Protecting a Moving Target: Addressing Web Application Concept Drift

Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna
{maggi,wkr,chris,vigna}@cs.ucsb.edu

Computer Security Group
UC Santa Barbara

Abstract. Because of the *ad hoc* nature of web applications, intrusion detection systems that leverage machine learning techniques are particularly well-suited for protecting websites. The reason is that these systems are able to characterize the applications' normal behavior in an automated fashion. However, anomaly-based detectors for web applications suffer from false positives that are generated whenever the applications being protected change. These false positives need to be analyzed by the security officer who then has to interact with the web application developers to confirm that the reported alerts were indeed erroneous detections. In this paper, we propose a novel technique for the automatic detection of changes in web applications, which allows for the selective retraining of the affected anomaly detection models. We demonstrate that, by correctly identifying legitimate changes in web applications, we can reduce false positives and allow for the automated retraining of the anomaly models. We have evaluated our approach by analyzing a number of real-world applications. Our analysis shows that web applications indeed change substantially over time, and that our technique is able to effectively detect changes and automatically adapt the anomaly detection models to the new structure of the changed web applications.

Keywords: Anomaly Detection, Web Application Security, Concept Drift, Machine Learning.

1 Introduction

According to a recent study by Symantec [1], web vulnerabilities represent 60% of all reported security flaws. In particular, site-specific vulnerabilities (i.e., those that affect custom web applications) are receiving increased attention from online criminals [2,3]. This is because by exploiting a single vulnerability in a popular site (e.g., a social networking site or a high-traffic portal), an attacker can infect a large number of end hosts by spreading malware via web browser exploits (e.g., drive-by download attacks). Therefore, there is a need for security tools and techniques to protect web applications and deal with their *ad hoc*, dynamic nature.

Anomaly-based intrusion detection techniques have been shown to be effective in protecting web applications against attacks [4,5,6,7,8]. In contrast to misuse detection systems, which contain fingerprints of all *known* attacks patterns,

anomaly-based detectors leverage models of the normal behavior of the monitored web applications to detect attacks, under the assumption that attacks cause anomalies, and anomalies are always associated with malicious activity. Besides an initial configuration, these tools typically neither require maintenance nor manual updates to provide protection. For these reasons, they have the advantage of offering a black-box solution to web application security, even against 0-day exploits and site-specific attacks. Some anomaly-based web attack detection techniques are mature enough to be implemented in commercial tools [9,10,11].

A class of anomaly detectors for web applications leverages machine learning techniques to automatically build models of the normal behavior of the monitored web applications. In this context, the term *normal behavior* generally refers to a set of characteristics (e.g., the distribution of the characters of string parameters, the mean and standard deviation of the values of integer parameters) extracted from HTTP messages that are observed during normal operation. Detection is performed under the assumption that attacks cause significant changes (i.e., anomalies) in the application behavior. Thus, any activity that does not fit the expected, learned models is flagged as malicious. Obviously, the detection accuracy strongly depends upon the quality of the models that describe the normal behavior. On one hand, over-specialization can lead to false positives [12,13]; on the other hand, over-generalization often results in false negatives [14,15,16].

One issue that has not been well-studied is the difficulty of adapting to changes in the behavior of the protected applications. By *behavior of a web application*, we refer to the features and the functionalities that the application offers and, as a consequence, the content of the inputs (i.e., the requests) that it process and the outputs (i.e., the responses) that it produces. This is an important problem because today's web applications are user-centric. That is, the demand for new services causes continuous updates to an application's logic and its interfaces.

Our analysis reveals that significant changes in the behavior of web applications are frequent. We refer to this phenomenon as *web application concept drift*. In the context of anomaly-based detection, this means that legitimate behavior might be misclassified as an attack after an update of the application, causing the generation of false positives. Normally, whenever a new version of an application is deployed in a production environment, a coordinated effort involving application maintainers, deployment administrators, and security experts is required. That is, developers have to inform administrators about the changes that are rolled out, and the administrators have to update or re-train the anomaly models accordingly. Otherwise, the amount of false positives will increase significantly. We propose a solution that makes these tedious tasks unnecessary. Our technique examines the responses (HTML pages) sent by a web application. More precisely, we check the forms and links in these pages to determine when new elements are added or old ones removed. This information is leveraged to identify legitimate changes.

Our technique recognizes when anomalous inputs (i.e., HTTP requests) are due to previous, legitimate updates (changes) in a web application. In such cases, false positives are suppressed by automatically and selectively re-training models. Moreover, when possible, model parameters can be automatically updated without requiring any re-training. Often, a complete re-training would be expensive

in terms of time; typically, it requires $O(P)$ where P represents the number of HTTP messages required to train a model. More importantly, such re-training is not always feasible since new, attack-free training data is unlikely to be available immediately after the application has changed. In fact, to collect a sufficient amount of data the new version of the application must be executed and real, legitimate clients have to interact with it in a controlled environment. Clearly, this task requires time and efforts. More importantly, those parts that have changed in the application must be known in advance.

Our approach takes a different perspective. We focus on the fundamental problem of *detecting* those parts of the application that have changed and that will cause false positives if no re-training is performed. Therefore, our technique is agnostic with respect to the specific training procedure, which can be different from the one we propose.

In summary, this paper proposes a set of change detection techniques to address the concept drift problem by treating the protected web applications as oracles. We show that HTTP responses contain important insights that can be effectively leveraged to update previously learned models to take changes into account. The results of applying our technique on real-world data show that learning-based anomaly detectors can automatically *adapt to changes*, and by doing this, are able to reduce their false positive rate without decreasing their detection accuracy.

In this paper, we make the following contributions.

- We detail the problem of concept drift in the context of web applications, and we provide evidence that it occurs in practice, motivating why it is a significant problem for deploying learning-based anomaly detectors in the real world.
- We present novel techniques based on HTTP response models that can be used to distinguish between legitimate changes in web applications and web-based attacks.
- We evaluate a tool incorporating these techniques over an extensive real-world data set, demonstrating its ability to deal with web application concept drift and reliably detect attacks with a low false positive rate.

