A Stateful Intrusion Detection System for World-Wide Web Servers

Giovanni Vigna William Robertson Vishal Kher Richard A. Kemmerer

Reliable Software Group
Department of Computer Science
University of California, Santa Barbara
{vigna,wkr,vkher,kemm}@cs.ucsb.edu

Abstract

Web servers are ubiquitous, remotely accessible, and often misconfigured. In addition, custom web-based applications may introduce vulnerabilities that are overlooked even by the most security-conscious server administrators. Consequently, web servers are a popular target for hackers. To mitigate the security exposure associated with web servers, intrusion detection systems are deployed to analyze and screen incoming requests. The goal is to perform early detection of malicious activity and possibly prevent more serious damage to the protected site. Even though intrusion detection is critical for the security of web servers, the intrusion detection systems available today only perform very simple analyses and are often vulnerable to simple evasion techniques. In addition, most systems do not provide sophisticated attack languages that allow a system administrator to specify custom, complex attack scenarios to be detected. This paper presents WebSTAT, an intrusion detection system that analyzes web requests looking for evidence of malicious behavior. The system is novel in several ways. First of all, it provides a sophisticated language to describe multi-step attacks in terms of states and transitions. In addition, the modular nature of the system supports the integrated analysis of network traffic sent to the server host, operating system-level audit data produced by the server host, and the access logs produced by the web server. By correlating different streams of events, it is possible to achieve more effective detection of web-based attacks.

Keywords: World-Wide Web, Security, Intrusion Detection

1 Introduction

Attacks against web servers and web-based applications account for a substantial portion of the security incidents on the Internet [11]. The large number of web servers and the continuous disclosure of vulnerabilities associated with web-based applications makes web servers a popular target for malicious hackers and worms [3, 4]. In fact, in the period between April 2001 and March 2002 web-related vulnerabilities accounted for 23% of the total number of vulnerabilities disclosed to the public [19].

Intrusion detection plays an important role in addressing the security problems of web servers, by providing timely identification of malicious activity and supporting effective response to attacks. Unfortunately, detection of attacks has been performed by applying simple pattern-matching techniques to the contents of HTTP requests or by identifying trends in a large set of web-related events. In addition, most intrusion detection systems focus on a single event stream, such as the network traffic directed to a server host or the access logs produced by a server application. The lack of a stateful detection model and the inability to analyze different event streams in an integrated way severely limits the effectiveness of current intrusion detection approaches.

To improve the detection of web-based attacks, we propose an integrated approach that performs intrusion detection using stateful analysis of multiple event streams. The approach is centered around the State-Transition Analysis Technique (STAT) [9], which supports the modeling of multi-step, complex attacks in terms of states and transitions. STAT-based intrusion detection systems can be developed in a modular fashion, by extending an application-independent runtime with components that deal with specific application domains.

This paper describes WebSTAT, a STAT-based intrusion detection system that supports the modeling and detection of sophisticated attacks. WebSTAT operates on multiple event streams, and it is able to correlate both network-level and operating system-level events with entries contained in server logs. This integrated approach supports more effective detection of web-based attacks and generates a reduced

number of false positives.

The remainder of this paper is structured as follows. Section 2 describes related work on the detection of web-based intrusions. Section 3 describes the STAT framework, which has been used as the basis for the development of WebSTAT. The characteristics of WebSTAT are presented in Section 4. Section 5 evaluates the performance impact of the system on deployed web servers. Finally, Section 6 draws conclusions and outlines future work.

2 Related Work

Intrusion detection is performed by analyzing one or more input event streams, looking for the manifestation of an attack. Historically, detection has been achieved by following one of two different approaches: anomaly detection or misuse detection. Anomaly detection relies on models of the "normal" behavior of a computer system. These models can focus on the users, the applications, or the network. Behavior profiles are built by performing a statistical analysis on historical data [8, 10] or by using rule-based approaches to specify behavior patterns [12, 24, 25]. An anomaly detector then compares actual usage patterns against established profiles to identify abnormal patterns of activity.

Misuse detection systems take a complementary approach. Misuse detection systems are equipped with a number of attack descriptions. These descriptions (or "signatures") are matched against a stream of audit data to find evidence that the modeled attack is occurring [9, 14, 16].

Anomaly and misuse detection both have advantages and disadvantages. Anomaly detection systems have the advantage of being able to detect previously unknown attacks. This advantage is paid for with a large number of false positives and the difficulty of training a system for a very dynamic environment. Misuse detection systems can detect only those attacks that have been modeled, even though, in some cases, it is possible to detect variations of the attacks. This limitation is balanced by the highly focused analysis of the audit data that these systems can perform. As a consequence, misuse detection system are less prone to the generation of false positives. Because of this, misuse detection is by far the most popular approach to intrusion detection. Misuse detection systems can be further classified using the type of analysis they perform and the source event stream they use.

