ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

IJARMATE
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

An Approach for the Implementation of Double Guard method for detecting the IDs in Web applications

Y.Md.Riyazuddin[1], G.Susmithavalli[2], G.Victor Daniel[3]

Asst Prof, Department of IT, GITAM, Hyderabad, India [1]

Prof, Department of CSE, MLRIT, Hyderabad, India [2]

Asst Prof, Department of IT, GITAM, Hyderabad, India [3]

Abstract—**An intrusion detection system (IDS) is a device or software application that monitors network or system activities for malicious activities or policy violations and produces reports to a management station. Some systems may attempt to stop an intrusion attempt but this is neither required nor expected of a monitoring system. Intrusion detection and prevention systems (IDPS) are primarily focused on identifying possible incidents, logging information about them, and reporting attempts. In addition, organizations use IDPSes for other purposes, such as identifying problems with security policies, documenting existing threats and deterring individuals from violating security policies. Internet services and applications have become an inextricable part of daily life, enabling communication and the management of personal information from anywhere. To accommodate this increase in application and data complexity,web services have moved to a multi-tiered design wherein the webserver runs the application front-end logic and data is outsourced to a database or file server.**

In this paper, we present DoubleGuard, an IDS system that models the network behavior of user sessions across both the front-end web server and the back-end database. By monitoring both web and subsequent database requests, we are able to ferret out attacks that an independent IDS would not be able to identify. Furthermore, we quantify the limitations of any multi-tier IDS in terms of training sessions and functionality coverage.

We implemented DoubleGuard using an Apache web server with MySQL and lightweight virtualization. We then collected and processed real-world traffic over a 15-day period of system deployment in both dynamic and static web applications. Finally,using DoubleGuard, we were able to expose a wide range ofattacks with 100% accuracy while maintaining 0% false positives for static web services and 0.6% false positives for dynamic web services.

Index Terms—**Anomaly Detection,Double Guard,Intrusion Detection,Multi tired Web services.**

I. INTRODUCTION

Web-delivered services and applications have increased in both popularity and complexity over the past few years. Daily tasks, such as banking, travel, and social networking, are all done via the web. Such services typically employ a web server front-end that runs the application user interface logic, as well as a back-end server that consists of a database or file server. Due to their ubiquitous use for personal and/or corporate data, web services have always been the target of attacks. These attacks have recently become more diverse, as attention has shifted from attacking the front-end to exploiting vulnerabilities of the web applications in order tocorrupt the back-end database system(e.g., SQL injectionattacks).A plethora of Intrusion Detection Systems(IDS) currently examine network packets individually with in both the web server and the database system. However, there is very little work being performed on multi-tiered Anomaly Detection (AD) systems that generate models of network behavior for both web and database network interactions. In such multi-tiered architectures, the back-end database server is often protected behind a firewall while the web servers are remotely accessible over the Internet. Unfortunately, though they are protected from direct remote attacks, the back-end systems are susceptible to attacks that use web requests as a means to exploit the back-end.

To protect multi-tiered web services, Intrusion detection systems (IDS) have been widely used to detect known attacks by matching misused traffic patterns or signatures. A class of IDS that leverages machine learning can also detect unknown attacks by identifying abnormal network traffic that deviates from the so-called "normal" behavior previously profiled during the IDS training phase. Individually,the web IDS and the database IDS can detect abnormal network traffic sent to either of them. However, we found that these IDS cannot detect cases wherein normal traffic isused to attack the web server and the database server. For example, if an attacker with non-admin privileges can log in to a web server using normal-user access credentials, he/she can find a way to issue a privileged database query by exploiting vulnerabilities in the web server. Neither the web IDS nor the database IDS would detect this type of attack since the web IDS would merely see typical user login traffic and the database IDS would see only the normal traffic of a privileged user. This type of attack can be readily detected if the database IDS can identify that a privileged request from the web servers not associated with user-privileged access. Unfortunately,within the current multi-threaded web server architecture, it is not feasible to detect or profile such causal mapping between web server traffic and DB server traffic since traffic cannot be

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

clearly attributed to user sessions.

We have implemented our Double Guard container architecture using OpenVZ, and performance testing shows that it has reasonable performance overhead and is practical for most web applications. When the request rate is moderate (e.g.,under 110 requests per second), there is almost no overhead in comparison to an unprotected vanilla system. Even in a worst case scenario when the server was already overloaded, we observed only 26% performance overhead. The container-based web architecture not only fosters the profiling of causal mapping, but it also provides an isolation that prevents future session-hijacking attacks. Within a lightweight virtualization environment, we ran many copies of the web server instances in different containers so that each one was isolated from the rest. As ephemeral containers can be easily instantiated and destroyed, we assigned each client session a dedicated container so that, even when an attacker may be able to compromise a single session, the damage is confined to the compromised session; other user sessions remain unaffected by it.

To address this challenge while building a mapping model for dynamic web pages, we first generated an individual training model for the basic operations provided by the web services. We demonstrate that this approach works well in practice by using traffic from a live blog where we progres- sively modeled nine operations. Our results show that we were able to identify all attacks, covering more than 99% of the normal traffic as the training model is refined.

