MEASUREMENT OF JAVASCRIPT APPLICATIONS' READINESS TO UNTRUSTED DATA USING BAYESIAN NETWORKS

Ekincan UFUKTEPE

Izmir Institute of Technology

July, 2014

MEASUREMENT OF JAVASCRIPT APPLICATIONS' READINESS TO UNTRUSTED DATA USING BAYESIAN NETWORKS

A Thesis Submitted to
The Graduate School of Engineering and Sciences of
Izmir Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

In Computer Engineering

by
Ekincan UFUKTEPE

July 2014

İZMİR

We approve the thesis of **Ekincan UFUKTEPE**

Examining Comittee Members:

_____

Assist. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering, Izmir Institute of Technology

_____

Prof. Dr. Fevzi BELLİ
Department of Computer Engineering, Izmir Institute of Technology

_____

Prof. Dr. Yaşar Güneri ŞAHİN
Department of Software Engineering, Izmir University of Economics

14 July 2014

_____

Assist. Prof. Dr. Tuğkan TUĞLULAR
Supervisor, Department of Computer Engineering
Izmir Institute of Technology

_____

Prof. Dr. Halis PÜSKÜLCÜ
Head of Department of Computer Engineering

_____

Prof. Dr. R. Tuğrul SENGER
Dean of Graduate School of Engineering and Science

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Asst. Prof. Dr. Tuğkan TUĞLULAR, for his guidance, encouragement and teaching me the concepts of computer security. Furthermore, sharing his industrial and academic experiences has given me a priceless perspective to the outside world.

I would also like to express my gratitude to my professor Dr. Burak Galip ASLAN, for introducing me the concepts of Bayesian Networks and giving me some advice on constructing my Bayesian Network.

I am also grateful to my colleagues in Izmir Institute of Technology Department of Computer Engineering, especially Gürcan GERÇEK. It was a great pleasure to work with smart, but also fun people.

Finally, I would like give my greatest gratitude to my dad Prof. Dr. Ünal UFUKTEPE, my mother Günnur UFUKTEPE and brother Eren UFUKTEPE, by supporting me and always forcing me never to give up. I can never forget the things what they have done for me.

ABSTRACT

MEASUREMENT OF JAVASCRIPT APPLICATIONS' READINESS TO UNTRUSTED DATA USING BAYESIAN NETWORKS

Web applications have become an integral part of our daily lives. People mostly provide their important needs, such as people keep their private data, do their banking transactions, shopping etc. through web applications. Therefore, web applications have been an attractive target to malicious individuals and organizations.

The usage of JavaScript language by web application developers is increasing very fast, especially after JavaScript started to service back-end developers as well. Therefore, JavaScript has incorporated both front-end and back-end developers. Concurrently, due to flexibility and its most popular library called jQuery, JavaScript has become an attractive to web application developers.

OWASP updates the top 25 security vulnerabilities regularly. According the results, SQL Injection (CWE-89) and Operating System Command Injection (CWE-78) has taken the 1st place and Cross-Site Scripting (XSS) (CWE-79) has taken the 3rd place. The results shows that three input validation based vulnerabilities appear in the top three; therefore, it can be said that input validation vulnerabilities have become critical vulnerabilities of web applications. However, developers still fail to validate the inputs or use libraries to protect their web applications against input validation vulnerabilities.

In this thesis, JavaScript application's functions are analyzed to determine if their parameters are validated or not. Then, according to the invalidated inputs, a Bayesian Network to measure its readiness to input validation vulnerabilities is generated.

ÖZET

JAVASCRIPT UYGULAMALARININ GÜVENİLİR OLMAYAN VERİLERE KARŞI HAZIRLIĞININ BAYESİAN AĞLARI İLE ÖLÇÜLMESİ

Günümüzde web uygulamaları hayatımızın ayrılmaz bir parçası olmuş durumda. Kullanıcılar birçok önemli ihtiyaçlarını web uygulamaları aracılığı ile yapmaktadır. Örneğin; şahsi veya özel bilgilerini saklamak, banka işlemlerini, alışverişlerini yapmak vb. Bu yüzden web uygulamaları saldırılmak için, hacker'lar gibi kötü niyetli şahıslara çok cazip gelmektedir.

Ek olarak, JavaScript dilinin kullanımı özellikle arka-uç geliştiricilerine de servis vermeye başladıktan sonra, web uygulamaları geliştiricileri tarafından kullanımı hızlıca artmaya başlamıştır. Böylece JavaScript hem ön-uç hem de arka-uç geliştiricilerini tek çatı altına toplamaya başarmıştır. Bununla beraber, JavaScript'in sağladığı esneklikler ve jQuery gibi bir kütüphanenin popüler olması, web uygulamaları geliştiricileri için cazip olmuştur.

OWASP'ın son güncellemeleri bize ilk 25 güvenlik zafiyetini vermiştir. Sonuçlara göre, SQL enjeksiyonları (CWE-89) ve işletim sistemleri komut enjeksiyonları (CWE-78) listenin 1. ve XSS (CWE-79) saldırıları 3. sırada yer almıştır. Sonuçlar bize listenin ilk 3 sırasında 3 girdi zafiyetini göstermiştir. Böylece, girdi zafiyetlerinin bizim için kritik olduğu söyleyebiliriz. Ancak, geliştiriciler halen girdilerini doğrulama ya da girdi zafiyetlerine karşı koruma sağlayabilecek kütüphaneleri kullanmakta başarısız oluyorlar.

Bu tezde, JavaScript uygulamalarındaki fonksiyonların parametrelerinin doğrulanıp doğrulanmadığı analiz ediyoruz. Daha sonra, doğrulanmayan girdilere göre Bayesian Ağı oluşturarak, güvenilir olmayan verilere karşı hazırlığını ölçüyoruz.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

The usage of web applications has been increasing very fast in recent years since it provides great conveniences to users. On the other hand these conveniences include some critical processes like online banking, storing personal information etc.

Similarly, JavaScript usage in web applications is also increasing since JavaScript used to address only front-end developers. However, it also addresses back-end developers as well, now. Therefore, JavaScript has added both front-end and backend developers to its users. Additionally, JavaScript is used by many popular web applications such as Google, Facebook, Twitter, YouTube, LinkedIn, etc. According to October 2013's results of programming language popularity [17], by the normalized statistics from; Github Repositories, Google Files, Ohloh, Craiglist and Google Search, JavaScript language has taken the 4th place.

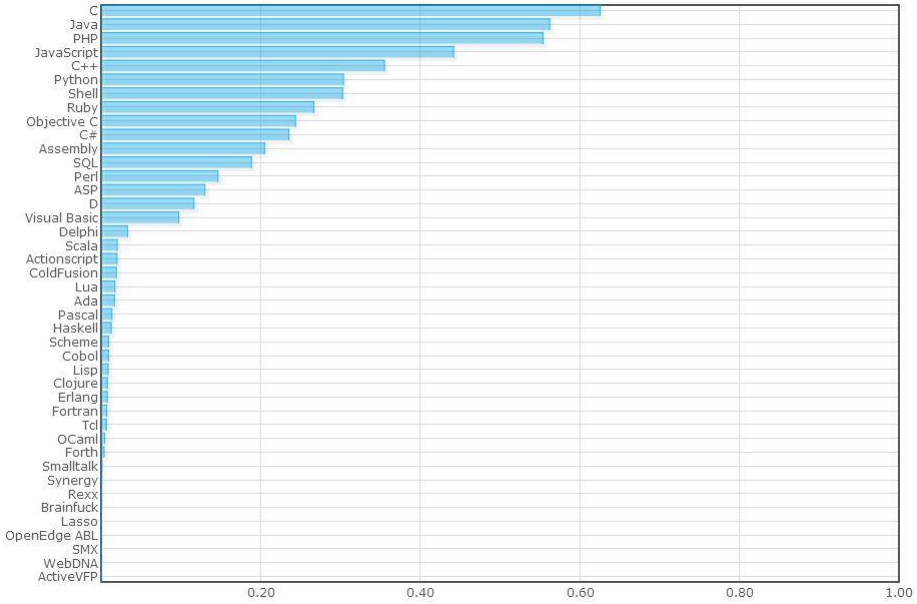


Figure 1.1. Table of Programming language popularity [17].

Not only the increasing popularity of JavaScript, but also a JavaScript library called jQuery is used by many developers, since it has useful functions and it provides conveniences to the developers. JavaScript and jQuery are becoming attractive for web

application developers, so it gives us a motivation to focus on both language and library.

Input validation vulnerabilities are introduced during the development phase of web applications. It occurs when the developers forget or fail to validate their inputs. Faults make a web application vulnerable against input validation vulnerabilities, such as; XSS, SQL injection, code injection and more related attacks that are based on inputs.

Measurement of a specific software or program quality has always been a challenging problem. According to Gilb's principle[1], "Every measurement action must be motivated by a particular goal or need that is clearly defined and easily understandable". As we define a goal of software to measure it, including a lot of attributes, which makes the measurement process complex. As part of software quality, this also includes measuring how much the software is secure as well. To get better measurement for the software we use Bayesian Networks where we use statistics, probability theory and cause and effect relations on each function that accepts input as parameters.

In this thesis, first our works on policy generation to prevent input validation vulnerabilities on server-side web applications are described. We experimented on PHP, server-side Java and JavaScript web applications. To prevent input validation vulnerabilities, we generate a type based policy on inputs that we analyzed. Then, before the request is sent to the server, we check the request value of the input and check with our generated policy if it satisfies the constraints. Later on, we aimed to use static analysis and Bayesian Network generation according to the policies we have generated.

We focus on statically detecting input validation vulnerabilities of JavaScript applications and measure their readiness against untrusted data by using Bayesian Networks. Finally, according to the measurement results, developers will be able to see the cause and effect relationships residing in their applications. Based on the measurement results, the developers can improve their applications and they can follow a new strategy to prevent input validation vulnerabilities.

[1] Gilb, T., & Finzi, S. (1988). *Principles of software engineering management*(Vol. 4). Reading, MA: Addison-Wesley.

CHAPTER 2

RELATED WORKS

In this chapter, the literature summary about the measurement of JavaScript applications' readiness to untrusted data by using Bayesian networks is given.

Static code analysis is an analysis that is performed at the implementation phase of security development lifecycle over a source code of a program. It is used to detect potential vulnerabilities over within a source code without running it. Static analysis uses techniques such as; data flow analysis, control flow graph (CFG), taint analysis, lexical analysis [11].

Data flow analysis is used to collect run-time (dynamic) information about data in software while it is in a static state. Control flow graph is an abstract graph representation of software by use of nodes that represent basic blocks. A node in a graph represents a block; directed edges are used to represent jumps (paths) from one block to another. If a node only has an exit edge, this is known as an "entry" block, if a node only has an entry edge, this is known as an "exit" block. Taint Analysis attempts to identify variables that have been "tainted" with user controllable input and traces them to possible vulnerable functions also known as a "sink". If the tainted variable gets passed to a sink without first being sanitized, it is flagged as vulnerability. Lexical Analysis converts source code syntax into "tokens" of information in an attempt to abstract the source code and make it easier to manipulate [11].

In 2009, Jensen et al [18] presented a static analysis program for JavaScript called TAJS that performs a type analysis on JavaScript code. The type analysis is performed on a lattice of "Value" and from transfer functions that have been derived from a dataflow analysis and flow graph. For example, if a function is called with a "*toString()*" function, they came up with an inference of the data type of "String". In this work, they claimed that, their type analyzer is the first sound and detailed tool of JavaScript code. The use of monotone framework with an elaborate lattice structure, combined with recent abstraction, results in an analysis with good precision on demanding benchmarks.

In 2011, Jensen et al [36] proposed lazy propagation as a technique for flow- and context-sensitive interprocedural analysis of programs with objects and first-class functions where transfer functions may not be distributive. The technique is described formally as a systematic modification of a variant of the monotone framework and its theoretical properties are shown. It is implemented in a type analysis tool for JavaScript where it results in a significant improvement in performance.

In 2012, Jensen et al [35] presented an approach to soundly and automatically transform many common uses of *eval* into other language constructs to enable sound static analysis of web applications. By eliminating calls to *eval*, they expanded the applicability of static analysis for JavaScript web applications in general. The transformation they proposed works by incorporating a refactoring technique into a dataflow analyzer. They reported on their experimental results with a small collection of programming patterns extracted from popular web sites. Although there are inevitable cases where the transformation must give up, their technique succeeded in eliminating many nontrivial occurrences of *eval*.

In 2009, Chugh et al [26] presented an information-flow based approach to infer the effects that a piece of JavaScript has on the website in order to ensure that key security properties are not violated. To handle dynamically loaded and generated JavaScript, they proposed a framework for staging information flow properties. Their framework propagates information flow through the currently known code in order to compute a minimal set of syntactic residual checks that are performed on the remaining code when it is dynamically loaded. They have implemented a prototype framework for staging information flow. They described their techniques for handling some difficult features of JavaScript and evaluated their system's performance on a variety of large real world websites. Their experiments show that static information flow is feasible and efficient for JavaScript, and that their technique allows the enforcement of information-flow policies with almost no run-time overhead.

