The International Journal Of Engineering And Science (IJES)
||Volume|| 2 ||Issue|| 10 ||Pages|| 17-22 ||2013||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

Identifying Distributed Dos Attacks in Multitier Web Applications

Thatikonda.Namratha
Department of Computer Science and Engineering, KITS, Warangal.
Andhra Pradesh, India.

Sunkari Venkatramulu
Associate Professor
Department of Computer Science and Engineering, KITS, Warangal.
Andhra Pradesh, India.

ABSTRACT

World-wide-web services as well as applications have become an inextricable component of daily living, enabling communication as well as the management of sensitive information from wherever. To allow for this boost in application and info complexity, web solutions have moved to your multi-tiered design and style wherein the web server runs the application form front-end logic and info are outsourced to your database or even file server. In this particular paper, we present an IDS system that product the behavior associated with user across the front-end web server as well as the back-end repository. By keeping track of both World Wide Web and pursuing database asks for, we can ferret out attacks in which independent IDS would not be capable of identify. On top of that, we quantify the restrictions of virtually any multitier IDS regarding training consultations and features coverage.

Keywords *- Intrusion Detection, Web Applications.*

Date of Submission: 10th, October, 2013 ⟺ Date of Acceptance: 30th, October, 2013

I. INTRODUCTION

Intrusion detection plays one of the key roles in computer system security techniques. 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 alerts. 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[10]. 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 to corrupt the back-end database system. A plethora of Intrusion Detection Systems (IDSs) currently examine network packets individually within both the web server and the database system. However, there is very little work being performed on multitier Anomaly Detection[1] (AD) systems that generate models of network behavior for both web and database network interactions. In such multitier 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 multitier web services, Intrusion detection systems 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 IDSs cannot detect cases wherein normal traffic is used 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[10].

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 server is not associated with user-privileged access. Unfortunately, within the current multithreaded 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 clearly attributed to user sessions.

II. PREVIOUS WORK

Researchers also proposed dynamic techniques against SQLIAs that do not rely on tainting[8]. These techniques include Intrusion Detection Systems (IDSs) and automated penetration testing tools[3]. Scott and Sharp propose Security Gateway[5], which uses developer-provided rules to filter Web traffic, identify attacks, and apply preventive transformations to potentially malicious inputs. The success of this approach depends on the ability of developers to write accurate and meaningful filtering rules. Similarly, Valeur et al. developed an IDS that uses machine learning to distinguish legitimate and malicious queries. Their approach, like most learning-based techniques, is limited by the quality of the IDS training set. Machine learning was also used in WAVES, an automated penetration testing tool that probes Web sites for vulnerability to SQLIAs. Like all testing tools, WAVES cannot provide any guarantees of completeness. SQLrand appends a random token to SQL keywords and operators in the application code. A proxy server then checks to make sure that all keywords and operators contain this token before sending the query to the database. Because the SQL keywords and operators injected by an attacker would not contain this token, they would be easily recognized as attacks. The drawbacks of this approach are that the secret token could be guessed, thus making the approach ineffective, and that the approach requires the deployment of a special proxy server.

Model-based approaches against SQLIAs include AMNESIA, SQL-Check, and SQLGuard[7]. AMNESIA, previously developed by two of the authors, combines static analysis and runtime monitoring to detect SQLIAs. The approach uses static analysis to build models of the different types of queries that an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Queries that do not match the model are identified as SQLIAs. A problem with this approach is that it is dependent on the precision and efficiency of its underlying static analysis, which may not scale to large applications. Our new technique takes a purely dynamic approach to preventing SQLIAs, thereby eliminating scalability and precision problems. SQLCheck identifies SQLIAs by using an augmented grammar and distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. The main weakness of this approach is that it requires the manual intervention of the developer to identify and annotate untrusted sources of input, which introduces incompleteness problems and may lead to false negatives. Our use of positive tainting eliminates this problem while providing similar guarantees in terms of effectiveness.