II. Related Work

A network Intrusion Detection System (IDS) can be classified into two types: anomaly detection and misuse detection. Anomaly detection first requires the IDS to define and characterize the correct and acceptable static form and dynamic behavior of the system, which can then be used to detect abnormal changes or anomalous behaviors. The boundary between acceptable and anomalous forms of stored code and data is precisely definable. Behavior models are built by performing a statistical analysis on historical data or by using rule-based approaches to specify behavior patterns. An anomaly detector then compares actual usage patterns against established models to identify abnormal events. Our detection approach belongs to anomaly detection,and we depend on a training phase to build the correct model.As some legitimate updates may cause model drift, there area number of approaches that are trying to solve this problem. Our detection may run into the same problem; insuch a case, our model should be retrained for each shift.

Intrusion alerts correlation provides a collection of components that transform intrusion detection sensor alerts into succinct intrusion reports in order to reduce the number of replicated alerts, false positives, and non-relevant positives.It also fuses the alerts from different levels describing a single attack, with the goal of producing a succinct overview of security-related activity on the network. It focuses primarily on abstracting the low-level sensor alerts and providing compound, logical, high-level alert events to the users. DoubleGuard differs from this type of approach that correlates alerts from independent IDSes. Rather, DoubleGuard operates on multiple feeds of network traffic using a single IDS that looks across sessions to produce an alert without correlating or summarizing the alerts produced by other independent IDSs.

These softwares, such as Green SQL , work as a reverse proxy for database connections. Instead of connecting to a database server, web applications will first connect to a database firewall. SQL queries are analyzed; if they're deemed safe, they are then forwarded to the back-end database server.The system proposed in composes both web IDS and database IDS to achieve more accurate detection, and it also uses a reverse HTTP proxy to maintain a reduced level of service in the presence of false positives. However, we found that certain types of attack utilize normal traffics and cannot be detected by either the web IDS or the database IDS. In such cases, there would be no alerts to correlate.Some previous approaches have detected intrusions or vulnerabilities by statically analyzing the source code or executables.Others dynamically track the information flow to understand taint propagations and detect intrusions. In DoubleGuard, the new container-basedweb server architecture enables us to separate the different information flows by each session. This provides a means of tracking the information flow from the web server to the database server for each session. Our approach also does not require us to analyze the source code or know the applicationlogic. For the static web page, our DoubleGuard approach doesnot require application logic for building a model. However, as we will discuss, although we do not require the full application logic for dynamic web services, we do need to know the basic user operations in order to model normal behavior.

Virtualization is used to isolate objects and enhance security performance. Full virtualization and para-virtualization are not the only approaches being taken. An alternative is a lightweight virtualization, such as OpenVZ, Parallels Virtuozzo, or Linux-VServer. In general, these are based on some sort of container concept. With containers, a group of processes still appears to have its own dedicated system, yetit is running in an isolated environment. On the other hand,lightweight containers can have considerable performance advantages over full virtualization or para-virtualization. Thousands of containers can run on a single physical host. There are also some desktop systems that use lightweight virtualization to isolate different application instances. Such virtualization techniques are commonly used for isolation and containment of attacks. However, in our DoubleGuard, we utilized the container ID to separate session traffic as a wayof extracting and identifying causal relationships between web server requests and database query events.

III. Threat Model & System Architecture

We initially set up our threat model to include our assump-

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

tions and the types of attacks we are aiming to protect against.We assume that both the web and the database servers are vulnerable. Attacks are network-borne and come from the web clients; they can launch application-layer attacks to compromise the web servers they are connecting to. The attackers canbypass the web server to directly attack the database server. We assume that the attacks can neither be detected nor prevented by the current web server IDS, that attackers may take over the web server after the attack, and that afterwards they can obtain full control of the web server to launch subsequent attacks. For example, the attackers could modify the application logic of the web applications, eavesdrop or hijack other users' web requests, or intercept and modify the database queries to steal sensitive data beyond their privileges.

On the other hand, at the database end, we assume that the database server will not be completely taken over by the attackers. Attackers may strike the database server through the web server or, more directly, by submitting SQL queries, they may obtain and pollute sensitive data within the database.

A.Architecture and confinement:

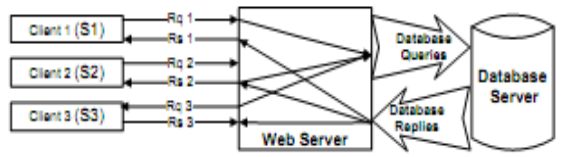

Fig 1: Classic 3-tier Model. The web server acts as the front-end, with the file and database servers as the content storage back-end.

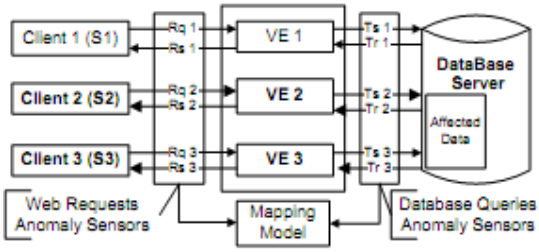

Fig 2: Web server instances running in containers

All network traffic, from both legitimate users and adversaries, is received intermixed at the same web server. If an attacker compromises the web server, he/she can potentially affect all future sessions (i.e., session hijacking). Assigning each session to a dedicated web server is not a realistic option, as it will deplete the web server resources. To achieve similar confinement while maintaining a low performance and resource overhead, we use lightweight virtualization.