In 2009, Guarnieri et al [24] identified a class of vulnerabilities, *cross-origin JavaScript capability leaks*, that arise when the browser leaks a JavaScript pointer from one security origin to another. These vulnerabilities undermine the same-origin policy and prevent Web sites from securing themselves against Web attackers. They presented an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the "points to" relation between JavaScript objects in the JavaScript heap. They implemented their detection algorithm in WebKit and used it to find new cross-origin JavaScript

capability leaks by running the WebKit regression test suite in their instrumented browser. Having discovered these leaked pointers, they turned their attention to exploiting these vulnerabilities. They constructed exploits to illustrate the vulnerabilities and find that the root cause of these vulnerabilities is the mismatch in security models between the DOM, which uses access control, and the JavaScript engine, which uses object-capabilities. Instead of patching each leak, they recommend that browser vendors repair the underlying architectural issue by implementing access control checks throughout the JavaScript engine. Although a straight-forward implementation that performed these checks for every access would have a prohibitive overhead, they demonstrated that a JavaScript engine optimization, the inline cache, reduces this overhead to 1–2%.

In 2010, Guarnieri et al [25] addressed that static analysis is a useful technique for applications ranging from program optimization to bug finding. Their paper explores staged static analysis as a way to analyze streaming JavaScript programs. They advocated the use of combined offline-online static analysis as a way to accomplish fast, online analysis at the expense of a more thorough and costly offline analysis on the static code. The offline stage may be performed on a server ahead of time, whereas the online analysis would be integrated into the web browser. Through a wide range of experiments on both synthetic and real life JavaScript code, they found that in normal use, where updates to the code are small, they could update static analysis results within the browser quickly enough to be acceptable for everyday use. They demonstrated this form of staged analysis approach to be advantageous in a wide variety of settings, especially in the context of mobile devices.

In 2013, Madsen et al [23] presented an approach that combines traditional pointer analysis and a novel use analysis to analyze large and complex JavaScript applications. They experimentally evaluated their techniques based on a suite of 25 Windows 8 JavaScript applications, averaging 1,587 lines of code, in combination with about 30,000 lines of stubs each. The median percentage of resolved calls sites goes from 71.5% to 81.5% with partial inference, to 100% with full inference. Full analysis generally completes in less than 4 seconds and partial in less than 10 seconds. They demonstrated that their analysis is immediately effective in two practical settings in the context of analyzing Windows 8 applications: both full and partial find about twice as many WinRT API calls compared to a naive pattern-based analysis; in their auto-

completion case study they out-perform four major widely-used JavaScript IDEs in terms of the quality of autocomplete suggestions.

A Bayesian network is a graphical model that encodes probabilistic relationships among variables of interest. When used in conjunction with statistical techniques, the graphical model has several advantages for data analysis. In 1996, Heckerman [16] defined these as follows:

1) Because the model encodes dependencies among all variables, it readily handles situations where some data entries are missing.
2) A Bayesian network can be used to learn causal relationships, and, hence, can be used to gain understanding about a problem domain and to predict the consequences of intervention.
3) Because the model has both a causal and probabilistic semantics, it is an ideal representation for combining prior knowledge (which often comes in causal form) and data.
4) Bayesian statistical methods in conjunction with Bayesian networks offer an efficient and principled approach for avoiding the over fitting of data.

Measuring how secure an application is, is a difficult process due to the relativity of attacks, application's functionality, application's architecture etc. Therefore, while measuring security as quality property of an application that researchers who work in this area generally focus on specific attack types, or the application which is written in programming language, and etc.

Moreover, tools that measure the security as quality property of an application are used as a recommendation system. Thus, these tools give an approximate perspective on application secureness. They might include Type I (false positive) and Type II (false positive) errors, so they never give an exact measurement.

In 2008, Frigault et al [37] propose to model probability metrics based on attack graphs as a special Bayesian Network. This approach provides a sound theoretical foundation to such metrics. It can also provide the capabilities of using conditional probabilities to address the general cases of interdependency between vulnerabilities.

In 2008, Frigault et al [38] in their previous work [37] explored the causal relationships between vulnerabilities encoded in an attack graph. However, the evolving nature of vulnerabilities and networks has largely been ignored. In this paper, they proposed a Dynamic Bayesian Networks (DBNs)-based model to incorporate temporal factors, such as the availability of exploit codes or patches. Starting from the model,

they studied two concrete cases to demonstrate the potential applications. This novel model provides a theoretical foundation and a practical framework to continuously measure network security in a dynamic environment.

In 2010, Kondakçı [15] proposed a network security risk assessment model by using Bayesian Belief Networks (Bayesian Networks). In this work, he introduced a generic threat model that can also be applied to risk computation of various types of IT assets and dependable computing environments. He modeled the classification of information security threats (human-related, internal and external) as a compound structure with four dependable parameters. He also developed a new risk propagation model using the conditional probability method and the average score scheme, by which risk levels can be easily estimated and quantified for different assessment systems. Furthermore, he mentioned the reason for using Bayesian Networks, which is its usage of representing knowledge and developing automated reasoning systems. The traditional inference methods are difficult to apply to determine posterior distributions of risk factors in dynamically changing IT environments. Therefore, dependence analysis in large scale networks can be easily performed by applying Bayesian approaches.

In 2010, Wagner [12] used Bayesian Networks to assess and predict software quality by using activity-based quality models. To construct the Bayesian Network, Wagner defined three types of nodes and follows four steps, to generate his network. The first step is a goal-based derivation of relevant activities and their indicators. The second is the step where we identify the facts and the sub-activities. The third step is where we add suitable indicators for the facts that are included. In the fourth step, we define the node probability tables to show quantitative relationships. The nodes are created by a map of software's "Situations and Activities". "Activities" holds information that has an influence on the activity i.e. anything that is done with the system. For example, *Maintenance* and *Use* are high-level activities. The "Situations" contains, for example, the *System*, its *Environment* and the development *Organization*. For each situation that has a positive or a negative effect on an activity, add a node to the facts. If the situation does not have any effect on the activity, do not add a node. However, do not forget to also add a node for each activity which is defined to observe the sub-activities and facts influences on the activities.

CHAPTER 3

OVERVIEW ON VULNERABILITIES

3.1. Analyzing Vulnerabilities

In 2013, OWASP announced the top 10 vulnerabilities [7]. Injection flaws, such as SQL Injections and OS command injections have taken the first place, on the other hand Cross-Site Scripting (XSS) errors have taken the third place, and these gave us the motivation to focus on input validation vulnerabilities. Furthermore, to support our motivation we used data and reported vulnerabilities from National Vulnerability Database (NVD) from January 2000 to January 2014 [8]. In Table 3.1, it can be seen that most of the input validation vulnerabilities appear in the top ten.

Table 3.1. Distribution of Reported Vulnerabilities since Jan. 2000 – Jan. 2014 (OWASP[7], NVD[8]).

OWASP Code	CWE Code	Vulnerability	Decision Tree Acronyms	Percentage (NVD)	Input Validation Vulnerability
-	CWE-119	Buffer Errors	BUFF	12,69	NO
A3	CWE-79	Cross-Site Scripting (XSS)	XSS	12,64	YES
-	No Mapping	Insufficient Information	-	12,29	NO
A1	CWE-89	SQL Injections	SQL	10,07	YES
A7	CWE-264	Permissions, Privileges and Access Control	PPA	8,5	NO
-	CWE-20	Input Validation	CCC	7,35	YES
-	CWE-399	Resource Management Errors	RES	5,54	NO
A1	CWE-94	Code Injection	CODE	5,01	YES
A4	CWE-22	Path Traversal	TRAV	4,17	YES
-	CWE-200	Information Leak / Disclosure	INFO.LEAK	4,07	NO
-	No Mapping	Other	-	3,06	NO
-	CWE-189	Numeric Errors	NUM	2,96	NO
A2	CWE-287	Authentication Errors	AUTHENT	2,2	NO
-	No Mapping	Design Error	-	1,91	NO

(cont. on next page)

Table 3.1. (cont.)

A8	CWE-352	Cross-Site Request Forgery (CSRF)	WEB.CSRF	1,77	NO
A6	CWE-310	Cryptographic Issues	CRYPTO	1,48	NO
-	CWE-255	Credential Management	-	1,12	NO
-	CWE-59	Link Following	PATH.LINK	0,93	NO
A5	CWE-16	Configuration	-	0,76	NO
-	CWE-362	Race Conditions	RACE	0,75	NO
-	CWE-134	Format String Vulnerability	FORMAT	0,42	NO
A1	CWE-78	OS Command Injections	OS	0,3	YES
-	No Mapping	Not In CWE	-	0,02	NO

The vulnerabilities have been selected according to the ones that follow an input validation strategy to avoid such attacks. We look into each vulnerability's CWE code given in Table 3.1 and search that uses an "input validation" strategy to mitigate the attacks. After this search we decided on the input validation vulnerabilities which are; XSS, SQL Injection, OS Command Injection, Input Validation, Code Injection, and Path Traversal.

We eliminate the other vulnerabilities to see the distribution of commonness of input validation vulnerabilities in Table 3.2 when we decide which vulnerabilities are input validation type vulnerabilities. For this reason, we are able to use these values as an impact factor/weight for our Bayesian Network we will construct in chapter 6.

Table 3.2. Distribution of Reported Input Validation Vulnerabilities since Jan. 2000 - Jan. 2014 (NVD[8])

Input Validation Vulnerability	**Total Reported Vulnerabilities**	**Percentage**
XSS	4584	32%
SQL Injection	3652	25%
Input Validation	2665	19%
Code Injection	1818	13%
Path Traversal	1511	10%
OS Command Injection	108	1%
TOTAL	14338	100%

3.2. Classification of Input Validation Vulnerabilities

We use Christey's [9] work to classify input validation vulnerabilities to extract an input validation decision tree. In Figure 3.1, we only select the nodes that are related to input validation we discussed in section 3.1. In Figure 3.2, we consider Code Injection (CODE) vulnerability as an Injection Problem (INJ), therefore we assign CODE node as a child node of INJ. In the last form of our tree (see Figure 3.2), we have an input validation vulnerability decision tree.

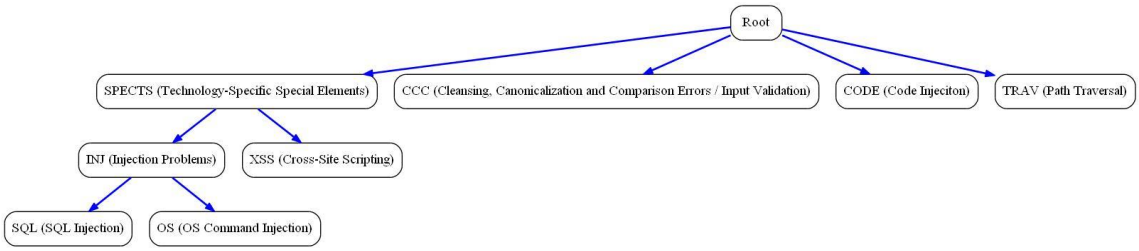


Figure 3.1. Input Validation Vulnerability Tree

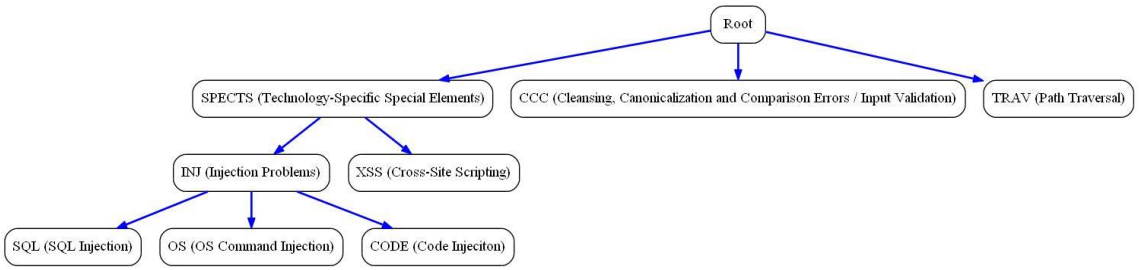


Figure 3.2. Updated Input Validation Vulnerability Tree

3.2.1. Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it [1].

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page [1].

3.2.2. SQL Injection

A SQL injection attack consists of an insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands [2].

3.2.3. Code Injection

Code injection is the general term for attack types which consist of injecting code that is then interpreted/ executed by the application. This type of attack exploits poor handling of untrusted data. These types of attacks are usually made possible due to a lack of proper input/output data validation, for example:

- allowed characters (standard regular expressions classes or custom)
- data format
- amount of expected data

Code injection differs from command injection in that an attacker is only limited by the functionality of the injected language itself. If an attacker is able to inject PHP code into an application and have it executed, he is only limited by what PHP is capable of. Command injection consists of leveraging existing code to execute commands, usually within the context of a shell [3].

3.2.4. Operating System Command Injection

The purpose of the command injection attack is to inject and execute commands specified by the attacker in the vulnerable application. In such a situation, the application, which executes unwanted system commands, is like a pseudo system shell, and the attacker may use it as any authorized system user. However, commands are executed with the same privileges and environment as the application has. Command injection attacks are possible in most cases because of lack of correct input data validation, which can be manipulated by the attacker (forms, cookies, HTTP headers etc.) [4].

