ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Networks xxx (2005) xxx–xxx

Computer Networks

www.elsevier.com/locate/comnet

A multi-model approach to the detection of web-based attacks

Christopher Kruegel *, Giovanni Vigna, William Robertson

Reliable Software Group, University of California, Santa Barbara, USA

Abstract

Web-based vulnerabilities represent a substantial portion of the security exposures of computer networks. In order to detect known web-based attacks, misuse detection systems are equipped with a large number of signatures. Unfortunately, it is difficult to keep up with the daily disclosure of web-related vulnerabilities, and, in addition, vulnerabilities may be introduced by installation-specific web-based applications. Therefore, misuse detection systems should be complemented with anomaly detection systems.

This paper presents an intrusion detection system that uses a number of different anomaly detection techniques to detect attacks against web servers and web-based applications. The system analyzes client queries that reference server-side programs and creates models for a wide-range of different features of these queries. Examples of such features are access patterns of server-side programs or values of individual parameters in their invocation. In particular, the use of application-specific characterization of the invocation parameters allows the system to perform focused analysis and produce a reduced number of false positives.

The system derives automatically the parameter profiles associated with web applications (e.g., length and structure of parameters) and relationships between queries (e.g., access times and sequences) from the analyzed data. Therefore, it can be deployed in very different application environments without having to perform time-consuming tuning and configuration.

Keywords: Intrusion Detection; World-Wide Web; Machine Learning; Anomaly Models

1. Introduction

Web servers and web-based applications are popular attack targets [20]. Web servers are usually accessible through firewalls, and web-based applications are often developed without following a sound security methodology. Attacks that exploit web servers or server extensions (e.g., programs invoked through the Common Gateway Interface [7] and Active Server Pages [28]) represent a substantial portion of the total number of vulnerabilities. By analyzing the CVE entries from

* Corresponding author.
E-mail addresses: chris@cs.ucsb.edu (C. Kruegel), vigna@cs.ucsb.edu (G. Vigna), wkr@cs.ucsb.edu (W. Robertson).

1389-1286/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2005.01.009

Table 1
Percentage of web-related attacks in the Common Vulnerabilities and Exposure database, per year

Type	Year	Total	Web-related	Percentage
CVE	1999	809	109	13.5
	2000	800	186	23.3
	2001	588	120	20.4
	2002	376	100	26.6
	Total	2573	515	20.0
CAN	1999	742	115	15.5
	2000	403	96	23.8
	2001	806	236	29.3
	2002	1197	385	32.2
	2003	1050	207	18.7
	2004	797	182	23.8
	Total	4995	1221	24.4

The table presents the results for both accepted entries (CVE) and candidate entries (CAN).

1999 to 2004, we identified that web-based attacks account for 20–30% of the attacks (see Table 1). In addition, the large installation base makes both web applications and servers a privileged target for worm programs that exploit web-related vulnerabilities to spread across networks [5].

To detect web-based attacks, intrusion detection systems (IDSs) are configured with a number of signatures that support the detection of known attacks. For example, at the time of writing, Snort 2.2 [37] devotes 1037 of its 2464 signatures to detecting web-related attacks. Unfortunately, it is hard to keep intrusion detection signature sets updated with respect to the large numbers of continuously discovered vulnerabilities. In addition, vulnerabilities may be introduced by custom web-based applications developed in-house. Developing ad hoc signatures to detect attacks against these applications is a time-intensive and error-prone activity that requires substantial security expertise.

To overcome these issues, misuse detection systems should be complemented by anomaly detection systems, which support the detection of new attacks. In addition, anomaly detection systems can be trained to detect attacks against custom-developed web-based applications. Unfortunately, to the best of our knowledge, there are no available anomaly detection systems tailored to detect attacks against web servers and web-based applications.

This paper presents an anomaly detection system that detects web-based attacks using a number of different techniques. The anomaly detection system takes as input web server log files that conform to the Common Log Format (CLF) and produces an anomaly score for each web request. More precisely, the analysis techniques used by the tool take advantage of the particular structure of HTTP queries that contain parameters [11]. The access patterns of such queries and their parameters are compared with established profiles that are specific to the program or active document being referenced. This approach supports a more focused analysis with respect to generic anomaly detection techniques that do not take into account the specific program being invoked.

This paper is structured as follows. Section 2 presents related work on the detection of web-based attacks and on learning-based anomaly detection. Section 3 describes an abstract model for the data analyzed by our intrusion detection system. Section 4 presents the anomaly detection techniques used. Then, Section 5 contains the experimental evaluation of the system with respect to real-world data and discusses the results obtained so far and the limitations of the approach. Finally, Section 6 draws conclusions and outlines future work.

2. Related work

The work presented here is related to three different areas of intrusion detection, namely learning-based anomaly detection, application-level intrusion detection, and, more specifically, detection of attacks against web servers. In the following, we discuss how previous works in these three areas relate to our research.

Anomaly detection relies on models of the intended behavior of users and applications and interprets deviations from this "normal" behavior as evidence of malicious activity [10,19,14,26,12]. A basic assumption underlying anomaly detection is that attack patterns differ from normal behavior. In addition, anomaly detection assumes that this "difference" can be expressed either quantitatively or qualitatively. This approach is complementary with respect to misuse detection, where a number of attack descriptions (usually in the form of signatures) are matched against the stream of audited events, looking for evidence that one of the modeled attacks is occurring [16,33,30]. Mainstream intrusion detection systems, e.g., Snort [37] and ISS's RealSecure [18], are mostly misuse-based even though they incorporate some anomaly-based techniques (e.g., protocol verification components).

The classification of intrusion detection approaches into anomaly-based and misuse-based is orthogonal with respect to the particular technique used to characterize normal or malicious actions. There are cases where "signatures" or "policies" are used to specify the expected behavior of users or applications, and, therefore, the detection of an anomaly is achieved leveraging models that are mostly used in misuse-based intrusion detection systems, such as state-transition systems (see, for example, [39,21,46]). Also, there are approaches where statistical analysis is used to derive a characterization of malicious activity and develop signatures of attacks. Therefore, even though this technique is similar to the ones used in a number of anomaly detection approaches, the resulting system is to be considered misuse-based (e.g., [27]). In the following, we will use the term "learning-based anomaly detection" to denote the class of approaches that relies on training data to build profiles of the normal, benign behavior of users and applications.

Different types of learning-based anomaly detection techniques have been proposed to analyze different data streams. A common approach is to use data-mining techniques to characterize network traffic. For example, in [36] the authors apply clustering techniques to unlabeled network traces to identify intrusion patterns. Statistical techniques have also been used to model the behavior of network worms (see for example [29]). Other approaches use statistical analysis to characterize user behavior. For example, the seminal work by Denning [10] builds user profiles using login times and the actions that users perform.

A particular class of learning-based anomaly detection approaches focuses on the characteristics of specific applications and the protocols they use. For example, in [13] and [48] sequence analysis is applied to system calls produced by specific applications in order to identify "normal" system call sequences for a certain application. These application-specific profiles are then used to identify attacks that produce previously unseen sequences. As another example, in [31] the authors use statistical analysis of network traffic to learn the normal behavior of network-based applications. This is done by analyzing both packet header information (e.g., source/destination ports, packet size) and the contents of application-specific protocols.

Our approach is similar to these techniques because it characterizes the benign, normal use of specific programs, that is, server-side web-based applications. However, our approach is different from these techniques in two ways. First of all, we use a number of different models to characterize the parameters used in the invocation of the server-side programs. By using multiple models it is possible to reduce the vulnerability of the detection process with respect to mimicry attacks [47,43]. Second, the models target specific types of applications, and, therefore, they allow for more focused analysis of the data transferred between the client (the attacker) and the server-side program (the victim). This is an advantage of application-specific intrusion detection in general [24] and of web-based intrusion detection in particular [25].

The detection of web-based attacks has recently received considerable attention because of the increasingly critical role that web-based services are playing. For example, in [1] the authors present a system that analyzes web logs looking for patterns of known attacks. A different type of analysis is performed in [2] where the detection process is integrated with the web server application itself. In [45], a misuse-based system that operates on multiple event streams (i.e., network traffic, system call logs, and web server logs) was proposed. Systems that focus on web-based attacks show that by taking advantage of the specificity of a particular application domain it is possible to achieve better detection results. However, these systems are mostly misuse-based and therefore suffer from the problem of not being able to detect attacks that have not been previously modeled. Our approach is similar to these systems because it focuses on specific applications. However, the goal of our system is to perform unsupervised, learning-based anomaly detection. The system can be deployed on a host that contains custom-developed server-side programs and it is able to automatically derive models of the normal invocation of these programs. These models are then used to detect known and unknown attacks.