Misuse detection analysis can be stateless or stateful. Stateless analysis examines each event in the input stream independently, while stateful analysis considers the relationships between events and is able to detect event "histories" that represent attacks. This analysis is more powerful and allows one to detect more complex attacks. At the same time, stateful approaches are more expensive in terms of CPU and memory requirements and may be vulnerable to denial-of-service attacks that target the intrusion detection system itself.

Misuse detection analysis can be performed on different event streams. Traditionally, event sources are either represented by packets transmitted on a network segment or by the audit records produced by the auditing facility of an operating system. Examples of event streams are the audit records generated by the Solaris Basic Security Module (BSM) [17] and traffic logs collected using tcpdump [18]. Recently, the focus of analysis has been broadened to include other event sources, including the logs produced by applications [1] and the alerts produced by the intrusion detection systems themselves [20].

Misuse detection of web-based attacks has been performed both at the network level, analyzing network traffic [16], and at the application level, analyzing server logs [7]. Both of these approaches have some limitations.

Network-based intrusion detection is vulnerable to insertion and evasion attacks [15]. These attacks attempt to desynchronize the view of the intrusion detection system (IDS) with respect to the view of the actual target, that is, the web server. In addition, intrusion detection systems do not take into account the application-level logic of the web server, and, therefore, they cannot identify attacks that exploit the organization and configuration of the server application. Finally, only a few network-based intrusion detection systems support stateful analysis of web requests. Maintaining information about histories of requests is an important aspect of intrusion detection, but this type of analysis is seldom performed because of the complexity introduced by the detection of multi-step attacks.

Application-based intrusion detection solves some of the problems that are inherent to network-based detection of web-based attacks. For example, in [2], the authors describe an intrusion detection system that is embedded in an Apache web server. The advantage of this solution is the ability to perform intrusion detection analysis at different stages in the processing of client requests. This approach makes evasion techniques ineffective, because the view of the intrusion detection system and the view of the server application are tightly integrated. On the other hand, a disadvantage of this approach is that by "in-lining" intrusion detection analysis the performance of the web server is impacted. In addition, the proposed solution is specific to the Apache web server and cannot be easily ported to different servers.

A different approach is followed by the system described in [1]. In this case, the intrusion detection system analyzes the logs generated by a web server, looking for patterns of malicious activity. This approach does not directly impact the efficiency of the server application. On the other hand, the attacks that were analyzed were limited to the exploitation of CGI scripts. Attacks that target the web server it-

self, such as buffer overflows, are not detected. In addition, the analysis performs simple pattern matching on the URL contained in a single request. Even though a module to combine alerts is provided, composite patterns are limited to the use of Boolean logic (e.g., conjunction of alert conditions). Therefore, attacks that may involve multiple steps, with complex timing relationships between requests, cannot be modeled.

The approach to intrusion detection presented in this paper performs stateful analysis on a number of web-related event streams. The approach has been implemented, and an intrusion detection system, called WebSTAT, has been developed by leveraging the STAT framework [22]. The resulting system has a number of advantages in terms of flexibility and performance.

First of all, complex multi-step attacks can be modeled in a high-level language, called STATL [6]. The expressiveness of the language allows the attack modeler to describe timing relationships between events, the branching of attack histories, and the unwinding of partially matched scenarios. Second, the system can operate on logs produced by different web servers, such as Apache and Microsoft IIS. Third, the system can perform integrated analysis of multiple event streams. The multi-threaded nature of the event collection module supports the concurrent analysis of multiple web server logs, and the correlation of these event streams with lower-level event streams, such as operating system-level audit records and network packets. Finally, the system allows an administrator to associate response actions with the intermediate steps of an attack. This feature supports timely deployment of countermeasures and the fine-tuning of intrusion detection responses with respect to the site's security policy.

3 The STAT Framework

The WebSTAT intrusion detection system has been developed by using the STAT framework. The framework provides the implementation of a domain-independent analysis engine that can be extended in a well-defined way to perform intrusion detection analysis in specific application domains.

The STAT framework centers around an intrusion modeling technique that characterizes attacks in terms of transitions between the security states of a system. This approach is supported by the STATL attack modeling language.

The STATL language provides constructs to represent an attack as a composition of *states* and *transitions*. States are used to characterize different snapshots of a system during the evolution of an attack. Obviously, it is not feasible to represent the complete state of a system (e.g., volatile memory, file system); therefore, a STATL scenario uses variables to record just those parts of the system state that are needed to define an attack signature (e.g., the value of a counter or the source of an HTTP request). A transition has an associated *action* that is a specification of the event that can cause the scenario to move to a new state. For example, an action can be the opening of a TCP connection or the execution of a CGI script. The space of possible relevant actions is constrained by a *transition assertion*, which is a filter condition on the events that can possibly match the action. For example, an assertion can require that a TCP connection be opened with a specific destination port or that a CGI application be invoked with specific parameters.

