Boosting Web Intrusion Detection Systems by Inferring Positive Signatures*

Damiano Bolzoni[1] and Sandro Etalle[1,2]

[1] University of Twente, Enschede, The Netherlands
damiano.bolzoni@utwente.nl
[2] Eindhoven Technical University, The Netherlands
s.etalle@tue.nl

Abstract. We present a new approach to anomaly-based network intrusion detection for web applications. This approach is based on dividing the input parameters of the monitored web application in two groups: the "regular" and the "irregular" ones, and applying a new method for anomaly detection on the "regular" ones based on the inference of a regular language. We support our proposal by realizing Sphinx, an anomaly-based intrusion detection system based on it. Thorough benchmarks show that Sphinx performs better than current state-of-the-art systems, both in terms of false positives/false negatives as well as needing a shorter training period.

Keywords: Web application security, regular languages, anomaly detection, intrusion detection systems.

1 Introduction

In the last decade, the Internet has quickly changed from a static repository of information into a practically unlimited on-demand content generator and service provider. This evolution is mainly due to the increasing success of so-called web applications (later re-branded web services, to include a wider range of services). Web applications made it possible for users to access diverse services from a single web browser, thereby eliminating reliance on tailored client software.

Although ubiquitous, web applications often lack the protection level one expects to find in applications that deal with valuable data: as a result, attackers intent on acquiring information such as credit card or bank details will often target web applications. Web applications are affected by a number of security issues, primarily due to a lack of expertise in the programming of secure applications. To make things worse, web applications are typically built upon multiple technologies from different sources (such as the open-source community), making it difficult to assess the resulting code quality. Other factors affecting the (in)security of web applications are their size, complexity and extensibility. Even

* This research is supported by the research program Sentinels (http://www.sentinels.nl). Sentinels is being financed by Technology Foundation STW, the Netherlands Organization for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

R. Meersman and Z. Tari (Eds.): OTM 2008, Part II, LNCS 5332, pp. 938–955, 2008.

with high quality components, the security of a web application can be compromised if the interactions between those components are not properly designed and implemented, or an additional component is added at a later stage without due consideration (e.g., a vulnerable web application could grant an attacker the control of another system which communicates with it).

An analysis of the Common Vulnerabilities and Exposures (CVE) repository [1] conducted by Robertson et al. [2] shows that web-related security flaws account for more than 25% of the total number of reported vulnerabilities from year 1999 to 2005 (this analysis cannot obviously take into account vulnerabilities discovered in web applications developed internally by companies). Moreover, the Symantec 2007 Internet Security Threat Report [3] states that most of the *easily exploitable vulnerabilities* (those requiring little knowledge and effort on the attacker side) are related to web applications (e.g., SQL Injection and Cross-site Scripting attacks). Most of the web application vulnerabilities are SQL Injections and Cross-site Scripting. These statistics show that web applications have become the Achilles' heel in system and network security.

Intrusion detection systems (IDSs) are used to identify malicious activities against a computer system or network. The growth of web applications (and attacks targeting them) led to adaptations of existing IDSs, yelding systems specifically tailored to the analysis of web traffic (sometimes called *web application firewalls* [4]). There exist two kinds of intrusion detection systems: *signature-* and *anomaly-based*. Here we focus on anomaly detection systems: as also argued by Vigna [2], signature-based systems are less suitable to protect web-services; among the reasons why anomaly-based systems are more suitable for protecting web applications we should mention that (1) they do not require any a-priori knowledge of the web application, (2) they can detect polymorphic attacks and (3) they can protect custom-developed web applications. On the negative side, anomaly-based systems are generally not easy to configure and use. As most of them employ mathematical models, users usually have little control on the way the system detects attacks. Often, system administrators prefer signature-based IDSs over anomaly-based ones because they are – according to Kruegel and Toth [5] – easier to implement and simpler to configure, despite the fact they could miss a significant amount of real attacks. Finally, anomaly-based systems usually show a high number of false positives [6], and – as we also argued in [7] – a high number of false positives is often their real limiting factor. These issues make the problem of protecting web servers particularly challenging.

Contribution. In this paper we present a new approach for anomaly detection devised to detect data-flow attacks [8] to web applications (attacks to the work flow are not taken into consideration) and we introduce Sphinx, an anomaly-based IDS based on it. We exploit the fact that, usually, most of the parameters in HTTP requests present some sort of regularities: by considering those regularities, we divide parameters into "regular" and "irregular" (whose content is highly variable) ones; we argue that, for "regular" parameters, it is possible to exploit their regularities to devise more accurate detection models. We substantiate this with a number of contributions:

- We introduce the concept of "positive signatures": to carry out anomaly-detection on the "regular" parameters, we first infer human-readable regular expressions by analyzing the parameter content, then we generate *positive signatures* matching *normal* inputs.
- We build a system, Sphinx, that implements our algorithm to automatically infer regular expressions and generate positive signatures; positive signatures are later used by Sphinx to build automaton-based detection models to detect anomalies in the corresponding "regular" parameters. For the parameters we call "irregular", Sphinx analyzes their content using an adapted version of our NIDS POSEIDON [9] (as it would not be "convenient" to generate a positive signature).
- We extensively benchmark our system against state-of-the-art IDSs such as WebAnomaly [10], Anagram [11] and POSEIDON.

We denote the generated signatures as "positive signatures", following the idea that they are as flexible as signatures but match positive inputs (in contrast with usual signatures used to match malicious inputs). Differently from mathematical and statistical models, positive signatures do not rely on any data frequency/presence observation or threshold. As shown by our benchmarks, positive signatures successfully detect attacks with a very low false positive rate for "regular" parameters.

