Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 1, January 2015, pg.374 – 382

RESEARCH ARTICLE

Intrusion Detection System for Multitier Web Based Application

Mr. Ratnakar S Jagale, Prof. M M Naoghare

Department of Computer Engineering, SVIT, Sinnar, Maharashtra & Savitribai Phule Pune University, India

Department of Computer Engineering, SVIT, Sinnar, Maharashtra & Savitribai Phule Pune University, India

rjagale@gmail.com; manisha.naoghare@gmail.com

Abstract— Computing 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 multitier design wherein the web server runs the application front-end logic and data are outsourced to a database or file server. In this report, we present Network Intrusion Detection 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 independent IDS would not be able to identify. Furthermore, we quantify the limitations of any multitier IDS in terms of training sessions and functionality coverage.

We implement Intrusion Detection System using an Apache web server with MySQL and lightweight virtualization. It is a system used to detect attacks in multitier web services. Our approach can create normality models of isolated user sessions that include both the web front-end (HTTP) and back-end (File or SQL) network transactions. To achieve this, we employ a lightweight virtualization technique to assign each user's web session to a dedicated container, an isolated virtual computing environment. We can use the container ID to accurately associate the web request with the subsequent DB queries. Thus, Network Intrusion Detection System can build a causal mapping profile by taking both the web server and DB traffic into account.

Keywords— Anomaly detection, virtualization, multitier web application

I. INTRODUCTION

We present an Intrusion Detection System, a system used to detect attacks in multi-tiered web services. Our approach can create normality models of isolated user sessions that include both the web front-end (HTTP) and back-end (File or SQL) network transactions. To achieve this, We employ a lightweight virtualization technique to assign each user's web session to a dedicated container, an isolated virtual computing environment. We use the container ID to accurately associate the web request with the subsequent DB queries. Thus, Network Intrusion Detection System can build a causal mapping profile by taking both the web server and DB traffic into account.

II. LITERATURE SURVEY

A network Intrusion Detection System 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 [10], [11]. 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 [12], [17], [20] or by using rule-based approaches to specify behavior patterns [19]. 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 are a number of approaches [18] that are trying to solve this problem. Our detection may run into the same problem; in such a case, our model should be retrained for each shift. Intrusion alerts correlation [17] 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. Our IDS differs from this type of approach that correlates alerts from independent IDSs. Rather, our IDS operates on multiple feeds of network traffic using single IDS that looks across sessions to produce an alert without correlating or summarizing the alerts produced by other independent IDSs.

An IDS such as in [16] also uses temporal information to detect intrusions. Our IDS, however, does not correlate events on a time basis, which runs the risk of mistakenly considering independent but concurrent events as correlated events. Our IDS does not have such a limitation as it uses the container ID for each session to causally map the related events, whether they are concurrent or not. 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 [2], Parallels Virtuozzo [3], or Linux-VServer [15]. 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, yet it 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 [14], [15] that use lightweight virtualization to isolate different application instances. Such virtualization techniques are commonly used for isolation and containment of attacks. However, in our IDS, we utilized the container ID to separate session traffic as a way of extracting and identifying causal relationships between web server requests and database query events.

III. PROBLEM STATEMENT

There are four major attacks will be possible on the multi-tier web based applications.

Privilege Escalation Attack

Let's assume that the website serves both regular users and administrators. For a regular user, the web request R_u will trigger the set of SQL queries Q_u for an administrator, the request R_a will trigger the set of admin level queries Q_a. 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 R_u and Q_a are legitimate requests and queries. Our approach, however, can detect this type of attack since the DB query Q_a does not match the request R_u, according to our mapping model. Fig. 3.1 shows how a normal user may use admin queries to obtain privileged information.

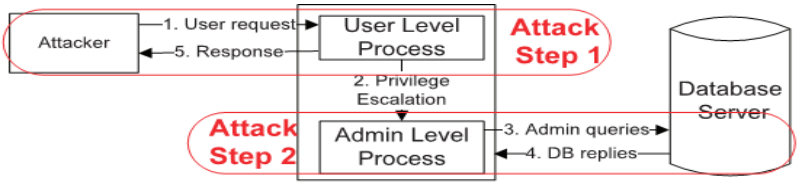

Fig. 3.1 Privilege Escalation Attack.

