Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses

Fang Yu Muath Alkhalaf Tevfik Bultan
Computer Science Department, University of California at Santa Barbara
Email: {yuf,muath,bultan}@cs.ucsb.edu

Abstract

Given a program and an attack pattern (specified as a regular expression), we automatically generate string-based vulnerability signatures, i.e., a characterization that includes all malicious inputs that can be used to generate attacks. We use an automata-based string analysis framework. Using forward reachability analysis we compute an over-approximation of all possible values that string variables can take at each program point. Intersecting these with the attack pattern yields the potential attack strings if the program is vulnerable. Using backward analysis we compute an over-approximation of all possible inputs that can generate those attack strings. In addition to identifying existing vulnerabilities and their causes, these vulnerability signatures can be used to filter out malicious inputs. Our approach extends the prior work on automata-based string analysis by providing a backward symbolic analysis that includes a symbolic pre-image computation for deterministic finite automata on common string manipulating functions such as concatenation and replacement.

1. Introduction

Web applications provide critical services over the Internet and frequently handle sensitive data. Unfortunately, Web application development is error prone and results in applications that are vulnerable to attacks by malicious users. According to the Open Web Application Security Project (OWASP)'s top ten list that identifies the most serious web application vulnerabilities, the top three vulnerabilities are: 1) Cross Site Scripting (XSS), 2) Injection Flaws (such as SQL injection) and 3) Malicious File Execution. All these vulnerabilities are due to improper string manipulation. Programs that propagate and use malicious user inputs without sanitization or with improper sanitization are vulnerable to these well-known attacks.

This work is supported by NSF grants CCF-0614002 and CCF-0716095.

In this paper, we propose a string analysis approach that 1) identifies if a web application is vulnerable to attacks, and 2) if it is vulnerable, generates a characterization of user inputs that might exploit that vulnerability. Such a characterization is called a vulnerability signature. We focus on vulnerabilities related to string manipulation such as the ones listed above. Vulnerabilities related to string manipulation can be characterized as attack patterns, i.e., regular expressions that specify vulnerable values for sensitive operations (called sinks).

Given an application, vulnerability analysis identifies if there are any input values that a user can provide to the application that could lead to a vulnerable value to be passed to a sensitive operation. Once a vulnerability is identified, the next important question is what set of input values can exploit the given vulnerability. A vulnerability signature is a characterization of all such input values [1]. A vulnerability signature can be used to identify how to sanitize the user input to eliminate the identified vulnerability, or it can be used to dynamically monitor the user input and reject the values that can lead to an exploit.

We use automata-based string analysis techniques for vulnerability analysis and vulnerability signature generation. Our tool takes an attack pattern specified as a regular expression and a PHP program as input and 1) identifies if there is any vulnerability based on the given attack pattern, 2) generates a DFA characterizing the set of all user inputs that may exploit the vulnerability.

Our string analysis framework uses deterministic finite automaton (DFA) to represent values that string expressions can take. At each program point, each string variable is associated with a DFA. To determine if a program has any vulnerabilities, we use a forward reachability analysis that computes an over-

```
1 <?php
2    $www = $_GET["www"];
3    $l_otherinfo = "URL";
4    $www = preg_replace(
              "/[^A-Za-z0-9 .-@://]/",
              "",
              $www);
5    echo $l_otherinfo . ": " . $www ;
6 ?>
```

Figure 1. A Small Example

approximation of all possible values that string variables can take at each program point. Intersecting the results of the forward analysis with the attack pattern gives us the potential attack strings if the program is vulnerable. The backward analysis computes an over-approximation of all possible inputs that can generate those attack strings. The result is a DFA for each user input that corresponds to the vulnerability signature.

The techniques proposed in this paper build on our earlier results on string analysis reported in [2], where we only discuss forward symbolic analysis and do not address vulnerability signature generation problem. The vulnerability signature generation approach presented in [3] is a backward analysis similar to the second phase of our analysis. However, they require loop invariants to be provided by the user in order to handle loops whereas we use an automated approach based on widening, and they focus on weakest precondition computation for binary programs whereas we focus on string manipulation operations. Compared to recent work on attack generation (for example [4]), we propose a sound static analysis approach that characterizes all possible inputs that can exploit a given attack pattern, rather than generation of concrete attacks using dynamic analysis techniques based on given exploits. Our key contributions in this paper are 1) The backward symbolic analysis based on backward image computation for string operations such as concatenation and replacement, and 2) A new approach to vulnerability signature generation problem that combines symbolic forward and backward analyses.

