Vulnerabilities Static Detection for Web Applications with False Positive Suppression

Jianjun Huang, Bin Liang, Jiagui Zhong, Qianqian Wang
1 Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE
2 School of Information, Renmin University of China
Beijing 100872, China
E-mail: {hjj, liangb, zhongjg, jessiewqq}@ruc.edu.cn

Abstract—Web applications become more and more important, and the corresponding security problems have been concerned about. This paper presents TASA, an ASP static analyzer, which employs a path-sensitive, inter-procedural and context-sensitive data flow analysis, mainly concerning the taint propagation and sanitization. This paper also discusses some techniques used in TASA, such as sanitization routines modeling, ASP specific features, alias analysis and path-related routines modeling, to prune false positives. Experiments on four open source applications show that TASA has a rate of false positive of 4.98% and it can avoid certain false warnings owing to the proposed approaches.

Keywords—ASP; data flow analysis; vulnerabilities static detection; false positive suppression

I. INTRODUCTION

Web applications have proliferated in recent years. They cover all kinds of online services ranging from news coverage, entertainment to banking and retailing. As the rapid development of Web applications, security vulnerabilities in those applications attract the attackers and they cause threats to either the service provider or the customers. The most common exploited vulnerabilities affecting Web applications are Cross-site Scripting (XSS) and SQL Injection, according to CWE [1]. These two defects are caused by lacking of input validation and data sanitization, typically as a result of the programming errors if the programmers do not program with security in mind.

One method to find these vulnerabilities is testing with a variety of abnormal data. However, testing is not as effective as supposed, taking into account that those attackers always exploit those holes with the least expected inputs and compromise the system. Static analysis is an alternative to find these errors in the source code. It is proved to be effective although it cannot find all the vulnerabilities and it always produces false warnings. Even so, static analysis for Web applications has increased in the past years, aiming at decreasing the risks being attacked. Most recent work pays attention to scripting languages like PHP and JSP [2,3,4].

This paper presents TASA, an ASP static analyzer. It automatically finds the taint-style vulnerabilities in ASP, especially for VBScript, by means of a path-sensitive, inter-procedural and context-sensitive data flow analysis. It utilizes the idea of security state propagation and sanitization to detect XSS and SQL Injection bugs. TASA also introduces some techniques to prune false positives. By observing and investigation, we find that TASA can be augmented at the following four aspects: modeling sanitization routines, handling ASP specific features, processing alias analysis and modeling path-related routines correctly. By utilizing these techniques, TASA is demonstrated to be effective enough with a false positive rate of 4.98% in a series of experiments on some real world code bases.

The rest of this paper is organized as follows. Section II gives an example to depict the taint-style vulnerability and section III describes the analysis engine TASA. False positive suppression is in section IV and experiments are in section V. Section VI compares our work to some related works and section VII concludes.

II. MOTIVATING EXAMPLE

Fig.1 gives a simplified real code snippet to demonstrate the SQL Injection, a major type of taint-style vulnerabilities. Lines 2 to 4 were not in this real code until the system was attacked. Without them, attackers can easily compromise the system by feeding the parameter NewsID with a tainted value. With “1 or 1 = 1”, the SQL statement becomes:

```vbnet
select * from News where NewsID=1 or 1 = 1,
which is equal to:
select * from News.
```

Then rs actually holds all the records in the table and may show them on the browser, causing information leak. If an attacker appends some other malicious shell commands to the SQL statement, it will lead to more serious damages.

Because the acceptable NewsID should be numeric, judging whether it is numeric before appending it to a SQL statement is an effective way to avoid attacks. Lines 2 to 4

```vbnet
1 NewsID=request.queryString("NewsID")
2 If not isNumeric (NewsID) then
3   response.end
4 End If
5 Set rs=server.CreateObject("adodb.recordset")
6 rs.Source="select * from News where NewsID=" & request.queryString("NewsID")
7 rs.Open rs.Source,conn,1,3
```

Figure 1. Motivating example for SQL Injection

fulfill the job.
A static detector may not handle this code correctly for the following reasons. First, the detector may not model the built-in routines correctly, e.g. isNumeric(). Second, some detectors (e.g. Pixy [3]) employ a traditional flow-sensitive analysis which combines the program states into a larger program state when both branches converge on Line 5. False warning emerges partially due to this kind of analysis and partially due to the misunderstanding of response.end(). Third, lack of alias analysis can also produce false warning even if the detector has handled the former two problems properly. Without understanding that the sanitized NewsID on Line 2 is the same as request.queryString(“NewsID”) on Line 6, the detector will make a mistake.

