Learning DFA Representations of HTTP for Protecting Web Applications

Kenneth L. Ingham\textsuperscript{a} Anil Somayaji\textsuperscript{b} Stephanie Forrest\textsuperscript{a,c} John Burge\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, University of New Mexico, Mail stop: MSC01 1130, Albuquerque, NM 87131-0001, USA
\textsuperscript{b}School of Computer Science, Carleton University, 5302 Herzberg Building, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada
\textsuperscript{c}Santa Fe Institute 1399 Hyde Park Road Santa Fe, New Mexico 87501, USA

Abstract

Web application security is increasingly important. To manage the large number of implementation platforms and custom applications, anomaly-based intrusion detection methods have been developed which use statistical methods to characterize normal HTTP requests. The training data for these systems, however, typically contain attack attempts that can corrupt the model of allowed behavior. Although such attempts are usually ineffective against properly patched systems, the presence of such events in normal training data can make statistics-based systems vulnerable to mimicry attacks. As an alternative, a more specific definition of normal HTTP requests, automatically tailored to individual web servers, can minimize this risk by avoiding overgeneralization. This paper describes how to construct such a system using a DFA induction algorithm combined with automata reduction heuristics. Data captured from production web servers provide evidence that the method detects attacks directed against servers running a variety of applications and that it has an acceptable false-positive rate. The paper describes an on-line one-class learning algorithm capable of handling arbitrary-length nonstationary data. Finally, we describe our database of HTTP attacks, a more comprehensive test set which is available to other researchers.

Key words: anomaly intrusion detection, finite automata induction, web server security

Preprint submitted to Elsevier Science 14 December 2005
1 Introduction

Web servers are a critical resource for many individuals and organizations, serving as vital conduits for communication, commerce, and services. Reflecting the high-value of web servers, attacks against web servers have become commonplace, as well as defenses to protect against these attacks. For example, for the snort [1] network intrusion detection system 27% of the October 25, 2005 Sourcefire Vulnerability Research Team rules [2] and 50% of the November 2, 2005 community rules were to detect attacks against web servers and web applications.

One reason for the large number of web application attack signatures is the variety of web applications, ranging from simple Perl common-gateway interface (CGI) scripts to complex database-driven e-commerce sites. Because the tools for creating web applications are easy to use, many of the people writing and deploying them have little background in security, and often come from fields other than computer science (e.g., publishing or graphic design). It is thus not surprising that many web applications have security flaws.

Given this situation, web applications are good candidates for post-hoc security approaches, in particular, those based on intrusion-detection techniques. However, the huge number of existing web attack signatures illustrates the difficulty of using signature-based IDS approaches for this problem. Also, the variety and insecure design of many web applications makes them poor candidates for formal specification-based security approaches. Recognizing these issues, researchers such as Kruegel and Vigna [3] and Tombini et al. [4] have proposed anomaly detection techniques for detecting dangerous web requests. To date, researchers studying anomaly detection for protecting web servers have used statistical methods to characterize the space of acceptable HTTP requests, for example, the distribution of request lengths or the characters in a request (e.g., [5,3]). Any request with a sufficiently different distribution is flagged as abnormal.

However, if the statistics are derived from data containing attacks (even ineffective ones), then the system is potentially vulnerable to mimicry attacks [6]. As an example, suppose the protected system accepted normal requests of the form:

```
GET /scripts/access.pl?user=johndoe&cred=admin
```

If the cred= portion had been vulnerable to a cross-site scripting (XSS) attack in the past, then the training data could include instances such as (the two examples are equivalent, but encoded differently)¹:

¹ These attack strings make use of examples from

---

Email addresses: ingham@cs.unm.edu (Kenneth L. Ingham),
soma@scs.carleton.ca (Anil Somayaji), forrest@cs.unm.edu (Stephanie Forrest), lawnguy@cs.unm.edu (John Burge).

---
GET /scripts/access.pl?user=johndoe&cred=<script>document.location='http://www.cgisecurity.com/cgi-bin/cookie.cgi?' +document.cookie</script>

GET /scripts/access.pl?user=johndoe&cred=%22%3e%3c%73%63%72%69%70%74%3e%64%6f%63%75%6d%65%6e%74%2e%6c%6f%63%61%74%69%6f%6e%3d%27%68%74%70%3a%2f%2f%77%77%77%2e%63%67%69%73%65%63%75%72%69%74%79%2e%63%6f%6f%6b%69%65%2d%67%69%3f%27%20%2b%64%6f%63%75%6d%65%6e%74%2e%63%6f%6b%69%65%3c%2f%73%63%72%69%70%74%3e

If several attacks such as these occur in the training data, there are several possible effects on the statistics describing normal behavior:

- The attribute and overall request length would be shifted toward larger values (XSS attacks are longer by the length of the attacking script plus required HTML or related code).
- The attribute character distribution would be biased toward cross-site scripting-type values.
- The requested program would be unchanged.
- The structure of attributes would be unchanged (user= and cred= would still both appear together and in the same order).