3. Data model

Our anomaly detection approach analyzes HTTP requests as logged by most common web servers (for example, Apache [3]). The analysis focuses on requests that use parameters to pass values to server-side programs or active documents. More formally, the input to the detection process consists of an ordered set $U = \{u_1, u_2, \ldots, u_m\}$ of URIs extracted from successful `GET` requests (requests whose return code is greater or equal to 200 and less than 300).

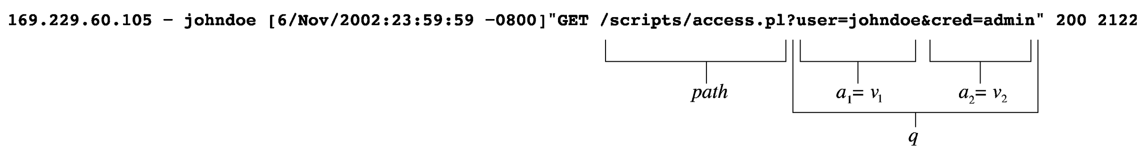
A URI u_i can be expressed as the composition of the path to the desired resource ($path_i$), an optional path information component ($pinfo_i$), and an optional query string (q). The query string is used to pass parameters to the referenced resource and it is identified by a leading "?" character. A query string consists of an ordered list of n pairs of parameters (or attributes) with their corresponding values. That is, $q = (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)$ where $a_i \in A$, the set of all attributes, and v_i is a string. The set S_q is defined as the subset $\{a_j, \ldots, a_k\}$ of attributes of query q. Fig. 1 shows an example of an entry from a web server log and the corresponding elements that are used in the analysis. For this example query q, $S_q = \{a_1, a_2\}$.

The analysis process focuses on the association between programs, parameters, and their values. URIs that do not contain a query string are irrelevant, and, therefore, they are removed from U. In addition, the set of URIs U is partitioned into subsets U_r according to the resource path. Therefore, each referenced program r is assigned a set of corresponding queries U_r. The anomaly detection algorithms are independently run on each set of queries U_r. This means that the modeling and detection processes are performed separately for each program r.

In the following text, the term "request" refers only to requests with queries. Also, the terms "parameter" and "attribute" of a query are used interchangeably.

4. Detection models

The anomaly detection process uses a number of different *models* to identify anomalous entries within a set of input requests U_r associated with a program r. A model is a set of procedures used to evaluate a certain feature of a query. Such a feature can be related to a single query attribute (e.g., the string length of a particular attribute value), to all query attributes (e.g., the order of attributes in a query), or to some relationship between a query and others in U_r (e.g., time delay between consecutive queries). Each model is associated with a query (or one of its attributes) by means of a *profile*. Consider, for example, the attribute length model, which analyzes a feature of a particular query attribute (in this case, the string lengths of this attribute). The attribute length model could



```
169.229.60.105 - johndoe [6/Nov/2002:23:59:59 -0800]"GET /scripts/access.pl?user=johndoe&cred=admin" 200 2122
```

Fig. 1. Sample web server access log entry.

be associated with the *username* attribute of a *login* program. In this case, the profile for the attribute length model captures the "normal" string length of the user name attribute of the login program.

The task of a model is to assign a probability value to either a query as a whole or one of the query's attributes. This probability value reflects the probability of the occurrence of the given feature value with regards to an established profile. The assumption is that feature values with a sufficiently low probability (i.e., abnormal values) indicate a potential attack.

Based on the model outputs, a query is either reported as a potential attack or as normal. This decision is reached by calculating a number of anomaly scores: one for the query itself, and one for each attribute. A query is reported as anomalous if at least one of these anomaly scores is above the corresponding detection threshold (detection thresholds are established during a training phase, as described below). Although alerting on a single anomalous attribute seems excessively cautious, it is necessary to prevent attackers from hiding a single malicious attribute in a query with many "normal" attributes.

The anomaly score for a query (or one its attributes) is derived from the probability values returned by the models that are associated with the query (or attribute). The anomaly score value is calculated using a weighted sum as shown in Eq. (1). In this equation, w_m represents the weight associated with model m, while p_m is its returned probability value. The probability p_m is subtracted from 1 because a value close to zero indicates an anomalous event that should yield a high anomaly score. In our experiments, only weights of 1 were used, meaning that all models were considered equal

$$\text{Anomaly Score} = \sum_{m \in \text{Models}} w_m * (1 - p_m). \tag{1}$$

A model can operate in one of two modes, training or detection. The training phase is required to determine the characteristics of normal events and to establish anomaly score thresholds to distinguish between regular and anomalous inputs. This phase is divided into two steps. During the first step, the system creates profiles for each server-side program and each of its attributes. During the second step, suitable thresholds are established. This is done by evaluating queries and their attributes using previously created profiles.

For each program and its attributes, the highest anomaly score is stored and the threshold is set to a value that is a certain, adjustable percentage higher than this maximum. By modifying this value, the user can perform a trade-off between the number of false positives and the expected detection accuracy. Because we assume that only normal queries are processed during the training phase, a threshold that is at least as large as the highest anomaly score is selected. In addition, to add a small safety margin, a somewhat higher value is recommended. As a default safety margin (also used in the experiments), we set a threshold that is 10% larger then the maximum anomaly score seen during training. Although different thresholds might yield better results for particular data sets, we wanted to evaluate our system with a reasonable default setting. In addition, the number of queries and attributes that are utilized to learn the profiles and the thresholds is determined by another adjustable parameter (with a default setting of 1000).

Once the models have learned the characteristics of normal events and suitable thresholds have been derived system switches to detection mode. In this mode, anomaly scores are calculated and anomalous queries are reported.

The following sections describe the algorithms that analyze the features that are considered relevant for detecting malicious activity. For each algorithm, the learning and training phases are discussed.

4.1. Attribute length

The length of a query attribute can be used to detect anomalous requests, especially when parameters are either fixed-size tokens (such as session identifiers) or short strings derived from human input (such as fields in an HTML form). In these cases, the length of the parameter values does not vary much between requests associated with a certain program. The situation may look different when malicious input is passed to the program. For example, to overflow a buffer in a target application, it is necessary to ship the shell code and additional padding, depending on the length of the target buffer. Therefore, the attribute may contain up to several hundred bytes. As another example, cross-site scripting attacks, which attempt to include scripts in pages whose content is determined by user-supplied data, may require an amount of data to be sent that can significantly exceed the length of legitimate parameters.

The goal of this model is to approximate the actual but unknown distribution of the parameter lengths and detect instances that significantly deviate from the observed normal behavior. Clearly, we cannot expect that the probability density function of the underlying real distribution will follow a smooth curve. We also have to assume that the distribution has a large variance. Nevertheless, the model should be able to efficiently identify significant deviations.

4.1.1. Learning

We approximate the mean $\dot{\mu}$ and the variance $\dot{\sigma}^2$ of the real attribute length distribution by calculating the sample mean μ and the sample variance σ^2 for the lengths $l_1, l_2, \ldots, l_n$ of the parameters processed during the learning phase (assuming that n queries with this attribute were processed). The cost of building this model is low. It is proportional to the number of queries n analyzed during the training phase as the mean and variance of the corresponding string lengths need to be calculated.

4.1.2. Detection

Given the estimated query attribute length distribution with parameters μ and σ^2 as determined by the previous learning phase, it is the task of the detection phase to assess the anomaly of a parameter with length l.

To assess the anomaly of a string with length l, we calculate the "distance" of the length l from the mean value μ of the length distribution. This distance is expressed with the help of the Chebyshev inequality [15], shown in Eq .(2). The Chebyshev inequality puts, for an arbitrary distribution with variance σ^2 and mean μ, an upper bound on the probability that the difference between the value of a random variable x and μ exceeds a certain threshold t

$$p(|x - \mu| > t) < \frac{\sigma^2}{t^2}. \tag{2}$$

When l is far away from μ, considering the variance of the length distribution, then the probability of any (legitimate) string x having a greater length than l should be small. Thus, to obtain a quantitative measure of the distance between a string of length l and the mean μ of the length distribution, we substitute t with the difference between μ and l.

