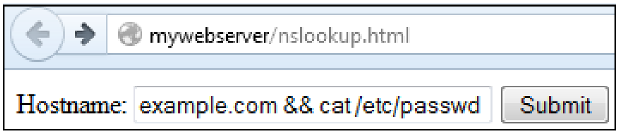


```
Server:         192.168.61.1
Address:        192.168.61.1#53

Non-authoritative answer:
Name:   example.com
Address: 93.184.216.119
```

Figure 8: DNS lookup output

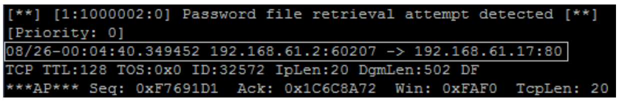
Now let us consider the same scenario during the attack. In this case the attacker is able to inject malicious input to the server to retrieve sensitive information stored in the server. Figure 9 shows the inband injection attempt by adding "`&& cat /etc/passwd`" command after the DNS name to read the content of the /etc/passwd file.



Hostname: example.com && cat /etc/passwd

Figure 9: Inband injection to view /etc/passwd file

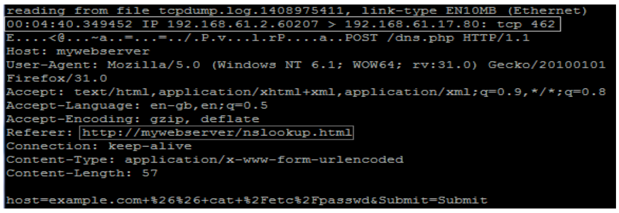
Figure 10 below shows the alert related to the detected password file retrieval attempt with alert time stamp. As highlighted on the figure, alert contains the timestamp of the incoming HTTP request on destination port 80 from the client IP address (192.168.61.2) to the web server IP address (192.168.61.17).



```
[**] [1:1000002:0] Password file retrieval attempt detected [**]
[Priority: 0]
08/26-00:04:40.349452 192.168.61.2:60207 -> 192.168.61.17:80
TCP TTL:128 TOS:0x0 ID:32572 IpLen:20 DgmLen:502 DF
***AP*** Seq: 0xF7691D1  Ack: 0x1C6C8A72  Win: 0xFAF0  TcpLen: 20
```

Figure 10: Alert of the password file retrieval attempt

Packet capture related to the identified attack is shown in Figure 11, which includes the accessed page (highlighted), vulnerable field and malicious input (at the bottom of the figure). It also has the same timestamp (second line highlighted) as the alert timestamp in Figure 10 above.

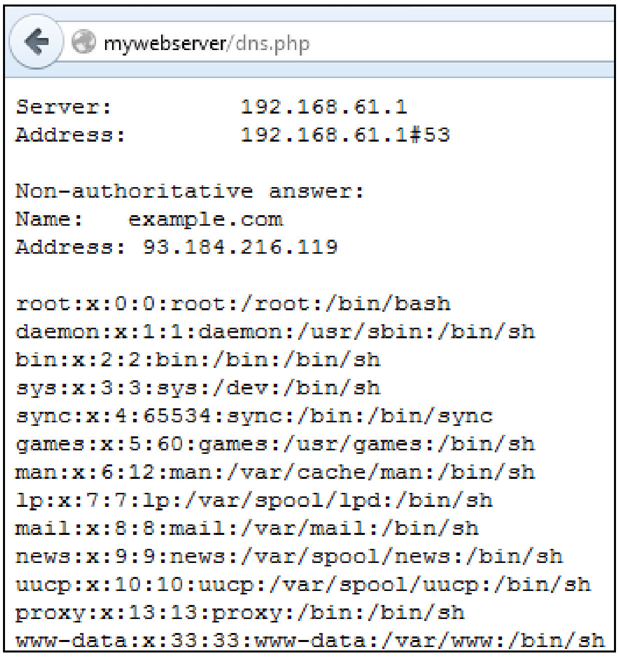


```
reading from file tcpdump.log.1408975411, link-type EN10MB (Ethernet)
00:04:40.349452 IP 192.168.61.2.60207 > 192.168.61.17.80: tcp 462
E....<@...~a..=...=../.P.v...l.rP....a..POST /dns.php HTTP/1.1
Host: mywebserver
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20100101
Firefox/31.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://mywebserver/nslookup.html
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 57

host=example.com+%26%26+cat+%2Fetc%2Fpasswd&Submit=Submit
```