2 Concept drift

To introduce the idea of concept drift, we will use a generalized model of learning-based anomaly detectors of web attacks. This model is based on the system presented in [5], but it is general enough to be adapted to virtually any learning-based anomaly detector for web applications. Also, we show that concept drift is a problem that exists in the real world, and we motivate why it should be addressed. Unless differently stated, we use the shorthand term *anomaly detector* to refer to anomaly-based detectors that leverage unsupervised machine learning techniques.

2.1 Anomaly detection for web applications

An anomaly detector builds models of normal behavior by observing HTTP messages exchanged between servers and clients. The traffic directed to the server

running a certain web application (e.g., an e-commerce application or a blog) can be organized into paths, or *resources*, $R = \{r_1, r_2, \dots, r_j, \dots\}$. Each resource corresponds to a different software module of the application (e.g., an account manager, a search component). Each resource r_j responds to requests, or *queries*, $Q = \{q_{j,1}, q_{j,2}, \dots, q_{j,i}, \dots\}$ that contain sets of name-value parameters transmitted by the client as part of the HTTP request. Each query $q_{j,i}$ is abstracted as a tuple $q_{j,i} = \langle r_j, P_q \rangle$, where $P_q = \{(p_1, v_1), (p_2, v_2), \dots, (p_k, v_k)\} \subseteq P_j$, and $P_j = P(r_j)$ is the set of *all* the parameters handled by r_j. For instance, the request `‘GET /page?id=21&uid=u43&action=del’` contains the resource $r_1 =$ `‘/page’` and the parameters $P_q = \{\langle p_1 = \texttt{id}, v_1 = 21\rangle, \langle p_2 = \texttt{uid}, v_2 = \texttt{`u43'}\rangle, \langle p_3 = \texttt{action}, v_3 = \texttt{`del'}\rangle\}$. Typically, an anomaly detector would use different models to capture legitimate values associated with each parameter.

In addition to requests, the structure of user sessions can be taken into account to model the normal states of a server-side application. In this case, the anomaly detector does not consider individual requests independently, but models their sequence. This model captures the legitimate order of invocation of the resources, according to the application logic. An example is when a user is required to invoke an authentication resource (e.g., `/user/auth`) before requesting a private page (e.g., `/user/profile`). In [5], a session S is defined as a sequence of resources in R. For instance, given $R = \{r_1, r_2, \dots, r_{10}\}$, a sample session is $S = \langle r_3, r_1, r_2, r_{10}, r_2 \rangle$.

Finally, HTTP responses that are returned by the server can also be modeled. For example, in [5], a model $m^{(\text{doc})}$ is presented that takes into account the structure of documents (e.g., HTML, XML, and JSON) in terms of partial trees that include security-relevant nodes (e.g., `<script />` nodes, nodes containing DOM event handlers, and nodes that contain sensitive data such as credit card numbers). These trees are iteratively merged as new documents are observed, creating a superset of the allowed document structure and the positions within the tree where client-side code or sensitive data may appear.

During the *learning* (or training) phase, given a training set of queries Q and the corresponding responses, the model parameters are estimated and appropriate anomaly thresholds are calculated. More precisely, each parameter of a resource r_i is associated with a set of models; this set of models is called a *profile*: $c_{(\cdot)} = \langle m_1, m_2, \dots, m_u \rangle$. The specific models in $c_{(\cdot)}$ and the strategy to combine their output determine the classes of attacks that can be detected. The interested reader is referred to [5,8,17] for more details.

During *detection*, for each new request q and corresponding response, the database of profiles is used to calculate an aggregated *anomaly score*, which takes into account the anomaly score of the request or the response according to all the applicable models. In general, an alert is raised if the aggregated anomaly score is above the threshold learned during training.

In this work, the set of models implemented in webanomaly [5] is used to show how anomaly detectors can be improved to cope with the problem of concept drift. However, the techniques we propose in this work can be easily applied to other anomaly-based detectors.

2.2 Web applications are not static

In machine learning, changes in the modeled behavior are known as *concept drift* [18]. Intuitively, the *concept* is the modeled phenomenon (e.g., the structure of requests to a web server, the recurring patterns in the payload of network packets). Thus, variations in the main features of the phenomena under consideration result in changes, or *drifts*, in the *concept*.

Although the generalization and abstraction capabilities of modern learning-based anomaly detectors are resilient to noise (i.e., small, legitimate variations in the modeled behavior), concept drift is difficult to detect and to cope with [19]. The reason is that the parameters of the models may stabilize to different values. For instance, a string length model could calculate the sample mean and variance of the string lengths that are observed during training. Then, during detection, the Chebyshev inequality is used to detect strings with lengths that significantly deviate from the mean, taking into account the observed variance. Clearly, small differences in the lengths of strings will be considered normal. On the other hand, the mean and variance of the string lengths can completely change because of legitimate and permanent modifications in the web application. In this case, the normal mean and variance will stabilize, or drift, to completely different values. If appropriate re-training or manual updates are not performed, the model will classify benign, new strings as anomalous. This might be a human-intensive activity requiring substantial expertise. Therefore, having an automated, black-box mechanism to adjust the parameters is clearly very desirable.

Changes in web applications can manifest themselves in several ways. In the context of learning-based detection of web attacks, those changes can be categorized into three groups: *request* changes, *session* changes, and *response* changes.

Request changes. Changes in requests occur when an application is upgraded to handle different HTTP requests. These changes can be further divided into two groups: *parameter value* changes and *request structure* changes. The former involve modifications of the actual value of the parameters, while the latter occur when parameters are *added* or *removed*. Parameter *renaming* is the result of removal plus addition.

Example. A new version of a web forum introduces internationalization (I18N) and localization (L10N). Besides handling different languages, I18N and L10N allow several types of strings to be parsed as valid dates and times. For instance, valid strings for the `datetime` parameter are `‘3 May 2009 3:00’`, `‘3/12/2009’`, `‘3/12/2009 3:00 PM GMT-08’`, `‘now’`. In the previous version, valid date-time strings had to conform to the regular expression `‘[0-9]{1,2}/[0-9]{2}/[0-9]{4}’`. A model with good generalization properties would learn that the field `datetime` is composed of numbers and slashes, with no spaces. Thus, other strings such as `‘now’` or `‘3/12/2009 3:00 PM GMT-08’` would be flagged as anomalous. Also, in our example, `tz` and `lang` parameters have been added to take into account time zones and languages. To summarize, the new version introduces two classes of changes. Clearly, the parameter domain of `datetime` is modified. Secondly, new parameters are added.