There is a variant of the code injection attack. The difference with code injection is that the attacker adds his own code to the existing code. In this way, the attacker extends the default functionality of the application without the necessity of executing system commands. Injected code is executed with the same privileges and environment as the application has [4].

An OS command injection attack occurs when an attacker attempts to execute system level commands through a vulnerable application. Applications are considered vulnerable to the OS command injection attack if they utilize user input in a system level command [4].

3.2.5. Input Validation

The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program. When the software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution [5].

The "input validation" term is extremely common, but it is used in many different ways. In some cases its usage can obscure the real underlying weakness or otherwise hide chaining and composite relationships. Some people use "input validation" as a general term that covers many different neutralization techniques to

ensure that the input is appropriate, such as filtering, canonicalization, and escaping. Others use the term in a narrower context to simply mean "checking if an input conforms to the expectations without changing it." [5].

3.2.6. Path Traversal

A Path Traversal attack aims to access files and directories that are stored outside the web root folder. By browsing the application, the attacker looks for absolute links to files stored on the web server. By manipulating variables that reference files with “dot-dot-slash (../)” sequences and its variations, it may be possible to access arbitrary files and directories stored on file system, including application source code, configuration and critical system files, limited by system operational access control. The attacker uses “../” sequences to move up to root directory, thus permitting navigation through the file system [6].

This attack can be executed with an external malicious code injected on the path, like the Resource Injection attack. To perform this attack it’s not necessary to use a specific tool; attackers typically use a spider/crawler to detect all URLs available. This attack is also known as “dot-dot-slash”, “directory traversal”, “directory climbing” and “backtracking” [6].

CHAPTER 4

POLICY GENERATION TO PREVENT INPUT VALIDATION VULNERABILITIES

In this chapter we describe our work on preventing input validation vulnerabilities on server-side web applications by data type analysis. The basis of this work comes from Scholte et al [34], "Preventing input validation vulnerabilities in web applications through automated type analysis". They developed a tool to called IPAAS (**I**nput **PA**rameter **A**nalysis **S**ystem) that automatically analyzes the data types of a PHP application and generates an input validation policy to prevent input validation vulnerabilities. In the following parts of this chapter, we introduce IPAAS and describe the improvements that I have made on IPAAS.

4.1. IPAAS

In this section, we introduce IPAAS, an Eclipse Plug-in to securing web application against Cross Site Scripting and SQL Injection attacks by using input validation. Normally IPAAS, was only developed to prevent input validation vulnerabilities for PHP web application, however, with our improvements on IPAAS, we aimed to prevent multiple languages such as Java and JavaScript web application, also including hybrid web application that includes many languages. In IPAAS, we succeed in adapting the Java web application, but not JavaScript, due to its dynamic nature. This has given us the motivation to target JavaScript separately.

IPAAS can be separated into three parts; parameter extraction, learning data types and runtime enforcement as seen in Figure 4.1.

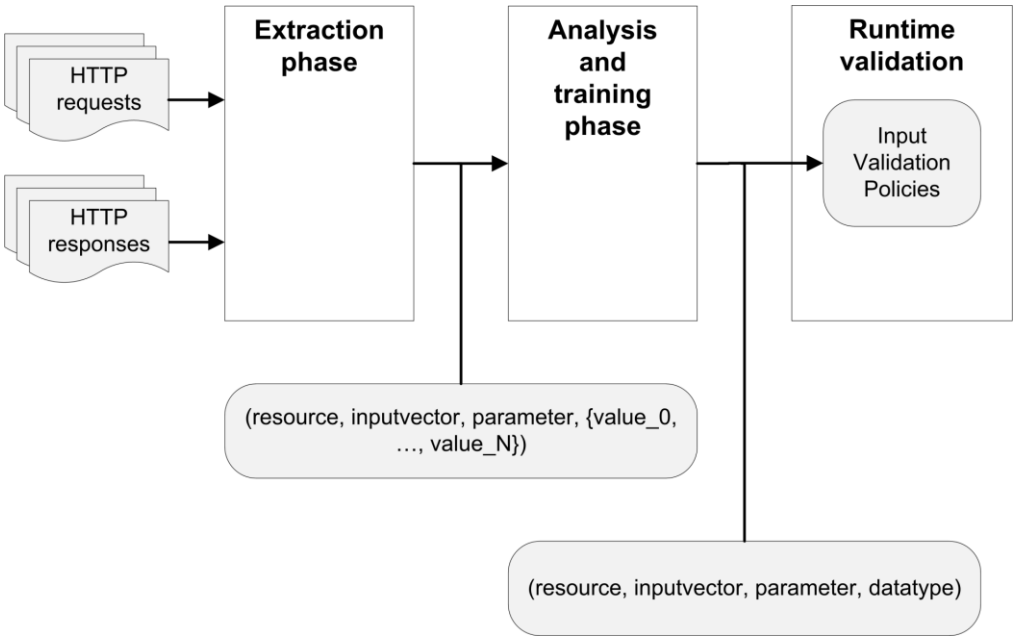


Figure 4.1.The Work-flow of IPAAS [34].

4.1.1. Parameter Extraction

First, data of parameters from the web applications are collected. Here, a proxy server has been used to intercept the HTTP requests and responses between the client and the web application during test. Then the HTTP messages that are been intercepted are parsed. For each request, all observed parameters are parsed into key-value pairs, associated with the requested resource, and stored in a database. Furthermore, each response containing an HTML document is processed by an HTML parser that extract the links and forms that have targets associated with the application that is under test. For each link containing a query string, key-value pairs are extracted similarly to the case of requests. For each form, all input elements are extracted. In addition, those input elements that specify a set of possible values (e.g., select elements) are traversed to collect those values [34].

4.1.2. Learning Data Types

In the second phase, each extracted parameter is observed according to its values obtained during testing. The labeling process for parameters is done by applying a set of validators.

Table 4.1. IPAAS types and their validators.

Type	Validator
Boolean	(0|1)|(true|false)|(yes|no)
Integer	(+|-)?[0-9]+
Float	(+|-)?[0-9]+(\+.[0-9]+)?
URL	*RFC 2396, RFC 2732*
Token	*static set of string literals*
Word	[0-9a-Za-Z@_-]+
Words	[0-9a-Za-Z@_- \r\n\t]+
free-text	*None*

```
FOREACH par IN parameters:
    int[] scores := NEW int[validators.length]
    string[] values := getValues(par)
    FOREACH value IN values:
        FOREACH validator IN validators:
            IF validate(validator, value) THEN
                scores[validator] :=
        scores[validator] + 1
    dataType := getValidatorWithHighestScore(scores)
```

Figure 4.2. Psuedo code of Data Type Learning.

In this phase, each parameter is assigned to a score vector with a length equal to the number of validators. Then each recorded value of a parameter is passed to a validator. If the validator accepts the value, the score is incremented. The data type with the highest score was chosen after all the values for a given parameter were processed. In Figure 4.3 we can see the work flow of the analysis and training phase of IPAAS.

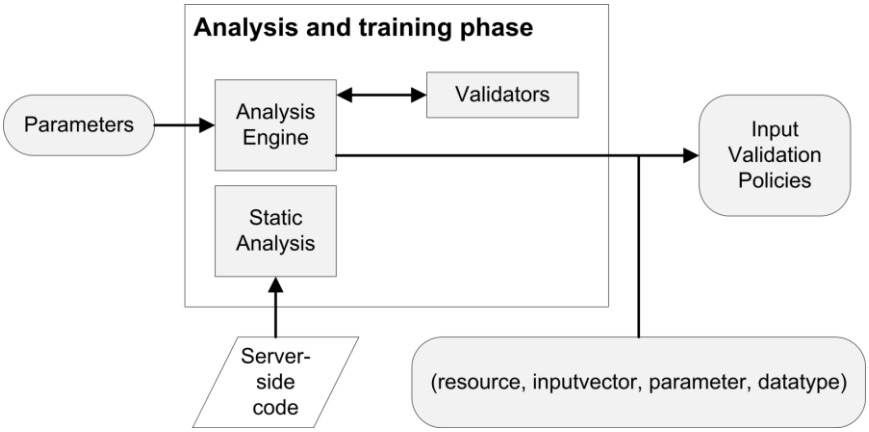


Figure 4.3. Work flow of analysis and training phase of IPAAS [34].

4.1.3. Runtime Enforcement

In our study, we focused on server-side PHP and Java web application. These two different types of projects' working mechanisms are different. We have to use different approaches for runtime enforcement.

The first two steps were performed during the test of web applications. However, in the third phase we performed it during deployment process. At runtime, IPAAS intercepts the incoming requests and checks each request with a validation policy created by IPAAS. If the input parameter does not satisfy the constraints of the policy, then IPAAS discards the input. On the other hand, if the input parameter satisfies the constraints of the policy the execution will continue.

4.2. Existing Features of IPAAS

Since we improved IPAAS and added some new features to it, first of all, explain the existing features of IPAAS and then explain the new features we integrated into IPAAS.

4.2.1. Data Type Analysis

This feature of IPAAS, dynamically analyzes the web application projects. It uses its test cases, which are held in a database. This database has two tables; parameters and requests.

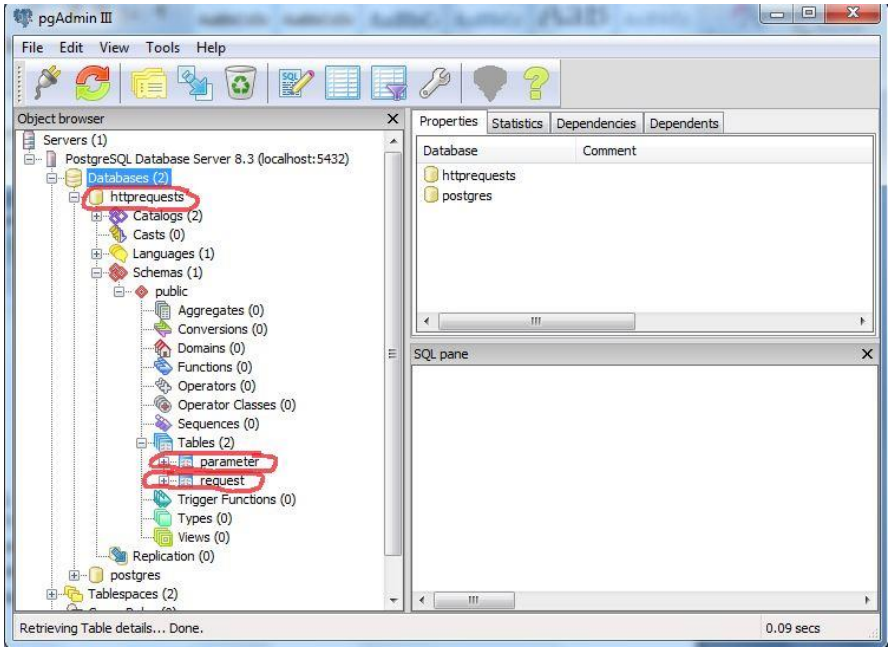


Figure 4.4. Example view of PostgreSQL

- **Parameters:** This table contains HTML parameter information. For example: Type (radio button, checkbox, combo box, etc.), value, name and cookie.
- **Requests:** This table contains information about which path has a POST and GET methods.

	requestid integer	name character var	value character var	type character var	cookie boolean
9368	2564	params[pagecla		single_line_text	FALSE
9369	2564	params[secure]	-1	radio	FALSE
9370	2564	params[secure]	0	radio	FALSE
9371	2564	params[secure]	1	radio	FALSE
9372	2564	link	index.php?optio	hidden	FALSE
9373	2564	option	com_menus	hidden	FALSE
9374	2564	id	53	hidden	FALSE

Figure 4.5. Example of Parameters Table.

	id [PK] integer	path character varying	method character varying
49	49	/joomla/installation/index.php	GET
50	50	/joomla/installation/index.php	POST
51	51	/joomla/installation/index.php	GET
52	52	/joomla/installation/index.php	GET
53	53	/joomla/installation/index.php	POST
54	54	/joomla/installation/index.php	POST
55	55	/joomla/installation/index.php	GET
56	56	/joomla/installation/index.php	POST

Figure 4.6. Example of Requests Table.

4.2.1.1. Perform Data Type Analysis

In Figure 4.7, the developer requests a Data Type Analysis and receives the Data Type Analysis results of the project which is uploaded. (See Figure 4.8 for an example of Data Type Analysis Result).

Figure 4.7. Use Case of Perform Data Type Analysis.

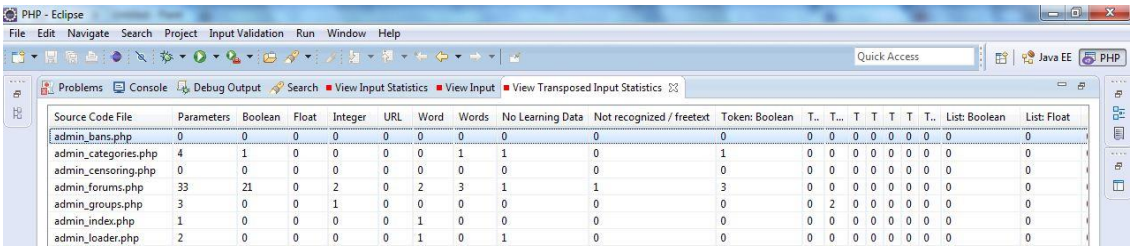


Figure 4.8 Example of Data Type Analysis Result.