Our new approach merges the ability of detecting new attacks without prior knowledge (common in anomaly-based IDSs) with the possibility of easily modifying/customizing the behaviour of part of the detection engine (common in signature-based IDSs).

Sphinx works with *any* web application, making custom-developed (or close-source) ones easily protected too. By working in an automatic way, Sphinx requires little security knowledge from system administrators, however expert ones can easily review regular expressions and make modifications.

We performed thorough benchmarks using three different data sets; benchmarks show that Sphinx performs better than state-of-the-art anomaly-based IDSs both in terms of false negatives and false positives rate as well as presenting a better learning curve than competing systems.

2 Preliminaries

In this section, we introduce the definitions and the concepts used in the rest of the paper.

Anomaly-based systems. For the purpose of this paper, we assume the presence of an application A that exchanges information over a network (e.g., think of a web server connected to the Internet running web applications). An input is any finite string of characters and we say that S is the set of all possible inputs.

Anomaly-based IDSs are devised to recognize regular activity and make use of a model $M_A \subseteq S$ of *normal* inputs: if an input $i \notin M_A$ then the IDS raises an alert. Typically, M_A is defined implicitly by using an abstract model M_{abs}

(built during a so-called training phase) and a similarity function $\phi(M_{abs}, i) \rightarrow \{yes, no\}$, to discern normal inputs from anomalous. For instance, an example of similarity function is the distance d having that $\{d(M_{abs}, i)$ *is lower than a given threshold* $t\}$.

Desiderata. The completeness and accuracy [12] of an anomaly detection system lie in the quality of the model M_A (i.e., the way it is defined and is built, later, during the training phase). We call *completeness* the ratio $TP/(TP+FN)$ and *accuracy* the ratio $TP/(TP+FP)$, where TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives the IDS raised. For M_A we have the following set of desiderata:

- M_A, to avoid false positives, should contain all foreseeable non-malicious inputs;
- M_A, to avoid false negatives, should be disjoint from the set of possible attacks;
- M_A should be simple to build, i.e., the shorter the training phase required to build M_A, the better it is.

The last point should not be underestimated: Training an anomaly detection system often requires having to put together a representative training set, which also has to be cleaned from malicious input (this is done off-line, e.g., using signature-based systems). In addition, applications change on a regular base (this is particularly true in the context of web applications, which are highly dynamic), and each time a software change determines a noticeable change in the input of the application, one needs to re-train the NIDS. The larger the training set required, the higher is the required workload to maintain the system.

Automata. An automaton is a mapping from strings on a given alphabet to the set {yes, no} such as $\alpha : Strings \rightarrow \{yes, no\}$; the language it accepts corresponds to $\{s \in Strings \mid \alpha(s) = yes\}$. Given a finite set of strings I it is easy to construct α_I, the automaton which recognizes exactly I.

3 Detecting Data-Flow Attacks to Web Applications

Let us describe how web applications handle user inputs. Web applications produce an output in response to a *user request*, which is a string containing a number of *parameter names* and the respective *parameter value* (for the sake of simplicity we can disregard parameterless HTTP requests, as attackers cannot inject attack payloads). RFC 2616 [13] defines the structure and the syntax of a request with parameters (see Figure 1).

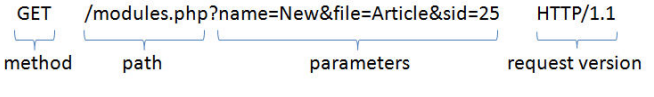

Fig. 1. A typical HTTP (GET) request with parameters

We can discard the request version and – for the sake of exposition – the method. Of interest to us is the presence of a path, a number of parameter names and of their respective values (in Figure 1 the parameter names are "name", "file" and "sid" and their respective values are "New", "Article" and "25"). The set of parameters is finite. A value can be any string (though, not all the strings will be accepted by the web application). Since no type is defined, the semantic of each parameter is implicitly defined within the context of the web application and such parameters are usually used in a consistent manner (i.e., their syntax is fixed). In the sequel, we refer to the natural projection function: Given an input i $path?p_1 = v_1 \& p_2 = v_2 \& \ldots \& p_n = v_n$, we define $p_n(i) = v_i$ as the function extracting the value of parameter p_n from input i.

Exploiting regularities. Intuitively, it is clear that the more "predictable" the input of the application A is, the easier it is to build a model M_A satisfying the desiderata (1), (2) and (3). For instance, if we knew that A accepted only – say – strings not containing any "special character" (a very predictable input), then building M_A as above would be trivial.

Our claim is that, in the context of web applications, it is possible to exploit the regularities which are not present in other settings to define and build M_A based on the inference of regular automata, which leads to the definition of an IDS that is more effective (yet simpler) than state-of-the-art systems.

Commonly, anomaly-based IDSs build (and use) a single model M to analyse network traffic. Our proposal takes advantage of the fact that requests to web applications present a fixed syntax, consisting of a sequence of *parameter* = *value*, and instead of building a single model to analyse the input, it builds an *ad hoc* model M_n for each parameter p_n (in practice, we create a separate model for many – not all – parameters). As already observed by Kruegel and Vigna in [14], this allows it to create a more faithful model of the application input. The idea is that of defining M_A implicitly by electing that $i \in M_a$ iff for each parameter n we have that $p_n(i) \in M_n$ (or that $p_n(i)$ is empty).

Regular and irregular parameters. So we first divide the parameters in two groups: the *regular parameters* and the *irregular parameters*. The core of our idea is that for the *regular* parameters it is better to define M_n as a *regular language* rather than using state-of-the-art anomaly-based systems. By "better" we mean that this method yields (a) lower false positive rate, (b) same (or higher) detection rate (c) a shorter learning phase. We support our thesis by presenting an algorithm realizing this.