It is possible for several occurrences of the same attack to be active at the same time. A STATL attack scenario, therefore, has an operational semantics in terms of a set of *instances* of the same scenario *specification*. The scenario specification represents the scenario's definition and global environment, and a scenario instance represents a particular attack that is currently in progress.

The evolution of the set of instances of a scenario is determined by the type of transitions in the scenario definition. A transition can be *nonconsuming*, *consuming*, or *unwinding*.

A nonconsuming transition is used to represent a step of an occurring attack that does not prevent further occurrences of attacks from spawning from the transition's source state. Therefore, when a nonconsuming transition fires, the source state remains valid, and the destination state becomes valid too. For example, if an attack has two steps that are the uploading of a file to a web server though FTP followed by an HTTP request for that file, then the second step does not invalidate the previous state. That is, another HTTP request for the same file can occur. Semantically, the firing of a nonconsuming transition causes the creation of a new scenario instance. The original instance is still in the original state, while the new instance is in the state that is the destination state of the fired transition. In contrast, the firing of a consuming transition makes the source state of a particular attack occurrence invalid. Semantically, the firing of a consuming transition does not generate a new scenario instance; it simply changes the state of the original one. Unwinding transitions represent a form of "rollback," and they are used to describe events and conditions that can invalidate the progress of one or more scenario instances and require the return to an earlier state. For example, the deletion of a file can invalidate a condition needed for an attack to complete, and, therefore, a corresponding scenario instance can be brought back to a previous state, such as before the file was created. For details about the semantics of the STATL language, see [6].

The STAT Core module is the runtime for the STATL language. The Core implements the concepts of state, transition, instance, timer, etc. In addition, the STAT Core is responsible for obtaining events from the target environment,

and matching this event stream against the actions and assertions corresponding to transitions in the active attack scenarios.

The STATL language and the Core runtime are domain-independent. They do not support any domain-specific features, which may be necessary to perform intrusion detection analysis in particular domains or environments. For example, network events such as an IP packet or the opening of a TCP connection cannot be represented in STATL natively. Therefore, the STAT framework provides a number of mechanisms to extend the STATL language and the runtime to match the characteristics of a specific target domain [23].

Domain-specific events and predicates are defined by subclassing specific C++ classes of the STAT Framework. These classes are encapsulated in a language extension module. The module is then compiled into a dynamically linked library (i.e., a "`.so`" file in a UNIX system or a DLL file in a Windows system). Once the event set and associated predicates for a language extension are available, it is possible to use them in a STATL scenario description by including them with the STATL `use` keyword. STATL scenarios are then translated into C++ and compiled into dynamically linked modules as well.

The input event streams analyzed by a sensor are provided by one or more event providers. An event provider collects events from the external environment (e.g., by obtaining packets from the network driver), creates events as defined in one or more STAT language extensions, encapsulates these events into generic STAT events, and inserts them into the input queue of the STAT Core. Event providers are compiled into dynamically linked modules, following a process that is similar to the one followed for language extensions.

In summary, a STAT-based sensor is created by developing a language extension that describes the particular domain of the application, an event provider that retrieves information from the environment and produces STAT events, and attack scenarios that describe attacks in terms of state-transition models of STAT events. In addition, it is possible to create response libraries that are specific to a certain domain. The response functions in the library can be dynamically associated with the states modeled in the attack scenarios.

4 An Intrusion Detection System for Web Servers

The WebSTAT intrusion detection system was developed by following the process outlined above. WebSTAT was built by developing new modules and reusing other modules developed for other sensors. More precisely, a language extension module that defines web-specific events was developed and an event provider that parses web server logs and generates the corresponding events was also developed. In addition, pre-existing language extensions and event providers that manage operating system-level and network-level events were included in the system. These modules were originally developed to build network-based and host-based intrusion sensors [21] but could be re-used in WebSTAT without modification, thanks to the modular nature of the STAT framework. As a last step, a number of STATL scenarios were developed to detect attacks against web servers. These attacks rely on one or more event streams to identify the evidence of an attack. The resulting system is shown in Figure 1.

The web language extension contains the definition of the basic client request event, auxiliary types, and web-specific predicates. Figure 2 shows a simplified version of the class for the `Request` event, which is an abstraction of an entry in the application log generated by a web server. Note that the class `Request` is a subclass of `STAT_Event`, which is the root event class of the STAT framework. Every new event introduced by a language extension must be a subclass of `STAT_Event`.

```
class Request : public STAT_Event
{
public:
  string request;        // Client request
  string userAgent;      // User agent
  string encodedRequest; // Decoded string
  bool isRequestEncoded; // Encoding flag

  [...]
}
```

Figure 2. The `Request` event defined in the web language extension.