In our design, we make use of lightweight process containers, referred to as "containers," as ephemeral, disposable servers for client sessions. It is possible to initialize thousands of containers on a single physical machine, and these virtualized containers can be discarded, reverted, or quickly reinitialized to serve new sessions. A single physical web server runs many containers, each one an exact copy of the original web server. Our approach dynamically generates new containers and recycles used ones. As a result, a single physical server can run continuously and serve all web requests.However, from a logical perspective, each session is assigned to a dedicated web server and isolated from other sessions.Since we initialize each virtualized container using a read-only clean template, we can guarantee that each session will be served with a clean web server instance at initialization.We choose to separate communications at the session level so that a single user always deals with the same web server. Sessions can represent different users to some extent, and we expect the communication of a single user to go to the same dedicated web server, thereby allowing us to identify suspect behavior by both session and user. If we detect abnormal behavior in a session, we will treat all traffic within this session as tainted. If an attacker compromises a vanilla web server, other sessions' communications can also be hijacked. In our system,an attacker can only stay within the web server containers that he/she is connected to, with no knowledge of the existence of other session communications. We can thus ensure that legitimate sessions will not be compromised directly by an attacker.

Figure 1 illustrates the classic 3-tier model. At the database side, we are unable to tell which transaction corresponds to which client request. The communication between the web server and the database server is not separated, and we can hardly understand the relationships among them. Figure 2 depicts how communications are categorized as sessions and how database transactions can be related to a corresponding session. According to Figure 1, if Client 2 is malicious and takes over the web server, all subsequent database transactions become suspect, as well as the response to the client. By contrast, according to Figure 2, Client 2 will only compromise the VE 2, and the corresponding database transaction set T2 will be the only affected section of data within the database.

B.Building the Normality Model:

This container-based and session-separated web server architecture not only enhances the security performances but also provides us with the isolated information flows that are separated in each container session. It allows us to identify themapping between the web server requests and the subsequent DB queries, and to utilize such a mapping model to detect abnormal behaviors on a session/client level. In typical 3-tiered web server architecture, the web server receives HTTP requests from user clients and then issues SQL queries to the database server to retrieve and update data. These SQL queries are causally dependent on the web request hitting the web server. We want to model such causal mapping relationships of all legitimate traffic so as to detect abnormal/attack traffic.

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

C.Attack Scenarios:

Our system is effective at capturing the following types of attacks:

Privilege Escalation Attack: Let's assume that the website serves both regular users and administrators. For a regular user,the web request ru will trigger the set of SQL queries Qu ; for an administrator, the request ra will trigger the set of admin level queries Qa . Now suppose that an attacker logs into the web server as a normal user, upgrades his/her privileges, and triggers admin queries so as to obtain an administrator's data.This attack can never be detected by either the web server IDS or the database IDS since both ru and Qa are legitimate requests and queries. Our approach, however, can detect this type of attack since the DB query Qa

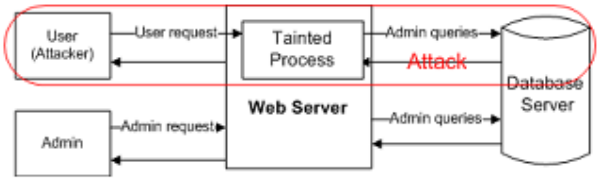

Fig.3: Privilege Escalation Attack

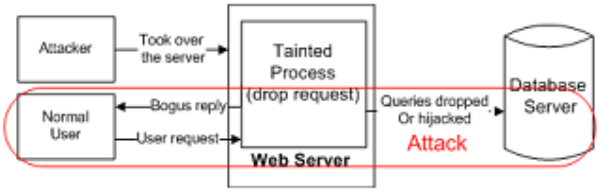

Fig .4: Hijack Future Session Attack

does not match therequest ru , according to our mapping model. Figure 3 shows how a normal user may use admin queries to obtain privileged information.

Hijack Future Session Attack: This class of attacks is mainly aimed at the web server side. An attacker usually takes over the web server and therefore hijacks all subsequent legitimate user sessions to launch attacks. For instance, by hijacking other user sessions, the attacker can eavesdrop, send spoofed replies,
and/or drop user requests. A session hijacking attack can be further categorized as a Spoofing/Man-in-the-Middle attack, an Exfiltration Attack, a Denial-of-Service/Packet Drop attack, or a Replay attack.

Figure 4 illustrates a scenario wherein a compromised web server can harm all the Hijack Future Sessions by not generating any DB queries for normal user requests. According to the mapping model, the web request should invoke some database queries (e.g., a Deterministic Mapping (section IV-A)), then the abnormal situation can be detected. However, neither a conventional web server IDS nor a database IDS can detect such an attack by itself.

Fig 5: Injection Attack.