2. An Overview

In this section we will give an overview of our analyses using the simple PHP script shown in Figure 1. This script is a simplified version of code from a real web application that contains a vulnerability. The script starts with assigning the user input provided in the `_GET` array to the `www` variable in line 2. Then, in line 3, it assigns a string constant to the `l_otherinfo` variable. Next, in line 4, the user input is sanitized using the `preg_replace` command. This replace command gets three arguments: the match pattern, the replace pattern and the target. The goal is to find all the substrings of the target that match the match pattern and replace them with the replace pattern. In the replace command shown in line 4, the match pattern is the regular expression `[^A-Za-z0-9 .-@://]`, the replace pattern is the empty string (which corresponds to deleting all the substrings that match the match pattern), and the target is the variable `www`. After the sanitization step, the PHP program outputs the concatenation of the variable `l_otherinfo`, the string constant `": "`, and the variable `www`.

The `echo` statement in line 5 is a sink statement since it can contain a Cross Site Scripting (XSS) vulnerability. For example, a malicious user may provide an input that contains the string constant `<script` and execute a command leading to a XSS attack. The goal of the replace statement in line 4 is to remove any special characters from the input to prevent such attacks.

Using string replace operations to sanitize user input is a common practice in web applications. However, this type of sanitization is error prone due to complex syntax and semantics of regular expressions. In fact, the replace operation in line 4 in Figure 1 contains an error that leads to a XSS vulnerability. The error is in the match pattern of the replace operation: `[^A-Za-z0-9 .-@://]`. The goal of the programmer was to eliminate all the characters that should not appear in a URL. The programmer implements this by deleting all the characters that do not match the characters in the regular expression `[A-Za-z0-9 .-@://]`, i.e., eliminate everything other than alpha-numeric characters, and the ASCII symbols `.`, `-`, `@`, `:`, and `/`. However, the regular expression is not correct. First, there is a harmless error. The subexpression `//` can be replaced with `/` since repeating the symbol `/` twice is unnecessary. More serious error is the following: The expression `.-@` is the union of all the ASCII symbols that are between the symbol `.` and the symbol `@` in the ASCII ordering. The programmer intended to specify the union of the symbols `.`, `-`, and `@` but forgot that symbol `-` has a special meaning in regular expressions when it is enclosed with symbols `[` and `]`. The correct expression should have been `.\-@`. This error leads to a vulnerability because the symbol `<` (which can be used to start a script to launch a XSS attack) falls between the symbol `.` and the symbol `@` in the ASCII ordering. So, the sanitization operation fails to delete the `<` symbol from the input, leading to a XSS vulnerability.

Now, we will explain how our approach automatically detects this vulnerability. First, the attack pattern for the XSS attacks can be specified as Σ^* `<script` Σ^*, i.e., any string that contains the substring `<script` matches the attack pattern. If, during the program execution, a string that matches the attack pattern reaches a sink statement, then we say that the program is vulnerable. For our small example, we simplify the attack pattern as Σ^* < Σ^*. Our analysis first generates the dependency graph for the input PHP program. Figure 2 shows the dependency graph for the PHP script in Figure 1. (the program segment that corresponds to a node and the corresponding line number are shown inside the node). Nodes 1 and 2 correspond to the assignment statement in line 2, nodes 3 and 4, correspond to the assignment statement in line 3, nodes 5, 6, 7 and 8 correspond to the replace statement in line 4, and nodes 9, 10, 11, and 12 correspond to the concatenation operations and the echo statement in line 5. Under each node we show the result of the forward and backward symbolic analyses as a regular expression.