Changes in HTTP requests directly affect the request models. First, parameter value changes affect any models that rely on the parameters' *values* to extract features. For instance, consider two of the models used in the system described in [5]: $m^{(\text{char})}$ and $m^{(\text{struct})}$. The former models the strings' character distribution by storing the frequency of all the symbols found in the strings during training, while the latter models the strings' structure as a stochastic grammar, using a *Hidden Markov Model* (HMM). In the aforementioned example, the I18N and L10N introduce new, legitimate values in the parameters; thus, the frequency of numbers in $m^{(\text{char})}$ changes and new symbols (e.g., `'-'`, `'[a-zA-Z]'` have to be taken into account. It is straightforward to note that $m^{(\text{struct})}$ is affected in terms of new transitions introduced in the HMM by the new strings. Secondly, request structure changes may affect any type of request model, regardless of the specific characteristics. For instance, if a model for a new parameter is missing, requests that contain that parameter might be flagged as anomalous.

Session changes. Changes in sessions occur whenever resource path sequences are *reordered*, *inserted*, or *removed*. Adding or removing application modules introduces changes in the session models. Also, modifications in the application logic are reflected in the session models as reordering of the resources invoked.

Example. A new version of a web-based community software grants read-only access to *non-authenticated* users, allowing them to display contents previously available to subscribed users only. In the old version, legitimate sequences were ⟨`/site`, `/auth`, `/blog`⟩ or ⟨`/site`, `/auth`, `/files`⟩, where `/site` indicates the server-side resource that handles the public site, `/auth` is the authentication resource, and `/blog` and `/files` were formerly private resources. Initially, the probability of observing `/auth` before `/blog` or `/files` is close to one (since users need to authenticate before accessing private material). This is no longer true in the new version, however, where `/files|/blog|/auth` are all possible after `/site`.

Changes in sessions impact all models that rely on the sequence of resources that are invoked during the normal operation of an application. For instance, consider the model $m^{(\text{sess})}$ described in [5], which builds a probabilistic finite state automaton that captures sequences of *resource paths*. New arcs must be added to take into account the changes mentioned in the above example. These types of models are sensitive to strong changes in the session structure and should be updated accordingly when they occur.

Response changes. Changes in responses occur whenever an application is upgraded to produce different responses. Interface redesigns and feature addition or removal are example causes of changes in the responses. Response changes are common and frequent, since page updates or redesigns often occur in modern websites.

Example. A new version of a video sharing application introduces Web 2.0 features into the user interface, allowing for the modification of user interface elements without refreshing the entire page. In the old version, relatively few nodes of documents generated by the application contained client-side code. In

the new version, however, many nodes of the document contain event handlers to trigger asynchronous requests to the application in response to user events. Thus, if a response model is not updated to reflect the new structure of such documents, a large of number of false positives will be generated due to *legitimate* changes in the characteristics of the web application responses.

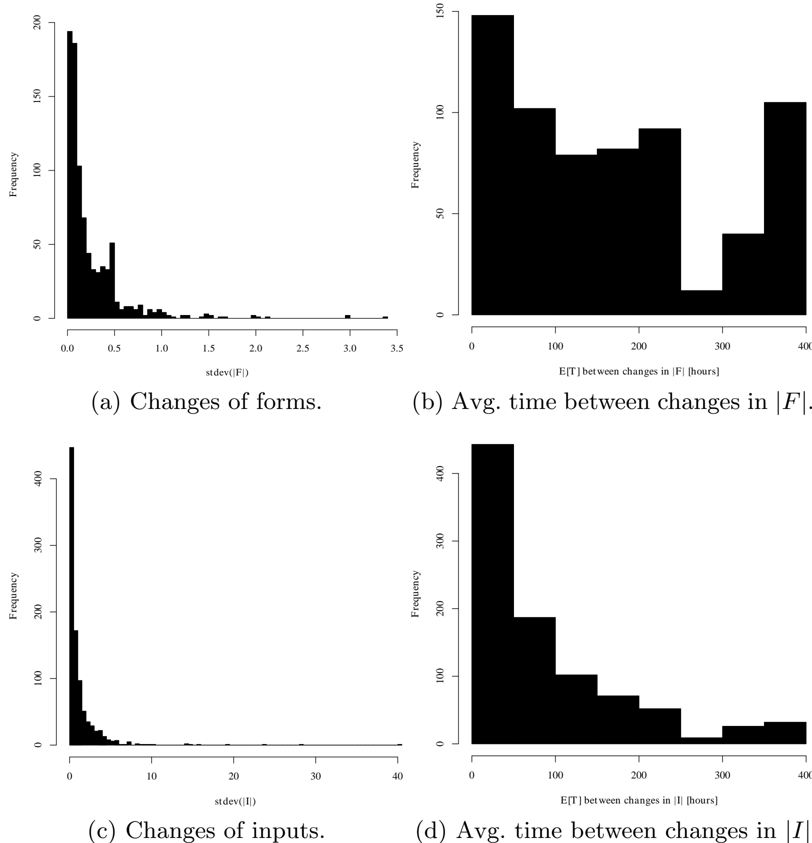
2.3 Prevalence of concept drift

To understand whether concept drift is a relevant issue for real-world websites, we performed three experiments. For the first experiment, we monitored 2,264 public websites, including the Alexa Top 500 and other sites collected by querying Google with popular terms extracted from the Alexa Top 500. The goal was to identify and quantify the changes in the forms and input fields of popular websites at large. This provides an indication of the frequency with which real-world applications are updated or altered.

