Challenges for Security Typed Web Scripting Languages Design

Co-author: **Zeba Nazleen Sajjade**
Research Scholar,
Sajjade Daraqshan Arshiya
Zill-E-Ilahi Shaheen

Co-author: **Gulfishan Tamkanat**
Sajjada (Research Scholar)
Department of computer science,
JJT University, Rajasthan

Abstract:

This paper focuses on the different challenges to design a security typed web scripting language. It uses the type system approach on a simple imperative language that captures a subset of the security typed web language constructs to express the security properties that must be held in the language with respect to its formal semantics to prevent insecure information flow in web application system and hence the common web application security vulnerabilities.

Keywords: *Information flow security, web application vulnerabilities, web scripting language, type system.*

1. Introduction

The web application vulnerabilities can be viewed as an insecure information flow problem and the different security mechanisms such as access control and encryption are not enough to guarantee a secure information flow, as they do not directly address the enforcement of information-flow security [13] policies in web application system.. Recently, a promising new approach has been developed: the use of programming-language techniques for specifying and enforcing information flow policies [7, 8]. However, to apply such languages in practice is not an easy task and needs much effort of the programmer to develop a real application. To this end we formalize the different web application vulnerabilities as an insecure information flow problem and illustrate the challenges for using the language based information flow control in the design of the web scripting languages [7] to track the insecure information flow in web application system and automatically patch it against the different attacks that exploit such vulnerabilities.
The remainder of this paper is organized as follows:
section 2 discusses the information flow control in web application system infrastructure. Section 3 introduces the most widely exploited web application vulnerabilities. Section 4 discusses the limitation of the existing web scripting languages in enforcing information flow security policy. Section 5 discusses the challenges. Finally we conclude in section 7.

2. Information flow control and integrating it with web infrastructure

Information-flow systems usually specify policies in terms of a lattice of security labels, where labels present the security levels of data. We assume that the security lattice L have the security levels *l*1, *l* 2. The ordering specifies the relationship between the two security levels. If *l*1 □ *l* 2, then information at level *l* 11is also visible at level *l* 2. However, if *l* 1≠ *l* 2, then information at level *l* 1 is invisible at level *l*2.

The top element in the lattice is T such that □ *l* □L, *l* ≤ T and the bottom element in the lattice is □ such that □ *l* □L, □ ≤ *l*. In web applications, The input data coming from the client web browser can be treated as having the security labels public and tainted, while the information on the server side have the security labels secret and untainted. Here the security levels are considered elements of the two simple security lattices, public □secret for confidentiality and untainted□ tainted for integrity as shown in figure (1).
Secret Tainted
Public Untainted

Figure1. Simple security lattices for confidentiality and integrity

3. Web application vulnerabilities

There are some web application vulnerabilities that are considered as a violation for the confidentiality policy of web application system, while there are others that are considered as a violation for the integrity policy.

One of the most common Web application vulnerabilities that violate the confidentiality policy is the ***broken access control*** [16] that exploits a weakness in the access control mechanism of web application program to break it and reveal sensitive The Fourth International Conference on Information Assurance and information. This is confidentiality policy violation, as the user access rights to the web application system may be broken by an attacker and the user's secret data can be revealed to the public without his or her permission. Another example of web application vulnerabilities that violate the confidentiality policy is the ***laundering attack*** [1], where the attacker can launder information through programs that make intentional information release or declassification for secret data.

One of the most common Web application vulnerabilities that violate the integrity policy is the ***Cross-site scripting (XSS)*** [3,4] , where a malicious script is prepared by an attacker and delivered on behalf of the Web server, it is granted the same trust level as the Web server, which at minimum allows the script to read user cookies set by that server. This often reveals passwords or allows for session hijacking.

Another example of web application vulnerabilities that violate the integrity policy is the ***SQL injection vulnerabilities*** [3,4] which are more dangerous than XSS and occur when un trusted values are used to construct SQL commands, resulting in the execution of arbitrary SQL commands given by an attacker. This technique is considered a violation for the database integrity, as it allows for the arbitrary manipulation of backend database.

Web application vulnerabilities formalization

From the description of the most common web application vulnerabilities mentioned above, we formalize them as a problem of confidentiality and integrity violation. The violation in data confidentiality occurs if the program reveals secret information or leaks more information than intended. The violation in data integrity occurs when un trusted data is used to construct trusted output without sanitization, leading to escalation in access rights that cause unauthorized release of secret information and hence result in confidentiality compromises.

4. Information flow security limitation in existing web languages

Currently web development languages lack the information flow security policy enforcement in web application system to protect its confidentiality and integrity. This creates many security vulnerabilities in its infrastructure that expose it to different types of attacks.

In the current web scripting languages, the confidentiality policy specification is expressed implicitly in web application system based on programmer experience and this specification is not expressed as security levels given to the data types. As a result, the access control policy may be broken in the web application system by unauthorized user. In addition, even if the current web scripting languages will be extended with new features to enforce the confidentiality policy explicitly in the program, they will lack a declassification mechanism to secure programs that make intentional information release for secrets, so an attacker can launder more information than intended through this type of programs, thus the confidentiality policy may be violated.