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. Fig. 3.2 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, and 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. Fortunately, the isolation property of our container based web server architecture can also prevent this type of attack. As each user's web requests are isolated into a separate container, an attacker can never break into other user's sessions.

Fig. 3.2 Hijack Future Session Attack

SQL Injection Attack

Attacks such as SQL injection do not require compromising the web server. Attackers can use existing vulnerabilities in the web server logic to inject the data or string content that contains the exploits and then use the web server to relay these exploits to attack the back-end database. Since our approach provides two-tier detection, even if the exploits are accepted by the web server, the relayed contents to the database server would not be able to take on the expected structure for the given web server request. For instance, since the SQL injection attack changes the structure of the SQL queries, even if the injected data were to go through the web server side, it would generate SQL queries in a different structure that could be detected as a deviation from the SQL query structure that would normally follow such a web request. Fig. 1.3 illustrates the scenario of a SQL injection attack.

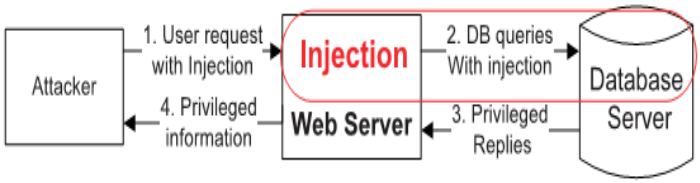

Fig. 3.3 SQL Injection Attack.

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 it would not detect it either. However, this type of attack can be caught with our approach since we cannot match any web requests with these queries. Fig. 3.4 illustrates the scenario wherein an attacker bypasses the web server to directly query the database.

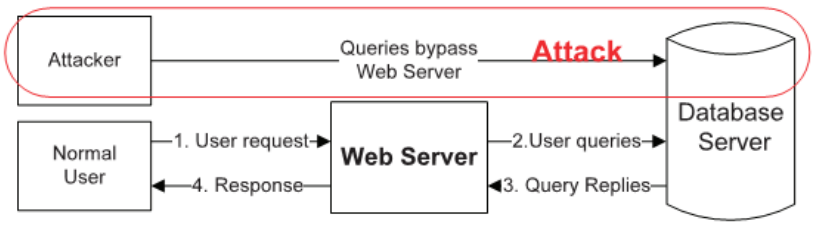

Fig. 3.4 Direct DB Attack.

IV. SYSTEM DESIGN

Architecture and Confinement

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 system 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 session's 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.

Classic three-tier model

The webserver acts as the front end, with the file and database servers as the content storage back end. Fig. 4.1 illustrates the classic three-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. In practice, we are unable to build such mapping under a classic three-tier setup. Although the web server can distinguish sessions from different clients, the SQL queries are mixed and all from the same web server. It is impossible for a database server to determine which SQL queries are the results of which web requests, much less to find out the relationship between them. Even if we knew the application logic of the web server and were to build a correct model, it would be impossible to use such a model to detect attacks within huge amounts of concurrent real traffic unless we had a mechanism to identify the pair of the HTTP request and SQL queries that are causally generated by the HTTP request.

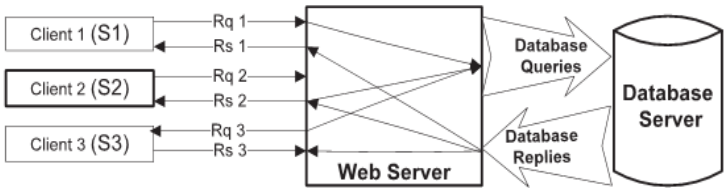

Fig. 4.1 Classic three-tier model.

Container based Model

Within our container-based web servers, it is a straightforward matter to identify the causal pairs of web requests and resulting SQL queries in a given session. Moreover, as traffic can easily be separated by session, it becomes possible for us to compare and analyze the request and queries across different sessions. Fig. 4.2 depicts how communications are categorized as sessions and how database transactions can be related to a corresponding session. According to Fig. 4.2, if Client 2 is malicious and takes over the webserver, all subsequent database transactions become suspect, as well as the response to the client. By contrast, according to Fig. 4.2, 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.