During forward analysis we characterize all the user input as Σ^*, i.e., the user can provide any string as input. Then, using our automata-based forward symbolic reachability analysis, we compute all the possible values that each string expression in the program can take. For example, during forward analysis, node 2, that corresponds to the value of the string variable `www` after the execution of the assignment statement in line 2, is correctly identified as Σ^*. More interestingly, node 8, the value of the the string variable `www` after the execution of the replace statement in line 4, is correctly identified as `[A-Za-z0-9 .-@:/]*` since any character that does not match the characters in the regular expression `[A-Za-z0-9 .-@://]` has been deleted.

Node 12 is the sink node. The result of the forward analysis identifies the value of the sink node as `URL:[A-Za-z0-9 .-@:/]*`. Next, we take the intersection of the result of the forward analysis with the attack pattern to identify if the program contains a vulnerability. If the intersection is empty then the program is not vulnerable with respect to the given attack pattern. Since our analysis is sound, this means that there is no user input that can generate a string that matches the attack pattern at the sink node. However, in our example, the intersection of the attack pattern and the result of the forward analysis for the sink node is not empty and is characterized by the following regular expression: `URL:[A-Za-z0-9 .-;=-@:/]*<[A-Za-z0-9 .-@:/]*`. The backward analysis starts from this intersection and traverses the dependency graph backwards to find out what input values can lead to string values at the sink node that falls into this intersection. Note that during backward analysis we do not need to compute any value for the nodes that are not on a path between an input node and a sink node. This means that during backward analysis we do not compute values for the nodes 3, 4, 5, 6, 9 and 10. The final result of the backward analysis is the result for the input node 1, which is characterized with the regular expression: `[^<]*<`Σ^*, i.e., any input string that contains the symbol < can lead to a string value at a sink node that matches the attack pattern. Using this information, the programmer can eliminate the vulnerability either by fixing the erroneous replace statement in line 4 or by adding another replace statement that removes the < symbol from the input.

3. Vulnerability Analysis

Our automata-based vulnerability analysis consists of two phases. In the first phase, we perform a forward symbolic reachability analysis from root nodes to compute all possible values that each node can take. We use this information to collect vulnerable program points, as well as the reachable attack strings of those vulnerable program points. If the program is vulnerable, i.e., there exists some vulnerable program points, we proceed to the second phase. In the second phase, we perform a backward symbolic reachability analysis from the vulnerable program points to compute all possible values of their predecessors that will result in attack strings at these vulnerable program points.

As shown in Algorithm 1, our analysis takes the following inputs: a dependency graph (denoted as G), a set of sink nodes (denoted as *Sink*), and an attack pattern (denoted as *Attk*). G is a directed graph that specifies how the values of user inputs flow to the sensitive functions. *Sink* denotes the nodes that are associated with sensitive functions that might lead to vulnerabilities. *Attk* is a regular expression represented as a DFA that accepts the set of attack strings. The set of string values is approximated as a regular language and represented symbolically as a DFA that accepts the language. To associate each node with its automata, we create two automata vectors *POST* and *PRE*. The size of both is bounded by the number of nodes in G. *POST*$[n]$ is the DFA accepting all possible values that node n can take. *PRE*$[n]$ is the DFA accepting all possible values that node n can take to exploit the vulnerability. Initially, all these automata accept nothing, i.e., their language is empty. *Vul* $\subseteq$ *Sink* is the set of vulnerable program points and initially is set to an empty set.

