Detection of Web Applications through Language Recognition

Eduardo GimÃĺnez

ICT4V

Abstract The abstract.

Keywords: ...

1 Context and motivations

This work focuses on preventing the misuse of Web Applications. A web application is piece of software that develops a coordinated set of functions based on a client-server arquitecture, in which the client runs on the userâĂŹs web browser. The information between the userâĂŹs client and the application server is transmitted using the HTTP protocol.

The kind of applications in which we are mainly interested are the ones used for e-government purposes. These are rather simple web applications which propose register and consulting functions, mostly made of static HTML. They either displays information to the user, lets the user to fill in a form to register some piece of information, or in order to return some piece of information to the user, such as an HTML page including information in a database, a PDF file or the like.

By its very nature, web applications are designed to be exposed in the Internet. This means that its owner can not govern which user actually connects to the application and therefore are available to any Internet user. This feature makes them a primary target of any attacker that wants to get access to the assets (data bases) that the application manages or to place baits (e.g., fake URLs to his own site) to lure honest users.

The code of a web application may contain vulnerabilities, and it is usually unknown which they are or where are they placed. For instance, a field in some form could be sensible to SQL injection, the database could have been previously infected with an URL that is displayed in the returned HTML page, or any of the vulnerabilities described in [OWASP]. Placed in this context, we are interested in developing automated tools that could help us to prevent the attackers from exploiting those vulnerabilities, or once exploited, can be easily deployed to content the incident minimizing the damages.

Furthermore, our main concern is not fixing the underlying problem, namely, the buggy web application code which contains the vulberabilities. The reason is that searching for those bugs is a time consuming activity and frequently the

application can not remain off-line until it has been successfully completed. For the very nature of the applications we are interested in, downtime has to be minimnized as much as possible. Moreover, in some cases the source code of the web application may not be easily available for inspection, either because the provider who sold it to the organization is reluctant to show the details of its code, or because the organization has no longer trace of the source code itself.

Instead of focusing in the web application code, we are rather interested in virtual patching. Virtual patching is a technique in which a second software component, external to the web application, is placed between the web application and its users. This component, usually called a Web Application Firewall (WAF), intercepts and inspects all the traffic between the server and the clients, searching for those packets that look like part of an attack. Once recognized, the suspicious packets are then processed in a different way (logged, suppressed, derived to a honeypot application or whatever). ModSecurity [ModSec] is an open source, widely used WAF enabling real-time web application monitoring, logging, and access control. It gives access to the HTTP traffic stream, in real-time, along with the ability to inspect it and take any of the aforementioned actions. The actions that ModSecurity undertakes are driven by rules. The administrator specifies rules about the contents of the HTTP packets through regular expressions. ModSecurity intercepts each packet from and to the protected web application. If the packet matches a rule, then the actions specified in that rules are fired. ModSecurity offers a default package of rules for tackling the most usual vulnerabilities spotted in the OWASP Top 10, known as the OWASP ModSecurity Core Rule Set (CRS).

The possibility for the user to specify its own rules rather than using a wired policy is one of the interesting features of ModSecurity, which enables to tailor the WAF to each specific application. However, an approach only based on rules also has some drawbacks. First, rules are a static and rigid by nature. They are well suited for describing known attack scenarios, but they may fail to cover slightly different ones, or even totally new ones. Also, as a side effect, covering all the attack variants usually result in quite complicated regular expressions and rules, which are difficult to understand, maintain and modify. Second, practice has shown that the CRS usually produce false positives. Such situations are handled by tuning the rules through the introduction of exceptions. Exception rules handle specific situations for specific fields or cases which require to allow traffic matching some of the CRS rules. However, this is usually a time consuming and error prone task, which has to be done for each specific web application.

In tradicional networks firewalls and IDS, the approach based on rules has been successfully complemented with other tools based on machine learning, anomaly detection and other statistical approaches which provides higher levels of flexibility and adaptability. Those approaches take advantage of sample data about what the âĂIJnormalâĂİ behavior of the web application is, in order to spot suspicious situations which fall out of this âĂIJnormal useâĂİ (anomalies), and which could correspond to on-going attacks. Our long term purpose is to improve ModSecurity with such anomaly detection techniques.