If a new XSS vulnerability were discovered, this time associated with the user= portion, it would likely be accepted as normal. A similar argument applies to other attacks, for example buffer-overflow attacks. If buffer overflows occur in the training data, a statistical IDS is likely to accept other buffer overflows as normal (due to long character sequences, possibly containing machine code, possibly encoded as printable ASCII). If both types of harmless attacks are in the training data, then variants and combinations of both attacks are likely to be accepted as normal. Combining multiple statistical detectors does not necessarily solve this problem, because each detector must generalize enough to accept the harmless attacks.

As an alternative, we propose a method for learning very specific definitions of normal web requests that is resistant to mimicry attacks. Our method learns more about the structure of the HTTP requests than the statistical methods, and it is therefore less vulnerable to mimicry. For the example given above, our method would learn the XSS attack against the cred= as normal, but it would not necessarily treat an XSS attack against the user= portion of the request as normal. This difference between our method and the statistical methods arises from the types of generalizations in the induction (learning) step; the DFA potentially generalizes less than a statistical system. Figure 1 illustrates this difference. By reducing the size of the generalization, we reduce the ability of attackers to mask dangerous

Fig. 1. The difference in generalization when an anomaly detector learns on data containing attacks. U is the universe of all possible HTTP requests, N is the set of normal requests, and A1, A2, and A3 are attacks. The lines surround what the anomaly detection system accepts. (a) A statistical system. (b) Our DFA.

web requests using fragments of normal ones. However, the smaller generalization also increases the risk of false positives. By using limited types of structural generalizations, model reduction heuristics and anomaly clustering, we can achieve acceptable false-positive rates with reasonably sized models of normal web requests. In addition, by tailoring each model to a single web site, we produce a definition of normal that both defeats mimicry attacks (because there is a smaller generalization) and provides diversity of defense, thus disrupting widely replicated attacks.

In the remainder of the paper, we first discuss requirements for a web-request anomaly intrusion-detection method (Section 2). We then describe a method for modeling HTTP requests according to these criteria (Section 3). Section 4 describes the datasets we used to test the system, Section 5 describes experimental results. Section 6 reviews related work, and we discuss the results and their implications in Section 7, and conclude with a summary and plans for future work in Section 8.

2 Requirements

There are several important criteria for a successful IDS: Accuracy, novelty detection, performance, diversity, and requirements on the learning algorithm. Accuracy is of primary importance and is measured in terms of false positives (the lower the better) and true positives (the higher the better). Detecting novel (zero day) attacks is also important in this environment, where attack techniques are continually evolving. In addition, the IDS must not present an undue burden on the administrator or the machine it is protecting. Anomaly intrusion detection mechanisms potentially satisfy all of these requirements. Such a system would use a corpus of training data to form a model of acceptable web server behavior, specifically ac-
ceptable HTTP requests. Once the system develops this model, it then can begin to use the model to classify events as normal or abnormal.

Web server content, applications, and clients differ from each other. An IDS should be able to exploit this diversity to eliminate (or, at least, minimize) the ability of an attacker to write one attack and have it succeed on all web servers running the same version of the software. Forcing attackers to craft a custom attack for each web site changes the timescale of the attack from one that occurs in computer time to one that occurs in human time. This change of timescale allows a human administrator to intervene and reduce the damage that the attacker can cause.

In addition to these generic criteria, there are additional constraints on how the system learns. These include the ability to perform one-class learning, time/space performance appropriate for on-line deployment, the ability to perform in nonstationary environments, and the ability to learn correctly in the presence of attack data.

Because we do not have a statistically significant number of attacks against any single web application, we desire a training method that performs one-class learning, i.e., one that learns from examples of normal behavior. A further constraint arises from the nature of HTTP. Because it is a stateless protocol, web application attacks typically are contained within a single variable-length HTTP request; thus, our training mechanism cannot rely on attacks consisting of multiple events.

To be suitable for online deployment in large-scale environments, an anomaly IDS mechanism must have modest storage and computational requirements, and it should be capable of learning with minimal human interaction. The IDS algorithm must also be capable of adapting as the protected web site changes over time. This is known as a nonstationary environment.

