UDC 004.492.3

Model of rules for malicious input parameters detection

Oleksandr Korzhenevskyi[1] and Mykola Graivoronskyi[1]

[1] *National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Institute of Physics and Technologyersity, Peremohy ave. 37, Kyiv, 03056, Ukraine*

Abstract
This article is devoted to detection of advanced techniques of malicious input parameter injections and web application firewall (WAF) bypass. The authors have proposed a hierarchical model for detection rules definition, which allows to edit different fragments separately. This model has been implemented with the usage of Backus-Naur form and ANTLR4 (generator of parsers and lexers). The solution has been tested using some popular web application scanners. The testing environment has been created with Python3. The results of research have been compared with the corresponding ones for the existing open source solution – libinjection. The main accent has been made to SQL injcetions and Cross-Site Scripting attacks.

Keywords: Web application security, web application firewall, injection, attack detection, Backus-Naur form, ANTLR4

Introduction

People's lives have always been connected with sharing information. This process is probably the basis for the existence of people in the form of society, not as individuals. Communication has taken place throughout the evolutionary path and has been a constant component of it, but its form has changed and evolved.

The rapid development of technologies of the end of the *XX* – the beginning of the *XXI* century has led to the fact that today new information comes to each of us at every moment. At the current stage of scientific progress, the most powerful technology for data exchange is certainly Internet. With all the comforts and benefits that humanity has received along with new technologies, information security professionals have received new challenges.

Everyone knows for sure that the most vulnerable place in any information system is a person. It is obvious that more qualified users make fewer mistakes and decrease all the risks connected with human factor. Nevertheless, the current situation is the following: anyone can upload data to the network today. However, even if only certain people were allowed to send materials, it would not make the system completely secure. The fact is that the interfaces responsible for the download can be vulnerable. These vulnerabilities often come down to the injection of data of a certain format, which changes the execution of the instructions in a way necessary for the attacker. Most of such vulnerabilities are well examined but researchers always find new ways to exploit existing defense mechanisms.

This article will focus on text parameter injection type attacks, which are usually constructions of a particular server-side language. The authors propose a solution for existing security issues.

1. Problem description

1.1. Attacks on input parameters of web applications

If we look at the list of the most critical security risks represented by so called OWASP Top 10 Project [1], we will find some vulnerabilities related to data input. These are Injection, Cross-Site Scripting (XSS) and XML External Entity. We will focus on the most popular of them in our work: SQL injection and XSS. They are well-known and greatly investigated, therefore they will be great examples to illustrate the purpose of this article.

«SQL injection is an attack in which the SQL code is inserted or appended into application/user input parameters that are later passed to a back-end SQL server for parsing and execution. Any procedure that constructs SQL statements could potentially be vulnerable, as the diverse nature of SQL and the methods available for constructing it provide a wealth of coding options» [2, page 22]. Security professionals define different types of SQL injections. There are probably the most popular ones:

- Boolean-based SQL injections (or 1=1;–);
- Time-based SQL injections (and SLEEP(30);–);
- Union-based SQL injections (1’ UNION ’1’, ’2’;–);
- Stacked queries (1; DROP TABLE users;–).

«XSS is an attack technique that forces a Web site to display malicious code, which then executes in a user’s Web browser» [3, page 68]. Since developers usually use JavaScript on the client side of an application, the features and operators of this language are most commonly used to exploit this kind of vulnerability.

The community of web researchers usually distinguishes such types of Cross-Site Scripting attacks:

- Reflected XSS;
- Stored XSS;
- DOM-based XSS.

1.2. Mechanisms of defense

The described attacks regard the 7th level of the OSI model. This is mostly the area of responsibility of web application firewalls. Other tools like intrusion detection/prevention system, Next Generation Firewalls or Unified Thread Management products use the same approaches for input data analysis.

There are some base techniques or methods for malicious payload detection. They were classified in the best way by Vladimir Ivanov, professional of Positive Technologies, in his investigation [4]. The following approaches were represented:

- Usage of regular expressions;
- Tokenization;
- Score building;
- Anomaly detection;
- Reputation analysis.

Score building and reputation analysis are mostly auxiliary methods that allow ranging risks and threats and rejecting the requests from suspicious sources. Anomaly detection is usually based on Artificial Intelligence, especially Machine Learning. This approach requires some time for gathering large sets of valid and malicious payloads and actually for learning on those data.