The log-based event provider reads the events stored in the server application log file as they are generated. The event provider parses the Common/Extended Log Format representation of the events and creates corresponding `Request` objects. These events are then inserted in the STAT Core event queue. The STAT Core extracts the events from the event queue and passes them to active attack scenarios for analysis (see Figure 1).

Attacks are represented by using STATL to specify state-transition models over the stream of events generated by the event providers. A number of attack scenarios have been developed; the following sections describe a subset of the existing scenarios.

A common pattern used by many of the WebSTAT scenarios is the "counting scenario" pattern. Each scenario that conforms to this pattern requires the following integer pa-

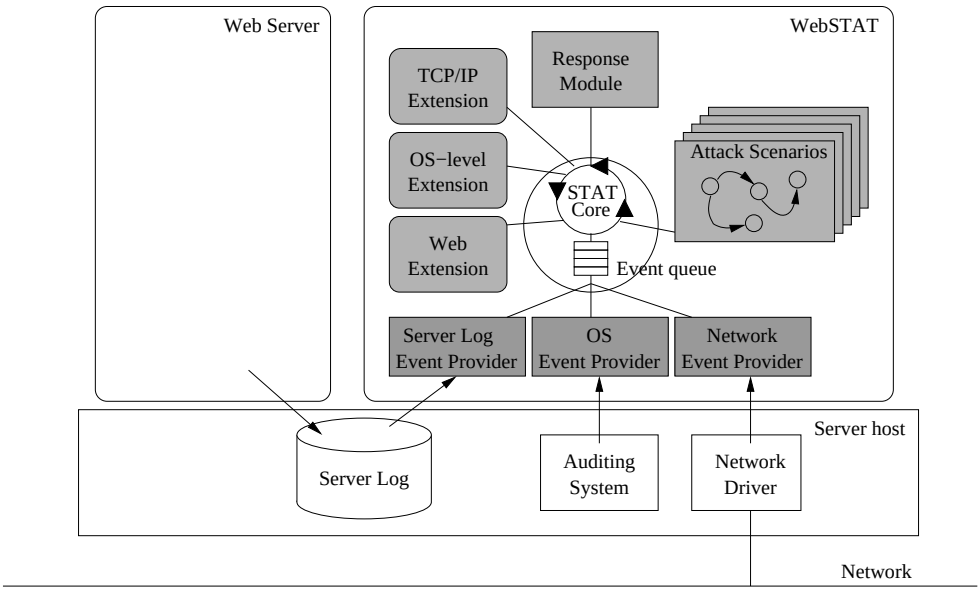

Figure 1. The WebSTAT component-based architecture. WebSTAT is obtained by composing the domain-independent STAT runtime with a number of language extensions, event providers, attack scenarios, and response functions.

rameters to be supplied: `threshold`, `alert_freq`, and `inactivity_timeout`. The first parameter specifies the number of occurrences of the event that need to appear in the event stream before an alert is raised. The second parameter indicates the frequency at which intermediate alerts are to be produced after the threshold has been overcome and before the attack is terminated. The attack is considered terminated when the attacker is inactive for the time specified by the `inactivity_timeout` parameter.

The web crawler, pattern matching, and repeated failed access scenarios presented in the following sections are examples of this generic counting scenario. The state-transition diagram of the counting scenario is shown in Figure 3. The reader should recall from section 3 that transitions can be nonconsuming, consuming, or unwinding. In a state-transition diagram the three types of transition are represented graphically using single headed, double-headed, or dashed-line arrows, respectively, and the initial state of a scenario is represented as state S_0.

The STAT framework also provides a number of generic response modules that scenarios can use to generate alerts. By default, WebSTAT uses the IDMEF response module, which generates alerts in IDMEF format [5]. However, WebSTAT can be extended dynamically with other response modules. Examples of possible response modules include the resetting of TCP connections and the dynamic reconfiguration of web servers in response to a detected attack.

4.1 Malicious Web Crawler Scenario

Web servers use a special file, `robots.txt`, to indicate what the acceptable behavior for robots (spiders/crawlers) visiting the site is. The web crawler scenario checks if robots indexing the contents of the web site hosted by the server adhere to the instructions specified in the `robots.txt` file. An alarm is raised if any web crawler violates the specified instructions. For this scenario to function, WebSTAT requires the User-Agent field to be logged; therefore, the server must be configured to log the requests in the Extended Log Format (ELF).