An important feature of this domain is the continual presence of attacks. Harmless attacks are a part of any Internet-connected web server’s normal traffic, including attacks targeted for other applications and attacks against patched vulnerabilities. Cleaning the training data in order to remove these harmless attacks is a time-consuming process and requires an extensive, ever growing corpus of known benign attacks. Some previous researchers have made use of web server log files in order to avoid this problem. However, using log files precludes the option of actually stopping an attack—by the time an attack is discovered in a log file, it is too late to stop it. Thus, we desire a method that can succeed in the presence of noisy (“dirty”) training data. We would also like a method that is resistant to mimicry attacks [6], i.e., we want a learning method whose generalization of normal behavior does not include dangerous web requests.
3 Modeling web requests

The evaluation criteria described in Section 2 are challenging and rule out most existing model induction algorithms. Consequently, we developed our own unsupervised, one-class, online DFA induction method for modeling HTTP requests. Although the formal grammar that describes HTTP in standards documents [7] is context-free (and the full HTTP language is in fact context sensitive), we discovered that in practice a given web server uses only a subset of the full HTTP language. As a result, normal HTTP requests can be described using the simpler DFA representation. Our algorithm combines an HTTP tokenizer, a DFA induction algorithm known as the Burge DFA induction algorithm, and two heuristics that help keep the learned DFA tractable and up-to-date. These are described below.

3.1 The HTTP tokenizer

The tokenizer breaks the HTTP request into tokens based on those described in the standard [7]. The tokens are a combination of the token type (e.g., pathname component) and optionally the value (e.g., cgi-bin). In practice, most of the values are necessary to properly distinguish attacks from normal requests. However, some tokens have highly variable values, and in these cases the algorithm checks to see if the values are well formed, and if so deletes them from further consideration. The tokens falling into this category are: hostnames (optionally validated via a domain name system lookup), IP addresses, dates, q-values (floating point values used for negotiation of languages and other preferences), entity tags (hashes), and PHP session identifiers (also hashes). The result is a stream of tokens combined with the associated values.
Fig. 3. The results of compressing the DFA in Figure 2c. States that have identical sources and destinations (S₁ and S₄) are compressed into the same state (“S₁, S₄”). Tokens that originally caused a transition into one of the uncompressed states cause transitions into the compressed state.

3.2 The Burge DFA Induction Algorithm

The Burge DFA induction algorithm is an $O(n)$ algorithm, where $n$ is the number of samples in the training data set. The algorithm does not require negative examples, and as we will describe, the resulting DFA can be easily modified to deal with the nonstationary environment in which we use it.

To model a web request, we construct an initial DFA as follows. First, let $\Sigma = \{T₁, \ldots, Tₙ\}$ be the set of $n$ unique tokens in the HTTP request, and let $L = (l₁, \ldots, lₜ)$ with $lᵢ \in \Sigma$ be the series of $t$ chronologically ordered tokens from the HTTP request. Let $G = (S, A)$ be a DFA with states $S$ and transitions $A$. $S = \{S_{\text{START}}, S₁, \ldots, Sₙ, S_{\text{FINISH}}\}$ where states $S₁, \ldots, Sₙ$ have a one-to-one correspondence with tokens $T₁, \ldots, Tₙ$, and $S_{\text{START}}$ and $S_{\text{FINISH}}$ are additional states (which could be thought of as having indices 0 and $n + 1$, respectively). $E(t)$ is a function that returns the state into which the token $t$ will cause a transition (i.e., $E(t)$ returns the state $s$ that corresponds to token $t$). $A = \{A_{i,j}\}$ where $A_{i,j}$ indicates a transition labeled $T_j$ between states $S_i$ and $S_j$.

Given these definitions, the algorithm proceeds as follows:

1. Parse an HTTP request into a list of tokens, $L$.
2. Set the current state $C = S_{\text{START}}$, $A = \emptyset$.
3. For $j = 1$ to $t$ :
   a) If $A_{C,E(lᵢ)} \notin A$ then $A \leftarrow A \cup A_{C,E(lᵢ)}$
   b) $C \leftarrow E(lᵢ)$
4. $A \leftarrow A \cup A_{C,S_{\text{FINISH}}}$

The algorithm works as follows. A DFA $G$ is constructed with one state for each unique token in the HTTP request, as well as the two additional states $S_{\text{START}}$ and $S_{\text{FINISH}}$. At the start of the algorithm, there are no transitions present in the DFA and a current state $C$ is set to the $S_{\text{START}}$ state. Then, in steps 3 through 5, tokens are sequentially read in from the HTTP request. In step 4, if there is no existing transition between the current state and the state corresponding to the current token, the transition is created and the new transition is labeled with the current token. $C$ is then updated to be the state corresponding to the current token. After all the tokens have been processed, a transition from $C$ to $S_{\text{FINISH}}$ is created. Figure 2 shows an example.
At this point, the DFA model has one state corresponding to each unique token present in the HTTP request, and the DFA has a transition between any two states that were seen consecutively in the request. The transition is labeled with the token corresponding to the destination state.

