2011

Protection Models for Web Applications

Karthick Jayaraman
Syracuse University

Abstract

Early web applications were a set of static web pages connected to one another. In contrast, modern applications are full-featured programs that are nearly equivalent to desktop applications in functionality. However, web servers and web browsers, which were initially designed for static web pages, have not updated their protection models to deal with the security consequences of these full-featured programs. This mismatch has been the source of several security problems in web applications.

This dissertation proposes new protection models for web applications. The design and implementation of prototypes of these protection models in a web server and a web browser are also described. Experiments are used to demonstrate the improvements in security and performance from using these protection models. Finally, this dissertation also describes systematic design methods to support the security of web applications.

Protection Models for Web Applications

by

Karthick Jayaraman
M.E, Anna University, 2004
B.E, Bharathiar University, 2001

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering to the Graduate School of Syracuse University

July 2011

Approved by:

Prof. Steve J. Chapin, Advisor

Prof. Wenliang Du, Co Advisor

Table of Contents

Contents iv

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Thesis and Contributions 5

1.2 Dissertation Organization 6

2 Related Work 7

2.1 Foundational Systems Work 7

2.1.1 Classic Security Principles 7

2.1.2 Classical Security Policies 9

2.1.3 Access-Control Lists and Capabilities 14

2.1.4 System Security 15

2.2 Web Application Security 18

2.2.1 New Browser Architectures 18

2.2.2 Language-based Information Flow 19

2.2.3 Mashup Solutions 20

2.2.4 Mitigation Methods 20

2.2.5 Anomaly Detection 25

3 Enforcing Request Integrity 27

3.1 Anatomy of Web Applications 30

3.2 Request Integrity (RI) Attacks 33

3.3 BAYAWAK 37

3.3.1 Behavior-preserving Diversification 37

3.3.2 Architecture 39

3.3.3 Avoiding RI Attacks 45

3.4 Experimental Evaluation 47

3.4.1 Resistance to RI Attacks. 48

3.4.2 Performance Overhead 50

3.5 Discussion 51

3.5.1 Usability Considerations 52

3.5.2 Attacks Against BAYAWAK 54

3.6 Summary 55

4 A Protection Model for Web Browsers 57

4.1 Protection Requirements for Web Applications 61

4.1.1 Principals and Objects 61

4.1.2 The Same-Origin Policy 64

4.1.3 Protection Needs 66

4.2 The ESCUDO Access-Control Model 67

4.2.1 Rings, Ring Assignment, and ACL 69

4.2.2 The ESCUDO MAC Policy 73

4.2.3 An Example 74

4.3 Implementation 75

4.3.1 Lobo Architecture 76

4.3.2 ESCUDO Implementation 77

4.4 Enforcing Policy Integrity in Escudo 84

4.4.1 Policy Manipulation Threats 84

4.4.2 ESCUDO Countermeasures 88

4.5 Evaluation 90

4.5.1 Building ESCUDO-based Web Applications 90

4.5.2 Compatibility with Legacy Applications 95

4.5.3 Defense Effectiveness 95

4.5.4 Performance Overhead 96

4.5.5 Applications of ESCUDO 97

4.6 Summary 98

5 Designing Secure Web Applications 99

5.1 The Web DFA Model 102

5.2 Design Principles 107

5.2.1 Non-sensitive GET/ Sensitive POST 109

5.2.2 Secret Token Validation 113

5.2.3 Intent Verification 115

5.2.4 Guarded Workflow 118

5.2.5 Defense in Depth 121

5.3 Case Study 122

5.3.1 Building INFBB 122

5.3.2 Modeling Legacy Web Applications 126

5.4 Summary 128

6 Conclusion and Future Work 129

6.1 Future Work 130

6.1.1 Future Directions in Request Integrity 130

6.1.2 Future Directions in Web-browser Access Control 133

6.1.3 Complementing Research Directions 135

BIBLIOGRAPHY **138**

List of Tables

2.1 Access-control matrix 14

3.1 RI attacks on example applications 49

3.2 Performance overhead from using BAYAWAK instances 51

4.1 Principals and objects inside the web browser. 61

4.2 Security requirements for phpBB web application. 91

4.3 ESCUDO policy configuration for phpBB. 91

4.4 Security requirements for PHP-Calendar. 93

4.5 ESCUDO policy for PHP-Calendar. 93

5.1 Design patterns to prevent web request forgery attacks 108

List of Figures

2.1 The Bell-La Padula Model 11

2.2 The Biba Model 12

2.3 Hierarchical protection rings (HPR) 13

3.1 RFG for an online message board 32

3.2 Cross-site-request forgeries 35

3.3 A workflow violation in a purchase transaction: Using a workflow attack, an attacker skips the third step and completes the order without paying. 36