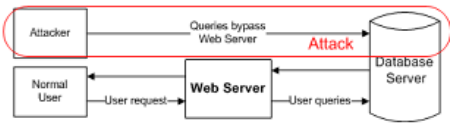

Fig.6:DB Query with out causing Web requests

Direct DB attack: It is possible for an attacker to bypass the web server or firewalls and connect directly to the database. An attacker could also have already taken over the web server and be submitting such queries from the web server without sending web requests. Without matched web requests for such queries, a web server IDS could detect neither. Furthermore, if these DB queries were within the set of allowed queries, then the database IDS itself would not detect it either. However,this type of attack can be caught with our approach since wecannot match any web requests with these queries. Figure 6 illustrates the scenario wherein an attacker bypasses the webserver to directly query the database.

D.Double Guard limitations:

In this section, we discuss the operational and detection limitations of DoubleGuard.

Vulnerabilities Due to Improper Input Processing: Cross Site Scripting (XSS) is a typical attack method wherein attackers embed malicious client scripts via legitimate user inputs.In DoubleGuard, all of the user input values are normalized so as to build a mapping model based on the structures of HTTP requests and DB queries. Once the malicious user inputs are normalized, DoubleGuard cannot detect attacks hidden in the values. These attacks can occur even without the databases. DoubleGuard offers a complementary approach to those research approaches of detecting web attacks based on the characterization of input values.

Possibility of Evading DoubleGuard:

Our assumption is that an attacker can obtain "full control" of the web server thread that she connects to. That is, the attacker can only take over the web server instance running in its isolated container. Our architecture ensures that every client be defined by the IP address and port container pair, which is unique for each session. Therefore, hijacking an existing container is not possible because traffic for other sessions is never directed to an occupied container. If this were not the case, our architecture would have been similar to the conventional one where a single web server runs many different processes. queries. However, this significantly increases the efforts for the attackers to launch successful attacks. In addition, users with non-admin permissions can cause minimal (and sometimes zero) damage to the rest of the

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

system and therefore they have limited incentives to launch such attacks.

Distributed DoS: DoubleGuard is not designed to mitigate DDoS attacks. These attacks can also occur in the server architecture without the back-end database.

IV.MODELING DETERMINISTIC MAPPING AND PATTERNS

Due to their diverse functionality, different web applicationsexhibit different characteristics. Many websites serve onlystatic content, which is updated and often managed Finally, in some cases, the web server will have some periodical tasks that trigger database queries without any web requests driving them. The challenge is to take all of these cases into account and build the normality model in such a way that we can cover all of them.

As illustrated in Figure 2, all communications from the clients to the database are separated by a session. We assign each session with a unique session ID. DoubleGuard normalizes the variable values in both HTTP requests and database queries, preserving the structures of the requests and queries. To achieve this, DoubleGuard substitutes the actual values of the variables with symbolic values. Figure15 depicts an example of the normalizations of the captured requests and queries.

Following this step, session i will have a set of requests, which is Ri , as well as a set of queries, which is Qi . If the total number of sessions of the training phase is N , then we have the set of total web requests REQ and the set of total SQL queries SQL across all sessions. Each single web request rm ∈ REQ may also appear several times in different Ri where i can be 1, 2 ... N . The same holds true for qn ∈ SQL.

A.Inferring Mapping Relations

If several SQL queries, such as qn , qp , are always found within one HTTP request of rm , then we can usually have an exact mapping of rm → {qn , qp }. However, this is not always the case. Some requests will result in different queries based on the request parameters and the state of the web server. For example, for web request rm , the invoked query set can sometimes be {qn ,qp } or, at other times, {qp } or {qq ,qn ,qs }.

The probabilities for these queries are usually not the same. For 100 requests of rm , the set is at {qn ,qp } 75 times, at {qp } 20 times, and at {qq ,qn ,qs } only 5 times. In such a case, we can find the mapping of rm → qp is 100%, with a rm → qn possibility of 80% and a rm → qs occurrence at 5% of all cases. We define this first type of mapping as deterministic and the latter ones as non-deterministic.

Below, we classify the four possible mapping patterns. Since the request is at the origin of the data flow, we treat each request as the mapping source. In other words, the mappings in the model are always in the form of one request to a query set rm → Qn . The possible mapping patterns are as in Figure 7

Deterministic Mapping: This is the most common and perfectly-matched pattern. That is to say that web request rm appears in all traffic with the SQL queries set Qn . The mapping pattern is then rm → Qn (Qn = ∅). For any session in the testing phase with the request rm , the absence of a query set Qn matching the request indicates a possible intrusion.On the other hand, if Qn is present in the session traffic without the corresponding rm , this may also be the sign of an intrusion.In static websites, this type of mapping comprises the majority of cases since the same results should be returned for each time a user visits the same link.