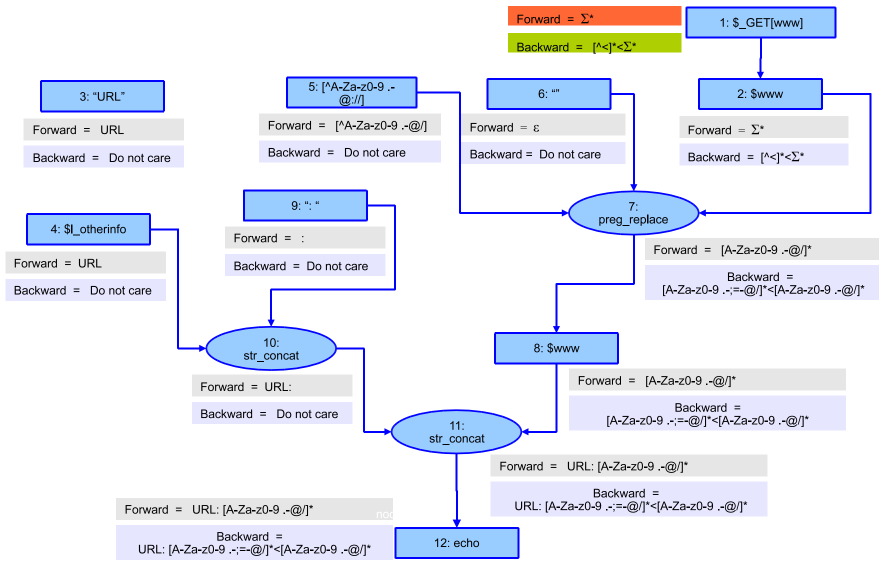


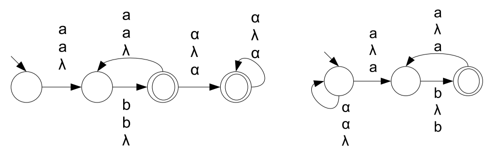
Figure 2. Results of Forward and Backward Analyses

At line 3, we first compute *POST* by calling the forward analysis. At line 4, for each node $n \in$ *Sink*, we generate a DFA *tmp* by intersecting the attack pattern and the possible values of n. If the language of *tmp*, i.e., $L(tmp)$, is not empty, we identify that n is a vulnerable program point and add it to *Vul* at line 7. In fact, *tmp* accepts the set of reachable attack strings at node n that can be used to exploit the vulnerability. Hence, we assign *tmp* to *PRE*$[n]$ at line 8. If *Vul* is not empty, we compute *PRE* by calling our backward analysis at line 12. Note that for $n \in$ *Vul*, *PRE*$[n]$ has been assigned. We report vulnerability signatures for each `input` node based on *PRE* at line 13-15. If *Vul* is an empty set, we report that the program is secure with respect to the attack pattern.

Algorithm 1 VULANALYSIS(G, *Sink*, *Attk*)

```
1:  Init(POST, PRE);
2:  set Vul := {};
3:  FWDANALYSIS(G, POST);
4:  for each $n \in Sink$ do
5:      tmp: = $POST[n] \cap Attk$;
6:      if $L(tmp) \neq \emptyset$ then
7:          Vul := $Vul \cup \{n\}$;
8:          $PRE[n] := tmp$;
9:      end if
10: end for
11: if $Vul \neq \emptyset$ then
12:     BWDANALYSIS(G, POST, PRE, Vul);
13:     for each input $n$ do
14:         Report the vulnerability signature $PRE[n]$;
15:     end for
16:     return "Vulnerable";
17: else
18:     return "Secure";
19: end if
```

The forward symbolic reachability analysis is based on a standard work queue algorithm. We iteratively update the automata vector *POST* until a fixpoint is reached [2]. Backward analysis uses the results of the forward analysis. Particularly, we are interested in computing all possible values of each node n that can exploit the identified vulnerability. The challenge of the backward analysis comes from the pre-image computation on string manipulating functions. To compute the pre-image of concatenation, we introduce concatenation transducers. A concatenation transducer M is a multi-track DFA that identifies the prefix and suffix relations precisely by binding the values of input and output tracks character by character. Below we show two examples of concatenation transducers that are used to compute the pre-image of the concatenation of a constant set with a variable. Let α indicate any character in Σ. In Figure 3 (a), the third track of M can be used to identify all suffixes of X that follow any string in $(ab)^+$. In Figure 3 (b), the second track of M can be used to identify all prefixes of X that are followed by any string in $(ab)^+$. To compute the pre-image of replace commands, e.g., `preg_replace("a","b",v)`, we replace the values of the replace pattern ($\{b\}$) with the values of both the match pattern and the replace pattern ($\{a, b\}$). This operation is achieved by using the language-based replacement proposed in [2]. If the replace operation performs deletion, e.g., `preg_replace("a","", v)`, the pre-image accepts that the values of the match pattern ($\{a\}$) to be repeated many times between any character. Details of our forward and backward analyses can be found in [5].