### 3.3 Maintaining the DFA model

In practice, the DFA induction algorithm described above leads to large, complex DFAs that potentially could grow without bound in dynamic environment with perpetually novel HTTP requests. We use two techniques to manage this complexity, one that compacts an existing model and a second that adds states and transitions incrementally as well as “forgetting” structures that have not been used recently. To reduce DFA size, our algorithm searches at regular intervals for nodes that have the same source and destination. These nodes are combined into one, as illustrated in Figure 3. To track web sites that change over time, the learned model needs to update itself automatically. When a normal HTTP request arrives that is not captured by the current DFA, the request is added to the DFA. For example, this situation might arise if a false positive were identified. To detect unused edges, a counter is updated every time an edge is traversed. Infrequently used edges are periodically dropped.

### 3.4 Determining similarity between a request and the DFA

When testing, incoming HTTP requests are tokenized and fed into the DFA model. In language recognition, a sentence would normally be rejected if any part of it failed to conform to the language definition. In our application, however, we are interested in knowing how close a sentence (an HTTP request) is to being in the language. During the learning phase, if a token is encountered with no corresponding transition, we create a new edge. During testing, however, we record the mismatch as a “miss event” and then attempt to resynchronize by reading the next token in the string. If the edge corresponding to this next token exists in the DFA, then the tester transitions to the edge, even if it requires jumping to a new state. If no such edge exists, a second miss event is recorded, and the tester reads yet another token. The number of “missed tokens” provides an estimate of how anomalous the request is with respect to the current DFA. Similarity between an HTTP request and the DFA is calculated by:

\[
\frac{\text{# of tokens where a transition is taken}}{\text{# of tokens in the HTTP request}} \in [0, 1]
\]
The similarity measure records the number of modifications that would need to be made to the DFA in order to accept the request (i.e., for each “jump” a new transition would have to be added). One benefit of this measure and its sensitivity to the token count is that the more complex requests (i.e., those specifying many of the HTTP header options) have more room for slight changes and still can be accepted as normal. A measure less sensitive to the token count is likely to have a higher false-positive rate due to this variation.

It is notable that our system does not record the frequency with which a transition is traversed when determining what is normal. Rarely accessed parts of a web site or rarely used configurations for web clients can thus be tolerated by the system. Of course, the DFA itself is induced from an observed sample of requests, and we expect the more common examples to be present in the sample.

3.5 Heuristics for reducing false positives

We have adopted two heuristics for reducing the false-positive rate. First, we discovered that many false positives are caused by HTTP requests with unusual lines. Looking forward to a deployed system, we noted that these lines are not critical for the web server to identify the requested resource. Therefore, if a request is considered anomalous, we try deleting one at a time the following HTTP header lines: Referer, Cookie, Accept-Language, Accept-Charset, and Accept. If, after deleting a line, the request passes the similarity test, then the request is accepted and processed without the anomalous header lines. With the exception of cookies, the worst impact this could have on a user would be that a web client might receive a default version of a web page instead of one customized to its preferred language, character set, or file format.

Deleting cookies is a potentially more serious operation, because they may encode some form of state (e.g., the PHP session identifier cookie) which is important for proper operation of the web site. Deleting a cookie can therefore potentially cause problems with the user’s ability to visit the web site. Unfortunately, we noticed that some web clients send cookies that should not be sent to the web site, and they are unrelated to legitimate cookies used by the local system. Legitimate cookies are properly handled by our system.

The second heuristic groups any putative attacks into classes. We added this heuristic after noticing a web robot which came online in the test data, but which was not present in the training data. This robot was responsible for nearly 10,000 false positives because its request format was different from the normal requests. By grouping related unusual requests, an administrator could look at a single exemplar of the class and determine if it is acceptable. If so, the request could be added to the DFA in the normal way, and future similar requests would be accepted as normal.
<table>
<thead>
<tr>
<th>Web site</th>
<th>Files</th>
<th>Database</th>
<th>Requests</th>
<th>Edges</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>aya.org</td>
<td>5,095</td>
<td>MySQL</td>
<td>40,149</td>
<td>13,310</td>
<td>1,800</td>
</tr>
<tr>
<td>explorenm.com</td>
<td>6,146</td>
<td>PostgreSQL</td>
<td>36,944</td>
<td>8,213</td>
<td>1,645</td>
</tr>
<tr>
<td>i-pi.com</td>
<td>6,644</td>
<td>none</td>
<td>7,694</td>
<td>5,819</td>
<td>1,269</td>
</tr>
<tr>
<td>cs.unm.edu</td>
<td>181,132</td>
<td>none</td>
<td>390,950</td>
<td>64,599</td>
<td>9,057</td>
</tr>
</tbody>
</table>

Table 1
The four web sites used for testing. **Files** is the number of distinct files on the web site, **Requests** is the number of HTTP requests in the training dataset, and **Edges** and **Nodes** show the size of the induced DFA.