Only strings with lengths that exceed μ are assumed to be malicious. This is reflected in our probability calculation as only the upper bound for strings that are longer than the mean is relevant. Note that an attacker cannot disguise malicious input by padding the string and thus increasing its length, because an increase in length can only reduce the probability value.

The probability value $p(l)$ for a string with length l, given that $l \geqslant \mu$, is calculated as shown below. For strings shorter than μ, $p(l) = 1$

$$p(|x - \mu| > |l - \mu|) < p(l) = \frac{\sigma^2}{(l - \mu)^2}. \tag{3}$$

We chose the Chebyshev inequality as an efficient metric to model decreasing probabilities for strings with lengths that increasingly exceed the mean. This technique is efficient because it is only necessary to determine the length of the input string followed by a simple computation.

In contrast to schemes that define a valid interval (e.g., by recording all strings encountered during the training phase), the Chebyshev inequality takes the variance of the data into account and provides the advantage of gradually changing probability values (instead of providing a simple "yes/no" answer). In general, the bound computed by the Chebyshev inequality is weak. Applied to our model, this weak bound results in a high degree of tolerance to deviations of attribute lengths given an empirical mean and variance. Although such a property is undesirable in many situations, in this case it is useful to flag only significant outliers.

4.2. Attribute character distribution

The attribute character distribution model captures the concept of a "normal" or "regular" query parameter by looking at its character distribution. The approach is based on the observation that attributes have a regular structure, are mostly human-readable, and almost always contain only printable characters. In case of attacks that send binary data (e.g., buffer overflow attacks), a completely different character distribution can be observed. This is also true for attacks that send many repetitions of a single character (e.g., a dot character in directory traversal exploits).

A large percentage of characters in regular attributes are drawn from a small subset of the 256 possible 8-bit values (mainly from letters, numbers, and a few special characters). As in English text, the characters are not uniformly distributed, but occur with different frequencies. Obviously, it cannot be expected that the frequency distribution is identical to a standard English text. Even the frequency of a certain character (e.g., the frequency of the letter "`e`") varies considerably between different attributes. Nevertheless, there are similarities between the character frequencies of query parameters.

This becomes apparent when the relative frequencies for all possible 256 characters in a string are *sorted in descending order*. In the following, the sorted, relative character frequencies of an attribute are called its *character distribution*. The character distribution of an attribute that is perfectly normal (i.e., non-anomalous) is called the attribute's *idealized character distribution* (*ICD*).

By sorting the relative frequency values, the connection between individual characters and their relative frequencies are lost. That is, it does not matter whether the character with the most occurrences is an "`a`" or a "`/`". For example, consider the parameter string "`passwd`" with the corresponding ASCII values of "112 97 115 115 119 100". The absolute frequency distribution is 2 for 115 and 1 for the four others. When these absolute counts are transformed into sorted, relative frequencies, the resulting values are 0.33, 0.17, 0.17, 0.17, 0.17 followed by 0 occurring 251 times. The same sorted, relative frequencies would be obtained from the string "`aabcde`".

For an attribute of a legitimate query, one can expect that the relative frequencies slowly decrease in value. In human-readable tokens, similar to English text, there are certain characters that appear more often than others, but there is no one that is clearly more prevalent than others. Moreover, when random identifiers are used, no character appears significantly more often then others, resulting in a nearly uniform character distribution. In case of malicious input, however, the frequencies often drop extremely fast. This can be the result of a certain padding character that is repeated many times in a buffer overflow attack or because of the many occurrences of the dot character in a directory traversal attempt.

4.2.1. Learning

The idealized character distribution is determined during the training phase. For each observed query attribute, its character distribution is stored. The idealized character distribution is then approximated by calculating the average of all stored character distributions. This is done by setting $ICD(n)$ to the mean of the nth entry of the stored character distributions $\forall n$: $0 \leqslant n \leqslant 255$. Because all individual character distributions sum up to unity, their average will do so as well, and the idealized character distribution is well-defined.

This calculation is efficient and linear in the number of attributes that are analyzed during the training phase. For each attribute, the character distribution has to be determined, an operation which has a cost that is proportional to the length of the attribute string, incurred by sorting the characters in the string.

4.2.2. Detection

Given an idealized character distribution *ICD*, the task of the detection phase is to determine the probability that the character distribution of a query attribute is an actual sample drawn from its *ICD*. This probability, or more precisely, the confidence in the hypothesis that the character distribution is a sample from the idealized character distribution, is calculated by a statistical test. The detection algorithm is based on a variant of the Pearson χ^2-test as the "goodness-of-fit" test [4].

The χ^2-test requires that the function domain is divided into a small number of intervals, or bins, and it is preferable that all bins contain at least "some" elements (the literature considers five elements to be sufficient for most cases). Although the test is sensitive to the choice of the bins, most reasonable choices of bins produce similar (though not identical) results [40]. As a reasonable choice of bins, we have selected the six bins for the domain of *ICD* as follows: $\{[0],[1,3],[4,6],[7,11],[12,15],[16,255]\}$. Although the choice of these six bins is somewhat arbitrary, it reflects the fact that the relative frequencies are sorted in descending order. Therefore, the values of $ICD(x)$ are higher when x is small, and thus all bins contain some elements with a high probability.

When a new query attribute is analyzed, the number of occurrences of each character in the string is determined. Afterward, the values are sorted in descending order and combined by aggregating values that belong to the same bin. The χ^2-test is then used to calculate the probability that the given sample has been drawn from the idealized character distribution as follows:

1. *Calculate the observed and expected frequencies*. The observed values O_i (one for each bin) are already given. The expected number of occurrences E_i are calculated by multiplying the relative frequencies of each of the six bins as determined by the *ICD* times the length of the attribute (i.e., the length of the string).
2. *Compute the χ^2-value* as $\chi^2 = \sum_{i=0}^{i<6}(O_i - E_i)^2/E_i$—note that i ranges over all six bins.
3. *Determine the degrees of freedom and obtain the significance*. The degrees of freedom for the χ^2-test is five in our case (the number of bins minus 1). The actual probability p that the sample is derived from the idealized character distribution is read from a predefined table using the χ^2-value as index.

The derived value p is used as the return value for this model. When the probability that the sample is drawn from the idealized character distribution increases, p increases as well. The cost of checking an input string using this model is the calculation of the χ^2-value. This requires that the character distribution for the input string is calculated (with a cost that is linear in the number of characters of the string), followed by a constant cost for the χ^2-test (namely, a few computations and a table lookup).

4.3. Structural inference

Often, the manifestation of an exploit is immediately visible in query attributes as unusually long parameters or parameters that contain repetitions of non-printable characters. Such anomalies are easily identifiable by the two models explained before.

There are situations, however, when an attacker is able to craft her attack in a manner that makes its manifestation appear more regular. For example, non-printable characters can be replaced by groups of printable characters. In such situations, we need a more detailed model of the query attribute that contains the evidence of the attack. This model can be acquired by analyzing the parameter's structure. For our purposes, the structure of a parameter is the regular grammar that describes all of its legitimate values.

The structural inference model is useful for detecting all kinds of attacks that require the parameter string to have a different structure than regular query arguments. This includes buffer overflow, directory traversal, and cross-site scripting attacks. Depending on the structure of regular arguments, it can also identify attacks that exploit errors in the program logic.

4.3.1. Learning

When structural inference is applied to a query attribute, the resulting grammar must be able to produce at least all training examples. Unfortunately, there is no unique grammar that can be derived from a set of input elements. When no negative examples are given (i.e., elements that should not be derivable from the grammar), it is always possible to create either a grammar that contains exactly the training data or a grammar that allows production of arbitrary strings.

The basic approach used for our structural inference is to generalize the grammar as long as it seems to be "reasonable" and stop before too much structural information is lost. The notion of "reasonable generalization" is specified with the help of Markov models and Bayesian probability.

In a first step, we consider the set of training queries as the output of a *probabilistic* grammar. A probabilistic grammar is a grammar that assigns probabilities to each of its productions. This means that some words are more likely to be produced than others, which fits well with the evidence gathered from query parameters. Some values appear more often, and this is important information that should not be lost in the modeling step.

A probabilistic regular grammar can be transformed into a non-deterministic finite automaton (NFA). Each state S of the automaton has a set of n_S possible output symbols o which are emitted with a probability of $p_S(o)$. Each transition t is marked with a probability $p(t)$ that characterizes the likelihood that the transition is taken. An automaton that has probabilities associated with its symbol emissions and its transitions can also be considered a Markov model.