Obviously, 2 methods remain. They both represent the signature-based approach. Most modern protection tools use regular expressions as a basic detection mechanism because of simplicity of rules creation process. These rules are special templates consisting of symbols and metasymbols. The main advantage of this approach is the fact that one rule often covers only one definite construction or a group of similar ones.

Tokenization is a process of detection a signature as a sequence of tokens. This method gives an opportunity to detect such attack as XSS and SQL injections rapidly. The most famous library using this method is libinjection that is widely used in open source web application firewalls, e.g. Modesecurity or Nginx Anti-XSS & SQL Injection. The main disadvantage is ability to paste «token breakers», which make a whole construction unrecognized.

1.3. Existing issues

There is a large class of attacks that abuse the breaches of the signature-based approach. We provide the example of some malevolent payload (1) for better understanding.

$$1000\ \mathit{AND\ SLEEP}\ (30);\ -- \tag{1}$$

Regular expressions approach will detect the usage of the word "SLEEP" and mark this input parameter as malicious. Tokenization method will return a sequence of tokens as it is shown on the illustration (figure 1).

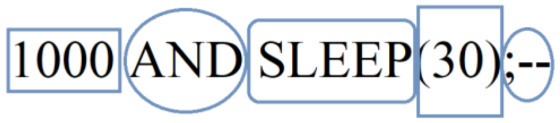

Figure 1: Illustration of tokenization

In this case both methods are able to detect a suspicious input parameter and protect against SQL injections. However let the construction change its shape. Let us have a look at another example (2).

1000 *AND SLEsleepEP* (30); –– (2)

Here is a logical question: what do we have now? Regular expressions still allow us detect a suspicious payload because «sleep» (lower case) will be found. Tokenization in its pure form will not give any result because «SLEEP» (upper case) is broken by «sleep» (lower case). This construction does not make any sense and does not affect an application. However its usage can be quite reasonable, when taking into consideration some security mechanisms that devices and software as WAF can have. Sometimes malevolent requests are not rejected but are filtered and forwarded further on the network. Let us imagine that we use regular expressions and then cut any matching constructions from payload. In this case the parameter (2) turns into (1), and a malicious parameter is sent to a server.

There are some other examples (3) - (6) below that illustrate another problem. We will pay attention to JavaScript code inclusion, which usually takes place while XSS attacks.

alert(“*IPT attacks*”); (3)

al\u0065rt(“*IPT attacks*”); (4)

al&x65rt(“*IPT attacks*”); (5)

al\145rt(“*IPT attacks*”); (6)

All the expressions (3) - (6) are different representations of the well-known construction *alert*(...) used by penetration testers for XSS vulnerabilities detection. We used only some encoding systems and manipulated only with one letter *e*. This gives new ways for attackers to disguise their payloads and makes much head ache for defenders.

During our investigation we tried to improve the tokenization approach and make it detect advanced attacks with the usage of cuttable (possibly cuttable) constructions and different kinds of character representation.

Our goal was not only to protect applications from real threats (which will do harm anyway) but also to reveal attempts of WAF bypass (the ones that will become harmful after filtration, decoding etc).

2. Solution

2.1. Model description

As it is mentioned in the previous section, we can define two main problems, which must be solved, in order to detect some advanced attacks:

- Injection of cuttable constructions
- Usage of different encoding

Simple tokenization technique can be represented in the following way (figure 2). We propose another more flexible model to rules description, which is based on several principles:

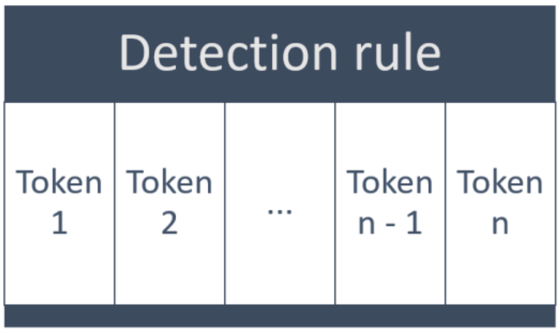

Figure 2: Illustration of tokenization

- Let us define the special token which will represent any construction of the ones that can be cut during the processing of input data. Let us also call this token «*CUTTABLE*».
- Let us build the rule by pasting *CUTTABLE** between each two tokens (figure 3). «*» means that there can be any number of these tokens, or they can be absent at all.