### 4 Test data

In this section we describe the datasets we used to test our system. First, we describe four production web sites used to collect our normal data, and then we describe our database of attacks.

#### 4.1 Normal Data

The training and test data are sets of HTTP requests from four web sites:

**aya.org**: Uses PHP extensively for dynamic content and a MySQL database.

**explorenm.com**: Uses Perl extensively and some PHP for CGI scripts, as well as a PostgreSQL database.

**i-pi.com**: Composed of files.

**cs.unm.edu**: Contains official departmental content and a diverse set of student and faculty web pages, ranging from simple content to complex, automatically generated content.

Our training data consists of the entire HTTP request sent by the client to the server, allowing us to make use of the HTTP header lines and test for attacks that are not contained in the requested resource path.

Table 1 shows the sizes of each of the four web sites and the size of their initial DFA. In general, our DFA edge count is $O(\text{alphabet size})$, and the alphabet in our tests includes the values of most tokens; hence it is large. Note also that the DFA structure encodes relationships between parts of the request. For example, if the requested language usually follows the preferred file types, the DFA will encode this structure.
4.2 Attack Data

Our attack database currently contains a collection of 65 attacks, some of which are different examples of the same vulnerability, either a different exploit for the same vulnerability or an exploit for the vulnerability on a different operating system. We collected the attacks from the following sources: Attacks against web servers we were testing (attacks in the wild); BugTraq and the SecurityFocus archives http://www.SecurityFocus.com/; The Open Source Vulnerability Database http://www.osvdb.org/; The Packetstorm archives http://Packetstorm.widexs.nl/; and Sourcebank http://archive.devx.com/sourcebank/. In many cases, we had to debug the attack programs in order to get them to produce malicious web requests. Note that we did not verify whether the attacks could actually compromise the targeted web application.

Our attack database contains the following categories of attacks: Buffer overflow; Input validation error (other than buffer overflow); Signed interpretation of unsigned value; and URL decoding error.

The attacks were against a collection of different web servers: Active Perl IS-API; AltaVista Search Engine; AnalogX SimpleServer; Apache with and without mod.php; CERN 3.0A; FrontPage Personal Web Server; Hughes Technologies Mini SQL; InetServ 3.0; Microsoft IIS; NCSA; Netscape FastTrack 2.01a; Nortel Contivity Extranet Switches; OmniHTTPd; and PlusMail.

The victimized operating systems for the attacks include the following: AIX; Linux (many varieties); Mac OS X; Microsoft Windows; OpenBSD; SCO UnixWare; Solaris x86; Unix; VxWorks; and any x86 BSD variant.

A snapshot of the database we are using is at http://www.i-pi.com/ingham/web-attacks/.

Our test data set is more extensive than those used by other researchers in this area. Researchers using the Lincoln Labs data [8,9] have at most four attacks to work with. This limit stems from the fact that when the data was collected, web servers represented a smaller profile for most organizations. Kruegel and Vigna [3] used 11 attacks, some of which were chosen specifically because they were against software used on some of the web servers they used as a source of training data. Tombini et al. [4] mentioned using 56 attacks in their testing of a portion of their IDS.

5 Results

In Section 2, we described criteria for evaluating IDS performance in the context of web servers and their applications. This section reports experimental results on
Fig. 4. Receiver Operating Characteristic Curves for UNM CS Dept. Web Site: False versus true positives for different similarity thresholds. True positives are the fraction of the attack set properly identified, while false positives are the fraction of the test data (containing no attacks) that were improperly identified as attacks. Each set of connected points represents a different configuration of the algorithm, including the algorithm that learns once (non-adaptive), the algorithm that learns continuously (adaptive), the algorithm that learns continuously and uses one heuristic (DL), and the algorithm that learns continuously and uses both heuristics (GRP). For comparison, a detector that makes random guesses is also plotted.

The performance of our system when measured by these criteria.

Accuracy results for IDSs are often reported using a receiver operating characteristic (ROC) curve, as shown in Figure 4. As the figure shows, our system detected most of the attacks in the attack database, with the fraction detected depending on the similarity threshold. We believe that the detection fraction can be improved with the future work described in Section 7.

In our system, the false-positive rate depends on the similarity threshold. We report the false-positive rate per day, because this measure reflects the actual load that a system administrator would perceive when using the system. The false-positive rate for the four sites is shown in Table 2 for different configurations of the system.

To show that the web server data are nonstationary and to study how continuous learning affects the performance of our system, we implemented a version of our system that learns only once. We trained it on 2004-11-12 data and tested it on
Table 2
False positive rates per day for four production web sites. Rates are reported for non-adaptive (learn once) and adaptive (continuously learn) methods of updating the model. The last two columns show the false positive rates for the continuous learning method when the two heuristics are used. The **DL** heuristic deletes unusual lines, and **Grp.** groups similar alarms together. The results were obtained using a similarity threshold of 0.852.