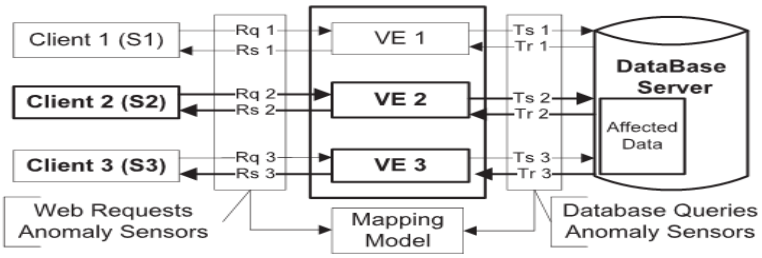

Fig. 4.2 Web server instances running in containers.

We put sensors at both sides of the servers. At the web server, our sensors are deployed on the host system and cannot be attacked directly since only the virtualized containers are exposed to attackers. Our sensors will not be attacked at the database server either, as we assume that the attacker cannot completely take control of the database server. In fact, we assume that our sensors cannot be attacked and can always capture correct traffic information at both ends. Fig. 5.2 shows the locations of our sensors.

V. MODELING DETERMINISTIC MAPPING AND PATTERNS

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 the mapping between the web server requests and the subsequent Database queries, and to utilize such a mapping model to detect abnormal behaviors on a session/client level. In typical three-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. Once we build the mapping model, it can be used to detect abnormal behaviors. Both the web request and the database queries within each session should be in accordance with the model. If there exists any request or query that violates the normality model within a session, then the session will be treated as a possible attack.

Due to their diverse functionality, different web applications exhibit different characteristics. Many websites serve only static content, which is updated and often managed by a Content Management System (CMS). For a static website, we can build an accurate model of the mapping relationships between web requests and database queries since the links are static and clicking on the same link always returns the same information. However, some websites (e.g., blogs, forums) allow regular users with non-administrative privileges to update the contents of the serve data. This creates tremendous challenges for IDS system training because the HTTP requests can contain variables in the pass parameters.

Deterministic Mapping

This is the most common and perfectly matched pattern. That is to say that web request R_m appears in all traffic with the SQL queries set Q_n. The mapping pattern is then $R_m \rightarrow Q_n$ ($Q_n \neq \theta$). For any session in the testing phase with the request R_m, the absence of a query set Q_n matching the request indicates a possible intrusion. On the other hand, if Q_n is present in the session traffic without the corresponding R_m, 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 $R_m \rightarrow \theta$. During the testing phase, we keep these web requests together in the set EQS.

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 NMR. During the testing phase, any query within set NMR is considered legitimate. The size of NMR depends on web server logic, but it is typically small.

Nondeterministic 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., $\{Q_n, Q_p, Q_q ...\}$). 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 → Q_i (Q_i ϵ {Q_n,Q_p,Q_q ...}). Therefore, it is difficult to identify traffic that matches this pattern. This happens only within dynamic websites, such as blogs or forum sites. Fig. 5.1 illustrates all four mapping patterns.