SQLGuard is an approach similar to SQLCheck. The main difference is that SQLGuard builds its models on the fly by requiring developers to call a special function and to pass to the function the query string before user input is added.

Other approaches against SQLIAs rely purely on static analysis. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQLIAs. Because of the inherently imprecise nature of the static analysis that they use, these techniques can generate false positives. Moreover, since they rely on declassification rules to transform untrusted input into safe input, they can also generate false negatives. Wassermann and Su propose a technique that combines static analysis and automated reasoning to detect whether an application can generate queries that contain tautologies. This technique is limited, by definition, in the types of SQLIAs that it can detect. Finally, researchers have investigated ways to statically eliminate vulnerabilities from the code of a Web application. Defensive coding best practices have been proposed as a possible approach, but they have limited effectiveness because they rely almost exclusively on the ability and training of developers. Moreover, there are many well-known ways to evade some defensive-coding practices, including “pseudo remedies” such as stored procedures and prepared statements. Researchers have also developed special libraries that can be used to safely create SQL queries. These approaches, although highly effective, require developers to learn new APIs, can be very expensive to apply on legacy code, and sometimes limit the expressiveness of SQL.

Dynamic taint analysis (also known as dynamic information flow analysis) consists, intuitively, in marking and tracking certain data in a program at run-time. This type of dynamic analysis is becoming increasingly popular. In the context of application security, dynamic-tainting[8] approaches have been successfully used to prevent a wide range of attacks, including buffer overruns, format string attacks, SQL and command injections, and cross-site scripting. More recently, researchers have started to investigate the use of tainting based approaches in domains other than security, such as program understanding, software testing, and debugging.

Unfortunately, most existing techniques and tools for dynamic taint analysis are defined in an ad-hoc manner, to target a specific problem or a small class of problems. It would be difficult to extend or adapt such techniques and tools so that they can be used in other contexts. In particular, most existing approaches are focused on data-flow based tainting only, and do not consider tainting due to the control flow within an application[6], which limits their general applicability. Also, most existing techniques support either a single taint marking or a small, fixed number of markings, which is problematic in applications such as debugging. Finally, almost no existing technique handles the propagation of taint markings in a truly conservative way, which may be appropriate for the specific applications considered, but is problematic in general. Because developing support for dynamic taint analysis is not only time consuming, but also fairly complex, this lack of flexibility and generality of existing tools and techniques is especially limiting for this type of dynamic analysis.

To address these limitations and foster experimentation with dynamic tainting[8] techniques, in this paper we present a framework for dynamic taint analysis. We designed the framework to be general and flexible, so that it allows for implementing different kinds of techniques based on dynamic taint analysis with little effort. Users can leverage the framework to quickly develop prototypes for their techniques, experiment with them, and investigate trade-offs of different alternatives. For a simple example, the framework could be used to investigate the cost effectiveness of considering different types of taint propagation for an application.

Intuitively, dynamic tainting tracks the information flow within a program by (1) associating one or more markings with some data values in the program and (2) propagating these markings as data values flow through the program during execution. Consider, for instance, the simple example. Imagine that we tainted the variables a at line 2 and b at line 3 with taint markings ta and tb, respectively. In such a case, we would expect, at the end of the execution, that the taint markings associated with variables x, y, and z would consist of sets {ta}, {tb}, and {ta, tb}, respectively. Taint marking ta, initially associated with a, would be associated with w because a's value is used to compute w. analogously, marking ta would be associated with y because the value of w, which is now tainted with ta, is used to compute y. The propagation of taint markings for the remaining variables is analogous.

III. PROPOSED WORK

3.1 Deterministic and Non Deterministic Mapping

The deterministic 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. 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. 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 webserver is made, a mapping relationship does not exist because only the web requests are observed. During the testing phase, we keep these web requests together in the set EQS. In some cases, the webserver 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 webserver logic, but it is typically small. The same web request may result in different SQL query sets based on input parameters or the status of the webpage 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. Each time that the same type of web request arrives, it always matches up with one of the query sets in the pool. Therefore, it is difficult to identify traffic that matches this pattern.