<table>
<thead>
<tr>
<th>System</th>
<th>Date</th>
<th>False positive fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>2004-11-17</td>
<td>0.03390455</td>
</tr>
<tr>
<td>Adaptive</td>
<td>2004-11-17</td>
<td>0.02701282</td>
</tr>
<tr>
<td>Static</td>
<td>2005-01-22</td>
<td>0.09516097</td>
</tr>
<tr>
<td>Adaptive</td>
<td>2005-01-22</td>
<td>0.02204199</td>
</tr>
</tbody>
</table>

Table 3
Effect of Continuous Learning on False Positives: False positives are shown for the DFA induced from cs.unm.edu 2004-11-12 data when applied to the CS data from 2004-11-17 and 2005-01-22. False-positive fractions are reported for a similarity threshold of 0.852, and they do not include the two heuristics in order to focus on how rapidly the data are changing. The algorithm labeled **static** learns only once and does not update its model as it runs, whereas the algorithm labeled **adaptive** does.

2004-11-17 and 2005-01-22 data. As Table 3 shows, in as little as 2.5 months the “learn once” system suffered performance degradation compared to the version that could adapt to ongoing changes in the web site. The two systems perform similarly when tested on the 2004-11-17 data, but the difference is much greater for the 2005-01-22 data. This difference represents only two months of changes to the web site.

In the data we have reported so far, all attacks were removed from the training data. We also tested learning when attacks occur in the training data. For a properly patched web server, these attacks (such as Code Red or Nimda) are harmless. If a successful attack occurs during training, then two problems will occur: First, the attack is learned as normal, and second, the web server administrator must re-install the operating system and retrain the DFA from scratch. Although many web server attacks exist, we rely on the fact that the zero-day attack rate is low enough that we are unlikely to encounter one during training (say one week).

When training on data containing harmless attacks, our system learned to tolerate
Site | False positive fraction
---|---
cs.unm.edu | 0.02701282
aya.org | 0.24090772
i-pi.com | 0.67835550
explorenm.com | 0.67999073

Table 4
DFA performance across web sites: The DFA induced from cs.unm.edu data is tested against data from other web sites. The false positive fractions are reported for a continually learning DFA with similarity threshold of 0.852 not using the two heuristics.

The idea here is that the training data contains a large fraction (for cs.unm.edu, the fraction is approximately 13%) of requests that are outdated attacks, that is, old attacks that the web servers had already been patched against. We would like our system to learn to tolerate these attack attempts in order to reduce the system administrator’s burden of looking at alarms that are not dangerous. On the other hand, we do not want the system’s tolerance of old attacks to bias it towards tolerating new attacks. Attacks in the training data were not recognized—after all, they were “normal.” The detection fraction for the other attacks remained unchanged, indicating that the system can learn harmless attacks as normal, but still recognize novel attacks.

Finally, to show that web sites receive different HTTP requests and that our system can exploit this diversity, Table 4 shows how the DFA built from the cs.unm.edu training data performs on the other web sites. The table shows that a DFA learned on data from one site does not generalize to other sites. The cs.unm.edu DFA misclassifies as many as $\frac{2}{3}$ of the requests destined for other web sites. The differences include different request paths and different client browser configurations. This result shows that we can automatically construct customized definitions of normal that are specific to a particular web site. This allows a tighter definition of normal (what is acceptable at one site is not normal at another) and provides resistance to widely replicated attacks.

6 Related Work

Earlier work on application-specific intrusion detection falls into three main categories: misuse (signature) detection, specification-based detection, and anomaly detection. The open-source intrusion detection system (IDS) snort [1] and most commercial systems are examples of the first category, in which preprogrammed rules are defined to recognize specific forms of attack. One signature-based system for protecting web servers was developed by Almgren et al. [10]. This system searches a log of requests for potential attack signatures against CGI programs.
Signature-based systems can detect previously seen attacks and are typically insensitive to behavioral changes in the protected system. Their major weaknesses are that they generally cannot detect novel (zero-day) attacks, and they can be prone to repeated false alarms in response to specific benign events (“squealing”) [11].

Formal specification systems (such as the system by Ko [12]) rely on an administrator or developer to specify rules defining acceptable system behavior. An alarm is then generated whenever these rules are broken. Although this approach can be effective, formal specifications take time to develop and require expertise beyond that which most programmers possess. Further, this labor-intensive exercise must be performed for every custom application that is to be protected.