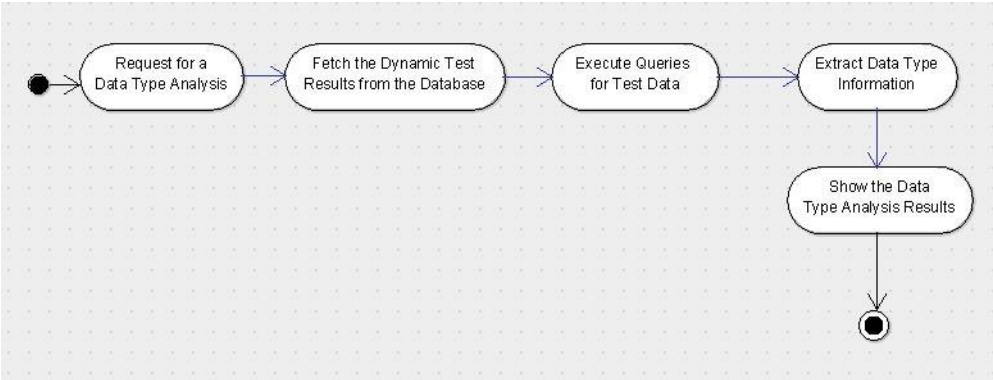


Figure 4.9. Activity Diagram of Data Type Analysis.

As in Figure 4.9, first the developer requests a data type analysis, when the request is received, IPAAS connects to the database, where the test data results are held. After the connection is established, IPAAS fetches the data from the database and executes a query. According to the query's results, IPAAS extracts information of data types from the inputs in the web application when the information extraction is done. The software development environment shows the results to the developer.

4.2.1.2 Perform Dynamic Test Analysis

In this step, where the tester performs dynamic tests on the running projects and a proxy server captures the responses of the test cases. Then the captured test case responses are stored in a database.

Figure 4.10. Use Case of Perform Dynamic Test Analysis.

In Figure 4.11, you can see an example of the working flow of “Performing Dynamic Test Analysis”.

The tester performs his tests through a web browser. Therefore, the tester has to open his web browser first and configure the web browser, so the proxy server can get the test results. After the configuration is done, the tester locates the web application through a web browser and performs tests on the web application.

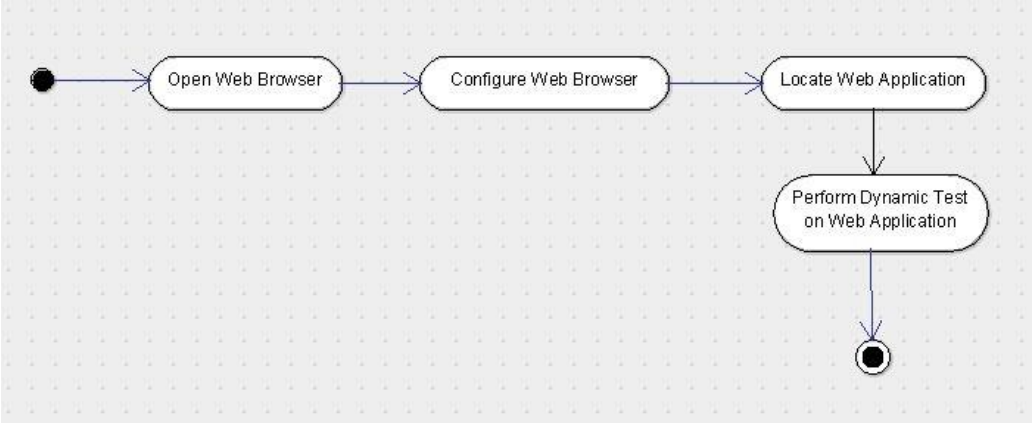


Figure 4.11. Activity Diagram of Performing Data Type Analysis.

4.2.2. Code Analysis

In this feature, we run a static analysis on the code. Therefore, we can extract information on; parameter name, vector name and where it is located. However, we note that this feature has not been used in this work.

In this phase, in Figure 4.12, the developer requests for a “Code Analysis” and the system runs a static analysis. At the end, the system returns the static analysis results to the developer. (See Figure 4.13 for an example of Code Analysis result).

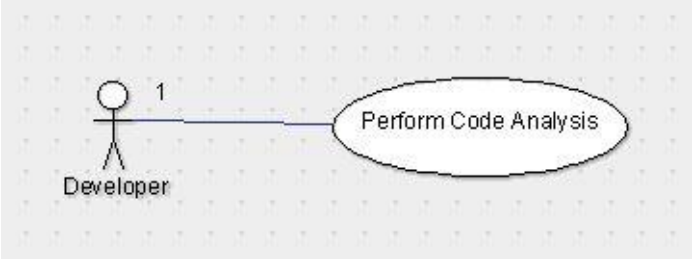


Figure. 4.12. Use Case Diagram of Code Analysis.

Problems | Console | Debug Output | Search | Code Scan: Input Parameter Names

File	Parameter Name	Vector Names
admin_bans.php	Parameter name in file: include/functions.php at line number: -1 is generated dynamically	COOKIE
admin_bans.php	Parameter name in file: include/functions.php at line number: -1 is generated dynamically	COOKIE
admin_bans.php	add_ban	GET
admin_bans.php	add_ban	POST
admin_bans.php	add_edit_ban	POST
admin_bans.php	ban_email	POST

Figure 4.13. Example of Code Analysis Result.

4.2.3. Generate Policy

In policy generation, IPAAS generates policies for input validation vulnerabilities that IPAAS has detected, so according to the policies, these potential vulnerabilities could be fixed in the following steps. (In Figure 4.14, we can see the use case diagram of performing a policy generation).

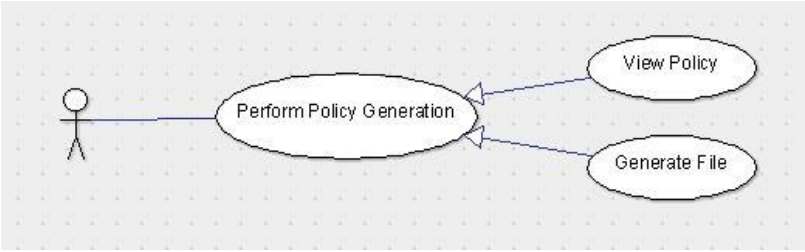


Figure 4.14. Use Case Diagram of Generate Policy.

In this step, as in Figure 4.14 “Perform Policy Generation” includes two functionalities, which are “View Policy” and “Generate Policy”. The developer is able

to request for a policy generation for his project and IPAAS provides the policies with a file. After the policies are created, the developer is also able to see the created policies for his project.

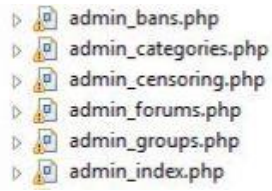


Figure 4.15. Before Policy Generation.

Figure 4.16. After Policy Generation.

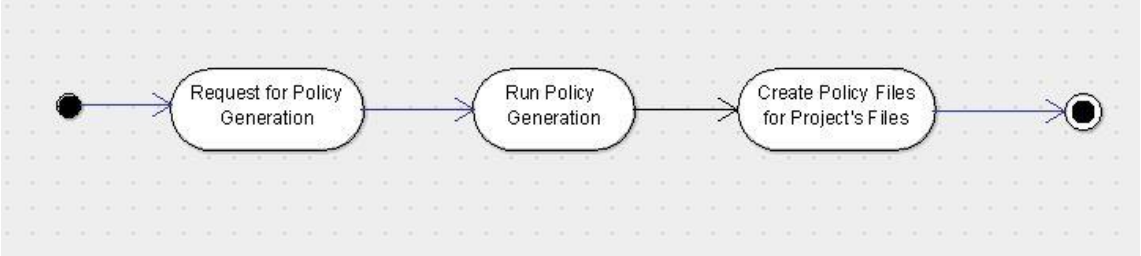


Figure 4.17. Activity Diagram of Policy Generation.

When the developer wants to create a policy for his project, first the developer requests for a policy generation, and then IPAAS runs a policy generation according to tester's dynamic test results and the results of data type analysis. In the end, the policies are provided to the developer as an ".inp" extension file (See Figure. 4.17 to see the activity diagram of Policy Generation). Then these files are used to fix the input validation vulnerabilities. (See Figures 4.15 and 4.16 for the differences between before and after policy generation).

4.3. New Features of IPAAS

We have discussed the existing features of IPAAS. In this section, the improvements and the added new features that would aid the developer and increase the perspective of IPAAS are overviewed.

4.3.1. Supporting Different Types of Projects and Extensions

`

IPAAS was developed to support server-side PHP web application projects. However, we made IPAAS also support server-side Java (Servlet) and JavaScript (ex; Node.JS) projects. The reason we wanted to focus on other type of projects is not only for popular usage of these types of projects but also to address developers as much as we can. We also wanted to secure more web applications by considering the increasing usage of clouds.

4.3.2. Delete Test Data

This feature provides the developer to reach the raw data of HTTP request and responses that include information about parameters and allows the developer to delete them. We gave this authority to the developer in case the developer updates the project. Then IPAAS will avoid coming up with false learned data types and input parameter policies.

We created this feature under "Data Type Analysis", because this process is preferred to be done before data type analysis. Furthermore, all the features under data type analysis work under the same database and same data. (See Figure 4.18)

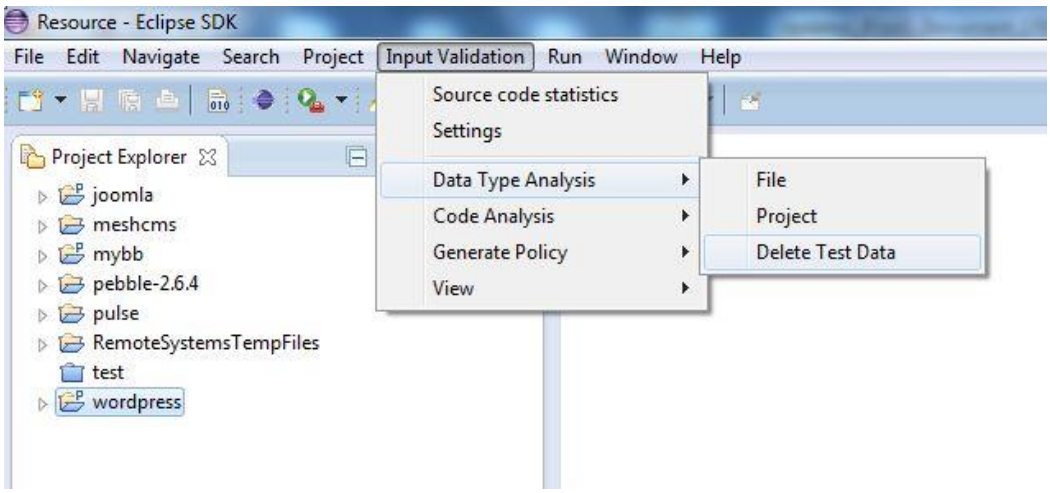
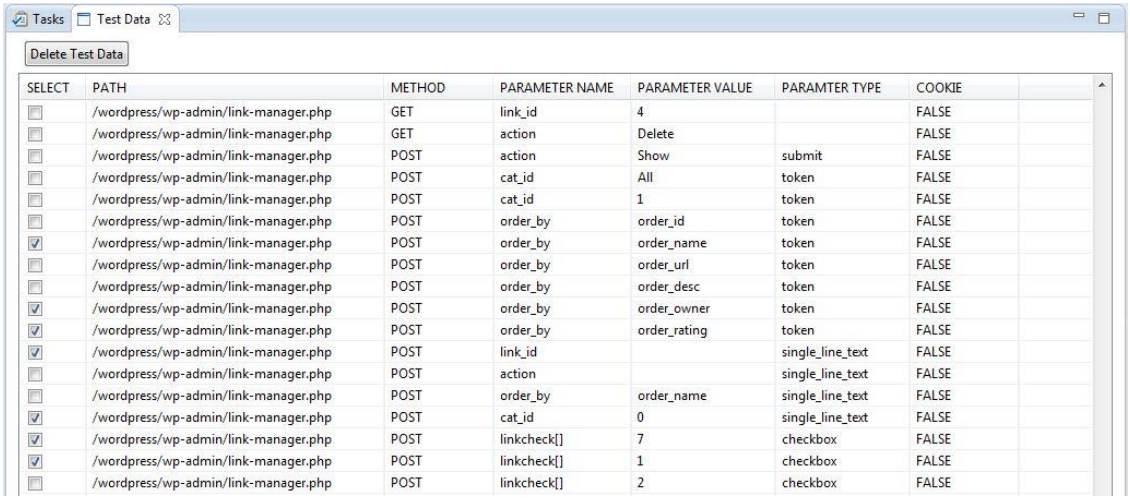


Figure 4.18. View of Delete Test Data Feature.