3.2 Modelling for Static Websites

In the case of a static website, the nondeterministic mapping does not exist as there are no available input variables or states for static content. We can easily classify the traffic collected by sensors[9] 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 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 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. After we confirm all deterministic mappings, we remove these matched requests and queries from REQ and SQL, respectively. Since multiple requests are often sent to the webserver within a short period of time by a single user operation, they can be mapped together to the same AQs. Some web requests that could appear separately are still present as a unit. For example, the read request always precedes the post request on the same webpage. During the training phase, we treat them as a single instance of web requests bundled together unless we observe a case when either of them appears separately. Our next step is to decide the other two mapping patterns by assembling a white list for static file requests, including JPG, GIF, CSS, etc. HTTP requests for static files are placed in the EQS set. The remaining requests are placed in REQ; if we cannot find any matched queries for them, they will also be placed in the EQS set. In addition, all remaining queries in SQL will be considered as No Matched Request cases and placed into NMR.

3.3 Modelling of Dynamic Patterns

In contrast to static webpage, dynamic webpage 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 nondeterministic mapping case 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.

3.3.1 SQL Tautologies Attacks

Tautology-based attacks are among the simplest and best known types of Attacks. The general goal of a tautology based attack is to inject SQL tokens that cause the query's conditional statement to always evaluate to true. Although the results of this type of attack are application specific, the most common uses are bypassing authentication pages and extracting data. In this type of injection, an attacker exploits a vulnerable input field that is used in the query's WHERE conditional. This conditional logic is evaluated as the database scans each row in the table. If the conditional represents a tautology, the database matches and returns all of the rows in the table as opposed to matching only one row, as it would normally do in the absence of injection.

3.3.2 SQL Union Queries

Although tautology-based attacks can be successful, for instance, in bypassing authentication pages, they do not give attackers much flexibility in retrieving specific information from a database. Union queries are a more sophisticated type of SQL Attacks that can be used by an attacker to achieve this goal, in that they cause otherwise legitimate queries to return additional data. In this type of SQLIA, attackers inject a statement of the form "UNION < injected query > ." By suitably defining < injected query > , attackers can retrieve information from a specified table. The outcome of this attack is that the database returns a data set that is the union of the results of the original query with the results of the injected query.

3.3.3 SQL Piggybacked Queries

Similar to union queries, this kind of attack appends additional queries to the original query string. If the attack is successful, the database receives and executes a query string that contains multiple distinct queries. The first query is generally the original legitimate query, whereas subsequent queries are the injected malicious queries. This type of attack can be especially harmful because attackers can use it to inject virtually any type of SQL command. In our example, an attacker could inject the text "0; drop table users" into the pin input field.

3.3.4 SQL Malformed Queries

Union queries and piggybacked queries let attackers perform specific queries or execute specific commands on a database, but require some prior knowledge of the database schema, which is often unknown. Malformed queries allow for overcoming this problem by taking advantage of overly descriptive error messages that are generated by the database when a malformed query is rejected. When these messages are directly returned to the user of the Web application[2], instead of being logged for debugging by developers, attackers can make use of the debugging information to identify vulnerable parameters and infer the schema of the underlying database. Attackers exploit this situation by injecting SQL tokens or garbage input that causes the query to contain syntax errors, type mismatches, or logical errors.

IV. RESULTS

The concept of this paper is implemented and different results are shown below, The proposed paper is implemented in Java technology on a Pentium-IV PC with minimum 20 GB hard-disk and 1GB RAM. The propose paper's concepts shows efficient results and has been efficiently tested on different Datasets.