The output of the Markov model consists of all paths from its start state to its terminal state. A probability value can be assigned to each output word w (that is, a sequence of output symbols $o_1, o_2, \ldots, o_k$). This probability value (as shown in Eq. (4)) is calculated as the sum of the probabilities of all distinct paths through the automaton that produce w. The probability of a single path is the product of the probabilities of the emitted symbols $p_{S_i}(o_i)$ and the taken transitions $p(t_i)$. The probabilities of all possible output words w sum up to 1

$$p(w) = p(o_1, o_2, \ldots, o_k) = \sum_{(paths\ p\ for\ w)} \prod_{(states \in p)} p_{S_i}(o_i) * p(t_i). \tag{4}$$

For example, consider the NFA in Fig. 2. To calculate the probability of the word "ab", one has to sum the probabilities of the two possible paths (one that follows the left arrow and one that follows the right one). The start state emits no symbol and has a probability of 1. Following Eq. (4), the result is

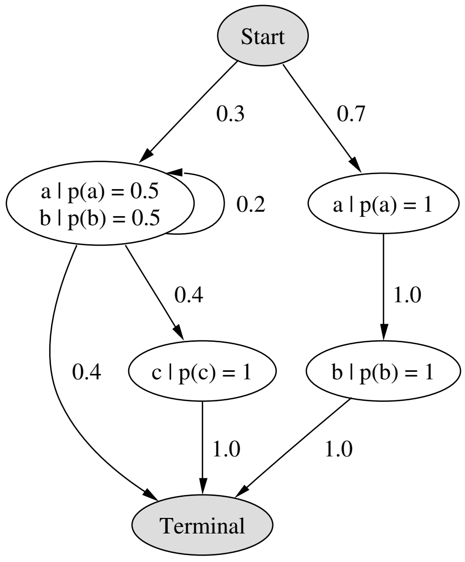


Fig. 2. Markov model example.

$$\begin{aligned} p(w) &= (1.0 * 0.3 * 0.5 * 0.2 * 0.5 * 0.4) \\ &\quad + (1.0 * 0.7 * 1.0 * 1.0 * 1.0 * 1.0) \\ &= 0.706. \end{aligned} \tag{5}$$

The goal of the structural inference process is to find an NFA that has the highest likelihood for the given training elements. An excellent technique to derive a Markov model from empirical data is explained in [41]. It uses the Bayesian theorem to state this goal as:

$$p(Model|TrainingData) = \frac{p(TrainingData|Model) * p(Model)}{p(TrainingData)}. \tag{6}$$

The probability of the training data is considered a scaling factor in Eq. (6) and it is subsequently ignored. As we are interested in maximizing the a posteriori probability (i.e., the left-hand side of the equation), we have to maximize the product shown in the enumerator on the right-hand side of the equation. The first term—the probability of the training data given the model—can be calculated for a certain automaton by adding the probabilities calculated for each input training element as discussed above. The second term—the prior probability of the model—is not as straightforward. It has to reflect the fact that, in general, smaller models are preferred. The model probability is calculated heuristically and takes into account the total number of states N as well as the number of transitions $\sum_S trans$ and emissions $\sum_S emit$ at each state S. This is justified by the fact that smaller models can be described with less states as well as fewer emissions and transitions. The actual value is derived as shown in Eq. (7)

$$p(Model) \alpha \frac{1}{\prod_{S \in States} (N+1)^{\sum_S trans} * (N+1)^{\sum_S emit}}. \tag{7}$$

The term that is maximized—the product of the probability of the model given the data, times the prior probability of the model itself—reflects the intuitive idea that there is a conflict between simple models that tend to over-generalize and models that perfectly fit the data but are too complex. By maximizing the product, the Bayesian model induction approach creates automations that generalize enough to reflect the general structure of the input without discarding too much information.

The model building process starts with an automaton that exactly reflects the input data and then gradually merges states. This state merging is continued until the a posteriori probability no longer increases. The cost of building the structural inference model in this straightforward fashion is $O((n * l)^3)$, where n is the number of input strings analyzed during the training phase and l is their maximum length. This cost is clearly prohibitive. The reason is that the Markov model could contain up to $(n * l)$ states, resulting in $(n * l)(n * l - 1)/2$ comparisons for each merging step in which every state needs to be compared with every other state. Because it is possible that the merging continues until only one state is left, up to $(n * l - 1)$ merging steps might required.

There are a number of optimizations such as the Viterbi path approximation and the path prefix compression that can be applied to make that process effective [41,42]. Using these optimizations, and by introducing intermediate state merging steps, the learning cost can be reduced to $O(n * l^2)$, which is acceptable even for large n and reasonable values of l.

4.3.2. Detection

Once the Markov model has been built, it can be used by the detection phase to determine the probability of query attributes. The probability of an attribute is calculated in a way similar to the likelihood of a training item as shown in Eq. (4). The problem is that even legitimate input that has been regularly seen during the training phase may receive a very small probability value, because the probability values of all possible input words sum up to 1. Therefore, we chose to have the model return a probability value of 1 if the word is a valid output from the Markov model and a value of 0 when the value cannot be derived from the given grammar. The cost of checking whether the input string can be derived from the Markov model is linear in the length of the string.

4.4. Token finder

The purpose of the token finder model is to determine whether the values of a certain query attribute are drawn from a limited set of possible alternatives (i.e., they are tokens of an enumeration). Web applications often require one out of a few possible values for certain query attributes, such as flags or indices. When a malicious user attempts to use these attributes to pass illegal values to the application, the attack can be detected. For arguments that can only take a limited number of legitimate values, the token finder model can be used to detect any attack that requires a different value to be passed. When no enumeration can be identified, it is assumed that the attribute values are random and no attacks can be detected by this model.

4.4.1. Learning

The classification of an argument as enumeration or as random value is based on the observation that the number of different occurrences of parameter values is bound by some unknown threshold t in the case of an enumeration while it is unrestricted in the case of random values.

When the number of different argument instances grows proportional to the total number of argument instances, the use of random values is indicated. If such an increase cannot be observed, we assume an enumeration. More formally, to decide if argument a is an enumeration, we calculate the statistical correlation ρ between the values of the functions f and g for increasing numbers $1, \ldots, i$ of occurrences of a. The functions f and g are defined as follows on $\mathcal{N}_0$:

$$f(x) = x, \tag{8}$$

$$g(x) = \begin{cases} g(x-1)+1, & \text{if the } x\text{th value for } a \text{ is new,} \\ g(x-1)-1, & \text{if the } x\text{th value was seen before,} \\ 0, & \text{if } x = 0. \end{cases} \tag{9}$$

The correlation parameter ρ is derived after the training data has been processed. It is calculated from f and g with their respective variances Var(f), Var(g) and the covariance Covar(f,g) as shown below

$$\rho = \frac{\mathrm{Covar}(f,g)}{\sqrt{\mathrm{Var}(f) * \mathrm{Var}(g)}}. \tag{10}$$

If ρ is less than 0, then f and g are negatively correlated and an enumeration is assumed. This is motivated by the fact that, in this case, increasing function values of f (reflecting the increasing number of analyzed parameters) correlate with decreasing values of $g(x)$ (reflecting the fact that many argument values for a have previously occurred). In the opposite case, where ρ is greater than 0, the values of a have shown sufficient variation to support the hypothesis that they are not drawn from a small set of predefined tokens. When an enumeration is assumed, the complete set of identifiers is stored for use in the detection phase.

The cost of building the token finder model is the calculation of the covariance between two simple functions. This cost depends on the number of queries analyzed.

4.4.2. Detection

Once it has been determined that the values of a query attribute are tokens drawn from an enumeration, any new value is expected to appear in the set of known values. When this happens, 1 is returned. If the value is not in the established set of values, 0 is returned. If it has been determined that the parameter values are random, the model always returns 1. The detection is efficiently performed by this model using a hash table lookup.

4.5. Attribute presence or absence

Most of the time, server-side programs are not directly invoked by users typing the input parameters into the URIs themselves. Instead, client-side programs, scripts, or HTML forms pre-process the data and transform it into a suitable request. This processing step usually results in a high regularity in the number, name, and order of parameters. Empirical evidence shows that hand-crafted attacks focus on exploiting a vulnerability in the code that processes a certain parameter value, and little attention is paid to the order or completeness of the parameters.