Also the current web scripting languages do not enforce the integrity policy, as there is no label checking used in static checking techniques to assign different trust levels in relation to different users such as the owner of the data and the users who is authorized to modify the data. As a result, the restriction of the modification of certain data types to be controlled by certain owner such as root or authorized users are not enforced explicitly in an information flow policy, but specified implicitly in the program by the programmer

effort. As a result, the web application system may be exposed to the well known Cross-Site Scripting attack. In addition, the language classes or methods to have no authority to run with limited privilege related to specific users or principals and hence an unsafe script could use unchecked input string to compose SQL queries and then have the DBMS execute the query without the privilege of a trusted user (the so-called SQL injection attack). Thus, the integrity policy for the web application system can be violated.

Nowadays the most widely used security mechanism in scripting languages related to information-flow is the "taint checking" mode for the scripting language Perl. With this feature enabled, Perl scripts mark any value that is retrieved from a source external to the script (outside the application) as **tainted** and raise an error if it is passed to potentially exploitable functions (such as system and exec calls).

Perl taint mode [10] enables a dynamic enforcement for integrity policy by making a set of special security checks during execution in an unsafe environment. So it does not track implicit flows that arise from the control structure of the program. Perl is not suitable for enforcing confidentially policy as it does not prevent leaking of secret information. Taint checking only seeks to reduce vulnerabilities, not eliminate them.

5. Challenges and proposed solution

The most widely used security typed languages such as Jif [2, 6, 12] and Flow Camel [9] languages are not web languages and they can't be used directly to write a secure web application program. So to design a security typed web language that uses static type checking technique to guarantee a secure information flow and enforce the confidentiality and integrity policy explicitly in web application program based on associating its ordinary data type such as int with a security level during its declaration is one of the most important challenges.

A program written in a sort of security-typed language is considered secure only if it is well-typed, which rules variables with high security level (confidential) in a well-typed program do not interfere or with variables with lower security level (public) by causing any variation for it and this is expressed as a kind of noninterference property [8].

On the other hand, enforcing noninterference in web application program is impractical, as sometimes there is a need to permit some intentional information release from secret to public. So to design a security typed web language with a declassification mechanism to leak some information from high security level to lower level in web application program when it is needed remains another important challenge. However there is often little or no guarantee about what is actually being leaked. As a consequence, web scripting language can be vulnerable to laundering attacks, which exploit declassification mechanisms to reveal more secret data than intended. So there is another challenge to consider this issue in our design for a security typed web language and consider it to have high impedance for such attack.

We briefly describe how a type system can be imposed on a simple imperative language that captures a subset of the security typed web language constructs to address common web application vulnerabilities. The language is similar to a security-typed imperative language given in [5,11]. It consists of Phrases, which are either expressions e or commands c.

(phrases) P ::= e | c
(expressions) e ::= x | *n* | e1 op e2| …… (1)
(commands) c ::= x := e | skip | if e then c1 else
c2| **while** e **do** c | c1; c2

Meta variable x ranges over identifiers, and n over integer literals. Integers are the only values. We use 0 for false and 1 for true, and op ranges over arithmetic and Boolean operations. We formalize the idea that program c does not leak information from H variables to L variables by the noninterference property which states that someone observing the final values of L variables cannot conclude anything about the initial values of H variables.

The Static type checking is typically expressed as an attempt to prove a type judgment. The formula Γ |− p: ρ typically has the meaning that in the environment Γ, the phrase p has the typeρ . If the phrase p is the entire program, this formula expresses the idea that the program is well-typed in a typical compiler. An identifier typing is a finite function mapping identifiers to ρ types*;* Γ (x) is the ρ type assigned to x by Γ.

Programs that are well typed under this type system are guaranteed to satisfy ***noninterference***. While noninterference is the usual way to define confidentiality and is useful for specification of security dependency, it is over restrictive for a program that is designed to release some confidential information as part of its functionality. So there is a need for downgrading policy [14,15] to make a new characterization for confidentiality that consider a delimited information release [1] and protect web application program against laundering attacks. Based on this need, we add some change to the expression syntax of the imperative language described above in (1) to declassify the security level of expression e intentionally to lower security level *l.*

(expressions) e ::= x | *n* | e1 op e2 |
declassify (e, *l*) …… (2)

Where *l* ranges over security levels. We assume that the security levels are elements of a security lattice L described in section 2. The specification of security must be relative to the expressions that appear under **declassify** operators. These expressions can be viewed as a part of the security policy, as they specify the "escape hatches" for information release. The new security specification delimits information release by only allowing release through escape hatch expressions as described in the definition of delimited information release given in [1]:

However, we believe that this definition is necessary in any web language that hopes to prevent laundering attack and provide secure web computing.