Once every hour, we visited one representative page for each of the 2,264 websites. In total, we collected 3,303,816 pages, comprising more than 1,390 snapshots for each website, between January 29 and April 13, 2009. One tenth of the representative pages were manually selected to have a significant number of forms, input fields, and hyperlinks with parameters (e.g., <a href="/login?anon=true&-lang=en" />). By doing this, we gathered a considerable amount of information regarding the HTTP messages generated by some applications. Examples of these pages are registration pages, data submission pages, or contact form pages. For the remaining websites, we simply used their home pages.

For each website w, each page sample crawled at time t is associated with a tuple $|F|_t^{(w)}, |I|_t^{(w)}$, the cardinality of the sets of forms and input fields, respectively. By doing this, we collected samples of the variables $|F|^w = |F|_{t_1}^w, \ldots, |F|_{t_n}^w$, $|I|^w = |I|_{t_1}^w, \ldots, |I|_{t_n}^w$, with $0 < n \lesssim 1,390$. Figure 1 shows the relative frequency of the variables $X_I = \mathrm{stdev}(|I|^{(w_1)})$, $\ldots$, $\mathrm{stdev}(|I|^{(w_k)})$ and $X_F = \mathrm{stdev}(|F|^{(w_1)})$, $\ldots$, $\mathrm{stdev}(|F|^{(w_k)})$. This demonstrates that a significant amount of websites exhibit variability in the response models, in terms of elements modified in the pages, as well as request models, in terms of new forms and parameters. In addition, we estimated the expected time between changes of forms and inputs fields, $E[T_F]$ and $E[T_I]$, respectively. In terms of forms, 40.72% of the websites drifted during the observation period. More precisely, 922 out of 2,264 websites have a finite $E[T_F]$. Similarly, 29.15% of the websites exhibited drifts in the number of input fields, i.e., $E[T_I] < +\infty$ for 660 websites. Figure 1 shows the relative frequency of (b) $E[T_F]$, and (d) $E[T_I]$. $E[T_F]$. This confirms that a non-negligible portion of the websites exhibit significantly frequent changes in the responses.

For the second experiment, we monitored in depth three large, data-centric web applications over several months: Yahoo! Mail, YouTube, and MySpace. We dumped HTTP responses captured by emulating user interaction using a custom, scriptable web browser implemented with HtmlUnit. Examples of these interactions are as follows: visit the home page, login, browse the inbox, send messages, return to the home page, click links, log out. Manual inspection revealed some major changes in Yahoo! Mail. For instance, the most evident change consisted of



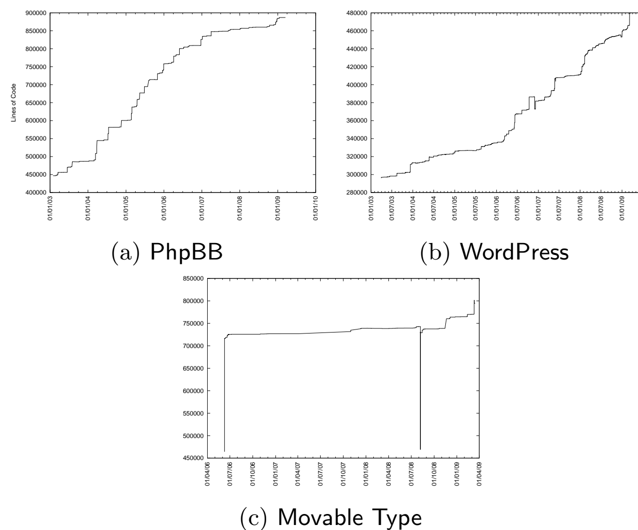
(a) Changes of forms. (b) Avg. time between changes in $|F|$.

(c) Changes of inputs. (d) Avg. time between changes in $|I|$

Fig. 1: Relative frequency of the standard deviation of the number of forms (a) and input fields (c). Also, the distribution of the expected time between changes of forms (b) and input fields (d) are plotted. A non-negligible portion of the websites exhibits changes in the responses.

a set of new features added to the search engine (e.g., local search, refined address field in maps search), which manifested themselves as new parameters found in the web search page (e.g. to take into account the country or the ZIP code). User pages of YouTube were significantly updated with new functionalities between 2008 and 2009. For instance, the new version allows users to rearrange widgets in their personal pages. To account for the position of each element, new parameters are added to the profile pages and submitted asynchronously whenever the user drags widgets within the layout. The analysis on MySpace did not reveal any significant change. The results of these two experiments show that changes in server-side applications are common. More importantly, these modifications often involve the way user data is represented, handled, and manipulated.

For the third experiment, we analyzed changes in the requests and sessions by inspecting the code repositories of three of the largest, most popular open-source web applications: WordPress, Movable Type, and PhpBB. The goal was to understand whether upgrading a web application to a newer release results in significant



(a) PhpBB (b) WordPress

(c) Movable Type

Fig. 2: Lines of codes in the repositories of PhpBB, WordPress, and Movable Type, over time. Counts include only the code that manipulates HTTP responses, requests and sessions.

changes in the features that are used to determine its behavior. In this analysis, we examined changes in the source code that affect the manipulation of HTTP responses, requests, and session data. We used StatSVN, an open-source tool for tracking and visualizing the activity of SVN repositories (e.g., the number of lines changed or the most active developers). We modified StatSVN to incorporate a set of heuristics to compute approximate counts of the lines of code that, directly or indirectly, manipulate HTTP session, request or response data. In the case of PHP, examples representative of such lines include, but are not limited to, `_REQUEST|_SESSION|_POST|_GET|session_|http_|strip_tags|addslashes`. In order to take into account data manipulation performed through library functions (e.g., WordPress' custom `Http` class), we also generated application-specific code patterns by manually inspecting and filtering the core libraries. Figure 2 shows, over time, the lines of code in the repositories of PhpBB, WordPress, and Movable Type that manipulate HTTP responses, requests and, sessions. These results show the presence of significant modifications in the web application in terms of relevant lines of code added or removed. More importantly, such modifications affect the way HTTP data is manipulated and, thus, impact request, response or session models.

The aforementioned experiments confirm that the class of changes we described in Section 2.2 is common in real-world web applications. Therefore, we conclude that anomaly detectors for web applications must incorporate procedures to prevent false alerts due to concept drift. In particular, a mechanism is needed to discriminate between legitimate and malicious changes, and respond accordingly.