2 Code injection attacks

A usual use case in e-governement web applications is to ask the user to fill a form wiht data, which is then used to assemble an SQL database query and return the retrieved information to the user. In many programming languages, the programmer assembles the query concatenating strings, including part of the data supplied by the user in the form. If such data is not correctly sanitized, an attacker may take advantage of this mthod to inject a carefully crafted piece of code that discloses unauthorized information or internally damages the application. For example, the form contains a field *user* that the application uses to assemble the following query as the string `query := "SELECT * FROM users WHERE name = '" +` *userName* `+ âĂIJ ';"`. If the field is filled with the data `John Smith`, then the record for John Smith is the users table of the database is returned and displayed. However, should the attacker enter the data `John SmithâĂŹ; DROP TABLE users; SELECT userinfo WHERE âĂŸtâĂŹ=âĂŸt`, the string results in the query `query := “SELECT * FROM users WHERE name = âĂŸJohn SmithâĂŹ;DROP TABLE users;SELECT userinfo WHERE âĂŸtâĂŹ=âĂŸtâĂŹ`, which removes the users table from the application database.

In its most general version, code injection consists in the malicious provision of unexpected input data which modifies the intended purpose of an executable, interpreted piece of code. Beyond SQL, this technique may be applied to many query languages that are usually used in web applications, such as XLDAP, OS commands, XPath, XML parsers, SMTP headers, NoSQL, among others.

3 Objectives

Being more specific, our objective is to extend ModSecurity with anomaly detection features based on probabilistic and statistical models which satisfy the following properties:

1. Increases the effectiveness rate with respect to the CRS, that is, detects more attack scenarios than just using the CRS rules alone. As a first experiment, we only focus on those attacks which exploit the use of user-supplied input without proper validation, such as SQL injection, cross-site scripting, remote file inclusion, etc.
2. Decreases the WAF deployment time, minimizing the amount of false positive and therefore of exceptions rules to be added.

The idea is to reuse ModSecurityâĂŹs core functionalities (packet interception, inspection and processing) but introducing an additional pluggable module for adding anomaly detection and machine learning features.

4 Bayesian Models

From a theoretical point of view, probabilistic methods are used in order to aproximate a computing function $f : X_1 \times X_2 \times \ldots X_n \rightarrow Y$ from a set of

training values $D \subseteq X_1 \times X_2 \times \ldots X_n \times Y$ which partially describe the function as a relation. Those methods provide a computable function that, given a tuple $(a_1, \ldots a_n) \in X_1 \times X_2 \times$ âĂęX$_n$, calculates a probability distribution $P(y = f(a_1, \text{âĂę}.a_n)\text{âĎd}|x_1 = a_1, \text{âĂę}.x_n = a_n)$ over a random variable $y \in Y$ conditioned on the random variables $x_1 \in X_1, \text{âĂę}x_n \in X_n$. The function f is therefore approximated by the value in Y with the highest probability in the distribution.

In order to achieve our objective, the first question to be answered is therefore what the function f to be approximated is. This amounts to establish what are the random variables to be considered in our model.

5 The HTTP Protocol

A web application is essentially a reactive system compliant with the HTTP protocol. HTTP is a simple communication protocol independent of the communication media and based on a client/server arquitecture. The client sends HTTP commands to the server, and the server reacts to them one by one by sending a command response to the client. HTTP is a connectionless protocol: the browser initiates a conection to the server for each command, then disconnects and waits for an answer. The server reconnects to the client each time in order to send the response back. HTTP is also a stateless protocol: each HTTP command is processed and answered independently from how the previously received commands were processed.

HTTP messages are encoded as text strings. On the client side, the HTTP client sends a request to the server specifying a request method (i.e., the type of operation to be perfomed), URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content.