SELECT	PATH	METHOD	PARAMETER NAME	PARAMETER VALUE	PARAMTER TYPE	COOKIE
☐	/wordpress/wp-admin/link-manager.php	GET	link_id	4		FALSE
☐	/wordpress/wp-admin/link-manager.php	GET	action	Delete		FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	action	Show	submit	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	cat_id	All	token	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	cat_id	1	token	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	order_by	order_id	token	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	order_by	order_name	token	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	order_by	order_url	token	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	order_by	order_desc	token	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	order_by	order_owner	token	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	order_by	order_rating	token	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	link_id		single_line_text	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	action		single_line_text	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	order_by	order_name	single_line_text	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	cat_id	0	single_line_text	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	linkcheck[]	7	checkbox	FALSE
☑	/wordpress/wp-admin/link-manager.php	POST	linkcheck[]	1	checkbox	FALSE
☐	/wordpress/wp-admin/link-manager.php	POST	linkcheck[]	2	checkbox	FALSE

Figure 4.19. After Performing Delete Test Data.

When the “Delete Test Data” action is performed, the plug-in shows the developer the test data’s path, method, parameter name, parameter value, parameter type, cookie and also gives the option to select the test data to delete.

Sometimes, the developer might miss to select all the parameters that have been updated. For instance, in Figure 4.19, at the bottom, the developer has two checkbox parameters, but he forgot to select the third one, which is related to the other two checkbox parameters. In this case, not deleting the third checkbox parameter will affect the data type analysis and validate a non-existing checkbox item. Therefore, to avoid this type of faults, the parameters which are related to each other are stored into the database with the same ID. When the developer selects at least one of the parameters, it deletes all the other parameters that are related to it.

CHAPTER 5

STATIC ANALYSIS OF JAVASCRIPT APPLICATIONS

5.1. Modified TAJS

TAJS [10] is a JavaScript static analysis tool, which is written in Java that has been developed by a research group from Aarhus University and Universität Freiburg. It was firstly developed for type analysis for JavaScript. Later on, they have improved TAJS and integrated new features such as interprocedural analysis with lazy propagation, detecting eval() function usages from a JavaScript source code and fixing them. While they develop such features, they use dataflow and control-flow analysis by generating their graphs.

Since we briefly mentioned TAJS's features, we want to detect the existing functions and its parameters that are defined in a JavaScript source code. TAJS has a feature to collect information on function parameters. Unfortunately, TAJS couldn't coordinate with jQuery to collect information on function parameters. Therefore, we target TAJS's flow graph feature to extract information of functions and parameters as well.

In TAJS's "flowgraph" package, in "Function.java" file we add a simple file operation to document the existing functions and their parameters with their of code lines where they have been defined. In that case we could easily select and locate the functions with their parameters. Otherwise, a function without parameters means that the function does not accept any inputs into itself.

5.2. Analyzing JavaScript

JavaScript (JS) is a dynamic computer programming language. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed [30]. It is also being used in server-side

programming, game development and the creation of desktop and mobile applications [31].

JavaScript is a prototype-based scripting language with dynamic typing and has first class functions. JavaScript was influenced its syntax by C and copies many names and naming conventions from Java, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JavaScript are taken from the Self and Scheme programming languages [33]. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles [31].

The application of JavaScript in use outside of web pages—for example, in PDF documents, site-specific browsers, and desktop widgets—is also significant. Newer and faster JavaScript VMs and platforms built upon them (notably Node.js) have also increased the popularity of JavaScript for server-side web applications. On the client side, JavaScript was traditionally implemented as an interpreted language but just-in-time compilation is now performed by recent (post-2012) browsers [31].

JavaScript was formalized in the ECMAScript language standard and is primarily used as part of a web browser (client-side JavaScript). This enables programmatic access to objects within a host environment [31].

When it comes to perform static analysis on JavaScript written applications, it is a challenging language due to its highly dynamic nature. Madsen [23] also add that, recently much attention has been directed at handling the peculiar feature of JavaScript in pointer analysis [24, 25, 26], data flow analysis [18, 27] and type systems [28, 29]; however, most of the work ignored the fact that JavaScript programs usually execute in a rich execution environment and mentions that JavaScript applications rely on large and complex libraries (i.e. jQuery) and frameworks, often written in a combination of JavaScript and native code such as C and C++.

5.3. Using TAJS

While we use TAJS, we utilize from the “flow graph” and “call graph” features. To briefly explain TAJS’s features;

In the flow graph, each node contains an instruction and each edge represents potential control flow between instructions in the program. The graph has a designated

program entry node corresponding to the first instruction of the global code in the program. Instructions refer to temporary variables, which have no counterpart in JavaScript, but which are introduced by the analyzer when breaking down composite expressions and statements to instructions. The nodes can have different kinds [18]:

- **declare-variable[x]:** declares a program variable named x with value undefined.
- **read-variable[x, v]:** reads the value of a program variable named x into a temporary variable v.
- **write-variable[v, x]:** writes the value of a temporary variable v into a program variable named x.
- **constant[c, v]:** assigns a constant value c to the temporary variable v.
- **read-property[v1, v2, v3]:** performs an object property lookup, where v1 holds the base object, v2 holds the property name, and v3 gets the resulting value.
- **write-property[v1, v2, v3]:** performs an object property write, where v1 holds the base object, v2 holds the property name, and v3 holds the value to be written.
- **delete-property[v1, v2, v3]:** deletes an object property, where v1 holds the base object, v2 holds the property name, and v3 gets the resulting value.
- **if[v]:** represents conditional flow for e.g. if and while statements.
- **entry[f, x1, . . . , xn]**, **exit**, and **exit-exc:** used for marking the unique entry and exit (normal/exceptional) of a function body. Here, f is the (optional) function name, and x1, . . . , xn are formal parameters.
- **call[w, v0, . . . , vn]**, **construct[w, v0, . . . , vn]**, and **after-call[v]:** A function call is represented by a pair of a call node and an after-call node. For a call node, w holds the function value and v0, . . . , vn hold the values of this and the parameters. An after-call node is returned to after the call and contains a single variable for the returned value. The construct nodes are similar to call nodes and are used for new expressions.
- **return[v]:** a function return.
- **throw[v]** and **catch[x]:** represent throw statements and entries of catch blocks.
- **<op>[v1, v2]** and **<op>[v1, v2, v3]:** represent unary and binary operators, where the result is stored in v2 or v3, respectively.

Since we modified TAJS, we run it through Eclipse from its source code. In TAJS, there are several features included. To use these feature we have to run them with

the feature's arguments. For example, to use the flow graph and call graph features of TAJS, we do;

-flowgraph "javascriptFile.js"

-callgraph "javascriptFile.js"

In case studies (Chapter 7), we use the **if[v]** nodes to check the if there are any validations performed on parameters and use the **read-variable[x, v]** node to check if the parameters is used before it is validated. TAJS does not provide a JavaScript's code with its image of flow graph. It provides its flow graph with a ".dot" (graph description language) extension file. To get a visual output of the ".dot" file, we use a graph visualization software called Graphviz[2].

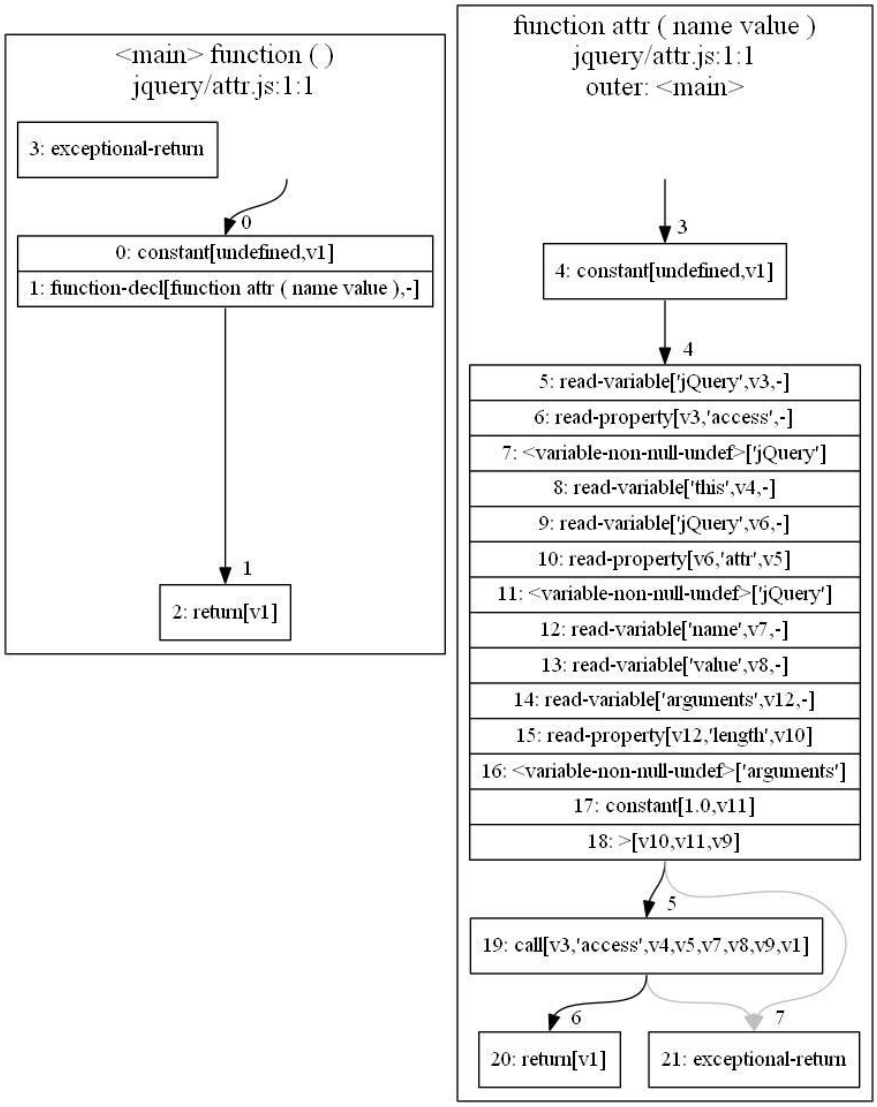


Figure 5.1. An Example of "attr()" function's flow graph.

[2] http://www.graphviz.org/

To use TAJS's call graph feature, we have to call the function in its defined file. Therefore, we could be able to observe it calling information and how deep it goes in the code. TAJS does not provide an output of a graphical image like its flow graph feature, but similarly provides a ".dot" extension file. To get a visual output of a functions call graph, we again used Graphviz.

CHAPTER 6

BAYESIAN NETWORK GENERATION

Bayesian Networks are a probabilistic directed acyclic graphical model that expresses the causal relationship between nodes based on Bayesian extraction. The reason why we choose to use Bayesian Networks as the base of methodology is that Bayesian Networks offer consistent semantics for representing uncertainty and an intuitive graphical representation of interactions between various causes and their effects. BBNs (Bayesian Belief Networks) are useful when the information about the past and/or the current situation is vague, incomplete, conflicting and uncertain [16].

We define a node probability table for each node, where a node represents the functions, vulnerabilities and the application. The tables will represent the relationships between the nodes, and the relationship variables are usually discrete with a fixed number of states. For each state, the probability that the variable is in this state is given. If there are parent nodes, i.e. a node that influences the current node, these probabilities are defined in dependence on the states of these parents [12].

In Wagner's [12] model the state probabilities are distributed equally. However, in our model, to obtain more realistic results we construct and assign values to the state probabilities using reported statistics and solid experimental results by using static analysis and refactoring. For instance, we use vulnerability commonness vulnerability damage, function depth, number of calls of a function.

6.1. Bayesian Network Generation

Wagner [12] used Bayesian Networks to assess software's quality. In his work, the first step goes through defining the activities, the facts and indicators of the problem. Wagner [12] gave an example of an optimization of security assurance and defines attacks as activities. In this case, our activities will correspond to our six input validation vulnerabilities we defined, which are; Cross-Site Scripting, SQL Injections, Code Injections, OS Command Injections, Input Validation, Path Traversal.

In the second step as Wagner [12] defined, we define the factors that are related

to the identified activities. Therefore, the vulnerable functions we detected from a JavaScript application will be represented as factors of our Bayesian Network.

In the last step, for indicators, we add an additional node for each fact and activity node we want to measure. As Wagner [12] mentioned the indicator how nodes will take part in the Bayesian Network, "the edges are directed from the activity and fact nodes to the indicators, i.e. the indicators are dependent on the facts and activities. An indicator is only an expression of the underlying factor it describes".

6.2. Node Probabilistic Table Generation

While we build the Bayesian Network, we show the relationships between the nodes and each predecessor node affects the successor nodes' probabilistic table (NPT), as it can be seen in Figure 6.1. Moreover, in Figure 6.1 it can be seen that the number of NPT is equal to the number of nodes that exists in the Bayesian Network.