3 Addressing concept drift

In this section, we first present our technique to distinguish between legitimate changes in web application behavior and evidence of malicious behavior. We then discuss how a web application anomaly detection system can effectively handle legitimate concept drift.

3.1 The web application as oracle

The body of HTTP responses contains a set of links L_i and forms F_i that refer to a set of target resources. Each form also includes a set of input fields I_i. In addition, each link $l_{i,j} \in L_i$ and form $f_{i,j} \in F_i$ has an associated set of parameters.

From a resource r_i, the client clicks upon a link $l_{i,j}$ or submits a form $f_{i,j}$. Either of these actions generates a new HTTP request to the web application with a set of parameter key-value pairs, resulting in the return of a new HTTP response to the client, r_{i+1}, the body of which contains a set of links L_{i+1} and forms F_{i+1}. This process continues until the session has ended (i.e., either the user has explicitly logged out, or a timeout has occurred).

Our key observation is that, at each step of a web application session, the set of potential target resources is given exactly by the content of the current resource. That is, given r_i, the associated sets of links L_i and forms F_i directly encode a significant sub-set of the possible r_{i+1}. Furthermore, each link $l_{i,j}$ and form $f_{i,j}$ indicates a precise set of expected parameters and, in some cases, the set of legitimate values for those parameters that can be provided by a client.

Example. Consider a hypothetical banking web application, where the current resource r_i = `/account` presented to a client is an account overview containing a set of links $L_i = \{$`/account/history?aid=328849660322`, `/account/history?aid=446825759916`, `/account/transfer`, `/logout`$\}$ and forms (represented as their target action) $F_i = \{$`/feedback`, `/search`$\}$.

From L_i and F_i, we can deduce the set of legal candidate resources for the next request r_{i+1}. Any other resource would, by definition, be a deviation from a legal session flow through the web application as specified by the application itself. For instance, it would not be expected behavior for a client to directly access `/account/transfer/submit` (i.e., a resource intended to submit an account funds transfer) from r_i. Furthermore, for the resource `/account/history`, it is clear that the web application expects to receive a single parameter `aid` with an account number as an identifier.

In the case of the form with target `/feedback`, let the associated input elements be:

```
<select name="subject">
  <option>General</option>
  <option>User interface</option>
  <option>Functionality</option>
</select>
<textarea name="message" />
```

It immediately follows that any invocation of the `/feedback` resource from r_i should include the parameters `subject` and `message`. In addition, the legal set of values for the parameter `subject` is given by enumerating the enclosed `<option />` tags. Similarly, valid values for the new `tz` and `datetime` parameters mentioned in the example of Section 2.2 can be inferred. Any deviation from these specifications could be considered evidence of malicious behavior.

We conclude that the responses generated by a web application constitute a specification of the intended behavior of clients and the expected inputs to an application's resources. As a consequence, when a change occurs in the interface presented by a web application, this will be reflected in the content of its responses. Therefore, as detailed in the following section, our anomaly detection system performs response modeling to detect and adapt to changes in monitored web applications.

3.2 Adaptive response modeling

In order to detect changes in web application interfaces, the response modeling of webanomaly has been augmented with the ability to build L_i and F_i from the HTML documents returned to a client. The approach is divided into two phases.

Extraction and parsing. The anomaly detector parses each HTML document contained in a response issued by the web application to a client. For each `<a />` tag encountered, the contents of the `href` attribute is extracted and analyzed. The link is decomposed into tokens representing the protocol (e.g., `http`, `https`, `javascript`, `mailto`), target host, port, path, parameter sequence, and anchor. Paths are subject to additional processing; for instance, relative paths are normalized to obtain a canonical representation. This information is stored as part of an abstract document model for later processing.

A similar process occurs for forms. When a `<form />` tag is encountered, the `action` attribute is extracted and analyzed as in the case of the link `href` attribute. Furthermore, any `<input />`, `<textarea />`, or `<select />` and `<option />` tags enclosed by a particular `<form />` tag are parsed as parameters to the corresponding form invocation. For `<input />` tags, the `type`, `name`, and `value` attributes are extracted. For `<textarea />` tags, the `name` attribute is extracted. Finally, for `<select />` tags, the `name` attribute is extracted, as well as the content of any enclosed `<option />` tags. The target of the form and its parameters are recorded in the abstract document model as in the case for links.

Analysis and modeling. The set of links and forms contained in a response is processed by the anomaly engine. For each link and form, the corresponding target resource is compared to the existing known set of resources. If the resource has not been observed before, a new model is created for that resource. The session model is also updated to account for a potential transition from the resource associated with the parsed document and the target resource by training on the observed session request sequence.

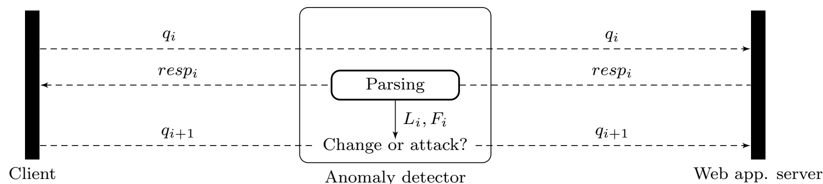


Fig. 3: A representation of the interaction between the client and the web application server, monitored by a learning-based anomaly detector. After request q_i is processed, the corresponding response $resp_i$ is intercepted and link L_i and forms F_i are parsed to update the request models. This knowledge is exploited as a change detection criterion for the subsequent request q_{i+1}.

For each of the parameters parsed from links or forms contained in a response, a comparison with the existing set of known parameters is performed. If a parameter has not already been observed (e.g., the new `tz` parameter), a profile is created and associated with the target resource model.

Any values contained in the response for a given parameter are processed as training samples for the associated models. In cases where the total set of legal parameter values is specified (e.g., `<select />` and `<option />` tags), the parameter profile is updated to reflect this. Otherwise, the profile is trained on subsequent requests to the associated resource.

As a result of this analysis, the anomaly detector is able to adapt to changes in session structure resulting from the introduction of new resources. In addition, the anomaly detector is able to adapt to changes in request structure resulting from the introduction of new parameters and, in a limited sense, to changes in parameter values.