However downgrading policy does not only control how confidential data is released to the publics but it also controls how trustworthy information is updated as it controls the way in which un trusted user inputs can be intentionally trusted when it is needed. Thus, by enforcing downgrading policy for integrity, we can guarantee flexibility in web application system structure in addition to the prevention for the kinds of web application vulnerabilities that are considered a violation for the integrity policy such as SQL-Injection and Cross site scripting (Xss) vulnerabilities. To illustrate this concept in type system, we consider the security lattice L described in section 2. It has two elements tainted as its upper bound and untainted as its lower bound. If untainted arguments are used to execute the command, then the command is typed checked. On the other hand if the arguments can be tainted or untainted then the execution path can trigger an integrity violation. To address this issue, we need to consider the function call in defining the expression of the imperative language mentioned previously in (1) as follows:

e ::= x | *n* | e1 op e2 | *f(x)*

Where *f(x)* represents a function call and we consider the command that do not induce integrity violation is considered a valid command. To check the program validity, we define type judgment and judgment rules.

Denoted as Γ |− C → Γ/, a type judgment specifies a type environment Γ in which the execution of a command C is valid, and becomes Γ/ as a result of the execution.

At call sites to sensitive functions, SATISFY (Γ, *f, x)* checks whether Γ(*x*) satisfies function *f*'s precondition. We derive type judgments according to command sequences and raise an error when SATISFY (Γ, *f, x) fails.* That is, given a program *P* and its initial type environment Γ0 (usually mapping all variables to untainted), then the validity of *P* depends on whether we can derive the judgment Γ0 |− *P* →Γ.

7. Conclusion:

In this paper we discussed the information flow security in web application environment. We stated the different web application vulnerabilities and formalized them as an insecure information flow problem. In addition, we mentioned the security limitations in the existing web scripting languages. Finally we

discussed the different challenges for designing of a web scripting language that uses the language-based information flow control approach to guarantee end-to-end security in web-based information systems.

We used the type system approach on a simple imperative language to address the security properties that must be held in the semantics of the web scripting language to prevent the common web application vulnerabilities. We believe that the challenges that we are discussed here are necessary to be considered in the design of any security typed web scripting language that hopes to provide secure web computing.

References

[1] Andrei Sabelfeld and Andrew C. Myers, "A Model for Delimited Information Release"**,** *Proceedings of the 2003 International Symposium on Software Security*. LNCS 3233, Springer-Verlag, pages 174–191, 2004.

[2] A. C. Myers, "JFlow: Practical mostly-static information flow control", *In Proc. ACM Symp. On Principles of Programming Languages, pages 228–241, January 1999.*

[3] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, Sy-Yen Kuo, "Verifying Web Applications Using Bounded Model Checking", *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*© 2004 IEEE.

[4] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo, "Securing web application code by static analysis and runtime protection",In *Proc. 13th International Conference on World Wide Web*. ACM Press, 2004.

[5] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis"**,** *J. Computer Security*, 4(3):167–187, 1996.

[6] Andrew C. Myers**,** *"Mostly-Static Decentralized Information Flow Control"***,** PhD thesis, Massachusetts Institute of Technology, January 1999.

[7] Peng Li and Zdancewic, S, **"**Practical information flow control in Web-based information systems"**,** *Proceedings of the 18th IEEE Computer Security Foundations Workshop* (CSFW'05) © 2005 IEEE.

[8] Andrei Sabelfeld, Andrew C. Myers,. "Language-Based Information-Flow Security"**,** *IEEE Journal on Selected Areas in Communications*, special issue on Formal Methods for Security, 21(1):5–19, January 2003.

[9] V. Simonet, "Flow Caml in a nutshell"**,** *Proceedings of the first APPSEM-II workshop*, In G. Hutton, editor, Nottingham, United Kingdom, Mar. 2003, pages 152–165

[10] Dan Ragle, "Introduction to Perl's Taint Mode", Revised on May, 2006. Available via: http://www.webreference.com/programming/perl/taint/

[11] Geoffrey Smith, "Principles of Secure Information Flow Analysis", Chapter 13 (pp. 291-307) of *Malware Detection*, edited by Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, Springer-Verlag, 2007.

[12] Stephen Chong, Andrew C. Myers, K. Vikram and Lantian Zheng, *"Jif Reference Manual"*, June 2006. Available via: http://www.cs.cornell.edu/jif/doc/jif-3.1.1/manual.html

[13] Steve Zdancewic, "Challenges for Information-flow Security", In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.

[14] S. Chong and A. Myers, "Security policies for downgrading", *In Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004.

[15] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference", *In Proc. 32th ACM Symp. On Principles of Programming Languages* (POPL), 2005.

[16] Martin G. Nystrom and CISSP-ISSAP, "9 Ways to Hack a Web App", *JavaOneSM Conference*, Session 5935, 2005.
Available via: http://gceclub.sun.com.cn/java_one_online/2005/TS-5935/ts-

******