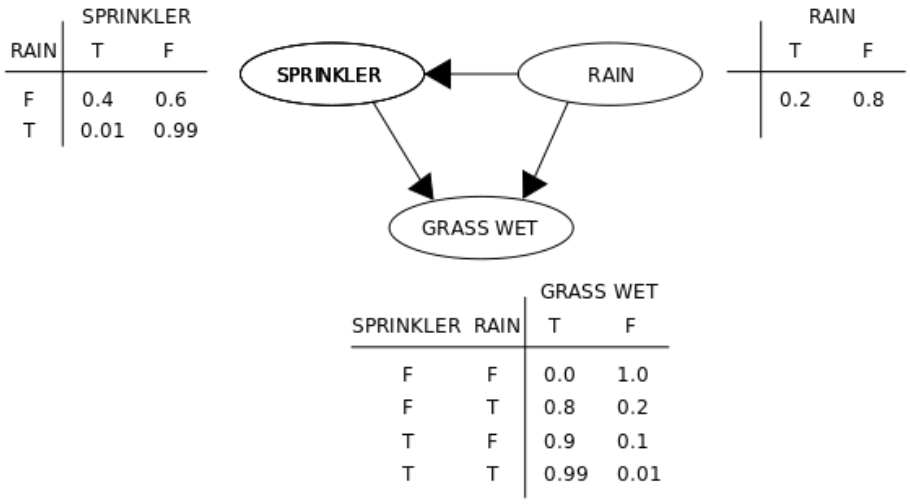


Figure 6.1. Example of a NPT of a Bayesian Network[3].

Therefore, the number of node probabilistic tables of our Bayesian Network will be;

$$1 + (number\_of\_vulnerabilities) + (number\_of\_functions) = NPT$$

The value 1 stands for our "Application node" and the number of vulnerabilities will be 6 later, which correspond to our 6 input validation vulnerabilities we defined

[3] http://en.wikipedia.org/wiki/Bayesian_network

previously. While we fill in our NPTs, we will observe them in three categories; Application's, input validation vulnerabilities' and functions' NPTs.

6.2.1. Application Node's Probabilistic Table Calculation

As it can be seen from Table 3.1, each vulnerability has different levels of damages. Therefore, we can easily say that each vulnerability has different effects on the "Application" node.

We have obtained the danger levels from OWASP [7] (6.1), however, the information we hold about danger levels is only a list of vulnerability danger ordering. To obtain a percentage value of danger, we assign weights to each input validation vulnerability, and process them with a "Weighted Percentile Formula" (6.1).

Table 6.1. Danger levels of input validation vulnerabilities with weights.

Danger	Input Validation Vulnerability	Weight
1	SQL Injection	4
1	Code Injection	4
1	OS Command Injection	4
2	XSS	3
3	Path Traversal	2
4	Input Validation	1

$$S_n = \sum_{k=1}^{n} w_n \ , P_n = \frac{100}{S_n}\left(S_n - \frac{w_n}{2}\right) \tag{6.1}$$

After we applied the "Weighted Percentile Formula" on our input validation vulnerabilities, we obtained the values on Table 6.2.

Table 6.2. Input validation vulnerabilities with danger percentages.

Danger	Input Validation Vulnerability	Weight	Danger %
1	SQL Injection	4	80%
1	Code Injection	4	80%

(cont. on next page)

Table 6.2. (cont.)

1	**OS Command Injection**	**4**	**80%**
2	XSS	3	75%
3	Path Traversal	2	66.7%
4	Input Validation	1	50%

As we evaluated our vulnerabilities' danger percentages, now we can shape our NPT for "Application" node. The NPT is consisted of 2^{n+1} values, the "n" value stands for the number of nodes that are connected the "Application" node, which is 6 (number of vulnerability categories) and the "+1" is to include the "false" values, which are the,

(calculated true value) + (calculated false value) = 1

Otherwise, we can only include the true values into our Bayesian Network and will not be able to see the negative changes in the Bayesian Network. In that case "Application" node's NPT will have 2^{6+1}= 128 values. To calculate Application's NPT values we write and apply the algorithm below and use a 1-0 normalization (6.2) to make it suitable for the NPT and Bayesian Network.

```
Set vul_com to Array of Vulnerabilities Commonness Statistics
Set vul_dmg to Array of Vulnerabilities Damage Statistics
For (i=0; i<6; i++)
{
        vul_score = vul_com[i] * vul_dmg[i]
}
Initialize totPosVulScores[vSubSets.size()]
Set sum to 0
For (i=0; i<vSubSets.size(); i++)
{
        Set sum to 0
        For(j=0; j<6; j++)
        {
                If subset[i] contains j
                        sum = sum + vul_score[j]
        }
        totPosVulScores[i] = sum
}
normalized_Scores = Normalize(totPosVulScores)
For(i=0; i<normalized_Scores; i++)
{
        normalized_Scores[i] =   normalized_Scores[i] –  standart_deviation;
        Get Absolute of  normalized_Scores[i]
}
```

Figure 6.2. Algorithm for Application node's NPT calculation.

$$Normalized(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}} \quad (6.2)$$

6.2.2. Node Probabilistic Table Calculation of Vulnerability Nodes

The function nodes are directed to every vulnerability node, which means that the function nodes' values will shape vulnerabilities' NPT. To evaluate NPT values of vulnerabilities', we use 3 types of information about functions.

The first information is the statistics of reported vulnerabilities from Jan. 2000 to Jan. 2014, which gives us an idea of the commonness of vulnerabilities we extracted from NVD [8] in Table 3.1 and 3.2. We used this information on functions instead of vulnerabilities, because since the reports are based on codes, it will be more sensible to use this information on function nodes to vulnerabilities. Additionally, according to Bayesian Network's nature, the predecessor (function) nodes inherit their information on the successor (vulnerability) nodes, therefore, the successor nodes will not be devoid of the commonness information.

The second information we used is the depth of functions. The depth term we used is; if we construct a function's call graph of an application, we obtain a directed graph that represents calling relationships between subroutines in a program. If we ignore the cycles of the call graph, the function's longest path will be the depth of the function. We assume that, the deeper the function gets in direct proportion the potential damage increases as well, by considering that might infect functions that are included in the call graph.

The third and last information we used is the number of calls of a specific function. It is obvious that if a function is called many times in different parts of a code, the coverage of a function in the source code will increase, which gives us not an exact, but a brief information of how many times it is used in the code.

After we collected our data, we wrote an algorithm below, to calculate our functions' NPT values;

```
For(i=0; i<numberOfFunctions; i++)
{
	For(j=0; j<numberOfVulnerabilities; j++)
	{
		func_score[i][j] = (func_depth +1)*(numberOfFuncCalls)*(vul_commoness);
	}
}
fSubSets = Call subset function for functions

For(i=0; i<numberOfVulnerabilities; i++)
{
	For(j=0; j< fSubSets.size; j++)
	{
		score = 0;
		For(k=0; k<numOfFunctions; k++)
		{
			If(fSubSets.get(j).contains(function[k]))
			{
				score = score + (fScores.get(i+(k*vulns.size())));
			}
		}
		If(subSetsOfFuncs.get(j).size() == 0)
		{
			score = 0;
		}
		FuncSubSetScores.add(score/numOfFunctions);
	}
}
normalized_FScores = Normalize(FuncSubSetScores);

int index = 0;

For(i=0; i<numberOfVulnerabilities; i++)
{
	For(j=index; j<index + fSubSets.size; j++)
	{
		normalized_FScores[j] = normalized_FScores[j] –  standart_deviation;
		Get Absolute of normalized_FScores[j]
	}
	index = index + fSubSets.size;
}
```

Figure 6.3. Algorithm for Vulnerability nodes' NPT calculation.

6.2.3. Function Nodes' Probabilistic Table Calculation

The Bayesian approach is based on belief and provides us with uncertain inferences. By uncertainty, we mean that we are not 100% sure of anything. Therefore, in Bayesian Networks we always give a small probability of standard deviation. Otherwise, if we give a certain probabilistic value like "1" and "0", this would be a frequentist approach, not a Bayesian. A Bayesian approach is similar to the case below.

> *"For billions of years, the sun has risen after it has set. The sun has set tonight. With very high probability (or I strongly believe that or it is true that) the sun will rise tomorrow. With very low probability (or I do not at all believe that or it is false that) the sun will not rise tomorrow."*[4]

For this reason, we assign "0.95" value as "True" if the function hasn't not used a validation and "0.05" value as "False" if the function contains an input validation code. We do not give certain values like 1 and 0 to abide the Bayesian approach. By not giving exact values we also include some special cases. For example; attacking by encoding malicious code, type-based validations etc.

6.2.4. Complexity of General Table Calculation

We mentioned the how we calculate the number of NPTs and how many values the NPTs will contain. Since are trying to build an automatized Bayesian Network Generator, we formalize the complexity of our Bayesian Network's NPT calculation.

While we calculate our "Application" node's NPT, the number of its values will be related to its predecessor ("Vulnerability") nodes. Then, we consider every combination of vulnerabilities that might occur in the "Application" node. Therefore we have $2^{(\text{number of vulnerabilities})}$ combinations of vulnerabilities for the "Application" node.

Similarly to "Application" node's NPT calculation, the number of values that will occur in the "Vulnerability" nodes will be related to its predecessor ("Function") nodes. We consider every possible combination of functions that might occur in a single "Vulnerability" node. Therefore, for a single "Vulnerability" node we have

[4] Korb, K. B., & Nicholson, A. E. (2003). *Bayesian artificial intelligence*. cRc Press.

$2^{(number\ of\ functions)}$ combinations of functions. However, by including every vulnerability we have *(number of vulnerabilities)* $\times 2^{(number\ of\ functions)}$.

For our "Function" nodes we assigned static values, therefore they do not require any calculation for their NPTs. To summarize our Bayesian Network's NPT calculation complexity, we formalize it as;

m = number of Vulnerabilities

n = number of Functions

$$O(2^m + (m \times 2^n))$$

CHAPTER 7

CASE STUDY

In this chapter we describe which test sample and why we select it. We also explain the case studies and experiments performed. In addition, we discourse the processes of how we gathered the necessary information to construct our Bayesian Network and to evaluate each node's NPT to get significant inferences.

7.1. Analyzing jQuery

JavaScript developers usually prefer the jQuery library in their applications since it is fast, small and feature-rich. It contains many functions that help the developers to make event handlings, manipulation, HTML document traversal and animation. Furthermore, it is used by many popular web sites such as Amazon, Microsoft, WordPress, Reddit, Instagram, Stack Overflow, the Guardian, Fox News. This has given the motivation to target jQuery to measure how secure it is against input validation vulnerabilities.

While analyzing jQuery, we target its functions and their parameters. The parameters of a function are actually inputs for a function and parameters, so parameters without validations are expected to be potential input validation vulnerabilities. Therefore, we check if there are any validations performed on parameters in the function, before it is used or processed.

In this thesis, we experiment on jQuery version 1.9.1 by using TAJS version 0.9.3. We use TAJS to detect functions and its parameters of jQuery. Since TAJS has a feature to provide us with a flow-graph of a JavaScript source code, we are able to reach the functions and its parameters as well. Therefore, we modify TAJS's flow-graph feature that runs file operation for functions and parameters to extract them, which we described in Chapter 5. Unfortunately, TAJS is unable to detect parameters of a function in its flow-graph feature. Considering jQuery's naming conventions and syntax style, we write a parser to extract function's parameters separately.

We obtain statistics of jQuery functions after we run extended TAJS. We have detected 579 functions and 896 parameters with TAJS,. Out of 579 functions, with our modification on TAJS, we detected that 463 functions have parameters and the rest of the 116 functions do not have any parameters, so we removed 116 functions from our experiments.

7.1.1. Selecting jQuery Functions for Experiment

We only based on jQuery's useful and most commonly used functions among jQuery 463 functions. Out of these functions we select functions with DOM (document object model) manipulation, AJAX, finding, looping, filtering results and events. Usually, functions with animations and effects do not fit the case for input validation vulnerabilities. Therefore, we eliminate such functions.

We create a list of jQuery functions that are mostly used by JavaScript developers as follows from Mardanov's [14] work and Deering's [13] article. Cooper paper [19] mentions that Bayesian Networks is a NP-hard problem, for this reason, we reduce the number of functions to 10, which will be supplied to the Bayesian Network;

- html()
- text()
- css()
- attr()
- val()
- each()
- data()
- hasData()
- removeData()
- find()

7.1.2. Detecting Vulnerable Functions

We assume function's parameters as inputs of a function before we detect vulnerable functions. Therefore, our strategy is to check if there is any validation

performed before the parameters are used. We follow this strategy manually to detect vulnerable functions. We divide this strategy into two phases:

1) We check if there are any specific validations performed on the parameter that checks if the parameter contains any codes, tags or escape characters. After our manual checks on our 10 jQuery functions, we do not encounter any specific validations for input validation vulnerabilities. These results encourage and support our hypothesis on jQuery functions which are not quite ready against untrusted data.

2) To ensure that the jQuery functions which we are experimenting on are vulnerable to input validation vulnerabilities. We look into the functions and try to see if there are any type checks for parameters. However, it is known that type checks are not a complete prevention method to avoid input validation vulnerabilities, especially for string type data. For example, to break defenses against for both type based and code based validations, attackers encode their malicious code and then perform their attacks. So they will be able to pass both string type and code based checks. Our manual checks on functions shows that some functions perform type checks on their parameters although all of the type based checks are performed after the parameters are used.