3.3 Advantages and limitations

Due to the response modeling algorithm described in the previous section, our web application anomaly detector is able to automatically adapt to many common changes observed in web applications as modifications are made to the interface presented to clients. Both changes in session and request structure such as those described in Section 2.2 can be accounted for in an automated fashion. For instance, the I18N and L10N modification of the aforementioned example is correctly handled as it consists in an addition of the `tz` parameter and a modification of the `datetime` parameter. Furthermore, we claim that web application anomaly detectors that do not perform response modeling cannot reliably distinguish between anomalies caused by legitimate changes in web applications and those caused by malicious behavior. Therefore, as will be shown in Section 4, any such detector that solely monitors requests is more prone to false positives in the real world.

Clearly, the technique relies upon the assumption that the web application has not been compromised. Since the web application, and in particular the documents it generates, is treated as an oracle for whether a change has occurred, if an attacker were to compromise the application in order to introduce a malicious

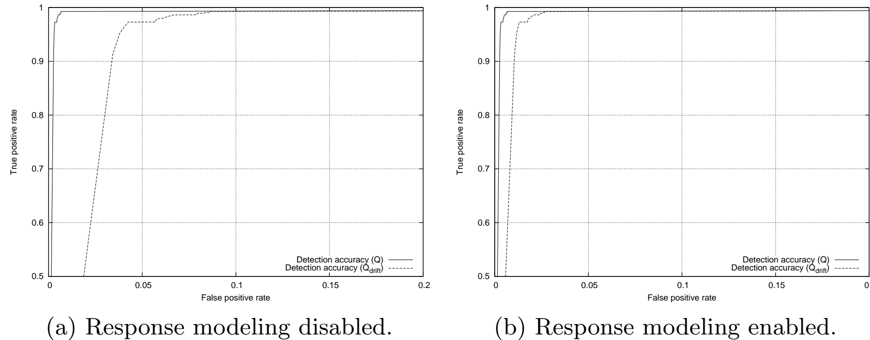
change, the malicious behavior would be learned as normal by our anomaly detector. Of course, in this case, the attacker would already have access to the web application. However, we remark that our anomaly detector observes all requests and responses to and from untrusted clients, therefore, any attack that would compromise response modeling would be detected and blocked. For example, an attacker could attempt to evade the anomaly detector by introducing a malicious change in the HTTP responses and then exploits the change detection technique that would interpret the new malicious request as a legit change. For instance, the attacker could incorporate a link that contain a parameter used to inject the attack vector. To this end, the attacker would have to gain control of the server by leveraging an existing vulnerability[1] of the web application (e.g., a buffer overflow, a SQL injection). However, the HTTP requests used by the attacker to exploit the vulnerability will trigger several models (e.g., the string length model, in the case of a buffer overflow) and, thus, will be flagged as anomalous. In fact, our technique does not alter the ability of the anomaly detector to detect attacks. On the other hand, it avoids many false positives, as demonstrated in Section 4.2.

Besides the aforementioned assumptions, three limitations are important to note. First, the set of target resources may not always be statically derivable from a given resource. For instance, this can occur when client-side scripts are used to dynamically generate page content, including links and forms. Accounting for dynamic behavior would require the inclusion of script interpretation. This, however, has a high overhead, is complex to perform accurately, and introduces the potential for denial of service attacks against the anomaly detection system. For these reasons, we have not included such a component in the current system, although further research is planned to deal with dynamic behavior. Moreover, as Section 4 demonstrates, the proposed technique performs well in practice.

Second, the technique does not fully address changes in the behavior of individual request parameters in its current form. In cases where legitimate parameter values are statically encoded as part of an HTML document, response modeling can directly account for changes in the legal set of parameter values. Unfortunately, in the absence of any other discernible changes in the response, changes in parameter values provided by clients cannot be detected. However, heuristics such as detecting when all clients switch to a new observable behavior in parameter values (i.e., all clients generate anomalies against a set of models in a similar way) could serve as an indication that a change in legitimate parameter behavior has occurred.

Third, the technique cannot handle the case where a resource is the result of a parametrized query and the previous response has not been observed by the anomaly detector. In our experience, however, this does not occur frequently in practice, especially for sensitive resources.

[1] The threat model assumes that the attacker can interact with the web application only by sending HTTP requests.



(a) Response modeling disabled.

(b) Response modeling enabled.

Fig. 4: Detection and false positive rates measured on Q and Q_{drift}, with HTTP response modeling enabled in (b).

4 Evaluation

In this section, we show that our techniques reliably distinguish between legitimate changes and evidence of malicious behavior, and present the resulting improvement in terms of detection accuracy.

The goal of this evaluation is twofold. We first show that concept drift in modeled behavior caused by changes in web applications results in lower detection accuracy. Second, we demonstrate that our technique based on HTTP responses effectively mitigates the effects of concept drift. In both the experiments, the testing data set includes samples of the most common types of attacks against web applications such as cross-site scripting (XSS) (e.g., CVE-2009-0781), SQL injections (e.g., CVE-2009-1224), and command execution exploits (e.g., CVE-2009-0258) that are reflected in request parameter values. In particular, we included a total of 1000 attacks, comprised of 400 XSS attacks, 400 SQL injections, and 200 command injections. The XSS attacks are variations on those listed in [20], the SQL injections were created similarly from [21], and the command execution exploits were variations of common command injections against the Linux and Windows platforms.

In both experiments, webanomaly was evaluated on a data set consisting of HTTP traffic drawn from real-world web applications. This data was obtained from several monitoring points at both commercial and academic sites. For each application, the full contents of each HTTP connection observed over a period of several months were recorded. The resulting flows were filtered using signature-based techniques to remove known attacks, and then partitioned into distinct training and test sets. In total, the data set contains 823 unique web applications, 36,392 unique resource paths, 16,671 unique parameters, and 58,734,624 HTTP requests.

4.1 Effects of concept drift

In the first experiment, we demonstrate that concept drift as observed in real-world web applications results in a significant negative impact on false positive

rates. First, webanomaly was trained on an unmodified, filtered data set. Then, the detector analyzed a test data set Q to obtain a baseline ROC curve.