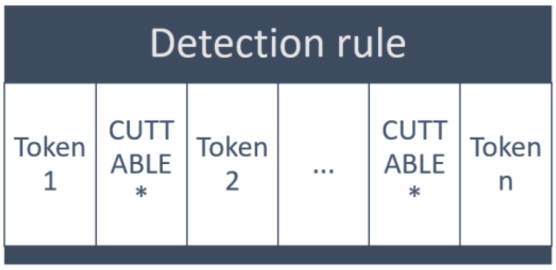

Figure 3: New rule description

- Let us split each token to fragments and paste *CUTTABLE** between each two fragments (figure 4).
- Let us define each fragment like token or as alternative among different representation.

We can see an example of token description («*K*») in the picture (figure 5).

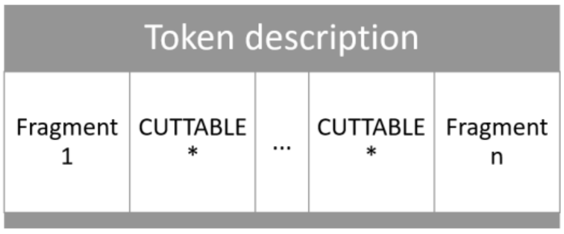

Figure 4: Token description

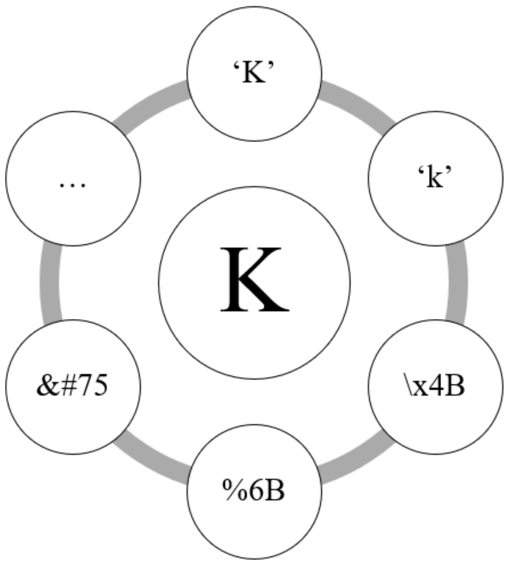

Figure 5: Token description

The main difference between fragment and token is that fragment is always used as a part of token and never independently. As a result we get some hierarchical model for detection rules definition.

2.2. Implementation tools

To implement this model, we decided to use Backus-Naur Form (BNF). BNF is a metalanguage for Context-Free Grammars. It is commonly used for syntax description of programming, query and markup languages. As our aim is to detect attacks like SQL injections and XSS, which contain operators and elements of the aforementioned languages, such kind of notation as BNF becomes very useful in this occasion. There are also some modifications of BNF like EBNF (extended) or ABNF (augmented). They provide some additional features that make rule creation process easier and faster.

To translate created rules into code of some programming language for further implementation in some programs we use such tool as ANTLR4 (the detailed information is available on the website[1]). This is a lexer and parser generator. The result of parsing stage is a special structure – *Abstract Syntax Tree* (*AST*). To investigate this tree, a developer can use Visitor or Listener. The interfaces for these objects are also generated by ANTLR4.

There are 2 main factors that made impact on our decision to choose exactly this tool:

- Programming languages: there are opportunities to generate code for C++, Java, Python, C#, Go, JavaScript and Swift, which are widely used for web application development and server creation.
- Velocity: although the theoretical time complexity of ANTLR4 is $O(n^4)$, the developers of this tool claim that it is much faster than GLR-parsers, which complexity is $O(n^3)$. This was approved in their article [5].

2.3. Examples of rules

We provide some examples of created rules in this subsection. ANTLR4 notation should be composed regarding the following points:

- If one deals with mixed grammar, he or she should define parser rules before lexer rules.
- The most specific constructions must be determined before most common.
- The order of search for rule matching is from left to right, from top to down.