Empty Query Set: In special cases, the SQL query set may be the empty set. This implies that the web request neither causes nor generates any database queries. For example, when a web request for retrieving an image GIF file from the same web server is made, a mapping relationship does not exist because only the web requests are observed. This type of mapping is called rm → ∅. During the testing phase, we keep these web requests together in the set EQS

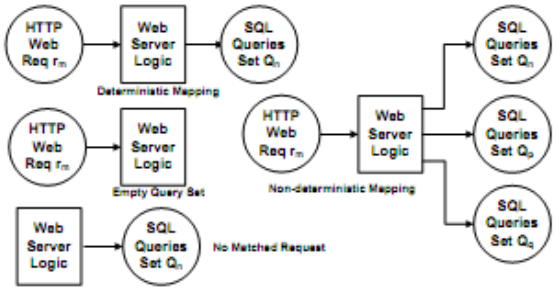

Fig 7: Overall representation of design patterns

.

No Matched Request: In some cases, the web server may periodically submit queries to the database server in order to conduct some scheduled tasks, such as cron jobs for archiving or backup. This is not driven by any web request, similar to the reverse case of the Empty Query Set mapping pattern. These queries cannot match up with any web requests, and we keep these unmatched queries in a set N M R. During the testing phase, any query within set N M R is considered legitimate.The size of N M R depends on web server logic, but it is typically small.

Non-deterministic Mapping: The same web request may result in different SQL query sets based on input parameters or the status of the web page at the time the web request is received. In fact, these different SQL query sets do not appear randomly, and there exists a candidate pool of query sets (e.g. {Qn , Qp , Qq ...}). Each time that the same type of web request arrives, it always matches up with one (and only one) of the query sets in the pool. The mapping pattern is rm → Qi (Qi ∈ {Qn , Qp , Qq ...}). Therefore, it is difficult to identify traffic that matches this pattern. This happens only within dynamic websites, such as blogs or forum sites.

Figure 7 illustrates all four mapping patterns.

B.Modeling for Static Websites:

In the case of a static website, the non-deterministic mapping does not exist as there are no available input variables or

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

states for static content. We can easily classify the traffic collected by sensors into three patterns in order to build the mapping model. As the traffic is already separated by session, we begin by iterating all of the sessions from 1 to N . For each rm ∈ REQ, we maintain a set ARm to record the IDs of sessions in which rm appears. The same holds for the database queries; we have a set AQs for each qs ∈ SQL to record all the session IDs. To produce the training model, we leverage the fact that the same mapping pattern appears many times across different sessions. For each ARm , we search for the AQs that equals the ARm . When ARm = AQs , this indicates that every time rm appears in a session then qs will also appear in the same session, and vice versa.

Given enough samples, we can confidently extract a mapping pattern rm → qs . Here, we use a threshold value t so that if the mapping appears in more than t sessions (e.g., the cardinality of ARm or AQs is greater than t), then a mapping pattern has been found. If such a pattern appears less than t times, this indicates that the number of training sessions is insufficient. In such a case, scheduling more training sessions is recommended before the model is built, but these patterns can also be ignored since they may be incorrect mappings. In our experiments, we set t to 3, and the results demonstrate that the requirement was easily satisfied for a static website with a relatively low number of training sessions

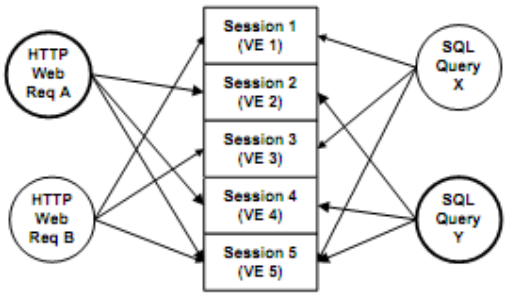

Fig .8:deterministic mapping using session ID of container (VE)

Figure 8 illustrates the use of the session ID provided by the container (VE) in order to build the deterministic mapping between http requests and the database requests. The request rA has the set ARA of {2,4,5}, which equals to AQY . Therefore, we can decide a Deterministic Mapping rA → qY . We developed an algorithm that takes the input of training dataset and builds the mapping model for static websites. For each unique HTTP request and database query, the algorithm assigns a hash table entry, the key of the entry is the request or query itself, and the value of the hash entry is AR for the request or AQ for the query respectively. The algorithm generates the mapping model by considering all three mapping patterns that would happen in static websites.

C.Testing for static websites

Once the normality model is generated, it can be employed for training and detection of abnormal behavior. During the testing phase, each session is compared to the normality model. We begin with each distinct web request in the session and, since each request will have only one mapping rule in the model, we simply compare the request with that rule. The testing phase algorithm is as follows:

1) If the rule for the request is Deterministic Mapping r → Q (Q = ∅), we test whether Q is a subset of a query set of the session. If so, this request is valid, and we mark the queries in Q. Otherwise, a violation is detected and considered to be abnormal, and the session will be marked as suspicious.
2) If the rule is Empty Query Set r → ∅, then the request is not considered to be abnormal, and we do not mark any database queries. No intrusion will be reported.
3) For the remaining unmarked database queries, we check to see if they are in the set N M R. If so, we mark the query as such.
4) Any untested web request or unmarked database query is considered to be abnormal. If either exists within a session, then that session will be marked as suspicious.

In our implementation and experimenting of the static testing website, the mapping model contained the Deterministic Mappings and Empty Query Set patterns without the No Matched Request pattern. This is commonly the case for static websites. As expected, this is also demonstrated in our experiments in section V.