After the baseline curve had been obtained, the test data set was processed to introduce new behaviors corresponding to the effects of web application changes, such as upgrades or source code refactoring, obtaining Q_{drift}. In this manner, the set of changes in web application behavior was explicitly known. In particular, as detailed in Table 1, 6,749 new session flows were created by introducing requests for new resources and creating request sequences for both new and known resources that had not previously been observed. Also, new parameter sets were created by introducing 6,750 new parameters to existing requests. Finally, the behavior of modeled features of parameter values was changed by introducing 5,785 mutations of observed values in client requests. For example, each sequence of resources ⟨`/login, /index, /article`⟩ might be transformed to ⟨`/login, /article`⟩. Similarly, each request like `/categories` found in the traffic might be replaced with `/foobar`. For new parameters, a set of link or form parameters might be updated by changing a parameter name and updating requests accordingly.

It must be noted that in all cases, responses generated by the web application were modified to reflect changes in client behavior. To this end, references to new resources were inserted in documents generated by the web application, and both links and forms contained in documents were updated to reflect new parameters.

webanomaly – without the HTTP response modeling technique enabled – was then run over Q_{drift} to determine the effects of concept drift upon detector accuracy. The resulting ROC curves are shown in Figure 4a. The consequences of web application change are clearly reflected in the increase in false positive rate for Q_{drift} versus that for Q. Each new session flow and parameter manifests as an alert, since the detector is unable to distinguish between anomalies due to malicious behavior and those due to legitimate change in the web application.

4.2 Change detection

The second experiment quantifies the improvement in the detection accuracy of webanomaly in the presence of web application change. As before, the detector was trained over an unmodified filtered data set, and the resulting profiles were evaluated over both Q and Q_{drift}. In this experiment, however, the HTTP response modeling technique was enabled.

Figure 4b presents the results of analyzing HTTP responses on detection accuracy. Since many changes in the behavior of the web application and its clients can be discovered using our response modeling technique, the false positive rate for Q_{drift} is greatly reduced over that shown in Figure 4a, and approaches that of Q, where no changes have been introduced. The small observed increase in false positive rate can be attributed to the effects of changes in parameter values. This occurs because a change has been introduced into a parameter value submitted by a client to the web application, and no indication of this change was detected on the preceding document returned to the client (e.g., because no `<select />` were found).

Change type	Anomalies	False Positives	Reduction
New session flows	6,749	0	100.0%
New parameters	6,750	0	100.0%
Modified parameters	5,785	4,821	16.6%
Total	19,284	4,821	75.0%

Table 1: Reduction in the false positive rate due to HTTP response modeling for various types of changes.

Table 1 displays the individual contributions to the reduction of the false positive rate due to the response modeling technique. Specifically, the total number of anomalies caused by each type of change, the number of anomalies erroneously reported as alerts, and the corresponding reduction in the false positive rate is shown. The results displayed were generated from a run using the optimal operating point (0.00144, 0.97263) indicated by the knee of the ROC curve in Figure 4b. For changes in session flows and parameters sets, the detector was able to identify an anomaly as being caused by a change in web application behavior in all cases. This resulted in a large net decrease in the false positive rate of the detector with response modeling enabled. The modification of parameters is more problematic, though; as discussed in Section 3.3, it is not always apparent that a change has occurred when that change is limited to the type of behavior a parameter's value exhibits.

From the overall improvement in false positive rates, we conclude that HTTP response modeling is an effective technique for distinguishing between anomalies due to legitimate changes in web applications and those caused by malicious behavior. Furthermore, any anomaly detector that does not do so is prone to generating a large number of false positives when changes do occur in the modeled application. Finally, as it has been shown in Section 2, web applications exhibit significant long-term change in practice, and, therefore, concept drift is a critical aspect of web application anomaly detection that must be addressed.

5 Related work

Anomaly-based IDSs have evolved considerably after Denning's seminal paper on intrusion detection [22]. Besides network-based detection [23], anomaly-based techniques have been also exploited to protect the operating system. In [24], the normal behavior of applications is captured by modeling system call sequences [25,26] along with features of their arguments. In [27], a mixture of machine learning techniques is exploited to detect anomalous system calls in the Linux kernel. Ad-hoc distances between system calls are defined to perform clustering in order to identify natural classes of similar calls. The reduced size of the clustered input makes the training of Markov chains efficient. The behavior of each host application is modeled as Markov chains on which probabilistic thresholds are calculated to detect misbehaving sequences.

PAYL [28] is a network-based anomaly detection system. It creates models of each service's normal behavior by recording byte frequencies of network streams. This approach has been further explored in [29], where higher-order n-grams are

used instead of frequencies. Instead, [30] exploits self-organizing maps to classify the payload of IP frames in order to separate normal packets from malicious ones.

Anomaly-based detectors of web attacks have been first proposed in [5], where a multi-model approach to characterizing the normal behavior of web application parameters is proposed.

A tool to protect against code-injection attacks has been recently proposed in [17]. The approach exploits a mixture of Markov chains to model legitimate payloads at the HTTP layer. The computational complexity of n-grams with large n is solved using Markov chain factorization, making the system algorithmically efficient.

HTTP responses are exploited in [8]. Besides other features, the DOM is modeled to enhance the detection capabilities of SQL injection and cross-site scripting attacks. The fact that it relies on HTTP responses makes this approach similar to ours. However, we exploit HTTP responses to *detect changes* and update *other* anomaly models accordingly, instead of modeling responses *per se*.

A complementary tool is proposed in [6], where an approach to improve the explanatory power of anomaly-based detectors is proposed along with a clustering and classification methodology to reduce their false positive rate. Another technique to increase detection accuracy is presented in [31], where Bayesian networks are exploited to combine models and define inter-model dependencies. The resulting system shows a significant reduction in false alerts.

Reduction of false positives in anomaly detection systems has also been studied in [13]. Similar behavioral profiles for individual hosts are grouped together using a k-means clustering algorithm. However, the distance metric used was not explicitly defined. Coarse network statistics such as the average number of hosts contacted per hour, the average number of packets exchanged per hour, and the average length of packets exchanged per hour are all examples of metrics used to generate behavior profiles. A voting scheme is used to generate alerts, in which alert-triggering events are evaluated against profiles from other members of that cluster. Events that are deemed anomalous by all members generate alerts.