For each regular parameter, we build a model using a combination of abstraction and regular expression inference functions that we are going to explain in the following section: We call this the *regular-text* methodology, following the intuition it is devised to build the model for parameters which are usually filled by data having a well-defined format (e.g., integer numbers, dates, user session cookies etc.). For the *irregular* parameters we use classical anomaly-based techniques, i.e., n-gram analysis: We call this the *raw-data* methodology, since it is meant to be more suitable for building the model of parameters containing e.g., pieces of blogs or emails, images, binary data etc.

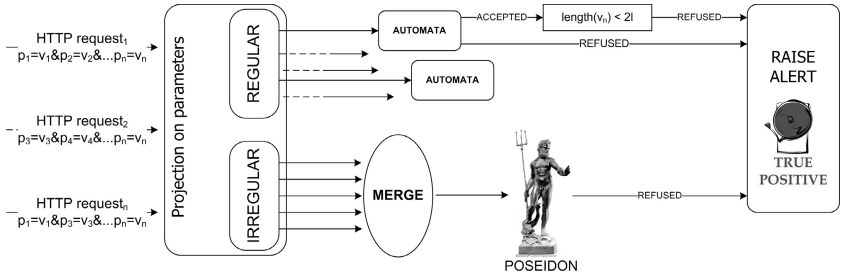

Fig. 2. Sphinx's internals

4 Sphinx's Detection Engine

To substantiate our claims, we built Sphinx: An anomaly-based intrusion detection systems specifically tailored to detect attacks in a web application data flows. Let us see how it works: building the intrusion detection system involves the following steps.

4.1 Building the Model

We first outline how we build the model M_A of the application given a *training set DS*; DS is a set of inputs (i.e., HTTP requests), which we assume does not contain fragments of attacks. Typically, DS is obtained by making a dump of the application input traffic during a given time interval, and it is cleaned (i.e., the malicious traffic is removed) off-line using a combination of signature-based intrusion detection techniques and manual inspection.

During the *first* part of the training, we discover the set of parameters used by the web application: DS is scanned a first time and the parameters $\{p_1, \ldots, p_n\}$ are extracted and stored. We call $DS_n = \{p_n(i) \mid i \in DS\}$ the training set for the parameter p_n (i.e., the projection of D_S on the parameter p_n).

In the second step we divide the parameters into two classes: the *regular* ones (for which we use the new anomaly detection algorithms based on regular expressions) and the irregular ones. In practice, to decide which parameters are the "regular" ones, in the sequel we use a simple a-priori syntactic check: If at least the 10% of the samples in DS_n contains occurrences of more than 5 distinct non-alphanumeric characters, we say that p_n is an irregular parameter, otherwise it is a regular one. This criterion for separating (or, better, defining) the regular parameters from the irregular ones is clearly arbitrary. Simply, our benchmarks have shown that it gives good results. We impose a minimum amount of samples (10%) to present more than 5 distinct non-alphanumeric characters to prevent Sphinx's engine from classifying a parameter as "irregular" because of few anoumalous samples. An attacker could in fact exploit this to force the system to classify any parameter as "irregular".

In the last step of the training phase we build a model M_n for each of the regular parameter p_n, using the training set DS_n. The *irregular* parameters, in turn, are again grouped together and for them we build a unique model: We could also build a single model per irregular parameter, but this would slow down the learning phase, which is already one of the weak spots of classical anomaly detection techniques.

4.2 The Regular-Text Methodology

This represents the most innovative aspect of our contribution. The *regular-text* methodology is designed to build a simple model of the "normal" input of the regular parameters. This model is represented by a regular expression and anomalies are detected by the derived finite automaton. We illustrate this methodology by presenting two algorithms realizing it: The first one is called the *simple regular expression generator* (SREG) and it is meant to illustrate the fundamental principles behind the construction of such regular language, the second one is called *complex regular expression generator* (CREG), and can be regarded as a further development of the first one. Here we should mention that standard algorithms to infer regular expressions (see [15] for a detailed overview) cannot be used for intrusion detection because they infer an expression matching exactly the strings in the training data set only, while we need to match a "reasonable" superset of it.

Simple regular expression generator. Here we introduce our first algorithm. We have a training set DS_n (the training set relative to parameter n) and we want to build a model M_n of the parameter itself, and a decision procedure to determine for a given input i whether $p_n(i)$ is contained in M_n or not.

Our first algorithm to generate M_n is based on applying two abstractions to DS_n. The first abstraction function, abs_1, is devised to abstract all letters and all digits (other symbols are left untouched), and works as follows:

$$abs_1(c_1 \dots c_n) = abs_1(c_1), \dots, abs_1(c_n)$$

$$abs_1(c_i) = \begin{cases} \text{“}a\text{”}, c_i \in \{\text{“}a\text{”}, \dots, \text{“}Z\text{”}\} \\ \text{“1”}, c_i \in \{\text{“0”}, \dots, \text{“9”}\} \\ c_i \quad otherwise \end{cases}$$

Thus abs_1 abstracts alphanumerical characters while leaving non-alphanumerical symbols untouched (for the reasons we clarified in Section 3). The reason for this choice is that, in the context of web applications, the presence of "unusual" symbols (or a concatenation of them) could indicate the presence of attack payloads.