With anomaly detection, acceptable behavior of a system is described by a model (typically learned empirically), and deviations from the model are labeled as anomalies. With some anomaly detection methods, only empirically observed behavior is considered to be acceptable (“normal”) [13], while with others all possible legal behavior (say, with respect to a given program binary) is considered acceptable [14]. As with specification-based methods, anomaly detection methods are capable of detecting novel attacks. It is, however, more difficult to characterize exactly what an anomaly-detector detects or what it might miss. Depending upon how well the model captures legitimate activities, some anomaly-detection systems can be vulnerable to false-positives.

One anomaly-based approach to web application intrusion detection is to treat web servers as a generic network service. Incoming and outgoing traffic, can then be modeled as a stream of bytes or as discrete packets. Wang and Stolfo [5], Mahoney [15], and Mahoney and Chan [16] showed that this approach can be effective in some circumstances. However, this earlier work relied on the 1999 MIT Lincoln Labs data for testing; this dataset has known problems (e.g., [17]), is quite dated as web behavior has evolved significantly over the past six years, and it contains only four different attacks against web servers.

A few researchers have developed web request-based anomaly intrusion detection methods. In 2003, Kruegel and Vigna [3] analyzed web server log files to avoid learning on dirty data, recording the CGI parameters appearing in GET requests. They used the CGI data to construct several models, which they then combined into a single anomaly signal. Their test suite consisted of 11 attacks, all of which abused CGI parameters of software installed at one of the sources of their data. Our attack database and testing extends this work to attacks that go beyond traditional CGI parameters. Only 60% of our database corresponds to attacks that target web applications using traditional CGI parameters. Adding to the evidence that many current attacks are not abusing CGI parameters, we note that CGI attacks comprise only 35% of the snort 2005-05-31 rules, and requests to CGI programs comprise only about 0.1% of all requests in our production data sets. One explanation for these data is that web applications have several methods for obtaining their data
(e.g., embedded in the path without the CGI formatting, using the POST method, etc). As an example, none of the dynamic content at explorenm.com uses the standard CGI parameter form, and this is true for much of aya.org also. Interestingly, Kruegel and Vigna removed both the “Code Red” and “Nimda” worms from the training sets, but they reported catching only Code Red, which used a buffer overflow exploit against a CGI program. In contrast, Nimda took advantage of a URL decoding error, and did not target a CGI program (and hence was invisible to their system).

More recently, Tombini et al. [4] combined misuse and anomaly detection to find attacks in logged HTTP requests. This approach takes advantage of the strengths of both misuse (signature) and anomaly detection to provide good results. They also analyzed how to resolve conflicts between the two methods in order to provide the best accuracy. They observed that out of 56 attacks, the misuse detector could detect 30—just over half. They did not publish results that tested how the combined system performed on attacks. In addition, they deleted dynamic user account data from their test suite (data that we retained in our test set), using only the dynamic content on the official web site, and then only after manually verifying the anomaly detector rules. They used this hand-tuned set of rules for their anomaly detector. They also used manual methods to identify safe or dangerous web requests, reporting that a person can perform this task within a few days.

One limitation of these published approaches is that nobody has reported positive results on attacks such as Apache Sioux [18], where the attack is not in the requested path, but instead occurs elsewhere in the HTTP request. Our system easily catches this attack. Finally, as noted earlier, web sites provide a nonstationary environment for IDSs to operate in. Only Wang and Stolfo [5] addressed this issue. All of the others would likely need regular retraining—retraining that often requires a dataset without any attacks in it. As mentioned earlier, removing attacks from training data is a time-consuming task that would prevent many one-shot learning systems from easy conversion to long-term production use.

Although our application of DFA induction is novel, the literature on DFA induction algorithms is quite extensive (for example, see [19,20]). Learning minimal DFAs from training data is desirable because such DFAs make the least number of assumptions about the underlying system. However, in the worse case, finding minimum DFAs is known to be NP-complete [21]. However, the average-case complexity of finding minimum DFAs is tractable using simple state-merging algorithms [22], and there has been significant research into inducing DFAs since this pivotal work [20,23]. Although the Burge algorithm finds a small enough DFA for our purposes, it is not guaranteed to be minimal. Other researchers (e.g., [24,25,?]) have studied anomaly detection using methods for constructing finite automata-based models of program behavior. We believe, however, that this is the first work using DFA induction methods for the web application protection problem.
In general, learning algorithms for one-class, nonstationary problem are rare, although this is an area of current interest in the machine learning community (e.g., see [16,26,27]). Adding the complications of noisy data, makes the problem even more difficult. Methods for handling nonstationary data include forgetting, as described by Salganicoff [28] in the context of learning how to grasp when portions of the gripper have failed. Neural networks with adaptive critics have been shown, e.g., by Prokhorov [29]. Unfortunately, these approaches require a fixed input size, a limitation that prevents use with HTTP requests. Littman and Ackley [30] looked at cases where the problem can be divided into two parts, variable and invariant; unfortunately, this approach does not apply to our environment, where little is invariant.