Considering these points, we begin with parser rule for time-based SQL injection with usage of *SLEEP* function (7).

sleep_sqli : SLEEP (*SEPARATOR*|*CUTTABLE*)*
L_PAREN (*SEPARATOR*|*CUTTABLE*)*
NUMBER
(*SEPARATOR*|*CUTTABLE*)* *R_PAREN* (7)

Token *SLEEP* is described in the following way (8).

fragment EP : *E* (*CUTTABLE*)* *P* ;
fragment EEP : *E* (*CUTTABLE*)* *EP* ;
fragment LEEP : *L* (*CUTTABLE*)* *EEP* ;
SLEEP : *S* (*CUTTABLE*)* *LEEP* ; (8)

1 https://www.antlr.org/

Token *CUTTABLE* is defined as an alternative among different constructions like comments, keywords, hashtags, tabulations etc (9).

```
CUTTABLE : (COMMENT | HASHTAG
           | NULL_BYTE | HOR_TAB
           | VER_TAB | LINE_FEED          (9)
           | CAR_RET | QUOTE
           | KEYWORD);
```

KEYWORD also represents an alternative among different keywords. *COMMENT* is defined as anything between /**/ or after # or // with *LINE_FEED* in the end. Other tokens are determined like *HASHTAG* (10). Here we can see hexadecimal, Unicode, HTML, URL and of course ASCII encoding. This list can be supplemented.

```
HASHTAG : ‘#’|‘%23’|‘\\043’|‘\\x23’|‘\\u0023’
        | ‘&#’ ‘0’? ‘0’? ‘0’? ‘0’? ‘0’? ‘35’ ‘;’
        | ‘&#’ [Xx]‘0’? ‘0’? ‘0’? ‘0’? ‘0’? ‘23’ ‘;’      (10)
        | ‘&’ [Nn][Uu][Mm] ‘;’ ?
```

Token *SEPARATOR* is defined as white space in different representations and symbol «+», which is used in URL. *L_PAREN* and *R_PAREN* correspond to «(» and «)» and are determined like (10). Token *NUMBER* is defined as following (11). *POINT* represents «.» and *HEX_DIGIT* is used for digits from 0 to *F*.

```
NUMBER : ‘-’ [1 – 9][0 – 9]* | [0 – 9]+
       | ‘-’ [1 – 9][0 – 9]* POINT [0 – 9]*
       | [0 – 9] + POINT [0 – 9]*                (11)
       | ‘0b’ [01] + | ‘0o’ [0 – 7]+
       | ‘0x’ HEX_DIGIT +
```

Similarly, the rules for some other constructions used for mentioned in the first section types of SQL injections are described.

Regarding XSS attacks, we have a look at something extraordinary. For example, the construction [“*XSS*”].*find*(*alert*) can be used by attackers to find security breaches of JavaScript processing in the web application. We have defined a rule (12) to detect such attempts of exploitation.

```
find_statement : L_BRACK .*? R_BRACK
                 CUTTABLE * FIND
                 CUTTABLE * L_PAREN          (12)
                 .*? R_PAREN
```

L_BRACK and *R_BRACK* correspond to «[» «]» and are defined like (10). The other tokens have already been described before.

We have distinguished the following constructions and written rules for each of them:

- script-tag statement: statement containing <script> or </script> or both of them;
- script statement: construction like javascript:alert(1);
- console statement: console object manipulation;
- popup statement: usage of functions invoking popup windows, e.g., prompt(«XSS»);
- DOM objects injection: usage as input parameter of such constructions as top[«alert»](«IPT»);
- toString usage: usage of function toString(), which allows to disguise malevolent payload, for example «8680439..toString(30)» is equal to «alert»;
- usage of JS functions: injection of some functions like eval, setInterval etc.;
- find statements: another way to disguise input parameters by using specifics of work with arrays in JS; example and the detection rule have been already mentioned above (12).

3. Solution assessment

3.1. Testing environment

The developers of ANTLR4 have created it using Java. They also provide some ready tools for parsing strings from files and visualizing results. Therefore we can evaluate some parameters without creation of any additional programs. You can see the abstract syntax tree (figure 6) for some time-based SQL injection using PG_SLEEP() function (PostgreSQL) and hiding it with injection of cuttable constructions and representation of characters in other encodings.

However our aim was to test our solution in the conditions, closest to the real ones. We have decided to choose Python as we have wanted to make a tool like host-based web application firewall. It means that our solution would be a part of a website back-end. Python provides a lot of different frameworks for web development. We have chosen Flask (i.e. Python3) because of simplicity and high speed of application creation.