(a) $X = (ab)^{+}.Z$ (b) $X = Y.(ab)^{+}$

Figure 3. Concatenation transducers

4. Experiments

We experimented on a number of benchmarks extracted from known vulnerable web applications: (1) `MyEasyMarket-4.1` (a shopping cart program), (2) `PBLguestbook-1.32` (a guestbook application), (3) `BloggIT-1.0` (a blog engine), and (4) `proManager-0.72` (a project management system).

In our experiments, we used an Intel machine with 3.0 GHz processor and 4 GB of memory running Ubuntu Linux 8.04. We use 8 bits to encode each character in ASCII. The performance of our vulnerability analysis is shown in Table 1 and Table 2.

	total time(s)	fwd time(s)	bwd time(s)	mem(kb)
1	0.569	0.093	0.474	2700
2	3.449	0.124	3.317	5728
3	1.087	0.248	0.836	18890
4	16.931	0.462	16.374	116097

Table 1. Total Performance

	CONCAT	REPLACE	PRECONCAT	PREREPLACE
	#operations/time(s)			
1	6/0.015	1/0.004	2/0.411	1/0.004
2	19/0.082	1/0.004	11/3.166	1/0.0
3	22/0.038	4/0.112	2/0.081	4/0.54
4	14/0.014	12/0.058	26/11.892	24/3.458

Table 2. String Function Performance

Table 3 shows the data about the DFAs that our analyses generated. Reachable Attack is the DFA that accepts all possible attack strings at the sink node. Vulnerability Signature is the DFA that accepts all possible malicious inputs that can exploit the vulnerability. We closely look at the vulnerability signature of (1) `MyEasyMarket-4.1`. The signature actually accepts α^* `<`α^* `s`α^* `c`α^* `r`α^* `i`α^* `p`α^* `t`α^* with respect to the attack pattern Σ^* `<script`Σ^*. α is the set of characters, e.g., `!`, that are deleted in the program. An input such as `<!script` can bypass the filter that rejects Σ^* `<script`Σ^* and exploit the vulnerability. This shows that simply filtering out the attack pattern can not prevent its exploits. On the other hand, the exploit can be prevented using our vulnerability signature instead. Both vulnerability signatures of (2) `PBLguestbook-1.32` accept arbitrary strings. By manually tracing the program, we find that both inputs are concatenated to an SQL query string without proper sanitization. Since an input can be any string, the pre-image of one input is the prefix of Σ^* `OR '1'='1'`Σ^* that is equal to Σ^*, while the pre-image of another input is the suffix of Σ^* `OR '1'='1'`Σ^* that is also equal to Σ^*. This case shows a limitation in our approach. Since we do not model the relations among inputs, we can not specify the condition that one of the inputs must contain `OR '1'='1'`.

	Reachable Attack (Sink)		Vulnerability Signature (Input)	
	#states	#bdd nodes	#states	#bdd nodes
1	24	225	10	222
2	66	593	2	9
3	29	267	92	983
4	131	1221	57	634
	136	1234	174	1854
	147	1333	174	1854

Table 3. Attack and Vulnerability Signatures

5. Conclusion

We presented symbolic string analysis techniques for identifying vulnerabilities and vulnerability signatures. Our approach is based on automata-based symbolic forward and backward reachability computations. We applied our approach to automated analysis of PHP programs. Our analyses successfully find vulnerabilities in existing web applications and generate vulnerability signatures identifying how these vulnerabilities can be eliminated.

References

[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, “Theory and techniques for automatic generation of vulnerability-based signatures,” *IEEE Trans. Dependable Sec. Comput.*, vol. 5, no. 4, pp. 224–241, 2008.

[2] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, “Symbolic string verification: An automata-based approach,” in *Proc. of SPIN*, 2008, pp. 306–324.

[3] D. Brumley, H. Wang, S. Jha, and D. X. Song, “Creating vulnerability signatures using weakest preconditions,” in *Proc. of CSF*, 2007, pp. 311–325.

[4] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of SQL injection and cross-site scripting attacks,” in *Proc. of ICSE*, 2009.

[5] F. Yu, M. Alkhalaf, and T. Bultan, “Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses,” Technical Report 2009-11, UCSB CS, 2009.