The second abstraction we use is actually a contraction:

$$abs_2(c_1 \dots c_n) = \begin{cases} abs_2(c_2 \dots c_n) \; if \; c_1 = c_2 = c_3 = \text{“}a\text{”} \; or \; \text{“1”} \\ c_1 c_1 \; abs_2(c_3 \dots c_n) \; if \; c_1 = c_2 \neq c_3 \; and \; c_1 = \text{“}a\text{”} \; or \; \text{“1”} \\ c_1 \cdot \; abs_2(c_2 \dots c_n) \; if \; c_1 \neq c_2 \; or \; c_1 = c_2 \; and \; c_1 \neq \text{“}a\text{”} \; and \; c_1 \neq \text{“1”} \end{cases}$$

Table 1. Some examples of applying abstractions abs_1 and abs_2 on different inputs

Input	$abs_1(i)$	$abs_2(i')$
11/12/2007	11/11/1111	11/11/11
addUser	aaaaaaa	aa
C794311F-FC92-47DE-9958	a111111a-aa11-11aa-1111	a11a-aa11-11aa-11

Intuitively, abs_2 collapses all strings of letters (resp. digits) of length greater or equal to two onto strings of letters (resp. digits) of length two. Again, symbols are left untouched, as they may indicate the presence of an attack. Table 1 provides some examples of application of abs_1 and abs_2 on different input strings.

These two abstraction algorithms are enough to define our first model, only one detail is still missing. If the samples contained in DS_n have maximum length say l, then we want our model M_n to contain strings of maximum length $2l$: an input which is much longer than the samples observed in the training set is considered anomalous (as an attacker could be attempting to inject some attack payload).

Definition 1. *Let DS_n be a training set. Let $l = max\{|x| \mid x \in DS_n\}$. We define the* simple regular-text model *of DS_n to be*

$$M_n^{simple} = \{x \mid |x| \leq 2l \wedge \exists y \in DS_n \; abs_2(abs_1(x)) = abs_2(abs_1(y))\}$$

During the *detection* phase, if $p_n(i) \notin M_n^{simple}$ then an alert is raised. The decision procedure for checking whether $i \in M_n^{simple}$ is given by the finite automaton $\alpha_{M_n^{simple}}$ that recognizes M_n^{simple}. Building $\alpha_{M_n^{simple}}$ is almost straightforward. It is implemented using a (unbalanced) tree data-structure, therefore adding a new node (i.e., a previously unseen character) costs $O(l)$, where l is the length of the longest observed input. The complexity of building the tree for n inputs is therefore $O(n \cdot l)$. The decision procedure to check, given an input i, if $p_n(i) \in M_n$ has complexity $O(l)$. To simplify things, we can represent this automaton as a regular expression.

Complex regular expression generator. The simple SREG algorithm is effective for illustrating how regular expressions can be useful in the context of anomaly detection and how they can be used to detect anomalies in regular parameters. Nevertheless we can improve on SREG in terms of FPs and FNs by using an (albeit more complex) algorithm, which generates a different, more complex model.

The algorithm goes through two different phases. In the *first phase* each DS_n is partitioned in groups with common (shared) prefixes or suffixes (we require at least 3 shared characters, to avoid the generation of useless regular expressions).

Table 2. Examples of how SREG works on different input sets

Training sets	$abs_2(abs_1(i))$	SREG
01/01/1970 30/4/85 9/7/1946	11/11/11 11/1/11 1/1/11	1(1/1(1/11|/11)|/1/11)
41E44909-C86E-45EE-8DA1 0F786C5B-940B-4593-B96D 656E0AB4-B221-422F-92AC	11a11-a11a-11aa-1aa1 1a11a1a-11a-11-a11a 111a1aa1-a11-11a-11aa	1(1(a11-a11a-11aa-1aa1| 1a1aa1-a11-11a-11aa)| a11a1a-11a-11-a11a)

Table 3. Examples of pattern selection, generated regular expressions for samples and the resulting one

Training set	Symbol Pattern	Shared Pattern	Intermediate Regular Expressions	Resulting Regular Expression
al.ias@atwork.com 3l_1t3@hack.it info@dom-ain.org	[. @ .] [_ @ .] [@ - .]	[@ .]	$(a^+\ [.])^+@(a^+).(a^+)$ $((a\|1)^+\ [\_])^+@(a^+).(a^+)$ $(a^+)@(a^+\ [-])^+.(a^+)$	$((a\|1)^+\ [.\ \|\ \_])^+@(a^+\ [-])^+.(a^+)$

In the *second phase*, the algorithm generates a regular expression for each group as follows. First, it applies abstractions abs_1 and abs_2 on each stored body (we call the *body* the part of the input obtained by removing the common prefix or suffix from it: prefixes and suffixes are handled later). Secondly, it starts to search for common symbol patterns inside bodies. Given a string s, we define the symbol pattern of s the string obtained by removing all alphanumerical characters from s.

As bodies could contain different symbol patterns, non-trivial patterns (i.e., patterns of length greater than one) are collected and the pattern matching the highest number of bodies is selected. If some bodies do not match the selected pattern, then the same procedure is repeated on the remaining bodies set till no more non-trivial patterns can be found.

For each non-trivial symbol pattern discovered during the previous step, the algorithm splits each body into sub-strings according to the symbol pattern (e.g., $s = s' - s'' - s'''$ is split in $\{s', s'', s'''\}$ w.r.t. the pattern). Corresponding sub-strings are grouped together (e.g., having strings s_1,s_2 and s_3 the algorithm creates $g1 = \{s'_1, s'_2, s'_3\}$, $g2 = \{s''_1, s''_2, s''_3\}$ and $g3 = \{s'''_1, s'''_2, s'''_3\}$) and a regular expression is generated for each group. This regular expression is the modified according to some heuristics to match also similar strings. Regular expressions are then merged with the corresponding symbol pattern, and any previously found prefix or suffix is eventually added (e.g., $re = (prefix)re_{g1} - re_{g2} - re_{g3}$). Table 3 depicts an example of the different steps of CREG.