D.Modeling of Dynamic Patterns

In contrast to static web pages, dynamic web pages allow users to generate the same web query with different parameters. Additionally, dynamic pages often use POST rather than GET methods to commit user inputs. Based on the web server's application logic, different inputs would cause different database queries. For example, to post a comment to a blog article, the web server would first query the database to see the existing comments. If the user's comment differs from previous comments, then the web server would automatically generate a set of new queries to insert the new post into the back-end database. Otherwise, the web server would reject the input in order to prevent duplicated comments from being posted (i.e., no corresponding SQL query would be issued.) In such cases, even assigning the same parameter values would cause different set of queries, depending on the previous state of the website. Likewise, this non-deterministic mapping case (i.e., one-to-many mapping) happens even after we normalize all parameter values to extract the structures of the web requests and queries. Since the mapping can appear differently in different cases, it becomes difficult to identify all of the one-to-many mapping patterns for each web request. Moreover,when different operations occasionally overlap at their possible query set, it becomes even harder for us to extract the one-to-many mapping for each operation by comparing matched requests and queries across the sessions.

E.Detection of Dynamic Websites

Once we build the separate single operation models, they can be used to detect abnormal sessions. In the testing phase,

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

traffic captured in each session is compared with the model. We also iterate each distinct web request in the session. For each request, we determine all of the operation models that this request belongs to, since one request may now appear in several models. We then take the entire corresponding query sets in these models to form the set CQS. For the testing session i, the set of DB queries Qi should be a subset of the CQS. Otherwise, we would find some unmatched queries. For the web requests in Ri , each should either match at least one request in the operation model or be in the set EQS. If any unmatched web request remains, this indicates that the session has violated the mapping model.

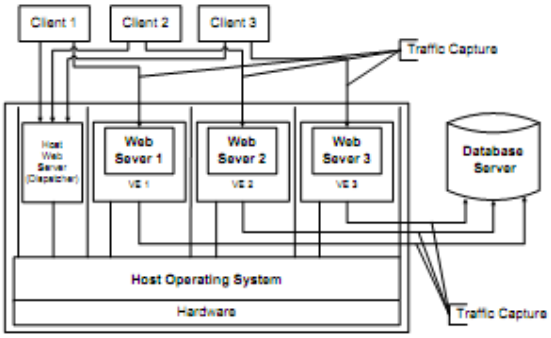

Fig.9: Overall architecture of Prototype

V.PERFORMANCE EVALUATION

We implemented a prototype of DoubleGuard using a web server with a back-end DB. We also set up two testing websites, one static and the other dynamic. To evaluate the detection results for our system, we analyzed four classes of attacks, as discussed in Section III, and measured the false positive rate for each of the two websites.

A. *Implementation*

In our prototype, we chose to assign each user session into a different container; however this was a design decision. For instance, we can assign a new container per each new IP address of the client. In our implementation, containers were recycled based on events or when sessions time out. We were able to use the same session tracking mechanisms as implemented by the Apache server (cookies, mod usertrack, etc) because lightweight virtualization containers do not impose high memory and storage overhead. Thus, we could maintain a large number of parallel-running Apache instances similar to the Apache threads that the server would maintain in the scenario without containers. If a session timed out, the Apache instance was terminated along with its container. In our prototype implementation, we used a 60-minute timeout due to resource constraints of our test server. However, this was not a limitation and could be removed for a production environment where long-running processes are required. Figure 9 depicts the architecture and session assignment of our prototype, where the host web server works as a dispatcher.

Initially, we deployed a static testing website using the Joomla Content Management System. In this static website, updates can only be made via the back-end management interface. This was deployed as part of our center website in production environment and served 52 unique web pages. For our analysis, we collected real traffic to this website for more than two weeks and obtained 1172 user sessions.

To test our system in a dynamic website scenario, we set up a dynamic Blog using the Wordpress blogging software. In our deployment, site visitors were allowed to read, post, and comment on articles. All models for the received front-end and back-end traffic were generated using this data.

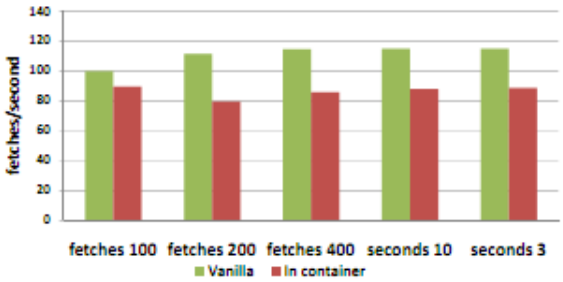

Fig 10: Performance evaluation using http load. The overhead is between 10.3% to 26.2%

B.*Container Overhead*

One of the primary concerns for a security system is its performance overhead in terms of latency. In our case, even though the containers can start within seconds, generating a container on-the-fly to serve a new session will increase the response time heavily. To alleviate this, we created a pool of web server containers for the forthcoming sessions akin to what Apache does with its threads. As sessions continued to grow, our system dynamically instantiated new containers. Upon completion of a session, we recycled these containers by reverting them to their initial clean states.