Beyond our manual checks, to support our hypothesis that our functions are vulnerable to untrusted data, we perform a double check by first extracting each function's source codes from "jQuery Source Viewer" [20]. Then we extract each function's flow-graph with TAJS and perform our code based and type based check strategies which we defined by manually checking the flow-graph again. After the manual checks on the flow-graph we validated once again that there isn't a validation performed before the parameters are used or any specific validation performed to prevent input validation vulnerabilities.

In summary, we observed that some functions include type based checks. However, they are performed after they are used. Additionally, we did not observe any code based checks in functions for their parameters. Therefore, we can conveniently say that the functions we experimented might be vulnerable against untrusted data.

7.2. Gathering Information of Selected jQuery Functions for Node Probabilistic Table

In Chapter 6, section 6.2, we explain how we evaluate the NPT values for Bayesian Network.

To find the number of calls of functions, we use Eclipse's Refactor[5] feature and JSRefactor [21], which is plugin developed for Eclipse. We are able to see the changes that are done in the source code and validate them if they are a function call or not with refactoring.

To calculate the depth of a function, first we have to extract the call graph of each function we will experiment on. Therefore, we are able to evaluate the distance from the selected function to its furthest leaf node, which is the depth of the function. To calculate the depth of each function we used TAJS's call graph feature to obtain each function's call graph.

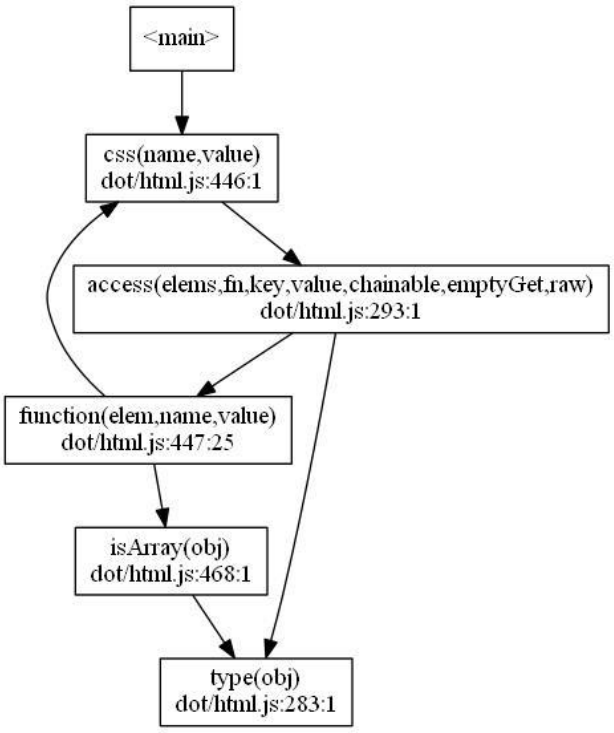


Figure 7.1. An example of "css()" function's call graph output driven by TAJS

[5] http://www.eclipse.org/

Table 7.1. Table of function information.

Function Name	Number of How Many Times the Function is called in jQuery	Depth
html()	2	2
text()	1	2
css()	32	4
attr()	3	2
val()	5	0
each()	59	0
data()	5	2
hasData()	2	0
removeData()	1	2
find()	19	0

7.3. Implementing an Automatized Bayesian Network Generator

We need a tool or a framework, which creates a file and keeps the information to build a Bayesian Network. The file should keep the Bayesian Network's connection information between the nodes and every node's NPT. We see there are many tools that are developed while we search for such a tool. Out of these tools we found OpenMarkov which is a Java open source software tool [22]. It provides a file close to an XML format, which is easy to parse and write. Therefore, we can create our Bayesian Network through OpenMarkov's file format and generate it. Finally, we have decided to use and integrate OpenMarkov to simulate our Bayesian Network.

We want our Bayesian Network to be more dynamic and flexible before we develop Bayesian Network Generator. So the developers can modify the Bayesian Network according to their JavaScript application as they desire to. To achieve this, we prepare a configuration file where our tool will first parse the configuration file. So the developers will be able to add or remove functions and same for vulnerabilities as well. Another advantage of our configuration file is to keep the functions' and vulnerabilities' information in it. This supports our tool to be kept updated. If a change occurs in the statistics of vulnerabilities or functions, it can be modified and be adapted to present time's values.

We integrate our algorithms (evaluate each vulnerability and function node's NPT) in chapter 6 to our tool. Then, we start building our Bayesian Network by initially defining the nodes. Then we establish the connections and relationships between nodes.

We define how our Bayesian Network architecture will be. It is constructed in three levels; the “Application”, the “Vulnerabilities” and the “Functions”. Briefly, while we introduce our nodes relationships to our tool, we direct all the vulnerability nodes to the “Application” node and all the function nodes to each vulnerability node which will be a complete bipartite graph. Finally, we assign the calculated NPTs to the nodes where it belongs.

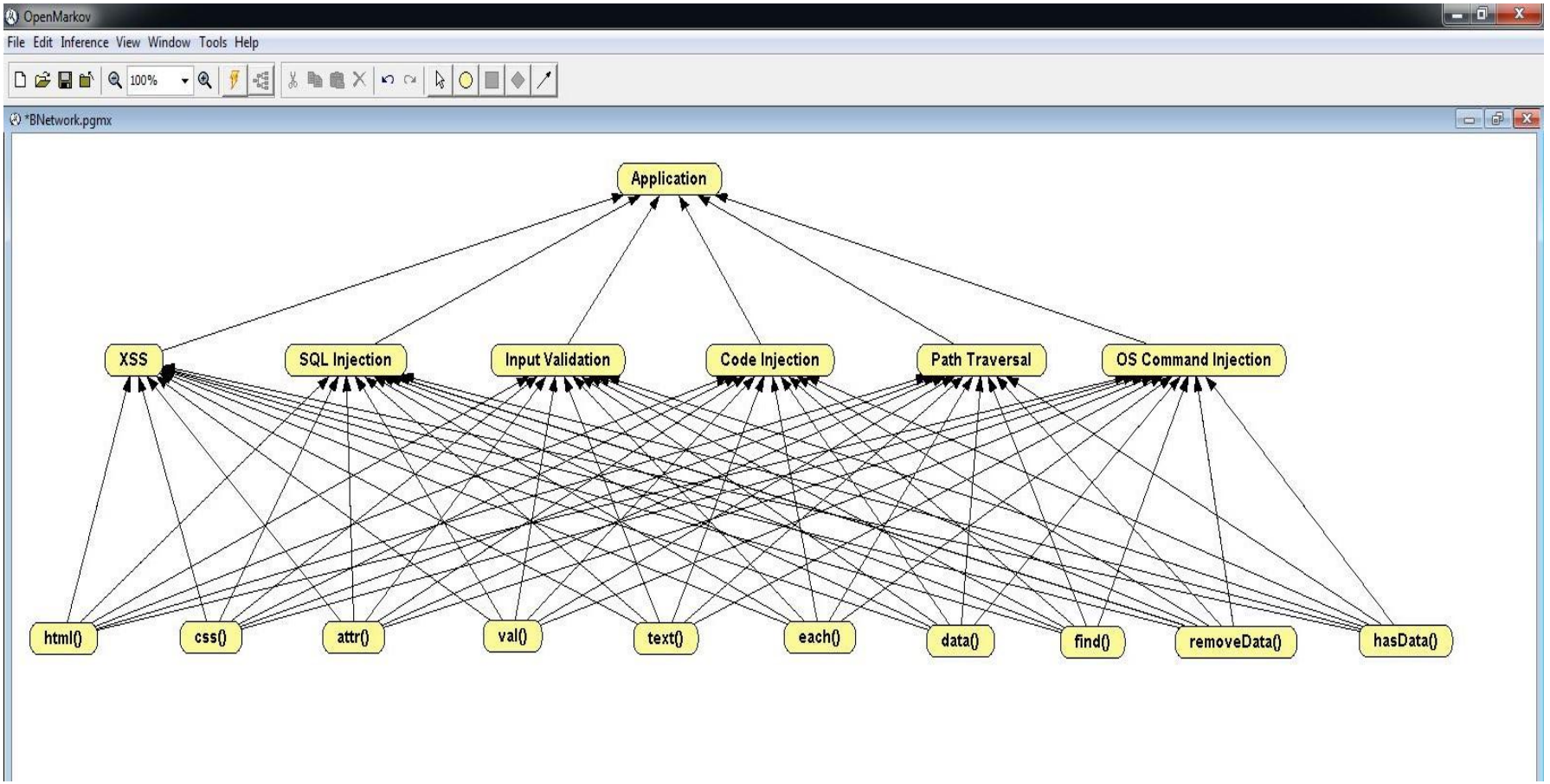


Figure 7.2. A screenshot of a constructed Bayesian Network by Bayesian Network Generator.

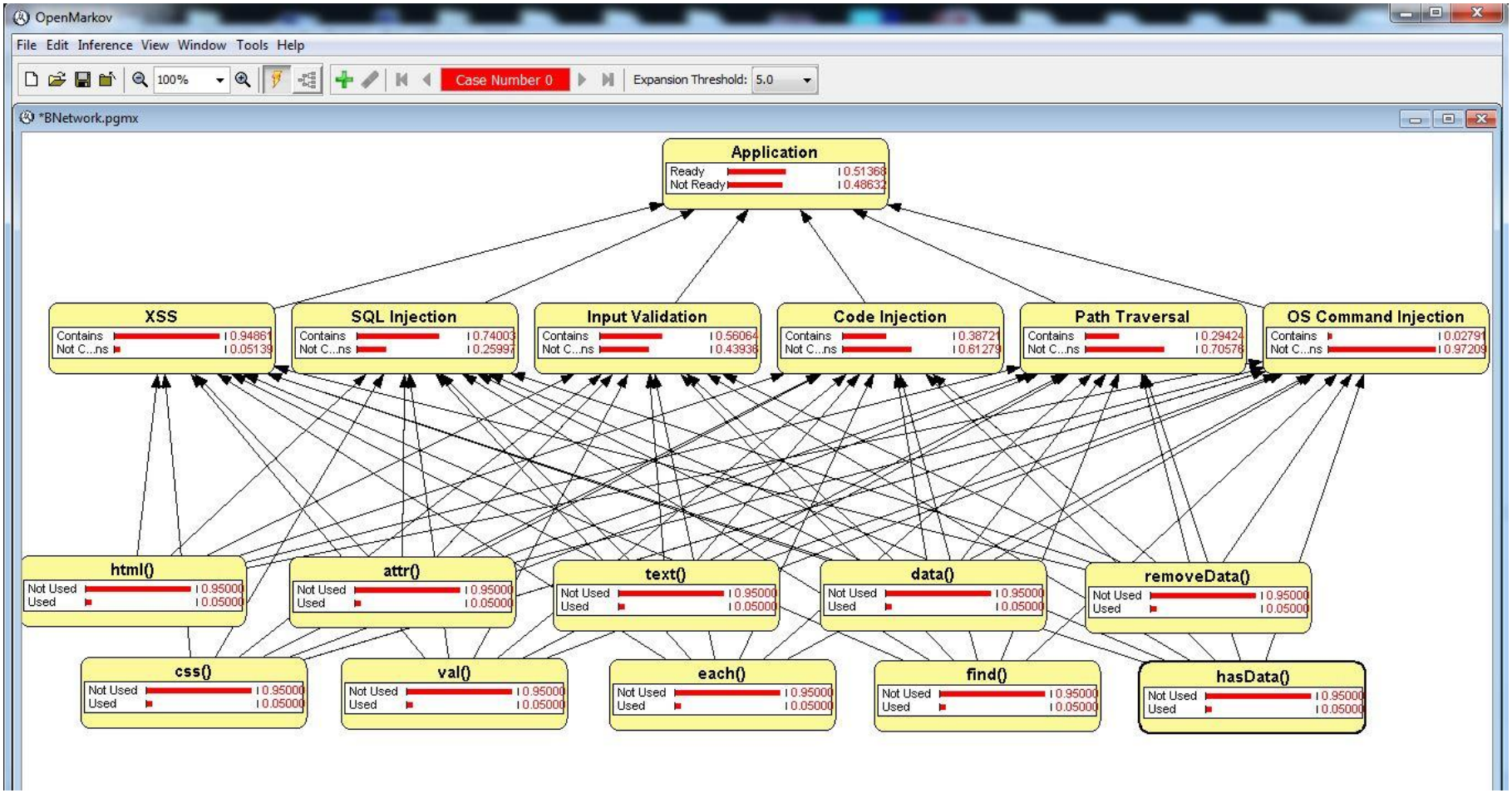


Figure 7.3. A screenshot of an executed Bayesian Network by Bayesian Network Generator.