In the following sections, we will see how TASA handles these problems in right ways and shows low false positive rate for this code as well as for the other real code.

III. TASA: THE ASP STATIC ANALYZER

This section mainly describes TASA, an ASP static analyzer. Generally, TASA preprocesses each ASP file, finds security holes by means of a path-sensitive, inter-procedural and context-sensitive data flow analysis with a predefined library of modeled routines and reports bugs to users. For the analysis, TASA tracks the propagation and sanitization of the security state of the variables holding user accessible data which originate from those so-called entry points. The security state propagates to another by assignment or calling a user defined method as a parameter; it can also be sanitized when passing through the sanitization routines or some ASP specific features.

A. Preprocessing

Preprocessing contains parsing the ASP files with an ANTLR[5] grammar based parser, constructing abstract syntax tree (AST), processing included files recursively and merging their ASTs into the main AST, collecting class related information for later detection.

ASP does not support composed include statement with arbitrary expression, which simplifies the preprocessing. After parsing each included file and building an AST for it, TASA replaces the include statement with the AST and finally makes a large tree. Notice that any file may contain one or more inclusions, so this work should be recursive.

Once accomplishing the AST construction, TASA traverses the whole AST to record the user defined methods and classes. They are mapped into a hash table for quick searching. For a class, TASA goes into the class body, traverses all the statements and puts <name, AST> pairs of the fields, properties and methods into hash tables.

B. Data flow analysis

This section will detail the data flow analysis used in TASA. There are two main security state defined in TASA: tainted and untainted. All user accessible data are initialized as tainted before they are sanitized; constant is considered as untainted and will not be tracked in order to save space. TASA tracks tainted variables and checks whether they are still tainted at a sensitive sink. If so, it reports a bug. Next, we will take a look at these features.

Take Fig.2 for example. In a traditional flow-sensitive but not path-sensitive analysis, the detector clones the program state at program point B and selects the next basic block (e.g. C) to continue the analysis until it reaches point E where the detector saves the current state, return to B and restores the program state to enter into the other branch D. Before reaching point E, detector merges two states along with each branch into a larger state and uses it in the following analysis. In a path-sensitive analysis, however, the detector does not stop and return when it first encounters the merging point at E. It returns to B after it has finished detecting the whole path A→B→D→E→F. Then it returns to B, restores the program state and follows the other path D→E→F. Merging at the end of branches often makes alias analysis and bug tracing more difficult. In contrast, a path-sensitive analysis is more straightforward though it may suffer path explosion. We have not solved this problem in TASA and leave it as future work.

Inter-procedural analysis is a way to get a global view of the whole program. It is also an effective way to find more real bugs and prune false warnings. TASA follows a call and enters into the callee with a refined program state. After exiting from the callee, TASA restores the program state at the call site using the returned state of the callee. For precision and coverage, TASA also handles those procedure in a user defined class. Namely, TASA is able to process the object-oriented features.

Context-sensitivity means that each call to a procedural is different to another even if they are calling to the same one because they have distinct contexts. To guarantee context-sensitivity, TASA clears all the local information bounded to a procedural when it exits from that one and creates new security state for each local variable when it reenters the method. Context-sensitivity is based on inter-procedural analysis but enhances the bug finding tool to find more real bugs and report less false warnings.