The `robots.txt` file is formatted according to the Robots Exclusion Protocol [13]. This file consists of a set of records in the form `<Field>: <Value>`. The record starts with one or more `User-agent` lines, specifying which robots the record applies to, followed by `Disallow` and `Allow` instructions to that robot. For example, consider the following records:

```
User-agent: *
Disallow: /cyberworld/map/
User-agent: cybermapper
Disallow:
```

This example specifies that no robots should visit any URL starting with "`/cyberworld/map/`", except for the robot called `cybermapper`. All robots must obey the first record in `/robots.txt` that contains a `User-agent`

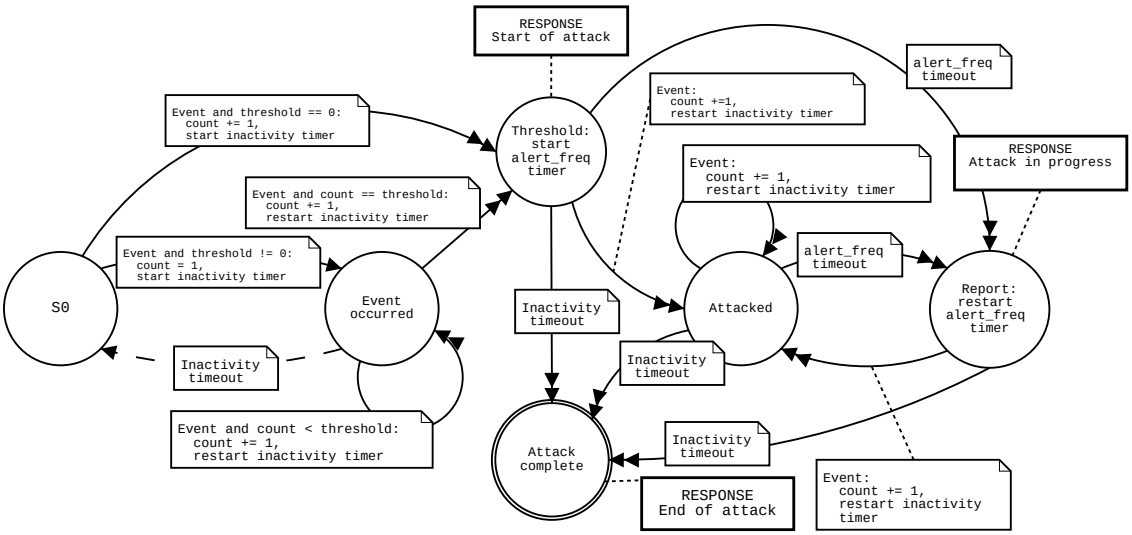

Figure 3. State-transition diagram for the "counting scenario" attack pattern.

field whose value contains the name of the robot as a substring. If no such record exists, they should obey the first record with a `User-agent` field with a "`*`" value, if present. If no record satisfies either condition, or no records are present, then the access is unlimited.

The malicious web crawler attack scenario reads the `robots.txt` file and, for each request, uses the request's URL and `User-Agent` header field to check whether the crawler is allowed to access the requested URL. If not, a compromised state is reached and a response function is invoked.

Because the malicious web crawler scenario is an instance of the generic counting scenario attack template (Figure 3), it requires parameters to be specified for the number of invalid robot accesses that are to be considered an attack, the frequency alerts are to be generated during an attack, and the length of the inactivity timeout.

4.2 Pattern Matching Scenario

WebSTAT can detect attacks embedded in URLs. For example, the presence of the substring `/default.ida` in a request is used to detect an attempted propagation of the Code Red worm. WebSTAT uses pattern matching to detect and report these malicious requests. Pattern matching scenarios take a list of regular expressions as a parameter; each item in the list can match one or more attacks. The regular expressions in the list are used to match against the current event's request (and headers, if the ELF is used). A compromised state is reached when one of these regular expressions matches a request or header. WebSTAT uses the Standard C regular expression library, which implements IEEE Std 1003.2-1992 ("POSIX.2") regular expressions. Currently, WebSTAT uses 66 regular expressions that represent attacks. New attacks can be detected by simply adding a new regular expression, which can be accomplished dynamically (i.e., without stopping WebSTAT's execution). Furthermore, one can group signatures and load multiple pattern matching scenarios, one per group, to match a request against every regular expression for that group. With this feature, the administrator can set threshold and timeout values for different groups depending upon the severity of the attack. To improve the performance of the system, regular expressions are compiled during the initialization of the scenario. The state-transition diagram for these scenarios is identical to the counting scenario shown in Figure 3.