The possible request methods enable to perform read/update/delete operations on the serverâĂŹs contents, as well as and some control operations for handling communications: GET: Retrieve information from the given server using a given URI. HEAD: Same as GET, but transfer the status line and the header section only. POST: Send data to the server using HTML forms. PUT: Replace the target resource with the uploaded content. DELETE: Remove the contents at the given by URI. CONNECT: Establishe a tunnel to the server identified by a given URI. OPTIONS: Describe the communication options for the target resource. TRACE: Perform a message loop back test along with the path to the target resource.

The server contents which are read, updated or deleted are organized into locations arranged in a tree structure, each one specified by a Universe Ressource Identifier (URI).

The HTTP server responds to a received request with a success or error code, followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content.

The HTTP protocol does not impose any structure on a request body. However, as was said in section 1.1, in the kind of web applications we are interested in, most of the interaction with the user consists in presenting HTML forms to be filled and answers to the filled forms. Once filled, HTML forms are transmitted to the server as either GET or POST requests including a form data set. Such data set is a list of pairs (field=value) included in the request body. The âĂIJ=âĂİ symbol is used to separate both components of the pair, even though this is not part of the standard.

6 Attacks considered

[The problem is about mixing programming language constructions into other languages, usually texto or numbers]. [Give an example of each attack] [Ww consider attacks in one step] [Could be seen as a problem of programming language recognition]

7 The CISC2010 dataset

[It is annonymized, consecuences]

8 The model

A first refinement of our problem is to consider the function fâĹűX$_1 \times X_2 \times \ldots X_n \rightarrow Y$ to be approximated as a classification function that tags each HTTP request received as either normal traffic or as part of an ongoing attack. From this point of view, an input random variable xi of the model presented in section 1.3 can be associated to a field of an HTTP request. The arrival of an HTTP request is therefore one sampling of such random variables. On the other hand, the domain Y of the function to be approximated is a finite domain with one value for normal traffic plus a list of values corresponding to a class of possible attacks.

Actually, the model could be refined by considering the request body as a list of $2m$ random variables, each one associated with one of the components of the (field=value) pairs in the request body. Despite being more accurate, this model seems harder to handle in practice. As different URLs have different associated forms, and therefore different set of random variables. Then, in such a model we do not have a single function to approximate, but a family of functions f_{url} of different arities indexed on the URL. Furthermore, the collection of functions to approach would be dependent of each application, and so would be the model.

A simpler approach is to consider each HTTP request field as a piece of text to be processed using document classification techniques. Such a model would be similar to the models used to develop spamming filters: the request field is seen as a bug of words, and a probability is assigned to each word in a dictionary, measuring the likelihood for the request to be an attack (of a give

class) depending on the number of ocurrences of this word in the request body. For instance, in SQL injection attacks, the attacker includes (part of) an SQL sentence in one of the body fields of the HTTP request. Therefore, the occurrence of an SQL keyword such as SELECT or WHERE increments the likelyhood for the request to be an attack. Actually, recognizing an injection attack amounts to recognizing that the HTTP contains a piece of programming language, and classifying which programming language is it from set of given ones (LDAP, SQL, Unix commands, Windows commands, etc.).

8.1 Attribute extraction: tokenization

The process of splitting each dataset document into words and counting the ocurrences of each word in it is usually called *tokenization*. Tokenization is therefore the process of attribute extraction in our model. In natuaral language documents, words are separated by special symbols used as delimiters: blank spaces, commas, dots, etc. The dictionnary to be used is therefore determined by splitting the the sequence of characters into those sub-sequences of symbols ("words") that strictly lay inside the boundaries of two consecutive delimiters (that is, delimiters excluded). In Weka, this functionality is provided by the `StringToWordVector` pre-processing function. However, for several reasons this tokenization approach does not fit well for processing HTTP requests. The relevant words to be spotted may be preceeded or followed for a large collection of characters. The keywords of the programming language which is contaminating the request (SQL, LDAP queries, OS commands, etc.) can be sourrended for may symbols: semicolons, commas, paretheses, white spaces, control returs, tabs, vertical bars, beginning of comments, etc. As a consequence, this tokenization process induces very large dictionnaries (about 80.000 for the CISC2010 dataset), and therefore too many random variables. This amount of random variables poses several efficiency problems for the learning process, the main being that exceeds the heap ressources available in Weka.