When we put the parameters at 3 and 10 seconds, the overhead was about 23%. We also tested using autobench, which is a Perl script wrapper around httperf. It can automatically compare the performance of two websites. We tested demanding rate ranging from 10 to 190, which means that a series of tests started at 10 requests per second and increased by 20 requests per second until 190 requests per second were being requested;any responses that took longer than 10 seconds to arrive were counted as errors. We compared the actual requests rates and the replay rates for bothservers

Figure11 shows that when the rate was less than 110 concurrent sessions per second, both servers could handle requests fairly well. Beyond that point, the rates in the container based server showed a drop: for 150 sessions per

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

second, the maximum overhead reflected in the reply rate was around 21% (rate of 130). Notice that 21% was the worst case scenario for this experiment, which is fairly similar to 26.2% in the http load experiment. When the server was not overloaded,and for our server this was represented by a rate of less than110 concurrent sessions per second, the performance overhead was negligible.

Fig .11. Performance evaluation using auto bench

C.Static Web page model in training phase

For the static website, we used the algorithm in Section IV-B to build the mapping model, and we found that only the Deterministic Mapping and the Empty Query Set Mapping patterns appear in the training sessions. We expected that the No Matched Request pattern would appear if the web application had a cron job that contacts back-end database server; however, our testing website did not have such a cron job. We first collected 338 real user sessions for a training dataset before making the website public so that there was no attack during the training phase.

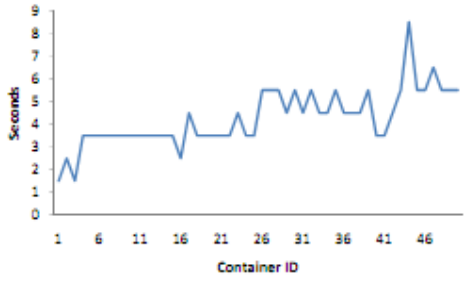

Fig 12. Time of starting new container

correctly build the entire model. Based on this training process accuracy graph, we can determine a proper time to stop the training.

D.Dynamic modeling and detection dates

We also conducted model building experiments for the dynamic blog website. We obtained 329 real user traffic sessions from the blog under daily workloads.

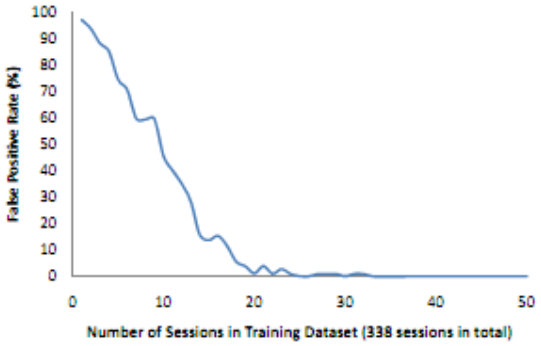

Fig 13.False positives vs Training time in static website

Single Operation	No. of requests	No. of queries
Read an article	3	23
Post an article	10	49
Make Comment to an article	2	9
Visit next page	2	18
List articles by categories	3	19
List articles by posted months	3	16
Read RSS feed	1	2
Cron jobs	1	11
Visit by page number	2	18

TABLE I
SINGLE OPERATION MODELS EXAMPLE

The model building for a dynamic website is different from that for a static one. We first manually listed 9 common operations of the website, which are presented in Table I. To build a model for each operation, we used the automatic tool Selenium to generate traffic.

E.Attack Detection

Once the model is built, it can be used to detect malicious sessions. For our static website testing, we used the production website, which has regular visits of around 50-100 sessions per day. We collected regular traffic for this production site, which totaled 1172 sessions.

1) Privilege Escalation Attack: For Privilege Escalation Attacks, according to our previous discussion, the attacker visits the website as a normal user aiming to compromise the web server process or exploit vulnerabilities to bypass authentication. At that point, the attacker issues a set of privileged (e.g., admin-level) DB queries to retrieve sensitive information. We log and process both legitimate web requests and database queries in the session traffic, but there are no mappings among them. during the training phase,DoubleGuard can capture the unmatched cases

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

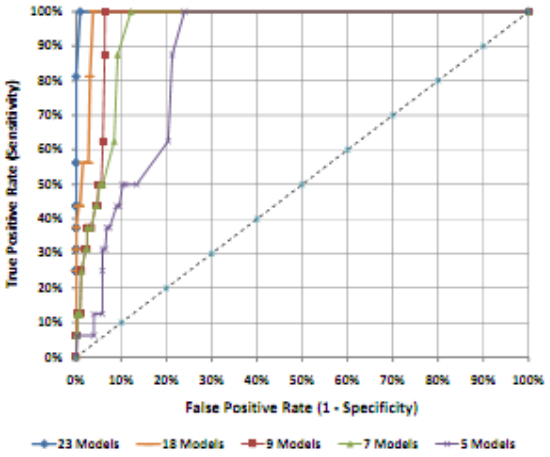

Fig 14.ROC curves for dynamic models

2) Hijack Future Session Attack (Web Server aimed attack): Out of the four classes of attacks we discuss, session hijacking is the most common, as there are many examples that exploit the vulnerabilities of Apache, IIS, PHP, ASP, and cgi, to name a few. Most of these attacks manipulate the HTTP requests to take over the web server. We first ran Nikto.