In addition to supplying the parameters common to all counting scenarios, the pattern-matching scenario takes a `string regex` parameter, which specifies a list of regular expressions in an XML-based format as shown below.

```
<REGEX>
  <expression name="CodeRed">
    .*default\.ida
  </expression>
  <expression name="phf">
    .+phf.+\%0a
  </expression>

  [...]

</REGEX>
```

4.3 Repeated Failed Access Scenario

The repeated failed access scenario checks if there are multiple client errors, including failed authentication at-

tempts, from a particular client or subnet. This type of activity is a strong indication that a malicious entity is attempting to probe the website to gain information for future attacks. An internal counter records the number of times a failed request originated from a certain subnet. If this counter exceeds the event threshold parameter, an alarm is raised. This scenario is another instance of the generic counting scenario.

4.4 Cookie Stealing Scenario

Cookies are a state management mechanism for HTTP (defined in RFC 2965) that is often used by web application developers to implement session tracking. The cookie stealing scenario detects if a cookie used as a session ID is improperly utilized by multiple users. This is often a manifestation of a malicious user attempting to hijack the session of a legitimate user to gain unauthorized access to protected web resources.

The scenario begins by recording the issuance or initial use of a session cookie by a remote client by mapping the cookie to an IP address. In addition, an inactivity timer is simultaneously set. Subsequent use of the session cookie by the same client results in a reset of the timer, while a cookie expiration or session timeout results in the removal of the mapping for that cookie. If, however, a client uses the valid session cookie of another client, then an attack is assumed to be underway and an alarm is raised. The cookie stealing scenario takes two parameters: `timeout` and `cookie_name`. The first parameter specifies a timeout that corresponds to the session timeout in seconds for the protected web application, while the second parameter specifies the name of the cookie used for session tracking by the protected web application. An additional requirement is that the web server has been configured to enable cookie logging. The state-transition diagram for this scenario is shown in Figure 4.

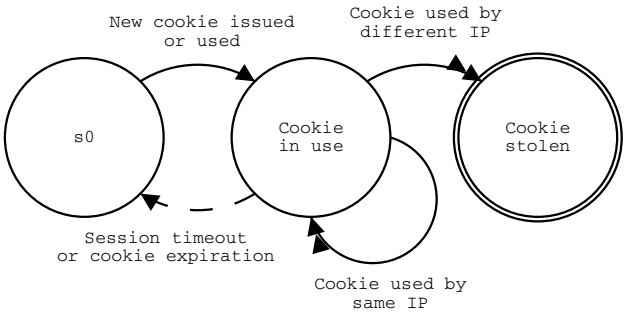

Figure 4. State-transition diagram for the cookie stealing scenario.

4.5 Buffer Overflow Scenario

Buffer overflows have historically been a popular attack against web servers. Aside from missing boundary checks within the web server itself, vulnerabilities within system libraries as well as third-party modules can allow remote attackers to gain illicit access to a host with the privileges of the web server process. The presence of binary data in a request or an extremely long request are strong indications of an attempt to exploit a buffer overflow. WebSTAT includes a scenario to detect these conditions. The buffer overflow scenario requires one parameter, `length`, which defines a request length threshold that must be exceeded for an alert to be raised. The state-transition diagram for this scenario is shown in Figure 5.

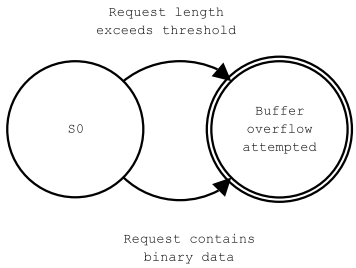

Figure 5. State-transition diagram to detect buffer overflow attacks.

Note that if a buffer overflow exploits a vulnerability in the web server code, it is possible that a log entry would never be created and the attack would go undetected.

4.6 Combining Network and Application-Level Buffer Overflow Detection

The STAT framework makes cross-domain intrusion detection scenarios possible by providing a comprehensive set of extension and provider modules to describe and examine various event sources. With multiple event providers loaded from different domains, attack scenarios can examine events and model attacks from several distinct points of view. All a scenario writer needs to do in order to leverage the capabilities provided by these modules is to import the requisite STATL language extensions, as shown in Figure 6.

By exploiting WebSTAT's ability to operate on different event streams, it was possible to improve the effectiveness of buffer overflow detection. More precisely, an improved buffer overflow attack scenario was developed. The scenario examines both web server access logs and the actual client requests as they traverse the network. This is accomplished by using a network-based event provider. The provider reads the TCP/IP streams between clients and a

```
use apache;
use tcpip;

scenario multi_domain_scenario
{
  [...]
}
```

Figure 6. Multi-domain scenario example.

web server from the network. The network packets are analyzed, looking for evidence of binary data. If binary data is found at the network-level, the scenario watches for a matching entry in the server logs. If none is found within a specified timeout period, then the scenario assumes that the attack has been successful and the web server process is now executing the code sent by the attacker. The state-transition diagram for this scenario is shown in Figure 7.