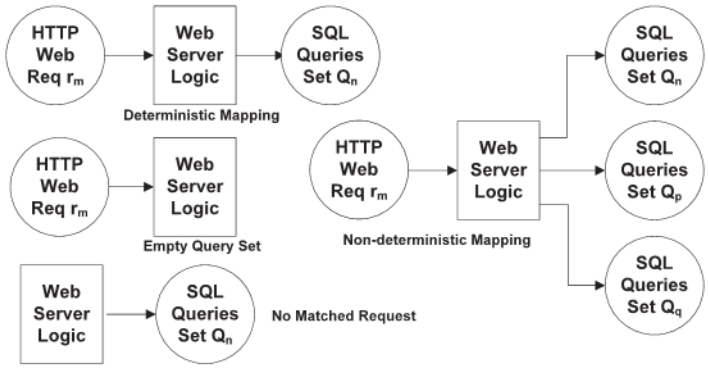

Fig. 5.1 Overall representation of mapping patterns

VI. ALGORITHM

Algorithm Name: Static Model Building Algorithm
Require: Training Data set, Threshold t
Ensure: The Mapping Model for static website
1. For each session separated traffic Ti do.
2. Get different HTTP requests r and DB queries q in this session.
3. for each different r do.
4. If r is a request to static file then.
5. Add r into set EQS.
6. Else
7. If r is not in set REQ then
8. Add r into REQ.
9. Append session ID i to the set ARr with r as the key.
10. for each different q do
11. If q is not in set SQL then
12. Add q into SQL
13. Append session ID i to the set AQq with q as the key
14. For each distinct HTTP request r in REQ do
15. For each distinct DB query q in SQL do
16. Compare the set ARr with the set AQq
17. If ARr = AQq and Cardinality (ARr) > t then
18. Found a Deterministic mapping from r to q
19. Add q into mapping model set MSr of r
20. Mark q in set SQL
21. Else
22. Need more training sessions.
23. Return False.
24. For each DB query q in SQL does.
25. If q is not marked then
26. Add q into set NMR.
27. for each HTTP request r in REQ do.
28. If r has no deterministic mapping model then
29. Add r into set EQS.
30. Return True.

VII. RESULT

Experimental dataset

We conducted model building experiments for dynamic blog website. We obtained 329 real user traffic sessions from blog under daily workloads. During this seven day phase, we made our website available only to internal users to ensure that no attacks would occur . Then we generate 20 attack traffic sessions mixed with these legitimate sessions, and the mixed traffic was used for detection. The model building for a dynamic website is

different from that for a static one. We first manually listed nine common operations of the website, which are presented in Table 7.1 In each session, we put only a sing le operation, which we iterated 50 times with different values in the parameters. Finally, We obtained separate models f or each sing le operation.

Sr.No.	Single Operation	Number of Request	Number of Queries
01	Read an article	03	23
02	Post an article	10	49
03	Make comment to an article	02	09
04	Visit next page	02	18
05	List article by categories	03	19
06	List article by posted month	03	16
07	Visit by page number	02	18

Table 7.1 Single operation model dataset.

Test Cases and Results

Test case is a commonly used term for a specific test. It consists of information such as test steps, verification steps, prerequisites, outputs, test environment, etc. It is a set of inputs, execution preconditions and expected outcomes. Test cases are executed against software products, and then expected outcomes are compared with actual results to determine test has failed or passed. The test cases to be verified for the project as per mentioned below:

Test case Id.	Test carried out	Snort	GSQL	IDS
TC01	Privilege Escalation Attack	NO	NO	YES
TC02	SQL Injection Attack	No	YES	YES
TC03	Direct DB Attack	NO	NO	YES
TC04	Web server aimed attack	YES	NO	YES

Table 7.2 Test cases and Results.

Comparative Result

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. We evaluated the overhead of our container-based server against a vanilla web server. For the http load evaluation, we used the rate of five (i.e., it emulated five concurrent users). we tested under the parameters of 100, 200, and 400 total fetches, as well as 3 and 10 seconds of fetches. For example, in the 100-fetches benchmark, http load fetches the URLs as fast as it can 100 times. Similarly, in the 10-seconds benchmark, http load fetches the URLs as fast as it can during the last 10 seconds. we picked 15 major URLs of the website and tested them against both servers. Fig. 8.1 shows our experiment results.

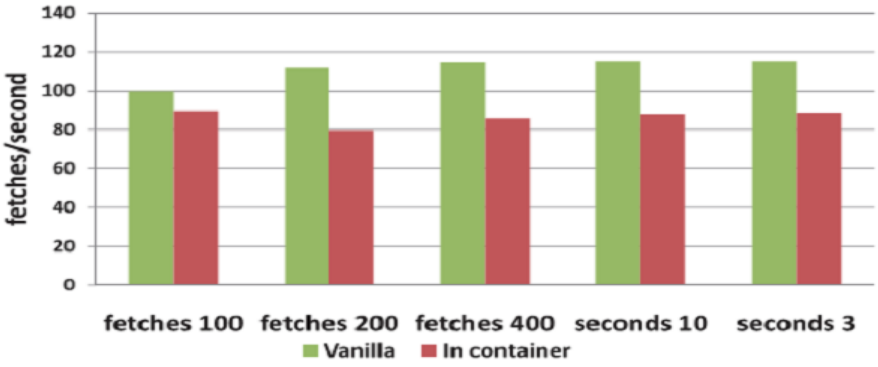

Fig 7.1 Performance evaluation using http load.