Finally, for bodies which do not share any non-trivial symbol pattern with the other members of the group, a dedicated general regular expression is generated. Table 4 shows some examples of generated regular expressions for different sets of strings.

Given the resulting regular expression (i.e., the positive signature), we build a finite automaton accepting the language it represents. The automaton is built in such a way that it accepts (as in the case of SREG) only strings of length less

Table 4. Examples of how CREG works on different input sets

Training sets	Pattern	Resulting Regular Expression
01/01/1970 30/4/85 9/7/1946	[/ / /]	$(1^+/1^+/1^+)$
addUser deleteUser viewUser	N/A	(a^+)User
41E44909_C86E_45EE_8DA1 0F786C5B-940B-4593-B96D 656E0AB4-B221_422F-92AC	Cannot find non-trivial patterns ⇒ general regular expression	$((a\|1)^+[-\|\_])^+$

than $2l$. In the detection phase, if an input is not accepted by the automaton, an alert is raised.

Effectiveness of positive signatures. Because of the novelty of our approach, let us see some concrete examples regarding the potential of positive signatures.

Think of a signature such as "id=1^+", accepting numeric values: the parameter *id* is virtually protected from any data-flow attack, since only digits are accepted. When we consider more complex signatures, such as "email=$((a|1)^+$ [. | _])$^+$@(a^+ [-])$^+$.(a^+)" (extracted from Table 3), it is clear that common attack payloads would be easily detected by the automaton derived from the regular expression, as they require to inject different symbol sets and in different orders.

One could argue that it could be sufficient (and simpler) to detect the presence of typical attack symbols (having them classified and categorized) or, better, the presence of a symbol set specific to an attack (e.g., " ' ", " , " and " - " for a SQL Injection). However, a certain symbol set is not said to be harmful per se, but it must be somehow related to the context: In fact, the symbol " , " used in SQL Injection attacks, can also be found in some representations of real numbers. Positive signatures, in contrast with usual state-of-the-art anomaly detection approaches, provide a sort of context for symbols, thus enhancing the detection of anomalies.

For instance, by May 2008, CVE contains more than 3000 SQL Injection and more than 4000 Cross-site Scripting attacks but only less than 250 path traversal and less than 400 buffer overflow attacks (out of a total of more than 30000 entries). Most of the SQL Injections happen to exploit "regular" parameters (integer-based), where the input is used inside a "SELECT" statement and the attacker can easily add a crafted "UNION SELECT" statement to extract additional information such as user names and their passwords. The same reasoning applies to Cross-site Scripting attacks. Sphinx's positive signatures can significantly enhance the detection of these attacks.

4.3 The Raw-Data Methodology

The raw-data methodology is used to handle the "irregular" parameters. For them, using regular automata to detect anomalies is not a good idea: The input is so heterogeneous that any automaton devised to recognize a "reasonable" super set of the training set would probably accept any input. Indeed, for this kind of heterogeneous parameters we can better use a classical anomaly detection engine, based on statistical content analysis (e.g., n-gram analysis). In the present embodiment of Sphinx we use our own POSEIDON [9] (which performs very well in our benchmarks), but we could have used any other anomaly-based NIDS. POSEIDON is a 2-tier anomaly-based NIDS that combines a neural network with n-gram analysis to detect anomalies. POSEIDON originally performs a packet-based analysis: Every packet is classified by the neural network, then, using the classification information given, the real detection phase takes place based on statistical functions considering the byte frequencies and distributions (the n-gram analysis). In the context of web applications, we are dealing with

streams instead of packets, therefore we have adapted POSEIDON to the context. POSEIDON requires setting a threshold value to detect anomalous inputs: to this end, in our tests we have used the automatic heuristic provided in [7].

4.4 Using the Model

When the training phase is completed, Sphinx switches to detection mode. As a new HTTP request comes, parameters are extracted applying the projections $p_1, \ldots, p_n$. Sphinx stores, for each parameter analyzed during the training phase, information regarding the model to use to test its input. For a parameter that was labelled as "regular", the corresponding automaton is selected. If it does not accept the input, then an alert is raised. If the parameter was labelled "irregular", the content is analyzed using the adapted version of POSEIDON (which uses a single model for all the irregular parameters). If the content is considered to deviate from the "normal" model, an alert is raised. Sphinx raises an alert also in the case a parameter has never been analyzed before and suddenly materializes in a HTTP request, since we consider this eventuality as an attempt to exploit a vulnerability by an attacker.

Editing and customizing positive signatures. One of the most criticized disadvantages of anomaly-based IDSs lies in their "black-box" approach. Being most of anomaly-based IDS based on mathematical models (e.g., neural networks), users have little or no control over the detection engine internals. Also, users have little influence on the false positive/negative rates, as they can usually adjust some threshold values only (the link between false positives/negatives and threshold values is well-known [16]).

Signatures, as a general rule, provide more freedom to customize the detection engine behaviour. The use of positive signatures opens the new possibility of performing a thorough tuning for anomaly-based IDSs, thereby modifying the detection models of regular parameters. Classical anomaly-based detection systems do not offer this possibility as their models aggregate collected information, making it difficult (if not impossible) to remove/add arbitrary portions of data.

Positive signatures allow IT specialists to easily (and quickly) modify or customize the detection models when there is a need to do so (see Table 5), like when (1) some malicious traffic, that was incorporated in the model during the training phase, has to be purged (to decrease the false negative rate) and (2) a new (previously unseen) input has to be added to the model (to decrease the false positive rate).