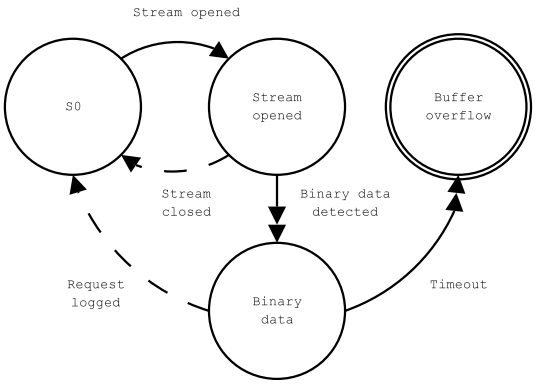

Figure 7. State-transition diagram for multi-domain detection of buffer overflows.

This scenario effectively detects the recent Apache chunked-encoding exploit, which typically does not leave an entry in the server logs.

4.7 Document Root Escape Attack

Cross-domain analysis is not limited to web server logs and network traffic, as in the previous scenario. For example, the root escape scenario examines events from the web server log and correlates them with operating system-level audit records to detect file system access violations. More precisely, the scenario detects if a client gained illicit access to a file outside a web server's document root. Figure 8 shows the state-transition diagram for the scenario.

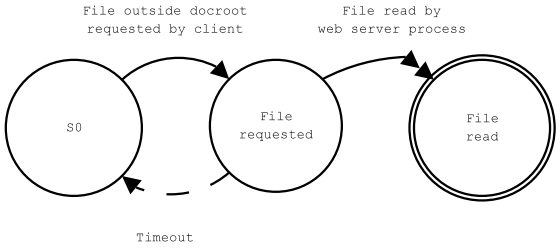

Figure 8. State-transition diagram for the document root escape scenario.

5 Performance Evaluation

An evaluation of the WebSTAT system was conducted to quantify the performance overhead WebSTAT would incur in a production web server. The experimental testbed used during this evaluation consisted of a single system acting as a web server loaded by multiple client systems. The web server was a Pentium IV 1.8 GHz machine with 1 GB of RDRAM running Apache 2.0.40 on a stock RedHat 8.0 installation. A network of Pentium IV 1.8 GHz machines with 1 GB of RDRAM each running the WebStone 2.5 benchmark on stock RedHat 8.0 installations acted as clients. All systems used Intel EtherExpress 10/100 Ethernet cards running in 100BaseT full-duplex mode, and were connected with a Cisco Catalyst 3500 XL switch.

The evaluation measured average throughput and response times under a typical real-world workload for both a host running standalone Apache and a host running Apache monitored by WebSTAT. WebStone was configured to perform five iterations of runs of 10 minutes each, varying the number of clients from 100 to 500 in increments of 50. The benchmark was configured to target an augmented mirror of the UC Santa Barbara Computer Science web server[1], which contained a mixture of static and dynamic pages. The client traffic included requests for static and dynamic pages as well as various attacks. Client requests were weighted to approximate typical observed access patterns. Apache was left in its default configuration with the exception of modifications to the server pool to increase the maximum number of possible concurrent client requests. Finally, WebSTAT was configured for online detection with all attack scenarios enabled.

Figure 9 displays the minimum, average, and maximum throughput for standalone Apache and Apache monitored by WebSTAT. From the graph, one can see that both systems under test performed near the theoretical limits of the testbed network hardware. In this case, Apache is clearly I/O-bound. WebSTAT, however, is primarily CPU-bound;

[1] `http://www.cs.ucsb.edu/`

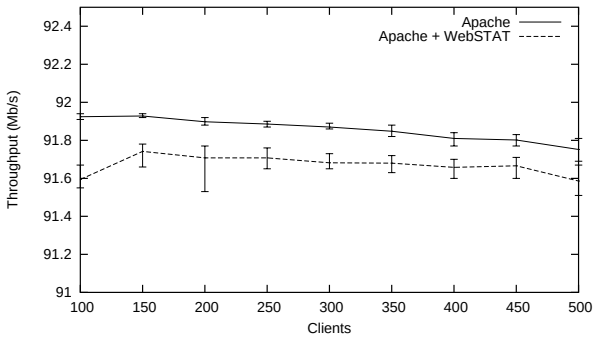

Figure 9. Minimum, average, and maximum throughput.

most of its time is spent performing regular expression matching against client requests, as detailed in Section 4.2. This CPU overhead explains the slight impact on average throughput shown in Figure 9. However, the performance degradation is limited to less than 0.5% in the average case.