C. State propagation

State propagation performs in two forms: assigning and calling a method. The latter one can be seen as a special case of assigning. Without losing generality, we focus on the assigning. Generally, an assignment takes the following form:

\[ v ← v_1 \{ \text{op} \} \{ \text{op} \} \]

One or more variables and constants at right hand side (rhs) are concatenated by operators and propagate their values as well as their security states to the variable at left hand side (lhs). TASA only processes string concatenating operator when it propagates the state, considering the target of detecting. The basic rules for state propagation are:

\[
\begin{align*}
\exists v_r \quad & \begin{cases} 11 & \text{page_no = request("\text{page_no}")} \\
\in \text{RHSV} : & 207 \begin{cases} \text{<a href="...?page_no=\%\text{page_no}-1 \%\&...} \\
& v_l \quad \text{is tainted} \\
& \Rightarrow v \quad \text{is tainted, and}
\end{cases}
\end{cases}
\end{align*}
\]

Figure 2. A sample control flow graph. Each node is a basic block.

Figure 3. Code sample with type conversion sanitization
∀ v_i ∈ RHSV: v_i is untainted → v is untainted, where RHSV denotes the set of all the variables at rhs and v is the variable at lhs.

In Fig.1, if request.queryString("NewsID") has not been sanitized on Line 6, it is tainted and the string literal is untainted, so rs.Source at lhs is tainted.

For method calls, formal parameters and actual parameters are seen as lhs and rhs respectively and the state propagation obeys the rules.

IV. FALSE POSITIVE SUPPRESSION

A bug finding tool should identify as many vulnerabilities as possible and try to avoid false warnings. A lot of false warnings will confuse the users and lower the value of the tool. This section mainly talks about the techniques used in TASA for false positive suppression. Some can be applied to other languages and some are ASP specific.

A. Sanitization routines modeling

There are many built-in routines in ASP. Some have nothing to do with static detection while some affect it in different ways. This subsection focuses on two kinds of routines that have sanitization effects. One is related to the output while the other is for the parameters. Many scripting languages provide equivalent sanitization routines we discuss here and they offer the basic prevention to web applications in most cases. The followings are two instances for each kind.

Server.HTMLEncode() belongs to the first kind. It encodes the HTML tags and outputs them to the browser. The browser displays these tags as normal text. It is useful to avoid XSS attacks to use this routine when the server outputs something to the browser. IsNumeric() is the second type of sanitization routines. Passing through the diagnosis means the parameter is numeric and then cannot trigger an XSS or SQL Injection attack. We can see it in Fig.1 where it states that Newsletter is aliased with request.queryString("NewsID") by a simple assignment on Line 1, and the sanitization on Line 3 also sanitizes request.queryString("NewsID"). This promises the rhs on Line 6 is untainted in the feasible path and then TASA does not report a false bug on Line 7.

B. ASP specific features

Different languages have different features which may impact the detector on false warning issues. Take PHP for example: the implicit type conversion mechanism converts a non-numeric value to zero, which results in a curious situation that string can be in an arithmetic expression. In ASP, however, it always throws an error when a variable need to be numeric but it actually holds a string. TASA takes advantage of this mechanism to prevent incorrect warnings.

Arithmetic expressions then can sanitize user accessible data if they meet an arithmetic operator. Operator “+” is an exception because it can be used as a string concatenating operator. It acts as the other operators provided either operand is a number.

Besides, conversion routines can fulfill this job too, e.g. Int(). It emits an error if the parameter is not an integer. This is different to the sanitization effect of IsNumeric() discussed in the last subsection.

The advantage of type conversion also occurs in a conditional statement when user accessible data are compared with a number. It involves all the comparison operators. In contrast, comparing to a string literal has no effect of sanitization except equivalence testing. Comparing with some specific constants like Empty and Null is the same as string comparison but they are indeed not ASP specific features.

Based on the analysis, TASA can convert each tainted source into untainted without any side effect if it passes though the above type conversion and so prunes false positives. Fig.3 shows a real program fragment. Without these considerations, a bug finding tool will report a false warning.

C. Alias analysis

Alias analysis is often used in static analysis for precision. In order to prune false bug report, TASA introduces a state-based alias analysis. Take the code in Fig.1 for example. Newsletter is aliased with request.queryString("NewsID") by a simple assignment on Line 1, and the sanitization on Line 3 also sanitizes request.queryString("Newsletter"). This promises the rhs on Line 6 is untainted in the feasible path and then TASA does not report a false bug on Line 7.