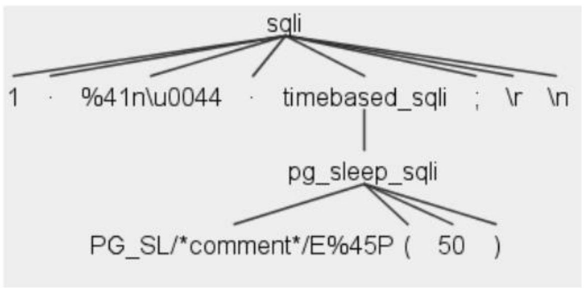

Figure 6: Abstract syntax tree for SQL injection with usage of PG_SLEEP

We have created a simple website with advertisements and a search form. That search form is an entry point, which can be exploited, and which must be protected. Therefore we apply our solution to it.

We have used for testing the following tools:

- sqlmap;
- OWASP ZAP;
- Netsparker;
- Acunetix.

The created application logs any requests into one file and detected attacks into another one. Than we compare these two files to assess the quality of our solution. We have not developed any web application to implement libinjection. We have had log file containing all the requests from scanner. We have used this file as input to console application, written in C/C++. When attack is detected, a message with the corresponding request is written to the output file. We can assess the efficiency of this product by comparing the number of detected attacks with the number of all requests then.

3.2. Analysis of results

We have tested our solution and the libinjection implementation as well. The obtained results are provided in the table 1. The field «Type» stands for type of attack: advanced means that the payload was modified by pasting some cuttable constructions or using encoding. The following 2 columns represent the number of requests, which have been detected as malicious. The last column shows the total amount of made requests.

Obviously, the results are practically identical for unmodified payload of scanners: libinjection detected 81.21% of SQL injections and 22.17% of XSS attempts; our solution detected 81.16% and 23.15%, respectively.

Table 1
Results of testing

Type	libinjection	Our solution	Total
SQLi	10422	10416	12833
SQLi advanced	2060	6783	12035
XSS	45	47	203
XSS advanced	110	150	209

Such low results for XSS are caused by some strange data provided, for example, by Netsparker: it sends «netsparker(1)» instead of «alert(1)» for some reasons. This is also fair for Acunetix. The larger number of instances and usage of sqlmap have provoked such a considerable advantage in the amount of test cases for SQL injections.

We can observe the significant difference in results for advanced attacks: libinjection has 52.63% for XSS and 17.12% for SQLi detection. Our solution was able to find 71.77% and 56.36% attacks respectively. We obtained this «advanced» payload by using the examples from OWASP pages, masking the payload from the scanners with the usage of previously described techniques and filtering it from useless requests (like the one with «netsparker(1)» in the previous paragraph). However to get the most objective results, both solutions should be tested in real systems with a larger number of requests.

Conclusions

The model we suggest and its implementation have demonstrated quite good results for detection of web application scanners activities and its excellence for some advanced payload in comparison with the existing solution. There are some advantages such as simplicity of modifications of existing rules (all rule components can be edited apart) and large set of implementation (once created rules can be translated to 7 languages). The main disadvantage is probably the possibility of REDOS attack due to unlimited usage of token CUTTABLE.

The proposed solution can be used as a component of different tools for web and network security such as Web Application Firewalls, Intrusion Prevention/Detection Systems etc. The current implementation allows to detect malevolent payload and block further query processing. In future it can be modified to filter input parameters by removing suspicious constructions.

The created solution can be used not only for real-time defense but also for forensics purposes, mostly for logs analysis.

References

[1] “Owasp top ten project,” The OWASP foundation, 2017.

[2] J. Clark, SQL Injection Attacks and Defense 2nd Edition. 225 Wyman Street, Waltham, MA 02451, USA: Syngress, 2 ed., 2012.

[3] J. Grossman, Cross Site Scripting Attacks: XSS Exploits and Defense. 30 Corporate Drive Burlington,MA 01803: Syngress, 2007.

[4] V. Ivanov, “Web application firewalls: Attacking detection logic mechanisms,” Positive Techologies, Black Hat USA 2016, 2016.

[5] T. Parr, S. Harwell, and K. Fisher, “Adaptive ll(*) parsing: The power of dynamic analysis,” 2014. Technical report.