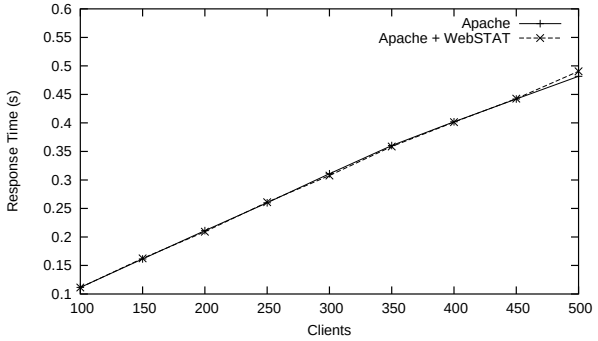

Figure 10. Minimum, average, and maximum response times.

Figure 10 displays the minimum, average, and maximum response times for standalone Apache and Apache monitored by WebSTAT. Readily apparent is the fact that Apache's response times increase linearly with the number of concurrent client requests. Equally apparent is the fact that WebSTAT has virtually no impact upon Apache's response times.

In this evaluation, the collected data demonstrates that WebSTAT incurs a small performance overhead in average web server throughput. Nevertheless, the drop in performance may well be acceptable given the advanced detection capabilities WebSTAT provides. Careful site-specific tuning, which was not applied in this evaluation, would also allow web server administrators to further reduce WebSTAT's impact on web server performance.

6 Conclusions and Future Work

This paper presented an approach for stateful intrusion detection, called WebSTAT. The approach is implemented by extending the STAT framework to create a sensor that performs detection of web-based attacks. WebSTAT is novel in that it provides a sophisticated language for describing multi-step attacks in terms of states and transitions, and these descriptions are automatically compiled into dynamically linked libraries, which has a number of advantages in terms of flexibility and extensibility. WebSTAT also operates on multiple event streams and is able to correlate both network-level and operating system-level events with entries contained in server logs. This supports more effective detection of web-based attacks and generates a reduced number of false positives.

The WebSTAT system has been evaluated in terms of its ability to detect attacks and the performance impact of the detection process on deployed web servers. The results achieved show that stateful intrusion detection can be performed on high performance servers in real-time.

Future plans are to further extend the system to perform more integrated analysis of web server logs and events collected from other domains such as network traffic streams or operating system-level event logs, as demonstrated in Section 4.6. More detailed performance experiments will also be run.

The current implementation of WebSTAT can be retrieved from `http://www.cs.ucsb.edu/~rsg/STAT/software`.

Acknowledgements

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, or the U.S. Government.

References

[1] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security*, San Diego, CA, February 2000.

[2] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. In *Pro-*

ceedings of Recent Advances in Intrusion Detection (RAID), LNCS, pages 22–36, Davis,CA, October 2001. Springer.

[3] CERT/CC. "Code Red Worm" Exploiting Buffer Overflow In IIS Indexing Service DLL. Advisory CA-2001-19, July 2001.

[4] CERT/CC. Apache/mod_ssl Worm. Advisory CA-2002-27, October 2002.

[5] D. Curry and H. Debar. Intrusion Detection Message Exchange Format: Extensible Markup Language (XML) Document Type Definition. `draft-ietf-idwg-idmef-xml-07.txt`, June 2002.

[6] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[7] R. Fielding. wwwstat: HTTPd Logfile Analysis Software. `http://ftp.ics.uci.edu/pub/websoft/wwwstat/`, November 1996.

[8] Paul Helman and Gunar Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. In *IEEE Transactions on Software Engineering*, volume Vol 19, No. 9, pages 886–901, 1993.

[9] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[10] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[11] D. Klein. Defending Against the Wily Surfer: Web-based Attacks and Defenses. In *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 1999.

[12] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.

[13] M. Koster. A Method for Web Robots Control. Internet Draft, draft-koster-robots-00.txt, December 1996.

[14] U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.

[15] T.H. Ptacek and T.N. Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, January 1998.

[16] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, November 1999.

[17] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module*. 2550 Garcia Ave., Mountain View, CA 94043, December 1991.

[18] Tcpdump and Libpcap Documentation. `http://www.tcpdump.org/`, June 2002.

[19] Security Tracker. Vulnerability statistics april 2001-march 2002. `http://www.securitytracker.com/learn/statistics.html`, April 2002.

[20] A. Valdes and K. Skinner. An Approach to Sensor Correlation. In *Proceedings of RAID 2000*, Tolouse, France, October 2000.

[21] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[22] G. Vigna, R.A. Kemmerer, and P. Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In W. Lee, L. Mè, and A. Wespi, editors, *Proceedings of the 4^{th} International Symposiun on Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 69–84, Davis, CA, October 2001. Springer-Verlag.

[23] G. Vigna, F. Valeur, and R.A. Kemmerer. Designing and Implementing a Family of Intrusion Detection Systems. In *Proceedings of the 9^{th} European Software Engineering Conference*, Helsinki, Finland, September 2003.

[24] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001. IEEE Press.

[25] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.