A second issue with this kind of tokenization approach is the fact that the numbero of relevant words for spotting an attack may actually be infinite. For instance, the suffix *(myparam=5)* inside an HTTP field is a relevant sequence for detecting an LDAP injection attack, but so does any other sequence obtained replacing myparam for any other parameter and *5* for another value. Moreover, this sequent may be relevant for a given web application which uses *myparam*, but is not for a web application which uses a different parameter identifier. In addition to this, such sequences can be juxtaposed to other similar ones with no deli  r separating them, like in *(myparam1=5)(myparam2=bla)*.

These reasons lead us to the conviction that in the context of classifying programming languages, the notion of word should be abstracted to the more general concept of a *lexical token*, that is, a string satisfying a regular expression, its lexical class. From this point of view, tokenization is nothing but a lexical analysis of the HTTP request, in the sense that compiler theory gives to this term. The notion of word and dictionnary is therefore replaced by a collection of

patterns, each one described by a regular expression, and describing a potentially infinite set of strings.

8.2 Attribute selection

The next step in the design of the model is therefore to determine which are the relevant patterns identifying a piece of programming language inside the HTTP request. Those patterns should satisy at least the following criteria:

- It should be a construction related to an attack. For instance, the keyword WHERE in SQL appears in many attacks, but the keyword XXX is hardly related to an attack.
- It should be long enough to have a low probability of ocurring in a different context, in particular in a valid HTTP request. For instance, the Unix command `cat` is frecuently used in many OS command injection, but it also occurs in many English words and therefore may apear in legal values as well.
- It should not be very long. The longest the sequence, more likely is for the attacker to find a slightly different construction which does the same but does not match the pattern.

A pre-processing tool was developed in Ocaml to perform the lexical analysis pre-processing of the dataset. This tool parses an ARFF file (the format for storing Weka datasetst), identifies the strings matching a collection of lexical tokens according to those criteria, and replace them by a representative string for each lexial class, its *lexical class name*. Each lexical class name is surrounded by the special character "Âą", which does not appear in the whole dataset. This character is then specified to Weka's tokenization process as the single delimiter to be used when separating documents in words. This preprocessing step drastically collapses the number of random variables of the model, as each lexical class name represents thousands of lexical tokens in the dataset.

In order to identify relevant lexical tokens we first studied the public knowledge about the web application attack classes in the dataset from the litterature from [**?**]. Then, we inspected the HTTP requests in the dataset, looking for regular expressions characterising the injected languages and satisfyng the above-mentioned criteria. Starting from a first kernel built by hand (for example, from the programming language keywords), we then tuned each regular expression using two different techniques, until an acceptable accuracy rate is reached. For the tunning process, we split the dataset in a collection of sub-sets, each one formed by the valid HTTP requests and the requests for a given class of attacks. This is possible because the syntax of the programming languages used in the injection attacks are quite different. Once the subsets sufficiently tunned, we re-assemble the whole and looked for interferences or unexpected classifications between attacks. An example of interference is the lexical class of file paths, which is used both in Xpath expressions and in OS commands.

Different methodologies were used for tuning the set of lexical classes. The kernel usually consisted in the keywords or basic lexical constructions of each