3) Injection Attack: Here we describe how our approach can detect the SQL injection attacks. To illustrate with an example, we wrote a simple PHP login page that was vulnerable to SQL injection attack. As we used a legitimate username and password to successfully log in, we could include the HTTP request in the second line of Figure 15 *4) Direct DB attack*: If any attacker launches this type of attack, it will easily be identified by our approach. First of all, according to our mapping model, DB queries will not have any matching web requests during this type of attack. On the other hand, as this traffic will not go through any containers, it will be captured as it appears to differ from the legitimate traffic that goes through the containers.

Operation	Snort	GSQL	DG
Privilege Escalation (WordPress Vul)	No	No	Yes
Web Server aimed attack (nikto)	Yes	No	Yes
SQL Injection (sqlmap)	No	Yes	Yes
DirectDB	No	No	Yes
linux/http/ddwrt_cgibin_exec*	No	No	Yes
linux/http/linksys_apply_cgi*	No	No	Yes
linux/http/piranha_passwd_exec*	No	No	Yes
unix/webapp/oracle_vm_agent_utl*	No	No	Yes
unix/webapp/php_include*	Yes	No	Yes
unix/webapp/php_wordpress_lastpost*	No	No	Yes
windows/http/altn_webadmin*	No	No	Yes
windows/http/apache_modjk_overflow *	No	No	Yes
windows/http/oracle9i_xdb_pass*	No	No	Yes
windows/http/maxdb_webdbm_database*	No	No	Yes

TABLE II
DETECTION RESULTS FOR ATTACKS (GSQL STANDS FOR GREENSQL, AND DG STANDS FOR DOUBLEGUARD, * INDICATES ATTACK USING METASPLOIT)

VI. CONCLUSION

We presented an intrusion detection system that builds models of normal behavior for multi-tiered web applications from both front-end web (HTTP) requests and back-end database (SQL) queries. Unlike previous approaches that correlated or summarized alerts generated by independent IDSes, DoubleGuard forms a container-based IDS with multiple input streams to produce alerts. Such correlation of different data streams provides a better characterization of the system for anomaly detection because the intrusion sensor has a more precise normality model that detects a wider range of threats.

We achieved this by isolating the flow of information from each web server session with a lightweight virtualization. Furthermore, we quantified the detection accuracy of our approach when we attempted to model static and dynamic web requests with the back-end file system and database queries.For static websites, we built a well-correlated model, which our experiments proved to be effective at detecting different types of attacks. Moreover, we showed that this held true for dynamic requests where both retrieval of information and updates to the back-end database occur using the web-server front end. When we deployed our prototype on a system that employed Apache web server, a blog application and a MySQL back-end, DoubleGuard was able to identify a wide range of attacks with minimal false positives. As expected, the number of false positives depended on the size and coverage of the training sessions we used. Finally, for dynamic web applications, we reduced the false positives to 0.6%..

REFERENCES

[1]C. Anley. Advanced sql injection in sql server applications. Technical report, Next Generation Security Software, Ltd, 2002.
[2]K. Bai, H. Wang, and P. Liu. Towards database firewalls. In DBSec 2005.
[3]B. I. A. Barry and H. A. Chan. Syntax, and semantics-based signature database for hybrid intrusion detection systems. Security and Commu- nication Networks, 2(6), 2009.
[4]D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In Proceedings of the 19th international conference on World wide web, 2010.
[5]M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns.
[6]M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In RAID 2007.
[7] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

IJARMATE
International Journal of Advanced Research in Management, Architecture, Technology and Engineering
(IJARMATE) Vol. IV, Issue VI, June 2018

of intrusion-
detection systems. Computer Networks, 31(8), 1999.
[8] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Au-
tomated Detection of Logic Vulnerabilities in Web Applications. In
Proceedings of the USENIX Security Symposium, 2010.
[9] Y. Hu and B. Panda. A data mining approach for database intrusion
detection. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M.
Liebrock, editors, SAC. ACM, 2004.
[10] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia. Efficiently tracking
application interactions using lightweight virtualization. In Proceedings
of the 1st ACM workshop on Virtual machine security, 2008.
[11] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed
worm signature detection. In USENIX Security Symposium, 2004.
[12] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In
Proceedings of the 10th ACM Conference on Computer and Communi-
cation Security (CCS '03), Washington, DC, Oct. 2003. ACM Press.
[13] Lee, Low, and Wong. Learning fingerprints for a database intrusion
detection system. In ESORICS: European Symposium on Research in
Computer Security. LNCS, Springer-Verlag, 2002.
[14] Liang and Sekar. Fast and automated generation of attack signatures:
A basis for building self-protecting servers. In SIGSAC: 12th ACM
Conference on Computer and Communications Security, 2005.
[15] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically
generating signatures for polymorphic worms. In IEEE Symposium on
Security and Privacy. IEEE Computer Society, 2005.
able Sec. Comput, 1(3), 2004.

.