The analysis performed by this model takes advantage of this fact and detects requests that deviate from the way parameters are presented by legitimate client-side scripts or programs. This type of anomaly is detected using two different algorithms. The first one, described in this section, deals with the presence and absence of attributes a_i in a query q. The second one is based on the relative order of parameters and is further discussed in Section 4.6. These models focus on two features that consider all query attributes simultaneously. This is different from the four previous ones, which focus on features of individual query arguments.

The algorithm discussed hereinafter assumes that the absence or abnormal presence of one or more parameters in a query might indicate malicious behavior. In particular, if an argument needed by a server-side program is missing, or if mutually exclusive arguments appear together, then the request is considered anomalous. This allows for the detection of attacks where server-side applications are probed or exploited by sending incomplete or malformed requests.

4.5.1. Learning

The test for presence and absence of parameters creates a model of acceptable subsets of attributes that appear simultaneously in a query. This is done by recording each distinct subset $S_q = \{a_i, \ldots, a_k\}$ of attributes that is seen during the training phase. The process is efficient as at most one value needs to be inserted into a hash table for each query encountered during the learning phase.

4.5.2. Detection

During the detection phase, the algorithm performs for each query a lookup of the current attribute set. When the set of parameters has been encountered during the training phase, 1 is returned, otherwise 0. This is done using a hash table lookup.

4.6. Attribute order

As discussed in the previous section, legitimate invocations of server-side programs often contain the same parameters in the same order. Program logic is usually sequential, and, therefore, the relative order of attributes is preserved even when parameters are omitted in certain queries. This is not the case for hand-crafted requests, as the order chosen by a human can be arbitrary and has no influence on the execution of the program.

The test for parameter order in a query determines whether the given order of attributes is consistent with the model deduced during the learning phase.

4.6.1. Learning

The order constraints between all k attributes (a_i: $\forall i = 1, \ldots, k$) of a query are determined during the training phase. An attribute a_s of a program *precedes* another attribute a_t when a_s and a_t appear together in the parameter list of at least one query and a_s comes before a_t in the ordered list of attributes of all queries where they appear together.

This definition allows one to introduce the order constraints as a set of attribute pairs O such that:

$$O = \{(a_i, a_j) : a_i \text{ precedes } a_j \text{ and } a_i, a_j \in (S_{q_j} : \forall j = 1, \ldots, n)\}. \qquad (11)$$

The set of attribute pairs O is determined as follows. Consider a directed graph G that has a number of vertices equal to the number of distinct attributes. Each vertex v_i in G is associated with the corresponding attribute a_i. For every query q_j, with $j = 1, \ldots, n$, that is analyzed during the training period, the ordered list of its attributes $a_1, a_2, \ldots, a_i$ is processed. For each attribute pair (a_s, a_t) in this list, with $s \neq t$ and $1 \leqslant s, t \leqslant i$, a directed edge is inserted into the graph from v_s to v_t.

At the end of the learning process, the graph G contains all order constraints imposed by queries in the training data. The order dependencies between two attributes are represented either by a direct edge connecting their corresponding vertices, or by a path over a series of directed edges. At this point, however, the graph could potentially contain cycles as a result of precedence relationships between attributes derived from different queries. As such relationships are impossible, they have to be removed before the final order constraints can be determined. This is done with the help of Tarjan's algorithm [44] which identifies all strongly connected components (SCCs) of a graph. For each component, all edges connecting vertices of the same SCC are removed. The resulting graph is acyclic and can be utilized to determine the set of attribute pairs O which are in a "precedes" relationship. This is obtained by enumerating for each vertex v_i all its reachable nodes $v_g, \ldots, v_h$ in G, and adding the pairs $(a_i, a_g) \cdots (a_i, a_h)$ to O.

The cost for running Tarjan's algorithm is $O(v + e)$, where v is bound by the number of different parameters that a program can take. Because this value is usually very small, the model can be built efficiently.

4.6.2. Detection

The detection process checks whether the attributes of a query satisfy the order constraints determined during the learning phase. Given a query with attributes $a_1, a_2, \ldots, a_i$ and the set of order constraints O, all the parameter pairs (a_j, a_k) with $j \neq k$ and $1 \leqslant j, k \leqslant i$ are analyzed to detect potential violations. A violation occurs when for any single pair (a_j, a_k), the corresponding pair with swapped elements (a_k, a_j) is an element of O. In such a case, the algorithm returns an anomaly score of 0, otherwise it returns 1. The detection cost for this model are i hash table lookups, where

i is the number of parameters of the analyzed query.

4.7. Access frequency

Different server-side applications are invoked with different frequencies, but the general access patterns for these applications remain relatively constant for a certain web site. The goal of the access frequency model is to capture these access patterns. This model and the following two are different from the previous models with respect to the analyzed features. While the previous models concentrate on individual queries and their attributes, this and the next two models focus on patterns of whole sequences of queries.

One feature of an access sequence is the frequency of program invocations. We distinguish between two types of access frequencies for each application. One is the frequency of the application being accessed from a certain client (based on the IP address), the other is the total frequency of all accesses. Consider the example of a web site with two scripts, one to authenticate users before they can access restricted parts and one to allow people to search the content of the pages. In this case, one would expect that the authentication script is called very infrequently for each individual client, because it is only necessary to login once. However, when the web site is accessed by many clients, the total access frequency for this login script is high. The search script, on the other hand, is accessed in bursts by an individual user who is looking for some information, and thus, the access frequency is high for this particular client. But most users do not use the search functionality because they know what to look for, and therefore, the total access frequency for the search script is low.

Changes in access patterns can indicate attacks. When an application is usually accessed infrequently but is suddenly exposed to a burst of invocations, this increase could be the result of an attacker probing for vulnerabilities or the result of an exploit that has to guess parameter values. A single determined attacker can evade detection by executing his actions slowly enough to keep the frequency low. However, most tools used by less skilled intruders execute brute force attacks without stealthiness in mind. Also, when the knowledge of a vulnerability becomes more widespread, many attackers independently attempt to exploit the vulnerability and raise the total frequency to a suspicious level.

4.7.1. Learning

To determine the expected normal access frequencies, the time period between the first and the last query in the training data is divided into consecutive time intervals of a fixed size (10 s in our implementation). Then, the total number of requests and the numbers of requests from distinct clients (or, more precisely, distinct IP addresses) are counted in each of these intervals. The counts for the total accesses and the counts for the accesses from distinct clients form two distributions with their respective means and variances. These two distributions are used during the detection phase, similarly to the attribute length model. The cost of building this model is proportional to the number of requests that are analyzed during the training period.

4.7.2. Detection

During detection, time is divided into intervals of the same fixed size that was used during the learning phase. When a query is evaluated, the number of total requests n_1 and the number of requests from this client n_2, both in the current time interval, are determined. Similar to the detection phase of the attribute length model (see Section 4.1), the Chebyshev inequality is used to calculate the probability of n_1, given the distribution of the total access frequencies, and the probability of n_2, given the access frequencies from distinct clients. The two probabilities are then added, the sum is divided by two and returned.

The detection cost is proportional to the number of requests during the current detection interval, as it is necessary to obtain the values n_1 and n_2 before the Chebyshev inequality can be applied.

4.8. Inter-request time delay

Abnormal client access patterns are often a sign that web servers or web applications are the target of surveillance or attack by malicious clients. One instance of such an access pattern is a set of requests that exhibit a regular delay between each successive request. This type of pattern is typically indicative of scripted probes or attacks against a web service, while legitimate user web browsing generally exhibits a wide variance in inter-request time delays. This model attempts to detect deviations from an expected distribution of time delays for each protected web application to detect scripted probes and attacks.

4.8.1. Learning

During the training phase, a distribution of "normal" time delays between successive client requests is created. To obtain this normal time delay distribution, the inter-request time delays are first stored for each different client. Then, delay values are put into small bins and the distributions for all different clients are combined into a single distribution by taking the average of the numbers in each bin. This is done similarly to the character distribution model (Section 4.2), where the individual character distributions are combined into the idealized character distribution. The cost of this model is, for each different client, linear in the number of requests received from this client during the learning phase.

4.8.2. Detection

During the detection phase, a distribution of time delays between successive requests from each client is compiled. The goal is then to determine the probability that the observed time delays between successive requests is a sample from the learned distribution. This is calculated using the Pearson χ^2-test, similarly to the character distribution model.