D. Path-related routines modeling

As we mentioned in section II, incorrect understanding of the semantics of built-in routines may lead to false warnings. An example is calling respond.end() in Fig.1. Misunderstanding of the routine will produce a wrong control flow graph as shown in Fig.4(a) while the right one is Fig.4(b).

Following the path 1->2->3->5->7 in Fig.4(a), a detector may report a bug at Line 7 even if it has handled the alias analysis described in last subsection.

V. EXPERIMENTS

We have performed a series of experiments on TASA to demonstrate its ability of detecting XSS and SQL Injection vulnerabilities. To this end, it ran on four open source ASP applications: ASP Portal v3.2.4 [6], aspWebCalendar FREE v1.1 [7], RapidClassified v3.15 free edition [8] and ASPNews v2.21 [9]. Almost each alone program was analyzed in several seconds in a laptop with 2.2 GHz Core Duo CPU and 2GB RAM.

Table 1 summarizes the empirical results or the experiments. TASA has found 144 XSS bugs and 77 SQL Injection vulnerabilities. Though some bugs occur at the same location, they come from different paths.
We manually checked each reported bug and found the rate of false positive is 4.98%. Before applying the techniques for false warning suppression discussed in section IV, the reported number of XSS flaws was 86 in ASP Portal. These techniques also decreased by 1 XSS and 6 SQL Injection warnings for RapidClassified and ASP Portal respectively. The decreased warnings are all false bugs.

VI. RELATED WORK

Y. Xie and A. Aiken present an inter-procedural static analysis algorithm for PHP, mainly focusing on the SQL injection vulnerabilities [2]. Their analysis employs novel three-tier architecture. They take into account sanitization routines like is_numeric() just as we do, but their tool requires user’s interaction which makes the analysis not totally automated. Pixy is an utterly automatic detector for taint-style vulnerabilities in PHP with a flow-sensitive, inter-procedural and context-sensitive data flow analysis [3]. The reported false positive rate of Pixy is 50% and it only concerns the first kind of sanitization routines and ignores the others like is_numeric(). It neither process object-oriented features.

Wassermann and Su introduce a static analysis technique to find subtle SQL injection flaws for PHP in [10]. They develop a mechanism to determine the possible string values of variables and check whether they are tainted when reaching a sensitive sink.

MSScasi is a static code analysis tool for finding SQL injection vulnerabilities in ASP code provided by Microsoft [11]. The tool seems to use a flow-sensitive and inter-procedural taint analysis. It is highly efficient when analyzing large scripting files. Tests show that it has some serious limitations. First, it only models Request.QueryString() and Request.Form() as tainted sources. It neglects the other routines that can also import untrusted user inputs such as Request.Cookies(). Second, it is intra-procedural, so it cannot find out those vulnerabilities across function boundaries. Last, it seems MSScasi never takes into consideration the sanitization effect we discussed in section IV and the tool report a SQL Injection warning for the code in Fig.1. When the tool is applied to the code in Fig.5, which is a snippet of ASPNews, MSScasi fails to report a SQL Injection flaw though it has modeled Request.Form(). TASA succeeds finding the bug.

VII. CONCLUSION

Web applications have become a popular and important interaction medium in our daily lives. Accompanying the proliferation of Web applications, security vulnerabilities that endanger users and service providers are discovered regularly. Certain vulnerabilities are due to the ignorance of security of programmers. They are embedded into the program logic and should be swept out before the applications are deployed. Therefore, this paper presents a static analysis approach, TASA, to detect taint-style vulnerabilities in ASP, particularly XSS and SQL Injection. We test the tool on real world applications and experiments show that TASA has a very low rate of false positive. We believe we have provided an effective solution to solve the security vulnerabilities in Web applications, therefore offering benefits to both users and application providers.

REFERENCES


Figure 5. A code snippet of ASPNews.