VIII. CONCLUSIONS

We presented an intrusion detection system that builds models of normal behavior for multitier 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 IDSs, Network IDS forms container-based IDS with multiple input streams to produce alerts. We have shown that such correlation of input 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, our IDS 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 percent.

ACKNOWLEDGEMENT

It is a great pleasure to acknowledge those who extended their support, and contributed time and psychic energy for the completion of this project work. At the outset, I would like to thank my project guide Prof. M. M. Naoghare, who served as sounding board for both contents and programming work. Her valuable and skilful guidance, assessment and suggestions from time to time improved the quality of work in all respect. I would like to take this opportunity to express my deep sense of gratitude towards him, for her invaluable contribution in completion of this project.

I am also thankful to Prof. S. M. Rokade, Head of Computer Engineering Department for his timely guidance, inspiration and administrative support without which my work would not have been completed. I am also thankful to the all staff members of Computer Engineering Department and Librarian, SVIT Chincholi, Nasik. Also I would like to thank my colleagues and friends who helped me directly and indirectly to complete this Project. Lastly my special thanks to my family members for their support and co-operation during this project work

REFERENCES

[1] SANS, The Top Cyber Security Risks, http://www.sans.org/top-cyber-securityrisks/, 2011.
[2] Openvz, http://wiki.openvz.org, 2011.
[3] Virtuozzo Containers, http://www.parallels.com/products/pvc45/, 2011.
[4] C. Anley, Advanced Sql Injection in Sql Server Applications, technical report, Next Generation Security Software, Ltd., 2002.
[5] K. Bai, H. Wang, and P. Liu, Towards Database Firewalls, Proc. Ann. IFIP WG 1.3 Working Conf. Data and Applications Security (DBSec 05), 2005.
[6] B. Parno, J.M. McCune, D. Wendlandt, D.G. Andersen, and A.Perrig, CLAMP: Practical Prevention of Large-Scale Data Leaks,Proc. IEEE Symp. Security and privacy, 2009.
[7] T. Pietraszek and C.V. Berghe, Defending against Injection Attacks through Context-Sensitive String Evaluation, Proc. Intl Symp. Recent Advances in Intrusion Detection (RAID 05), 2005.
[8] R. Sekar, an Efficient Black-Box Technique for Defeating Web Application Attacks, Proc. Network and Distributed System Security Symp. (NDSS), 2009.

[9] greensql, http://www.greensql.net/, 2011.
[10] H. Debar, M. Dacier, and A. Wespi, "Towards Taxonomy of Intrusion-Detection Systems," Computer Networks, vol. 31, no. 9, pp. 805-822, 1999.
[11] T. Verwoerd and R. Hunt, "Intrusion Detection Techniques and Approaches," Computer Comm., vol. 25, no. 15, pp. 1356-1365, 2002.
[12] C. Kruegel and G. Vigna, "Anomaly Detection of Web-Based Attacks," Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.
[13] Y. Huang, A. Stavrou, A.K. Ghosh, and S. Jajodia, "Efficiently Tracking Application Interactions Using Lightweight Virtualization," Proc. First ACM Workshop Virtual Machine Security, 2008.
[14] S. Potter and J. Nieh, "Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems," Proc. USENIX Ann. Technical Conf., 2010.
[15] Linux-vserver, http://linux-vserver.org/, 2011.
[16] A. Seleznyov and S. Puuronen, "Anomaly Intrusion Detection Systems: Handling Temporal Relations between Events," Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID '99), 1999.
[17] F. Valeur, G. Vigna, C. Kru ¨ gel, and R.A. Kemmerer, "A Comprehensive Approach to Intrusion Detection Alert Correlation," IEEE Trans. Dependable and Secure Computing, vol. 1, no. 3, pp. 146-169, July-Sept. 2004.
[18] A. Stavrou, G. Cretu-Ciocarlie, M. Locasto, and S. Stolfo, "Keep Your Friends Close: The Necessity for Updating an Anomaly Sensor with Legitimate Environment Changes," Proc. Second ACM Workshop Security and Artificial Intelligence, 2009.
[19] M. Roesch, "Snort, Intrusion Detection System," http://www.snort.org, 2011.
[20] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-Based Detection of State Violations in Web Applications," Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID '07), 2007.