Table 5. Some examples of signature customization

Positive Signature	Problem	Action
id=d$^+$ | (a$^+$ ['|,|-|;])$^+$	The payload of a SQL Injection attack was included in the training set	The IT specialist manually modifies the signature ⇒ id=d$^+$
date=1$^+$/1$^+$/1$^+$	A new input ("19-01-1981") is observed after the training phase, thereby increasing the false positive rate	The IT specialist re-train the model for parameter with the new input and the positive signature ⇒ date=1$^+$/1$^+$/1$^+$ | 1$^+$-1$^+$-1$^+$ is automatically generated

5 Benchmarks

The quality of the data used in benchmarks (and the way it was collected) greatly influences the number of successfully detected attacks and false alerts: Test data should be representative of the web server(s) to monitor, and the attack test bed should reflect modern attack vectors. Presently the only (large) public data set for testing intrusion detection systems is the DARPA data set [17], dated back to 1999. Although this data set is still widely used (since public data sets are scarce), it presents significant shortcomings that make it unsuitable to test our system: E.g., only four attacks related to web (and most of them target web server's vulnerabilities) are available and traffic typology is outdated (see [18,19] for detailed explanations about its limitations). So, to carry out our experiments we collected three different data sets from three different sources: Real production web sites that strongly rely on user parameters to perform their normal activity.

The first data set is a deployment of the widely-known PostNuke (a content-management system). The second comes from a (closed-source) user forum web application, and it contains user messages sent to the forum, which present a variable and heterogeneous content. The third data set has been collected from the web server of our department, where PHP and CGI scripts are mainly used. Each data set contains both GET and POST requests: Sphinx's engine can interpret the body of POST requests as well, since only the request syntax changes from a GET request, but the content, in case of regular parameters, looks similar. In case of encoded content (for instance, a white space is usually encoded as the hexadecimal value "%20"), the content is first decoded by Sphinx's engine and then processed.

We collected a number of samples sufficient to perform extensive training and testing (never less than two weeks of traffic and in one case a month, see also Table 6). Data used for training have been made attack-free by using Snort to remove well-known attacks and by manually inspecting them to purge remaining noise.

Comparative benchmarks. To test the effectiveness of Sphinx, we compare it to three state-of-the-art systems, which have been either developed specifically to detect web attacks or have been extensively tested with web traffic.

First, WebAnomaly (Kruegel et al. [14]) combines five different detection models, namely attribute length, character distribution, structural inference, attribute presence and order of appearance, to analyze HTTP request parameters. Second, Anagram (Wang et al. [11]) uses a Bloom filter to store any n-gram (i.e., a sequence of bytes of a given length) observed during a training phase,

Table 6. Collected data sets: code name for tests, source and number of samples

Data set	Web Application	# of samples (HTTP requests)
DS_A	PostNuke	∼460000 (1 month)
DS_B	(Private) User forum	∼290000 (2 weeks)
DS_C	CS department's web site (CGI & PHP scripts)	∼85000 (2 weeks)

without counting the occurrences of n-grams. During the detection phase, Anagram flags as anomalous a succession of previously unseen n-grams. Although not specifically designed for web applications, Anagram has been extensively tested with logs captured from HTTP servers, achieving excellent results. We set the parameters accordingly to authors' suggestions to achieve the best detection and false positive rates. Third, our own POSEIDON, the system we adapted to handle raw-text parameters in Sphinx. POSEIDON, during our previous experiments [9], showed a high detection rate combined with a low false positive rate in tests related to web traffic, outperforming the leading competitor.

We divide tests into two phases. We compare the different engines first by considering only the "regular" parameters. Later, we consider full HTTP requests (with both "regular" and "irregular" parameters).

The goal our tests is twofold. Next to the effectiveness of Sphinx, we are also interested in testing its the *learning rate*: Any anomaly-based algorithm needs to be trained with a certain amount of data before it is able to correctly flag attacks without generating a massive flow of false alerts. Intuitively, the longer the training phase is, the better the IDS should perform. But an anomaly detection algorithm that requires a shorter training phase it is certainly easier to deploy than an algorithm that requires a longer training phase.

Testing the regular-expression engine. In this first test, we compare our CREG algorithm to WebAnomaly, Anagram and POSEIDON using training sets of increasing size with "regular" requests only (requests where raw-data parameters have been previously removed). This test aims to demonstrate the effectiveness of our approach over previous methods when analyzing regular parameters. We use training sets of increasing size to measure the learning rate and to simulate a training phase as it could take place in a real environment, when a system is not always trained thoroughly. For the attack test bed, we selected a set of real attacks which truly affected the web application we collected the logs of and whose exploits have been publicly released. Attacks include path traversal, buffer overflow, SQL Injection and Cross-site Scripting payloads. Attack mutations, generated using the Sploit framework [20], have been included too, to reproduce the behaviour of an attacker attempting to evade signature-based systems. The attack test bed contains then 20 attacks in total. Table 7 reports results for tests with "regular" requests.