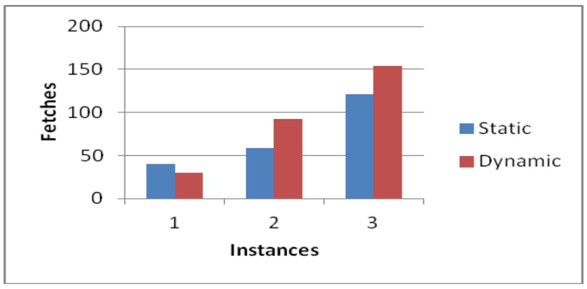

Fig. 1 Graphs shows number of fetches at different instances

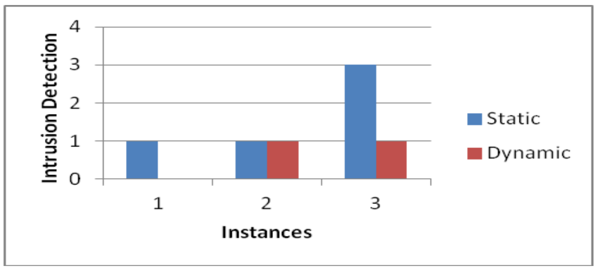

Fig. 1 Graph shows intrusion detection at different instances

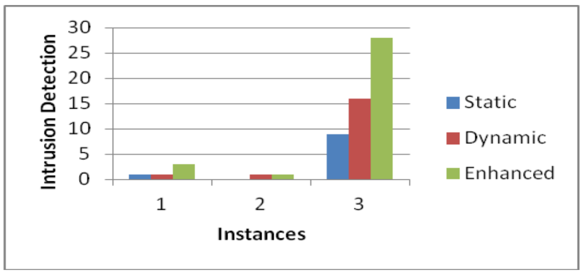

Fig. 3 Graph shows intrusion detection using enhanced technique

V. CONCLUSION

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.By using this we can able to find the Distributed Denial of Service attacks(DDOS). Unlike previous approaches that correlated or summarized alerts generated by independent IDSs, System 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[4]. 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.

REFERENCES

[1] D. Bates, A. Barth, and C. Jackson, “Regular Expressions Considered Harmful in Client-Side XSS Filters,” Proc. 19th Int’l Conf. World Wide Web, 2010.

[2] 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.

[3] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, “Toward Automated Detection of Logic Vulnerabilities in Web Applications,” Proc. USENIX Security Symp., 2010.

[4] 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.

[5] 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.

[6] S. Potter and J. Nieh, “Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems,” Proc. USENIX Ann. Technical Conf., 2010.

[7] Y. Shin, L. Williams, and T. Xie, “SQLUnitgen: Test Case Generation for SQL Injection Detection,” technical report, Dept. of Computer Science, North Carolina State Univ., 2006.

[8] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kru¨ gel, and G. Vigna, “Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis,” Proc. Network and Distributed System Security Symp. (NDSS ’07), 2007.

[9] 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.

[10] G. Vigna, F. Valeur, D. Balzarotti, W.K. Robertson, C. Kruegel, and E. Kirda, “Reducing Errors in the Anomaly-Based Detection of Web-Based Attacks through the Combined Analysis of Web Requests and SQL Queries,” J. Computer Security, vol. 17, no. 3, pp. 305-329, 2009.

Authors Profile:

Namratha Thatikonda
Obtained her Bachelor’s degree in Information Technology from SR Engineering College, Warangal, A.P.India. Pursuing her Master’s degree in Software Engineering from KITS, Warangal, AP, India.

Sunkari venkatramulu
Obtained his Bachelor’s degree in Computer Science and Engineering from Kakatiya University, India.Then he obtained his Master’s degree in Computer Science from JNTU and pursuing his PhD from Kakatiya University, Warangal. He is also life member of ISTE. He is currently an Associate Professor at the Faculty of Computer Science and Engineering, Kakatiya Institute of Technology & Science (KITS), Kakatiya University Warangal. His specializations include network security, Websecurity, Intrusion detection system.