query language. If the lexical analysis yield too many lexical tokens ocurring in valid HTTP requests, then more context was added to the lexical tokens corresponding to prefixes or suffixes. For example, the Unix command `cat`, which occurs in many OS commanding attacks, is a string that is also very likely to occur in other contexts. However, in OS commanding attacks, this string is usually preceeded by a semicolon ending a previous command, a vertical bar `|` assembling a Unix filter, a white space, or the file path `/usr/local/bin/`. Extending the lexical token to include such prefixes, the probability of capturing this lexical token in the context of, namely, English text, is much lower. However, describing the constructions of the query language with too much detail usually leads to the opposite case, namely, several lexical tokens are not identifyined during the pre-processing, and therefore not counted in the model. This is the case, for instance, of Xpath expressions containing an Xpath predicate, such as `/clients/xxx[blabla]`. There are too many possible expressions for the predicate in between the square brackts, trying to capture them with a regular expression is hard and usually leaves some of them outside. It is preferable to just consider that as free text, without checking its structure. A different tunning process was to run the tokenization process and use a InfoGain function to rank the information provided by the obtained attributes (words). For instance, the set of OS commands to be included in the OS Commanding class was enlarged using this method, as the name of the command appears in the pieces of text highly ranked by this function. Amazingly, this process was not the most fruitfull one for determining relevant lexical tokens. This process eventually lead us to a small number of classes, xx in total. The resulting lexical classes are presented in appendix XXXX.

A different way of minimizing false positives consist in tuning the HTTP fields that will be processed. Indeed, not any HTTP field can be exploited by the attacker to inject a query. The contents of many of them follow a highly structured pattern, which can be checked using an algorithm instead of using machine learning techniques. For instance, the XXX field shall contain an IP address and nothing else, and this can be positevely checked by the WAF. Furthermore, some of the fileds are not involved in the construction of any kind of query on the web application side, so it does not worth to inspect them for query language injections. Moreover, in the *Query* and *body* fields, the attacks only concern the values inside the assigned parameters of the request, that is, the right hand side of the *ident=value* pairs of whith these fields are made of, not the query parameter names themselves. Hence, it make sense not to consider those parameter names during the tokenization process.

The model used is a purely negative model: it only counts lexical classes that pin up possible attacks, regardless of the rest of the text appearing in the HTTP request. After the tokenization process, all the pieces of text sourrended the spotted lexical classes were discarded. Several reasons uphold this approach. First, to provide a model with as few words as possible speeds up the learning process and avoids Weka running out of memory. Second, most of the remaining material was either useless for identifying attacks or increase the possibility for introducing

overfitting in the model. As an example, consider the cookies or other encrypted material that is random in nature and uniquely associated to a collection of HTTP requests forming an HTTP session. The learning process could associate that when an HTTP request in the dataset has a given cookie, it corresponds to a session initiated by the attacker, and therefore it is likely to contain an attack. However, this information is hardly reusable for classifying future HTTP requests, as the identified cookie is not likely to be assigned again. Finally, as was mentioned before, the dataset was anonymized through a masking process which replaces the information inside valid HTTP requests by random data. This destructs any regularity in the valid HTTP requests that could be learned in order to recognize them. Using true, non anonymized web application data, it could worth to push further the lexical analysis, recognizing other lexical classes such as names, numbers, personal identification numbers, and other types. In the more general case, one could think of the classification problem as a problem of learning the correct datatype (alphabet) of each field and query parameter in valid HTTP requests.

[Ojo: el modelo cuenta las ocurrencias en toda la HTTP request, todos los fields, no solo aquellos que nos interesan. Esto es porque el formato de weka de pares no permite separarlos por field]

[buscar literatura sobre reconocimiento de lenguajes de programaciÃşn, as far as we know there is no much]

[Open questions: From a different perspective, the problema can be modeled using statistical language models and n-gram models introduced by Shannon. The problem to be solved is, having seen a piece of text in the field, which is the probability for the next word to be a âĂŸ or âĂIJ or SELECT or other suspicious words that could point to an SQL injection or other forms of attacks. Can the problem be modeled as Grammar induction? Should we considered isolated HTTP requests or rather the whole session?]

9 Experimental results

Table 1. Confusion Matrix

	SSI	OS Command	SQL Injection	Path Traversal	Valid	LDAP Injection	XXS	XPath Injection
SSI								
OS Command								
SQL Injection								
Path Traversal								
Valid								
LDAP Injection								
XXS								
XPath Injection								

10 Conclusions

Acknowledgments ...

In the bibliography, use `\textsuperscript` for "st", "nd", ...: E.g., "The 2nd conference on examples". When you use JabRef, you can use the clean up command to achieve that.

All links were last followed on October 5, 2014.