6 Conclusions

In this work, we have identified the natural dynamicity of web applications as an issue that must be addressed by modern anomaly-based web application anomaly detectors in order to prevent increases in the false positive rate whenever the monitored web application is changed. We refer to this frequent phenomenon the *web application concept drift*.

We propose the use of novel HTTP response modeling techniques to discriminate between legitimate changes and anomalous behaviors in web applications. More precisely, responses are analyzed to find new and previously unmodeled parameters. This information is extracted from anchors and forms elements, and then leveraged to update request and session models. We have evaluated the effectiveness of our approach over an extensive real-world data set of web application traffic. The results show that the resulting system can detect anomalies and avoid false alerts in the presence of concept drift.

As future work, we plan to investigate the potential benefits of modeling the behavior of JavaScript code, which is becoming increasingly prevalent in modern web applications. Also, additional, richer, and media-dependent response models must be studied to account for interactive client-side components, such as Adobe Flash and Microsoft Silverlight applications.

Acknowledgments

The authors wish to thank the anonymous reviewers and our shepherd, Manuel Costa, for their insightful comments. This work has been supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095, by the European Union though the grant FP7-ICT-216026-WOMBAT, and by Secure Business Austria (SBA).

References

1. Turner, D., Fossi, M., Johnson, E., Mark, T., Blackbird, J., Entwise, S., Low, M.K., McKinney, D., Wueest, C.: Symantec Global Internet Security Threat Report – Trends for July-December 07. Technical Report XII, Symantec Corporation (April 2008)
2. Ofer Shezaf and Jeremiah Grossman and Robert Auger: Web Hacking Incidents Database. `http://whid.xiom.org` (March 2009)
3. Open Security Foundation: DLDOS: Data Loss Database – Open Source. `http://datalossdb.org/` (March 2009)
4. Cho, S., Cha, S.: SAD: web session anomaly detection based on parameter estimation. In: Computers & Security. Volume 23. (2004) 312–319
5. Kruegel, C., Robertson, W., Vigna, G.: A Multi-model Approach to the Detection of Web-based Attacks. Journal of Computer Networks **48**(5) (July 2005) 717–738
6. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.A.: Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2006), San Diego, CA, USA (February 2006)
7. Guangmin, L.: Modeling Unknown Web Attacks in Network Anomaly Detection. In: Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology (ICCIT 2008), Washington, DC, USA, IEEE Computer Society (2008) 112–116
8. Zanero, S., Criscione, C.: Masibty: A Web Application Firewall based on Anomaly Detection. In: DeepSec - In-depth security conference. (November 2008)
9. Citrix Systems, Inc.: Citrix Application Firewall. `http://www.citrix.com/English/PS2/products/product.asp?contentID=25636` (January 2009)
10. F5 Networks, Inc.: BIG-IP Application Security Manager. `http://www.f5.com/products/big-ip/product-modules/application-security-manager.html` (January 2009)
11. Breach Security, Inc.: Breach WebDefend. `http://www.breach.com/products/webdefend.html` (January 2009)
12. Axelsson, S.: The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 1999), New York, NY, USA, ACM (1999) 1–7

13. Frias-Martinez, V., Stolfo, S.J., Keromytis, A.D.: Behavior-Profile Clustering for False Alert Reduction in Anomaly Detection Sensors. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2008), Anaheim, CA, USA (December 2008)
14. Escalante, H.J., Fuentes, O.: Kernel Methods for Anomaly Detection and Noise Elimination. In: Proceedings of the International Conference on Computing (CORE 2006), Mexico City, Mexico 69–80
15. Kim, S.i., Nwanze, N.: Noise-Resistant Payload Anomaly Detection for Network Intrusion Detection Systems. In: Proceedings of the Performance, Computing and Communications Conference (IPCCC 2008), Austin, TX, USA, IEEE Computer Society (December 2008) 517–523
16. Cretu, G.F., Stavrou, A., Locasto, M.E., Stolfo, S.J., Keromytis, A.D.: Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, CA, USA, IEEE Computer Society (May 2008) 81–95
17. Song, Y., Stolfo, S., Keromytis, A.: Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In: Proc of the 16th Annual Network and Distributed System Security Symposium (NDSS). (2009)
18. Schlimmer, J., Granger, R.: Beyond incremental processing: Tracking concept drift. In: Proceedings of the Fifth National Conference on Artificial Intelligence. Volume 1. (1986) 502–507
19. Kolter, J., Maloof, M.: Dynamic weighted majority: An ensemble method for drifting concepts. The Journal of Machine Learning Research **8** (2007) 2755–2790
20. Robert Hansen (RSnake): XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html` (June 2009)
21. Ferruh Mavituna: SQL Injection Cheat Sheet. `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/` (June 2009)
22. Denning, D.E.: An Intrusion-Detection Model. IEEE Transactions on Software Engineering **13**(2) (1987) 222–232
23. Lee, W., Stolfo, S.J.: A Framework for Constructing Features and Models for Intrusion Detection Systems. ACM Transactions on Information and System Security **3**(4) (2000) 227–261
24. Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomaly system call detection. ACM Transactions on Information and System Security **9**(1) (February 2006) 61–93
25. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P 1996), Oakland, CA, USA, IEEE Computer Society (May 1996) 120–128
26. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P 2001), Oakland, CA, USA, IEEE Computer Society (2001) 156–168
27. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing **99**(1) (5555)
28. Wang, K., Stolfo, S.J.: Anomalous Payload-based Network Intrusion Detection. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2004), Springer-Verlag (September 2004)
29. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2006), Hamburg, GR, Springer-Verlag (September 2006)

30. Zanero, S.: Analyzing tcp traffic patterns using self organizing maps. Lecture Notes in Computer Science **3617** (2005) 83
31. Kruegel, C., Mutz, D., Robertson, W., Valeur, F.: Bayesian Event Classification for Intrusion Detection. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV, USA (December 2003)