The anomaly score that is returned by this model depends on two factors: the likelihood that an observed distribution is a sample from the learned expected distribution as described above, and additionally the number of requests which have been monitored from a specific client for that application. The first consideration determines how anomalous the time delay distribution appears, while the second determines how much confidence should be placed in this anomaly score. This is important because the quality of the score produced depends on the number of requests that have been observed for a particular combination of client and application. Specifically, a small number of requests could produce a distribution that is considered anomalous; however, because of the small number of requests, one cannot conclude with a high degree of confidence that this access pattern is truly anomalous. Consequently, the anomaly score output from this model is scaled by a constant factor that increases with the number of observed requests. This scaling factor eventually reaches 1, and is thus effectively dropped, once the observed distribution has reached a size which is considered suitable for placing sufficient confidence in the results of this model.

The detection cost is the computation needed for the χ^2-test, which is proportional to the number of requests during the detection interval.

4.9. Invocation order

Web-based applications are often composed of a set of server-side programs, which, together, implement the application functionality. For example, a shopping application may have a login program to authenticate a user, a program to access a catalog, a program to add items to a virtual cart, and a program to perform checkout/payment. The nature of a web-based application may impose a well-defined ordering over the invocation of its component programs. For example, a user has to first login before being able to perform any other transaction.

This model captures the order of invocation of web-based applications for a certain client (again, based on the IP address). By characterizing the order of invocation per client, the model attempts to infer the regularity of the structure for a session, which may be associated with specific credentials (e.g., a cookie). The invocation order model is related to the structural inference model (see Section 4.3). The difference is that this model analyzes the structure of a sequence of queries instead of the syntax of the parameters of a simple query.

The model can be used to detect attacks against the application logic. This includes, for example, situations where an attacker attempts to bypass a login check and access a privileged program directly.

4.9.1. Learning

During the learning phase, the program invocations are grouped according to the source IP of the query (note that this is different from all other models, where the characterization was performed independently for each server-side application). This grouping identifies *sessions* composed of a list of program invocations. More precisely, a session S is defined as the series of resource paths $\langle \mathrm{path}_1, \mathrm{path}_2, \ldots, \mathrm{path}_n \rangle$ associated with the corresponding server-side program invocations. The invocations that are part of a session are determined by a constraint on inter-arrival time (that is, invocations that are close in time are aggregated into a session).

The model building process starts with a non-deterministic automaton that outputs exactly the "strings" that represent the sessions S_i with their corresponding sequences of program invocations encountered during the training phase. Similar to the process described in Section 4.3, the process continues by gradually merging the states of this automaton until its a posteriori probability (given the training data) reaches a maximum. This final automaton then represents all invocation sequences that are considered legal.

The learning cost of this model is similar to the cost of the structural inference model (refer to Section 4.3.1). In this case, however, the input are sequences of program accesses instead of sequences of characters.

4.9.2. Detection

During the detection phase, a query q is associated with its corresponding session S. When this session S can be derived from the automaton that was built during the training phase, the output of the model for q is 1. When S cannot be derived, the result for q (and all subsequent queries associated with this session) is 0. The detection cost of checking whether an access sequence can be produced by the Markov model is proportional to the length of the input sequence.

5. Evaluation

This section discusses our approach to validate the proposed models and to evaluate the detection effectiveness of our system. That is, we assess the capability of the models to accurately capture the properties of the analyzed attributes and their ability to detect potentially malicious deviations.

The evaluation was performed using three data sets. These data sets were Apache log files from a production web server at Google, Inc. and from two Computer Science Department web servers located at the University of California, Santa Barbara (UCSB) and the Technical University, Vienna (TU Vienna).

We had full access to the log files of the two universities. However, the access to the log file from Google was restricted because of privacy issues. To obtain results for this data set, our tool was run on our behalf locally at Google and the results were mailed to us.

Table 2 provides information about important properties of the data sets. The table shows the time interval during which the data was recorded, the log file size and the total number of HTTP queries in the log file. In addition, it lists the number of programs for which detection was performed (i.e., those programs that were invoked more than 1000 times so that the training phase could be completed) and the number of analyzed queries and attributes.

5.1. Model validation

This section shows the validity of the claim that our proposed models are able to accurately describe properties of query attributes. For this purpose, our detection tool was run on the three data sets to determine the distribution of the probability values for the different models. The length of the training phase was set to 1000 for this and all following experiments. This means that our system used the first thousand queries that invoked a certain server-side program to establish its profiles and to determine suitable detection thresholds.

Figs. 3 and 4 show a distribution of the probability values that have been assigned to the query attributes by the length and the character

Table 2
Data set properties

Data set	Time interval	Size (MByte)	HTTP queries	Programs	Program requests	Attributes
Google	1 h	236	640,506	3	490,704	1,611,254
UCSB	297 days	1001	9,951,174	2	4617	7993
TU Vienna	80 days	251	2,061,396	8	713,500	765,399

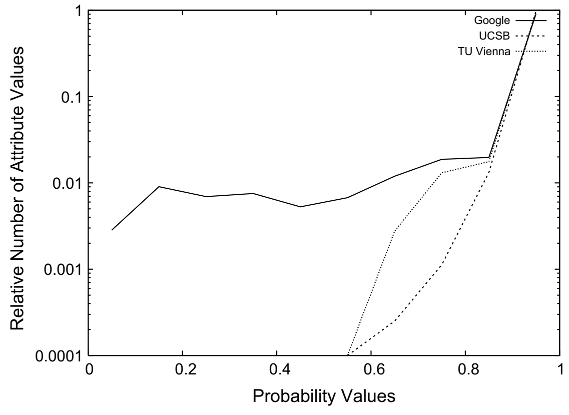


Fig. 3. Attribute length.

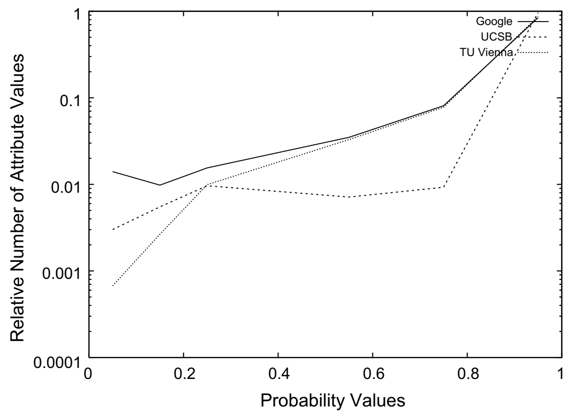


Fig. 4. Attribute character distribution.

distribution models, respectively. The y-axis shows the percentage of attribute values that appeared with a specific probability. For the figures, we aggregated the probability values (which are real numbers in the interval between 0.0 and 1.0) into ten bins, each bin covering an interval of 0.1. That is, all probabilities in the interval $[0.0, 0.1)$ are added to the first bin, values in the interval $[0.1, 0.2)$ are added to the second bin, and so forth. Note that a probability of 1 indicates a completely normal event. The relative number of occurrences are shown on a logarithmic scale.

Table 3 shows the *number of attributes* that have been rated as normal (with a probability of 1) or as anomalous (with a probability of 0) by the structural model and the token finder model. The table also provides the *number of queries* that have been classified as normal or as anomalous by the presence/absence model, the attribute order model, the access frequency model, the inter-request time delay model, and the invocation order model.

The distributions of the anomaly scores in Figs. 3 and 4, and Table 3 show that all models are capable of capturing the normality of their corresponding features. The vast majority of the analyzed attributes and queries are classified as normal (reflected by an anomaly score close to one in the figures) and only few instances deviate from the established profiles. The graphs in Figs. 3 and 4 quickly drop from above 90% of "most normal" instances in the last bin to values below 1%. It can be seen that the data collected by the Google server shows the highest variability (especially in the case of the attribute length model). This is due to the fact that the Google search string is included in the distribution. Naturally, this string, which is provided by users via their web browsers to issue Google search requests, varies to a great extent.

Note also that there are no values shown for the access frequency model, the inter-request time delay model, and the invocation order model in the Google column. This is due to the fact that these models require the IP addresses of the clients that issue the web requests. However, this information was missing in the Google log file because of anonymization efforts to protect customer privacy.