Figure 11: Request capture associated with alert

The unvalidated input shown in Figure 9 caused the web server to extract entire password file and display it on the client browser. Figure 12 shows a trimmed version of the output.



```
Server:         192.168.61.1
Address:        192.168.61.1#53

Non-authoritative answer:
Name:   example.com
Address: 93.184.216.119

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
```

Figure 12: Output of injected command

Validating the response to the detected attack has the content of a password file, the system identified the page is vulnerable to that attack and blocked any future access to the identified URL (i.e. http://mywebserver/nslookup.html) containing the vulnerable web page by adding a new drop rule.

6 Conclusion

In this paper we have analysed different types of attacks on the web server and proposed techniques to deal with these attacks. We have shown that the proposed architecture can identify zero-day attacks by aggressively detecting attacks and analysing responses from the web application. This mechanism can be used to effectively safeguard web applications from such attacks and deny accessing the vulnerable page until a resolution or patch has been implemented. In the future work we will develop techniques for securing the web applications with dynamic web pages.

References

Barham, B., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. & Warfield, A. (2003), Xen and the art of virtualization, *in* 'SIGOPS Oper. Syst. Rev.', Vol. 38, No. 5, ACM Press, New York, NY, USA, pp. 164–177.

Chen, G., Jin, H., Zou, D., Zhou, B., Liang, Z., Zheng, W. & Shi, X. (2014), SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities, *in* 'Dependable and Secure Computing, IEEE Transactions', Vol. 10, No. 6, pp. 368–379.

Corona, I. & Giacinto, G. (2010), Detection of Server-side Web Attacks, *in* 'JMLR: Workshop and Conference Proceedings', Vol. 11, pp. 160–166.

Fu, Y. & Lin, Z. (2012), Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection, *in* 'IEEE Symposium on Security and Privacy', pp. 586–600.

Haldar, V., Chandra, D. & Franz, M. (2005), Dynamic taint propagation for java, *in* 'ACSAC ?05: Proceedings of the 21st Annual Computer Security Applications Conference', pp. 303–311.

He, Q., Guo, Y., Wang, Y. & Qiang, B. (2013), A web site protection oriented remote backup and recovery method, *in* 'Proceedings of the 8th International ICST Conference on Communications and Networking in China (CHINACOM)', pp. 395–399.

HT Bridge (2014), High-Tech Bridge Research: Web Application Security Trends in 2013, `https://www.technologyreview.com/web/21537/`, Accessed 14 June 2014.

ISTR (2014), Symantec Internet Security Threat Report, Vol. 19, Symantec Corporation.

Jayarathna, D., Tupakula, U. & Varadharajan, V. (2014), Hypervisor-based Security Architecture for Validating DNS Services, *in* 'Proceedings of the 12th Australasian Information Security Conference (AISC 2014)', Auckland, New Zealand, Vol. 149, pp. 83–86.

Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. & Hazelwood, K. (2005), Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *in* 'ACM SIGPLAN Conf. Programming Language Design Implementation', pp. 190–200.

Ma, J., Saul, L. K., Savage, S. & Voelker, G. M. (2011), Learning to detect malicious URLs, *in* 'ACM Trans. Intell. Syst. Technol.', Vol. 2, No. 3, pp. 30:1–24.

Nguyen-tuong, A., Guarnieri, S., Greene, D., Shirley, J., & Evans, D. (2005), Automatically hardening web applications using precise tainting, *in* 'Proceedings of the 20th IFIP International Information Security Conference', pp. 372–382.