Our tests show that with a rather small training set (20000 requests, originally collected in less than two days), CREG generates 43 false positives (∼0,009%), less than 2 alerts per day. The "sudden" decrease in FPs shown by CREG (and

Table 7. Results for CREG and comparative algorithms on "regular" requests only

#training samples		CREG	WebAnomaly	Anagram	POSEIDON
5000	Attacks	20/20	18/20	20/20	20/20
	FPs	1062	1766	144783	1461
10000	Attacks	20/20	16/20	20/20	20/20
	FPs	1045	1529	133023	1387
20000	Attacks	20/20	16/20	20/20	20/20
	FPs	43	177	121484	1306
50000	Attacks	20/20	14/20	20/20	20/20
	FPs	16	97	100705	1251

Webanomaly) when we train it with at least 20000 requests is due to the fact that, with less than 20000 training samples, some parameters are not analyzed during training (i.e., some URLs have not been accessed), therefore no model is created for them and by default this event is considered malicious. One surprising thing is the high number of false positives shown by Anagram [11,21]. We believe that this is due to the fact that Anagram raises a high number of false positives on specific fields whose content looks pseudo-random, which are common in web applications. Consider for example the following request parameter *sid=0c8026e78ef85806b67a963ce58ba823* (it is a user's session ID automatically added by PostNuke in each URL link), being this value randomly generated as a new user comes: Such a string probably contains a number of n-grams which were not observed during the training phase therefore, and Anagram is likely to flag any session ID as anomalous. On the other hand, CREG exploits regularities in inputs, by extracting the syntax of the parameter content (e.g., the regular expression for *sid* is $(a^+|d^+)^+$), and easily recognizes similar values in the future. WebAnomaly shows (unexpectedly, at least in theory) a worse detection rate as the training set samples increase. This is due to the fact that the content of new samples is similar to some attack payloads, thus the system is not able to discern malicious traffic.

Testing Sphinx on the complete input. We show the results of the second test which uses the complete input of the web application (and not only the regular parameters). We use the two data sets DS_B and DS_C: DS_B contains 78 regular and 10 irregular parameters; DS_C respectively 334 and 10. We proceed as before, using different training sets with increasing numbers of samples. To test our system, we have used the attack database presented in [21] which has already been used to assess several intrusion-detection systems for web attacks. We adapted the original attack database and added the same attack set used in our previous test session. We found this necessary because [21] contains some attacks to the platforms (e.g., a certain web server vulnerability in parsing inputs) rather than to the web applications themselves (e.g., SQL Injection attacks are missing). Furthermore, we had to exclude some attacks since they target web server vulnerabilities by injecting the attack payload inside the HTTP headers: Although Sphinx could be easily adapted to process header fields, our logs do not always contain a HTTP header information. In total, our attack bed contains

Table 8. Results for Sphinx and comparative algorithms on full requests from DS_B: we report separate false positive rates for Sphinx (RT stands for "regular-text" models and RD for "raw-data" model)

# training samples		Sphinx FPs RT	Sphinx FPs RD	WebAnomaly	Anagram	POSEIDON
5000	Attacks	80/80		67/80	80/80	80/80
	FPs	162	1955	2593	90301	3478
10000	Attacks	80/80		67/80	80/80	80/80
	FPs	59	141	587	80302	643
20000	Attacks	80/80		53/80	80/80	80/80
	FPs	43	136	451	71029	572
50000	Attacks	80/80		47/80	80/80	80/80
	FPs	29	127	319	61130	433

Table 9. Results for Sphinx and comparative algorithms on full requests from DS_C: we report detailed false positive rates for Sphinx (RT stands for "regular-text" models and RD for "raw-data" model)

# training samples		Sphinx FPs RT	Sphinx FPs RD	WebAnomaly	Anagram	POSEIDON
5000	Attacks	80/80		78/80	80/80	80/80
	FPs	36	238	607	16779	998
10000	Attacks	80/80		77/80	80/80	80/80
	FPs	24	109	515	13307	654
20000	Attacks	80/80		49/80	80/80	80/80
	FPs	10	98	459	7417	593
50000	Attacks	80/80		46/80	80/80	80/80
	FPs	3	47	338	4630	404

80 vectors, including the 20 attacks previously used to test DS_A (adapted to target the new data set).

The tests show that the presence of irregular parameters significantly influences the false positive rate of Sphinx. We need an extensive training to achieve a rate of 10 false positives per day: this is not surprising, since we observed a similar behaviour during previous tests (see [7,9]).

6 Related Work

Despite the fact that web applications have been widely developed only in the last half-decade years, the detection of web-based attacks has immediately received considerable attention.

Ingham et al. [22] use a deterministic finite automaton (DFA) to build a profile of legal HTTP requests. It works by tokenizing HTTP request parameters, and storing each token type and (optionally) its value. Pre-defined heuristic functions are used to validate and generalize well-known input values (e.g., dates, file types, IP addresses and session cookies). Each state in the DFA represents an unique token, and the DFA has a transition between any two states that were seen consecutively (from a chronological point of view) in the request. A similarity function determines if a request has to be considered anomalous. It reflects the changes (i.e., for each missed token a new transition would have to be added) that would have to be made to the DFA for it to accept the request.

Despite its effectiveness, this approach relies on predefined functions which can be used to analyse only certain (previously known) input types. Furthermore, for some parameters (e.g., blog messages) it could be difficult to find a good function to validate the content. Sphinx, on the other hand, is able to learn in an automatic way the syntax of most of parameter values and uses a content-based anomaly detector for parameters whose syntax cannot be extracted.

WebAnomaly (Kruegel et al. [10]) analyses HTTP requests and takes significant advantage of the parameter-oriented URL format common in web applications. The system applies up to nine different models at the same time to detect possible attacks, namely: attribute length and character distribution, structural inference, token finder, attribute presence and order, access frequency, inter-request time delay and invocation order. We have compared WebANomaly to Sphinx in our benchmarks.

Jovanovic et al. [23] present a static-analysis tool (called Pixy) for web applications. The tool detects data flow vulnerabilities by checking how inputs could affect the (intended) behaviour of the web application, leading to an outflow of information. This approach requires the sources code of the web application to be available.

Finally, we should mention that Almgren et al. [24] and Almgren and Lindqvist [25] present similar systems which are based on signature-based techniques and either analyse web server logs ([24]) or are integrated inside the web server itself.