Table 3
Probability values

	Google		UCSB		TU Vienna	
	Normal	Anomalous	Normal	Anomalous	Normal	Anomalous
Structure	1,595,516	15,738	7992	1	765,311	98
Token	1,603,989	7265	7974	19	765,039	370
Presence/absence	490,704	0	4616	1	713,425	75
Order	490,704	0	4617	0	713,500	0
Access frequency	–	–	4617	0	713,408	92
Inter-request time delay	–	–	4614	3	713,310	190
Invocation order	–	–	4561	56	704,394	9106

5.2. Detection effectiveness

This section analyzes the number of hits and false positives raised during the operation of our tool. For the calculation of the anomaly scores of a query and its attributes by means of Eq. (1), all weights its were set to 1.

To assess the number of false positives that can be expected when our system is deployed, the intrusion detection system was run on our three data sets. For this experiment, we assumed that the training data contained no real attacks. Although the original log files showed a significant number of entries from Nimda or Code Red worm attacks, these queries were excluded both from the model building and detection process. This is due to the fact that all three sites use the Apache HTTP server, which fails to locate the targeted vulnerable program and thus, fails to execute it. As we only include queries that result from the invocation of existing programs into the training and detection process, these worm attacks were ignored.

The false positive rate can be easily calculated by dividing the number of reported anomalous queries by the total number of analyzed queries. The results are shown for each data set in Table 4.

The relative numbers of false positives are very similar for all three sites, but the absolute numbers differ tremendously, reflecting the different web server loads. Although almost five thousand alerts per day for the Google server appears to be a very high number at a first glance, one has to take into account that this is an initial result. The alerts are the output produced by our system running with default parameters. One approach to reduce the number of false positives is to modify the training and detection thresholds to account for the higher variability in the Google traffic. Nearly half of the number of false positives are caused by anomalous search strings that contain instances of non-printable characters (probably requests issued by users with incompatible character sets) or extremely long strings (such as URLs directly pasted into the search field). Another approach is to perform post-processing of the output, maybe using a signature-based intrusion detection system to discard anomalous queries with known deviations. In addition, it is not completely impossible to deal with this amount of alerts manually. One or two full-time employees could browse the list of alerts, quickly discarding obviously incorrect instances and concentrating on the few suspicious ones.

Table 4
False positive rates

Data set	Number of alerts	Number of queries	False positive rate	Alarms per day
Google	206	490,704	0.000419	4944
UCSB	3	4617	0.000650	0.01
TU Vienna	137	713,500	0.000192	1.71

When analyzing the output for the two university log files, we encountered several anomalous queries with attributes that were not malicious, even though they could not be interpreted as correct in any way. For example, our tool reported a character string in a field used by the application to transmit an index. By discussing these queries with the administrators of the corresponding sites, it was concluded that some of the mistakes may have been introduced by users that were testing the system for purposes other than security.

After estimating the false alarm rates, the detection capabilities of our tool were analyzed. For this experiment, we used the data set of TU Vienna. We have chosen this data set for two reasons. First, we had access to the log file and could append new log file entries; something that was impossible for the Google data set. Second, the vulnerable programs that were attacked had already been installed at this site and were regularly used. This allowed us to base the evaluation on real-world training data.

We used eleven real-world exploits downloaded from popular security sites [6,35,38] for our experiment. The set of attacks consisted of a buffer overflow against `phorum` [34], a php message board, and three directory traversal attacks against `htmlscript` [32]. Two XSS (cross-site scripting) exploits were launched against `imp` [17], a web-based email client, and two XSS exploits against `csSearch` [8], a search utility. `Webwho` [9], a web-based directory service was compromised using three variations of input validation errors. All attacks were actually executed against the TU Vienna web server, on which (patched) versions of the victim programs were installed. The log entries that corresponded to the attacks were then appended to the TU Vienna log file.

We also wanted to assess the ability of our system to detect worms such as Nimda or Code Red. However, as mentioned above, all log files were created by Apache web servers. Apache is not vulnerable against the attacks, as both worms exploit vulnerabilities in Microsoft's Internet Information Server (IIS). We solved the problem by installing a Microsoft IIS server and, after manually creating training data for the vulnerable program, launching a Code Red attack [5]. Then, we transformed the IIS log into Apache format and appended the log file entries for both the training and the attack queries to the TU Vienna data set.

Finally, our system was run on the TU Vienna log file, using the same thresholds that were used to evaluate the false alarm rate for this data set. All eleven attacks and the Code Red worm were detected. Although the attacks were known to us, all are based on existing code that was used unmodified. In addition, the log entries that corresponded to malicious queries were introduced into the log files after the model algorithms were designed and the false alarm rate was assessed. No manual tuning or adjustment was necessary.

Table 5 shows the models that reported an anomalous query or an anomalous attribute for each class of attacks. It is evident that there is no model that raises an alert for all attacks. This underlines the importance of choosing and combining different properties of queries and attributes to cover a large number of possible attack venues. Note that the access frequency model, the inter-request time delay model, and the invocation order model could not contribute to the detection of the injected attacks and, therefore, are not shown in Table 5. The reason is that these models analyze features of query sequences. However, only single attacks were injected and no malicious sequences were present. It also seemed unreasonable to create artificial attack sequences, which would be inevitably biased to our threat assumptions.

It can be seen that the length model, the character distribution model, and the structural model are very effective against a broad range of attacks that inject a substantial amount of malicious payload into an attribute string. Attacks such as buffer overflow exploits (including the Code Red worm, which bases its spreading mechanism on a buffer overflow in Microsoft's IIS) and cross-site script-

Table 5
Detection capabilities

Attack class	Length	Char. distr.	Structure	Token	Presence	Order
Buffer overflow	x	x	x		x	
Directory traversal		x	x			
XSS (cross-site scripting)	x	x	x		x	
Input validation				x		x
Code red	x	x	x			

ing attempts require a substantial amount of characters, thereby increasing the attribute length noticeably. Also, a human operator can easily tell that a maliciously modified attribute does not "look right". This observation is reflected in its anomalous character distribution and a structure that differs from the previously established profile.

Input validation errors, including directory traversal attempts, are harder to detect. The required number of characters is smaller than the number needed for buffer overflow or XSS exploits, often in the range of the legitimate attribute. Directory traversal attempts stand out because of the unusual structure of the attribute string (repetitions of slashes and dots). Unfortunately, this is not true for input validation attacks in general. The three attacks that exploit an error in Webwho did not result in an anomalous attribute for the character distribution model or the structural model. In this particular case, however, the token finder raised an alert, because only a few different values of the involved attribute were encountered during the training phase.

The presence/absence and the parameter order model can be evaded with little effort by an adversary that has sufficient knowledge of the structure of a legitimate query. Note, however, that the available exploits used in our experiments resulted in reported anomalies from at least one of the two models in 8 out of 11 cases (one buffer overflow, four directory traversal, and three input validation attacks). We therefore decided to include these models into our IDS, especially because of the low number of false alarms they produce.

The results presented in this section show that our system is able to detect a high percentage of attacks with a very limited number of false positives (all attacks, with less than 0.06% false alarms in our experiments). Some of the attacks are also detectable by signature-based intrusion detection systems such as Snort, because they represent variations of known attacks (e.g., Code Red, buffer overflows). Other attacks use malicious manipulation of the query parameters, which signature-based system do not notice. These attacks are correctly flagged by our anomaly detection system.

A limitation of the system is its reliance on web access logs. Attacks that compromise the security of a web server before the logging is performed may not be detected. The approach described in [2] advocates the direct instrumentation of web servers in order to perform timely detection of attacks, even before a query is processed. This approach may introduce some unwanted delay in certain cases, but if this delay is acceptable then the system described here could be easily modified to fit that model.

6. Conclusions

Web-based attacks should be addressed by tools and techniques that compose the precision of signature-based detection with the flexibility of anomaly-based intrusion detection system.

This paper introduces a novel approach to perform anomaly detection of web-based attacks, using as input HTTP queries containing parameters. It is, to the best of our knowledge, the first anomaly detection system specifically tailored to the detection of web-based attacks, taking advantage of application-specific correlation between server-side programs and parameters used in their invocation.

Ideally, the system will not require any installation-specific configuration and learns all query characteristics from training data, even though the level of sensitivity to anomalous data can be configured via thresholds to suit different site policies. The system has been tested on data gathered at Google, Inc. and two universities in the United States and Europe, showing promising results.