OWASP Top 10 (2014), The Ten Most Critical Web Application Security Risks, The OWASP Foundation, `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, Accessed 20 Feb 2014.

OWASP WAP (2014), Web Application Firewall, The OWASP Foundation, `https://www.owasp.org/index.php/Web_Application_Firewall`, Accessed 29 Jul 2014.

Paxson, V. (1999),): Bro: A System For Detecting Network Intruders In Real-Time, *in* 'Computer Networks', Vol. 31, No. 23–24, pp. 2435–2463.

Payne, B., Carbone, M. & Lee, W. (2007), Secure and Flexible Monitoring of Virtual Machines, *in* 'Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)'.

Pfoh, J., Schneider, C. & Eckert, C. (2009), A formal model for virtual machine introspection, *in* 'VMSec '09: Proceedings of the 1st ACM workshop on Virtual machine security', New York, NY, USA, 2009. ACM, pp. 1–10.

Robertson, W. K. & Vigna (2009), Static enforcement of web application integrity through strong typing, *in* 'USENIX '09: Proceedings of the 18th conference on USENIX security symposium'. pp. 283–298.

Robertson, W. K., Vigna, G., Kruegel, C. & Kemmerer, R. A. (2006), Using generalization and characterization techniques in the anomaly-based detection of web attacks, *in* 'Proceedings of the Network and Distributed System Security Symp. (NDSS 2006)', San Diego, CA.

Roesch, M. (1999), Snort - Lightweight Intrusion Detection for Networks, *in* 'Proceedings of the 13th USENIX conference on System administration', November 07-12, 1999, Seattle, Washington, pp. 229–238.

Rosenblum, M. & Garfinkel, T. (2005), Virtual Machine Monitors: Current Technology and Future Trends, *in* 'Computer', Vol. 38, No. 5, IEEE Computer Society, Los Alamitos, CA, USA, pp. 39–47.

Samuel, M., Saxena, P. & Song, D. (2011), Context-sensitive auto-sanitization in web templating languages using type qualifiers. *in* 'CCS?11: Proceedings of the 18th ACM conference on Computer and communications security'. pp. 587–600.

Sarrouy, O., Totel, E., & Jouga, B. (2009), Application Data Consistency Checking for Anomaly Based Intrusion Detection, *in* 'Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '09)', Springer-Verlag, Berlin, Heidelberg, pp 726–740.

Seifert, C., Welch, I., Komisarczuk, P., Aval, C. & Endicott-Popovsk, B. (2008), Identification of malicious web pages through analysis of underlying DNS and web server relationships, *in* 'Proceedings of the 4th IEEE LCN Workshop on Network Security (WNS 2008)', pp. 935–941.

Shar, L. & Tan, H. (2012), Automated removal of cross site scripting vulnerabilities in web applications, *in* 'Inf. Softw. Technol.', Vol. 54, No. 5, pp. 467–478

Skrupsky, N., Bisht, P., Hinrichs, T., Venkatakrishnan, V. & Zuck, L. (2013), TamperProof: a server-agnostic defense for parameter tampering attacks on web applications, *in* 'Proceedings of the third ACM conference on Data and application security and privacy (CODASPY '13)', ACM, New York, NY, USA, pp. 129–140.

Vigna, G., Robertson, W., Kher, V. & Kemmerer, R. A. (2003), A stateful intrusion detection system for world-wide web servers, *in* 'Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)', Las Vegas, NV, pp 34–43.

Windom Sr., C. (2013), Security Considerations for VMware Horizon View 5.2, *in* 'VMware Technical White Paper', VMware Inc.

Grossman, J. (2013), How Does Your Website Security Stack Up Against Your Peers?, *in* 'WhiteHat Security Website Statistics Report', No. 12, WhiteHat Security, Inc.

Yang, H., Lee, D. & Yoo, S. (2014), A study on stable web server system using virtualization technology against attacks, *in* 'Multimedia Tools and Applications, Springer Science+Business Media, New York.