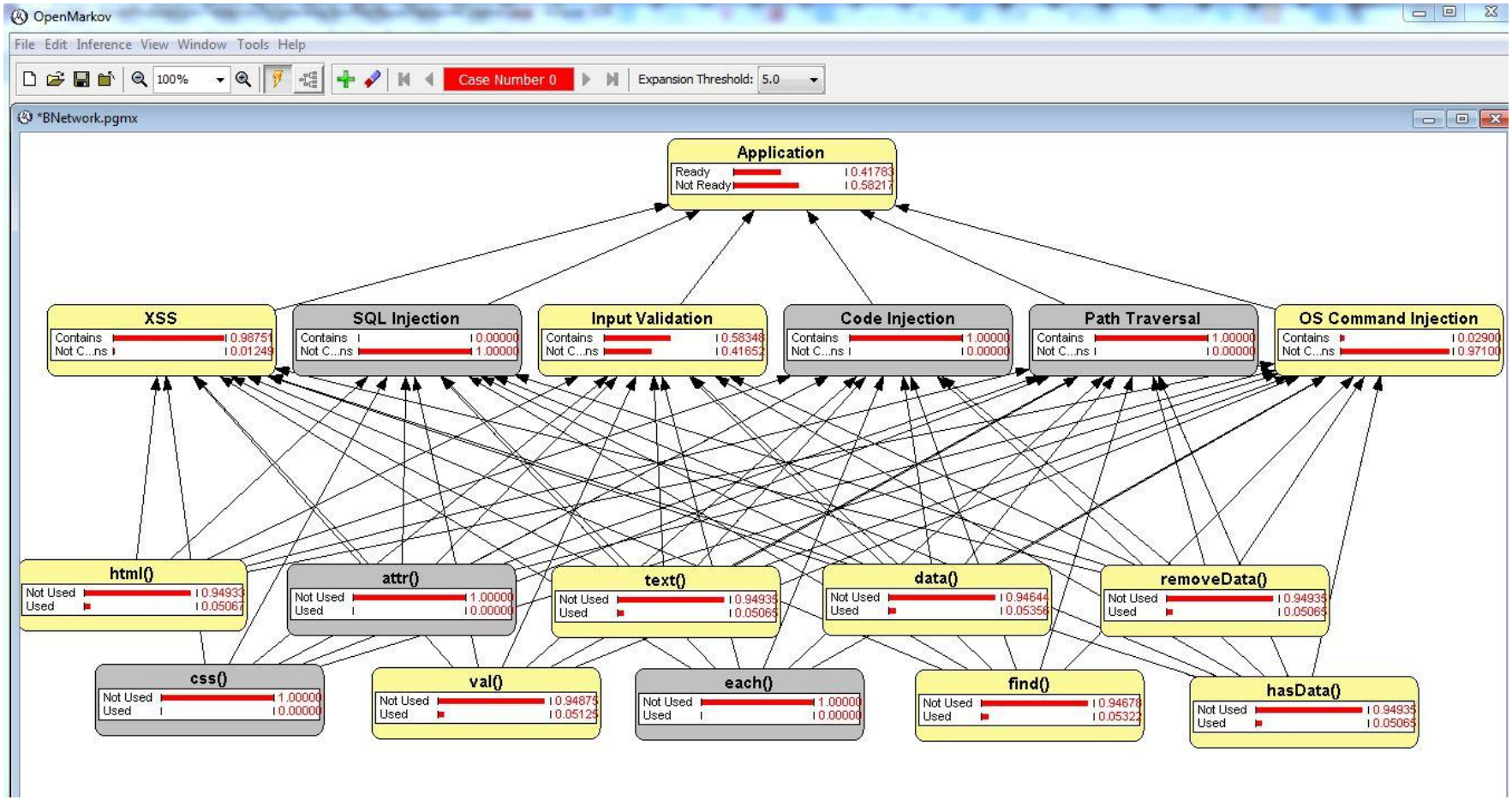


Figure 7.4. An example of some selected states on the Bayesian Network by a developer on Bayesian Network Generator.

7.4. Readiness to Input Validation Vulnerabilities for jQuery Applications

Not only we have observed the possibilities of which type of vulnerabilities might occur in the application after we construct and execute our Bayesian Network with Bayesian Network Generator, but also how much the application is ready against untrusted data which are input validation vulnerabilities.

The Bayesian Network shows us that;

Table 7.2. Table of vulnerabilities' percentages that might be included in the application.

Vulnerability	% of Vulnerability that might be included in the Application
XSS	95%
SQL Injection	74%
Input Validation	56%
Code Injection	39%
Path Traversal	29%
OS Command Injection	3%

Table 7.3. Measured percentage of readiness against untrusted data of a JavaScript Application with Bayesian Networks.

Measured Application	% of Readiness
JavaScript Application/System	51%

Without any selection by the developers the readiness against untrusted data and input validation vulnerabilities is measured as 51%. However, by selecting the used or unused functions in the application, the readiness can be decreased or increased as well. Furthermore, if the application does not use any database in their application, the developers can easily set the "SQL Injection" vulnerability node's state to "Not Contains", because an application with no database will be no use for SQL Injection attacks, so if the state is set "Not Contains" the readiness will increase.

CHAPTER 8

CONCLUSION

In this thesis, we have proposed a model to measure JavaScript applications' readiness against untrusted data by using Bayesian Networks and also developed a tool to construct an automatized Bayesian Network Generator to measure JavaScript applications. To define untrusted data, we collect data (10 years of statistics of reported vulnerabilities) to see their actions which are related to input validation vulnerabilities. We investigate each vulnerability's mitigations in a software development life-cycle of the statistics we obtained. Through our investigations, we have detected 6 input validation vulnerabilities and classified them with a decision tree. The mitigations to perform input validations for input validation vulnerabilities are recommended in the design and development phases of a software development life-cycle. Therefore, we have chosen static analysis not only to detect potential input validation vulnerabilities but also to analyze an application's functions to extract their call graph to measure their depth. In addition, we have used refactoring to find the number of how many times a function is called.

While we construct our Bayesian Network, we express it in three levels; facts, activities and application's readiness. The facts represent the function nodes we experimented on, which include information of input validation vulnerabilities' commonness, the function node's depth, the function node's number of calls. The activity level represents our input validation vulnerability nodes, which it includes each vulnerability damage percentages that can cause damage on the application. The final level there is a sink node that gives us the final measurement of how ready our application against untrusted data is.

In the case study, we have performed our tests on JavaScript's most popular library called jQuery, due its usage and preference by JavaScript developers. In the end, based on our function selections to experiment on, without any state selection on the nodes of our Bayesian network, we have measured that jQuery is 51% ready against untrusted data. We have not only measured the readiness against untrusted data, but also

observed the possibilities of each input validation vulnerabilities that might occur in the applications.

In conclusion, in many works that use Bayesian Network, the information is usually based on equally distributed data; however, in our model and tool, we use real information and statistics, which makes our work and measurement realistic and unique. Another advantage is that the tool we represent is flexible and dynamic. Therefore, the developer can easily introduce its own functions and their information.

Tom DeMarco[6] states that "You can't control what you can't measure". If we adapt DeMarco's quote to measurement of readiness on untrusted data, it is almost impossible to react or take precaution without any measurement. However, in this thesis we have modeled and developed a tool that measures the readiness of untrusted data and aids the developers to see how ready their JavaScript application is. So if developers can see the measurement results, they can also fix their applications and react.

[6] Controlling Software Projects, Management & Measurement Estimation, pg. 3

CHAPTER 9

FUTURE WORK

In our work, calculating the depth, number of function calls and detecting input validation vulnerabilities with a static analysis tool could be automatized and given to the Bayesian Network, so the work load on the developer could be decreased.

Since, software measurement has been a challenging problem to solve, security/vulnerability measurement of system gets a credit from this as well. That is why we used Bayesian Network. The Bayesian Network we used is based on a univariate analysis. However, we believe that if we change our Bayesian Network structure on a multivariate analysis, we will be able obtain more exact results. For instance, let's assume that we have 2 different databases; one that keeps the list of schools in the world and the second database that keeps usernames and passwords. If these two databases are exposed with the same attack, the consequences will not be the same. Obviously, we can say that the second database's importance is higher than that of the first database and this where we will use multivariate values. The second database will have a higher value while the other database gets a lower value. There are applications of multivariate in Bayesian networks. Therefore, we expect integrating multivariate analysis to our model is possible.

Currently our Bayesian Network serves the developer as a recommendation system. The developer observes the changes according to the functions they use. We aim to provide a decision mechanism by using our Bayesian Network that finds the optimal solution, with a higher readiness rate against untrusted data. To construct a mechanism like this, we plan to use multivariate analysis with the game theory to aid and find the best solution.

REFERENCES

1. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS), (accessed:05.02.2014)

2. https://www.owasp.org/index.php/SQL_Injection, (accessed:5.02.2014)

3. https://www.owasp.org/index.php/Code_Injection, (accessed:5.02.2014)

4. https://www.owasp.org/index.php/Command_Injection, (accessed:5.02.2014)

5. http://cwe.mitre.org/data/definitions/20.html, (accessed:05.02.2014)

6. https://www.owasp.org/index.php/Path_Traversal, (accessed:05.02.2014)

7. N. Smithline, OWASP Top 10 2013, 2013, https://www.owasp.org/index.php/Top_10_2013-Top_10, (accessed:03.02.2014)

8. National Vulnerability Database, http://web.nvd.nist.gov/view/vuln/statistics, (accessed:03.02.2014)

9. S. Christey, Preliminary List Of Vulnerability Examples for Researchers, *NIST Workshop Defining the State of the Art of Software Security Tools*, Gaithersburg, MD, August 2005.

10. TAJS, version: 0.9-3, http://www.brics.dk/TAJS/

11. https://www.owasp.org/index.php/Static_Code_Analysis, (accessed:03.02.2014)

12. S. Wagner, A Bayesian Network Approach to Assess and Predict Software Quality using Activity-based Quality Models. *Information and Software Technology*, **52**, 1230-1241, 2010. http://dx.doi.org/10.1016/j.infsof.2010.03.016

13. S. Deering, Useful jQuery Function Demos For Your Projects, 2012, http://www.smashingmagazine.com/2012/05/31/50-jquery-function-demos-for-aspiring-web-developers/

14. A. Mardanov, The list of most commonly used jQuery API functions, 2013, https://gist.github.com/azat-co/5898111

15. S. Kondakçı. Network Security Risk Assessment Using Bayesian Belief Networks, IEEE International Conference on Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust, 978-0-7695-4211-9/10, 2010

16. D. Heckerman. A Tutorial on Learning With Bayesian Networks, Technical Report, Microsoft Research, no. MSR-TR-96-06, 1996, http://research.microsoft.com/apps/pubs/?id=69588

17. Programming Language Popularity, http://langpop.com/, (accessed:12.03.2014)

18. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In Proc. 16th International Static Analysis Symposium, SAS '09, volume 5673 of LNCS, pages 238–255. Springer-Verlag, August 2009.

19. Cooper, Gregory F. "The computational complexity of probabilistic inference using Bayesian belief networks." *Artificial intelligence* 42, no. 2 (1990): 393-405.

20. jQuery Source Viewer, http://james.padolsey.com/jquery/, (accessed:05.04.2014)

21. JSRefactor - http://www.brics.dk/jsrefactor/, (accessed:12.05.2014)

22. OpenMarkov - http://www.openmarkov.org/, (accessed:20.04.2014)

23. Madsen, M., Livshits, B., & Fanning, M. (2013, August). Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 499-509). ACM.

24. Guarnieri, S., & Livshits, V. B. (2009, August). GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium* (pp. 151-168).

25. Guarnieri, S., & Livshits, B. (2010, June). Gulfstream: Incremental static analysis for streaming JavaScript applications. In *Proceedings of the USENIX Conference on Web Application Development*.

26. Chugh, R., Meister, J. A., Jhala, R., & Lerner, S. (2009, June). Staged information flow for JavaScript. In *ACM Sigplan Notices* (Vol. 44, No. 6, pp. 50-62). ACM.

27. Jensen, S. H., Madsen, M., & Møller, A. (2011, September). Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 59-69). ACM.

28. Thiemann, P. (2005). Towards a type system for analyzing javascript programs. In *Programming Languages and Systems* (pp. 408-422). Springer Berlin Heidelberg.

29. Thiemann, P. (2005, January). A type safe DOM api. In *Database Programming Languages* (pp. 169-183). Springer Berlin Heidelberg.

30. Flanagan, David; Ferguson, Paula (2006). JavaScript: The Definitive Guide (5th ed.). O'Reilly & Associates. ISBN 0-596-10199-6.

31. JavaScript - http://en.wikipedia.org/wiki/JavaScript, (accessed:22.05.2014)

32. "ECMAScript Language Overview" (PDF). 2007-10-23. p. 4. Retrieved 2009-05-03.

33. ECMAScript, E. C. M. A., and European Computer Manufacturers Association. "ECMAScript Language Specification." (2011).

34. Scholte, T., Robertson, W., Balzarotti, D., & Kirda, E. (2012, July). Preventing input validation vulnerabilities in web applications through automated type analysis. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual* (pp. 233-243). IEEE.

35. Jensen, S. H., Jonsson, P. A., & Møller, A. (2012, July). Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (pp. 34-44). ACM.

36. Jensen, S. H., Møller, A., & Thiemann, P. (2011). Interprocedural analysis with lazy propagation. In *Static Analysis* (pp. 320-339). Springer Berlin Heidelberg.

37. Frigault, M., & Wang, L. (2008). *Measuring network security using bayesian network-based attack graphs* (pp. 698-703). IEEE.

38. Frigault, M., Wang, L., Singhal, A., & Jajodia, S. (2008, October). Measuring network security using dynamic bayesian network. In *Proceedings of the 4th ACM workshop on Quality of protection* (pp. 23-30). ACM.