Although the models presented here are specific to the modeling of web-based attacks, our approach has been recently extended to cover the detection of anomalous system call invocations [23]. In this case, the parameters passed to a system call are characterized in a way that is similar to the one described in this paper. The use of multiple models, composed with a sophisticated Bayesian technique to compose their outputs [22], delivered very good results. This shows that the learning-based models described in this paper can be generalized to other domains as well, and that their application is not limited to the detection of web-based attacks.

Future work will focus on further decreasing the number of false positives by refining the algorithms developed so far, and by looking at additional features. The ultimate goal is to be able to perform anomaly detection in real-time for web sites that process millions of queries per day with virtually no false alarms.

Acknowledgements

We would like to thank Urs Hoelzle from Google, Inc. who made it possible to test our system on log files from one of the world's most popular web sites.

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, or the US Government.

References

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

[2] M. Almgren, U. Lindqvist, Application-integrated data collection for security monitoring, in: Proceedings of Recent Advances in Intrusion Detection (RAID), Davis, CA, October 2001, LNCS, Springer, 2001, pp. 22–36.

[3] Apache 2.0 Documentation, 2004. Available from: <http://www.apache.org/>.

[4] P. Billingsley, Probability and Measure, third ed., Wiley-Interscience, New York, 1995.

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

[6] CGI Security Homepage, 2004. Available from: <http://www.cgisecurity.com/>.

[7] K. Coar, D. Robinson, The WWW Common Gateway Interface, Version 1.1. Internet Draft, June 1999.

[8] csSearch, 2004. Available from: <http://www.cgiscript.net>.

[9] Cyberstrider Web Who. Available from: <http://www.web-who. co.uk>.

[10] D.E. Denning, An intrusion detection model, IEEE Transactions on Software Engineering 13 (2) (1987) 222–232.

[11] R. Fielding et al., Hypertext Transfer Protocol—HTTP/1.1. RFC 2616, June 1999.

[12] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, B. Miller, Formalizing sensitivity in static analysis for intrusion detection, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2004.

[13] S. Forrest, A sense of self for UNIX processes, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1996, pp. 120–128.

[14] A.K. Ghosh, J. Wanken, F. Charron, Detecting anomalous and unknown intrusions against programs, in: Proceedings of the Annual Computer Security Application Conference (ACSAC'98), Scottsdale, AZ, December 1998, pp. 259–267.

[15] Anthony Hayter, Probability and Statistics for Engineers and Scientists, second ed., Duxbury Press, Florence, KY, 2001.

[16] K. Ilgun, R.A. Kemmerer, P.A. Porras, State transition analysis: a rule-based intrusion detection system, IEEE Transactions on Software Engineering 21 (3) (1995) 181–199.

[17] IMP Webmail Client. Available from: <http://www.horde.org/imp/>.

[18] ISS, Realsecure. Available from: <http://www.iss.net/>.

[19] H.S. Javitz, A. Valdes, The SRI IDES statistical anomaly detector, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1991.

[20] D. Klein, Defending against the wily surfer: Web-based attacks and defenses, in: Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, April 1999.

[21] C. Ko, M. Ruschitzka, K. Levitt, Execution monitoring of security-critical programs in distributed systems: a specification-based approach, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1997, pp. 175–187.

[22] C. Kruegel, D. Mutz, W.K. Robertson, F. Valeur, Bayesian event classification for intrusion detection, in: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV, December 2003.

[23] C. Kruegel, D. Mutz, F. Valeur, G. Vigna, On the detection of anomalous system call arguments, in: Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS '03), Gjovik, Norway, October 2003LNCS, Springer-Verlag, 2003, pp. 326–343.

[24] C. Kruegel, T. Toth, E. Kirda, Service specific anomaly detection for network intrusion detection, in: Proceedings of the Symposium on Applied Computing (SAC), March 2002, ACM Scientific Press, 2002.

[25] C. Kruegel, G. Vigna, Anomaly detection of Web-based attacks, in: Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS'03), Wash-

ington, DC, October 2003, ACM Press, New York, 2003, pp. 251–261.
[26] T. Lane, C.E. Brodley, Temporal sequence learning and data reduction for anomaly detection, in: Proceedings of the ACM Conference on Computer and Communications Security, San Francisco, CA, ACM Press, New York, 1998, pp. 150–158.
[27] W. Lee, S. Stolfo, A framework for constructing features and models for intrusion detection systems, ACM Transactions on Information and System Security 3 (4) (2000) 227–261.
[28] J. Liberty, D. Hurwitz, Programming ASP.NET, O'Reilly, Sebastopol, CA, 2002.
[29] M. Liljenstam, D. Nicol, V. Berk, R. Gray, Simulating realistic network worm traffic for worm warning system design and testing, in: Proceedings of the ACM Workshop on Rapid Malcode, Washington, DC, 2003, pp. 24–33.
[30] U. Lindqvist, P.A. Porras, Detecting computer and network misuse with the production-based expert system toolset (P-BEST), in: IEEE Symposium on Security and Privacy, Oakland, CA, May 1999, pp. 146–161.
[31] M. Mahoney, P. Chan, Learning nonstationary models of normal network traffic for detecting novel attacks, in: Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining, Edmonton, Alberta, Canada, 2002, pp. 376–385.
[32] Miva HtmlScript. Available from: <http://www.htmlscript.com>.
[33] V. Paxson, Bro: a system for detecting network intruders in real-time, in: Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998.
[34] Phorum: PHP Message Board. Available from: <http://www.phorum.org>.
[35] PHP Advisory Homepage. Available from: <http://www.phpadvisory.com/>.
[36] L. Portnoy, E. Eskin, S. Stolfo, Intrusion detection with unlabeled data using clustering, in: Proceedings of ACM CSS Workshop on Data Mining Applied to Security, Philadelphia, PA, November 2001.
[37] M. Roesch, Snort—lightweight intrusion detection for networks, in: Proceedings of the USENIX LISA '99 Conference, Seattle, WA, November 1999.
[38] Security Focus Homepage. Available from: <http://www.securityfocus.com/>.
[39] R. Sekar, M. Bendre, P. Bollineni, D. Dhurjati, A fast automaton-based method for detecting anomalous program behaviors, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2001.
[40] G. Snedecor, W. Cochran, Statistical Methods, eighth ed., Iowa State University Press, 1998.
[41] A. Stolcke, S. Omohundro, Hidden Markov model induction by Bayesian model merging, in: Proceedings of Advances in Neural Information Processing Systems, 1993.
[42] A. Stolcke, S. Omohundro, Inducing probabilistic grammars by Bayesian model merging, in: International Conference on Grammatical Inference, 1994.
[43] K.M.C. Tan, K.S. Killourhy, R.A. Maxion, Undermining an anomaly-based intrusion detection system using common exploits, in: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID), Zurich, Switzerland, October 2002, pp. 54–73.
[44] R. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal of Computing 1 (2) (1972) 10–20.
[45] G. Vigna, W. Robertson, V. Kher, R.A. Kemmerer, A stateful intrusion detection system for world-wide Web servers, in: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV, December 2003, pp. 34–43.
[46] D. Wagner, D. Dean, Intrusion detection via static analysis, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2001, IEEE Press, 2001.
[47] D. Wagner, P. Soto, Mimicry attacks on host-based intrusion detection systems, in: Proceedings of the ACM Conference on Computer and Communications Security, Washington, DC, November 2002, pp. 255–264.
[48] C. Warrender, S. Forrest, B.A. Pearlmutter, Detecting intrusions using system calls: alternative data models, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, 1999, pp. 133–145.

Christopher Kruegel is an Assistant Professor with the Automation Systems Group at the Technical University Vienna. Before that, he was working as a research post-doc for the Reliable Software Group at the University of California, Santa Barbara. He received his Ph.D. with honors in computer science from the Technical University Vienna while working as a research assistant for the Distributed Systems Group. His research interests include most aspects of computer security, with an emphasis on network security, intrusion detection and vulnerability analysis.

Giovanni Vigna is an Associate Professor in the Department of Computer Science at the University of California in Santa Barbara. His current research interests include intrusion detection, security of mobile code systems, vulnerability analysis, and wireless systems. In particular, he worked on developing frameworks for the modular development of both misuse-based and anomaly-based intrusion detection systems. Giovanni Vigna received his M.S. with honors and Ph.D. from Politecnico di Milano, Italy, in 1994 and 1998, respectively.

William Robertson is a Ph.D. student with the Reliable Software Group at UC Santa Barbara. His research interests include static analysis, vulnerability analysis, reverse engineering, and system hardening.