7 Discussion

Protecting web servers is a challenging and important problem, especially in the context of custom web-based applications. Even if the server itself is secure, vulnerabilities in the application can render the system insecure. Our anomaly detection method addresses several of the more important problems that arise in protecting web servers. Specifically, we have described a one-class online learning algorithm that is capable of handling nonstationary data of arbitrary length and is capable of learning correctly, even when old attacks are present in the training data. However, any learning algorithm that meets the criteria of Section 2 could be applied to this anomaly-detection problem. Our algorithm shows how each of the criteria can be addressed.

In our work we encountered an interesting example of how quickly normal behavior can evolve—the novel web robot. This experience is not unique. Several previously published papers describe interesting difficulties with anomalous normal data. Tombini et al. [4] found a bug in how Count.cgi was used. Kruegel and Vigna [3] noticed a character string where an index was expected and traced it to user(s) testing the system in a non-malicious manner. These results imply that web behavior is evolving quickly.

Our automatically customized representations of normal behavior allow each web site to have a unique definition of normal behavior (no generalization across web sites). This feature gives our system the potential to stop “one attack fits all servers” style of attacks. Specifically, attacks must not only resemble acceptable requests for the targeted application, they must also resemble the requests that are sent to the specific installation. Because an attacker would need to craft an attack for each different web site, this additional workload would slow the attack rate to a level that can be handled by a human system administrator. Having said this, it is still possible that multiple installations could be compromised by a single attack targeting an application with very regular request structure; further research is required to
determine if this situation occurs in practice.

One weakness in our system is the fraction of the attacks missed by the system. Our true positive rate is approximately 70% (depending on the threshold for normal), with a corresponding false positive rate of up to 22 per day. For comparison, Wang and Stolfo [5] achieved a true positive rate of 80-100% (depending on the model and the acceptable false positive rate) on their four attacks. Kruegel and Vigna [3] achieved a true positive rate of 100% on their 11 attacks, with corresponding false positive rates of 1.89 to 4,994 per day. Note, however, that Kruegel and Vigna restricted their work to parameters for CGI scripts; 40% of the attacks in our database are not in this class, and only 35% of the snort 2005-05-31 rules are for this class of attacks.

Many of the false negatives in our results arise from attacks that use a large number of tokens, all but one or a few of which are normal. A different similarity metric from that used in this paper could improve our results, an area we plan to explore next. Another area of future work would be to develop additional heuristics, along the lines of those we described for false positives. In this case, however, our heuristics would be aimed at improving true-positive rates.

The online nature of the Burge DFA induction algorithm, and the fact that it can learn on a set of easily obtained normal requests, shows that our system could be deployed in a production IDS. Such an IDS would take the form of a proxy. Normal requests (including those that have a line deleted) are passed to the server. Abnormal requests would either be blocked or sent to a separate, less-functional but more secure server. The proxy approach also has the advantage that it can determine from the web server reply whether or not a request is valid. Currently our data set includes improper paths, syntax errors, and other examples of the problems web client, web proxy and web robot designers have in properly following the HTTP standards. The DFA learns these as normal, then (assuming that they are rare) they are deleted over time. Eliminating this source of noise would produce smaller DFAs.

8 Conclusion

Protecting web servers is an important unsolved problem. Even if the server itself is secure, vulnerabilities in web-based applications can render the system insecure. We have described an anomaly detection method that addresses several problems in protecting a web server. In particular, we developed a one-class online learning algorithm that can operate in dynamic environments on data of arbitrary length. The algorithm learns successfully even when old attacks persist in the data sets describing normal behavior.
Adequate testing and comparison of new intrusion-detection approaches is a time-consuming task. Our paper describes a data set that we developed for testing intrusion-detection systems in the web-application environment. Our attack database is significantly larger and more diverse than those reported by other researchers, and it is available for other researchers to use.

We foresee several avenues for extending this line of research. First, we are currently developing a prototype web defense system based upon the techniques reported in this paper. We are also developing a framework for testing additional algorithms under identical circumstances. By facilitating head-to-head comparisons, we believe such a prototype will help advance this area of research by identifying which algorithms work best, and under what circumstances they excel or fail. Finally, we believe that new heuristics and similarity measures can be developed which will improve the accuracy of our detector.

9 Acknowledgments

SF and KI gratefully acknowledge the support of the National Science Foundation (grants ANIR-9986555, CCR-0331580, and CCR-0311686), Defense Advanced Research Projects Agency (grants F30602-02-1-0146), and the Santa Fe Institute. AS gratefully acknowledges the support of Natural Sciences and Engineering Research Council of Canada (NSERC) and MITACS.

References