7 Conclusion

Sphinx is conceptually simple, and – as our benchmarks show – to detect attacks to web applications it performs better than competing systems. Here we want to stress that the system we have compared it to are really the best ones now available and that the set of benchmarks we have carried out (with 3 different data sets) is very extensive. Another aspect we want to stress is that Sphinx presents also a better learning curve than competitors (i.e., it needs a lower number of samples to train itself). This is very important in the practical deployment phase, when changes to the underlying application require that every now and then the system be retrained (and retraining the system requires cleaning up the training set from possible attacks, an additional operation which needs to be done – accurately – off-line).

Sphinx, instead of using solely mathematical and statistical models, takes advantage of the *regularities* of HTTP request parameters and is able to automatically generate, for most of the parameters, human-readable regular expressions (we call them "positive signatures"). This also means that the IT specialist, if needed, could easily inspect and modify/customize the signatures generated by Sphinx, thereby modifying the behaviour of the detection engine. This aspect should be seen in the light of the criticisms that is often addressed to anomaly-based systems: That they are as black-boxes which cannot be tuned by the IT specialists in ways other than modifying, e.g., the alert threshold [5]. Sphinx is – to our knowledge – the first anomaly-detection system which relies heavily on signatures which can be seen, interpreted, and customized by the IT specialists.

References

1. The MITRE Corporation: Common Vulnerabilities and Exposures database (2004), `http://cve.mitre.org`
2. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.A.: Using generalization and characterization techniques in the anomaly-based detection of web attacks. In: NDSS 2006: Proc. of 17th ISOC Symposium on Network and Distributed Systems Security (2006)
3. Symantec Corporation: Internet Security Threat Report (2006), `http://www.symantec.com/enterprise/threat-report/index.jsp`
4. Web Application Security Consortium: Web Application Firewall Evaluation Criteria (2006), `http://www.webappsec.org/projects/wafec/`

5. Kruegel, C., Toth, T.: Using Decision Trees to Improve Signature-based Intrusion Detection. In: Vigna, G., Kruegel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 173–191. Springer, Heidelberg (2003)
6. Axelsson, S.: The base-rate fallacy and the difficulty of intrusion detection. ACM Trans. Inf. Syst. Secur. (TISSEC) 3(3), 186–205 (2000)
7. Bolzoni, D., Crispo, B., Etalle, S.: ATLANTIDES: An Architecture for Alert Verification in Network Intrusion Detection Systems. In: LISA 2007: Proc. 21th Large Installation System Administration Conference, USENIX Association, pp. 141–152 (2007)
8. Balzarotti, D., Cova, M., Felmetsger, V.V., Vigna, G.: Multi-Module Vulnerability Analysis of Web-based Applications. In: CCS 2007: Proc. 14th ACM Conference on Computer and Communication Security, pp. 25–35. ACM Press, New York (2007)
9. Bolzoni, D., Zambon, E., Etalle, S., Hartel, P.: POSEIDON: a 2-tier Anomaly-based Network Intrusion Detection System. In: IWIA 2006: Proc. 4th IEEE International Workshop on Information Assurance, pp. 144–156. IEEE Computer Society Press, Los Alamitos (2006)
10. Kruegel, C., Vigna, G., Robertson, W.: A multi-model approach to the detection of web-based attacks. Computer Networks 48(5), 717–738 (2005)
11. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
12. Debar, H., Dacier, M., Wespi, A.: A Revised Taxonomy of Intrusion-Detection Systems. Annales des Télécommunications 55(7–8), 361–378 (2000)
13. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 (1999)
14. Kruegel, C., Vigna, G.: Anomaly Detection of Web-based Attacks. In: CCS 2003: Proc. 10th ACM Conference on Computer and Communications Security, pp. 251–261 (2003)
15. Fernau, H.: Algorithms for Learning Regular Expressions. In: Jain, S., Simon, H.U., Tomita, E. (eds.) ALT 2005. LNCS (LNAI), vol. 3734, pp. 297–311. Springer, Heidelberg (2005)
16. van Trees, H.L.: Detection, Estimation and Modulation Theory. Part I: Detection, Estimation, and Linear Modulation Theory. John Wiley and Sons, Inc., Chichester (1968)
17. Lippmann, R., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA offline intrusion detection evaluation. Computer Networks: The International Journal of Computer and Telecommunications Networking 34(4), 579–595 (2000)
18. Mahoney, M.V., Chan, P.K.: An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In: Vigna, G., Kruegel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 220–237. Springer, Heidelberg (2003)
19. McHugh, J.: Testing Intrusion Detection Systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. ACM Transactions on Information and System Security (TISSEC) 3(4), 262–294 (2000)
20. Vigna, G., Robertson, W.K., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. In: CCS 2004: Proc. 11th ACM Conference on Computer and Communications Security, pp. 21–30. ACM Press, New York (2004)

21. Ingham, K.L., Inoue, H.: Comparing Anomaly Detection Techniques for HTTP. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 42–62. Springer, Heidelberg (2007)
22. Ingham, K.L., Somayaji, A., Burge, J., Forrest, S.: Learning DFA representations of HTTP for protecting web applications. Computer Networks: The International Journal of Computer and Telecommunications Networking 51(5), 1239–1255 (2007)
23. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In: S&P 2006: Proc. 26th IEEE Symposium on Security and Privacy, pp. 258–263. IEEE Computer Society, Los Alamitos (2006)
24. Almgren, M., Debar, H., Dacier, M.: A lightweight tool for detecting web server attacks. In: NDSS 2000: Proc. of 11th ISOC Symposium on Network and Distributed Systems Security (2000)
25. Almgren, M., Lindqvist, U.: Application-integrated data collection for security monitoring. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 22–36. Springer, Heidelberg (2001)