3.4 Behavior-preserving diversification of the request-flow graph 37

3.5 Bayawak Architecture 39

3.6 Flowchart for request processing in BAYAWAK 43

4.1 ESCUDO protection model 68

4.2 Ring assignment using div tags. 71

4.3 Assigning cookies to rings 72

4.4 Example: Assigning DOM elements to rings. 75

4.5 Architecture of the Lobo web browser 76

4.6 ESCUDO reference monitor (ERM) 81

4.7 A JavaScript program that accesses a list of DOM elements 81

4.8 Tag boundary confusion 85

4.9 Attribute boundary confusion 85

4.10 JavaScript program that modifies the scope of ESCUDO policies . . . 86

4.11 JavaScript program that modifies the ESCUDO policy of existing DOM elements 87

4.12 JavaScript program that adds new DOM elements in ring 0 87

4.13 Div tag with nounces 88

4.14 Div regions with empty comment tags 89

4.15 Performance overhead in parsing and rendering (in 8 different scenarios). 96

5.1 A Web DFA model for an online shopping cart application. 104

5.2 Web DFA for our shopping application with appropriate request type 111

5.3 A checkout workflow annotated with required preconditions and postconditions. 120

5.4 INFBB DFA labeled with sensitive/ non-sensitive states and proper request type 123

5.5 Implementing secret-token validation 124

5.6 INFBB workflow annotated with preconditions and postconditions . 125

5.7 Possible CSRF attack on punBB bulletin board system 127

Acknowledgments

I am beholden to several people, without whose support I would not have completed my PhD. First and foremost, I would like to thank my advisor, Prof. Steve J. Chapin. I am a fan of his teaching and writing style. The lessons he taught me go beyond academic life. Effective thinking, execution, and communication skills are some examples. Brainstorming with him was always fun—in several instances he helped me transform problems into opportunities. Finally, I also thank him for the wonderful parties he hosted for all his advisees. He made me feel at home during my entire stay in Syracuse. I also thank my co-advisor, Prof. Wenliang Du, for his guidance on my dissertation, perspectives, and illuminating discussions.

I thank Prof. Shiu-Kai Chin, Prof. Susan Older, and Glenn Benson. I worked with them on the “Partner Key Management” project, and this experience gave me a number of perspectives that helped me succeed in my PhD. Prof. Chin is a great source of inspiration, and I am grateful to him for his mentorship. I am indebted to Glenn for giving me a number of industry perspectives and practical insights on system security. I also thank Prof. Jim Royer for patiently supervising me on an independent study on formal methods.

I thank Prof. Jim Fawcett and Prof. Carlos Caicedo for agreeing to be on my thesis committee. Their suggestions and comments were very useful for my research.

I have been lucky to have had some wonderful peers, who supported me during my PhD. In particular, I would like to thank Thumrongsak Kosiyatrakul, Gregg

Lewandowski, Norka Lucena, Christian Sarmoria, Deepak Subramaniam, Paul Talaga, Haizhi Xu, Zutao Zhu, and others for their great friendship.

I was fortunate to have had the opportunity to work with some great researchers outside of Syracuse University such as Vijay Ganesh, Adam Kiezun, and Mahesh Tripunitara. I am indebted to them for their support.

Last, but not the least, I thank my family. My father, Dr. R. Jayaraman, continues to be my greatest inspiration. I thank my mother, sister, and brother-in-law for support. I also thank my wife Hema for her support. All this support has been critical in helping me complete this work. Finally, I thank my lovely nieces Nadhini and Srinidhi for bringing a lot of happiness to the family. I dedicate this dissertation to Nandhini and Srinidhi.

Chapter 1

Introduction

Web applications have progressively evolved from static web pages to highly sophisticated applications. We use web applications for a range of every-day activities such as communication, shopping, paying bills, banking, and entertainment. To facilitate richer user interactions, these applications commonly use two features. First, modern web applications use significant portions of client-side JavaScript programs. Second, web pages combine trusted, semi-trusted, and untrusted content in their web pages. With these additions, web applications have become full-featured programs to the point of supplanting desktop applications such as e-mail clients and word processors.

Although web applications have evolved into sophisticated programs, methods for assuring their security are still rudimentary. When compared to desktop applications, an important difference is that web applications do not have the support of adequate protection models. Desktop applications do not implement their security from scratch, but build on top of **trusted** and **non-bypassable** operating system mechanisms. Application developers trust that operating system mechanisms are correct implementations and will behave consistently. These mechanisms are non-bypassable because operating systems strictly isolate user programs from the trusted code that enforces security restrictions, and this isolation is enforced using the support of a hardware feature called protection rings [89] (Chapter 2 provides a detailed description of protection rings).

The following example illustrates how desktop applications implement a significant portion of their security by just configuring the appropriate policy settings in the operating systems. Let us consider the *passwd* program in UNIX that users use to change their passwords. The */etc/shadow* file contains the passwords of all users. The security requirement is to enforce that a user can only update his own password. Two operating system mechanisms, file-based permissions and *set-uid root*, are used for realizing this security objective. The permissions of the */etc/shadow* file are configured such that only the root user can access or modify its contents. In addition, the *passwd* program is configured to be a *set-uid root* program. This mechanism has two implications. First, the program is owned by root, so a normal user cannot modify this program. Second, the program executes with root's permissions whenever it is invoked by a user. As a result of these restrictions, the only way in which a normal user can update his password entry in the */etc/shadow* file is using the *passwd* program, which restricts a user to update his password only.

Web applications do not have similar protection models for meeting their security needs. Web servers and browsers that provide a run-time environment for executing web applications were initially designed for static web pages. However, modern applications have a richer set of principals, objects, and interactions compared to static web pages. Although web servers and browsers have kept up with functional needs of these rich applications, they have not updated their protection models to meet the security needs. In effect, we have a mismatch between the protection needs of web applications and the protection models provided in web browsers and web servers. This mismatch has led to concrete problems that can be observed in web applications such as the following:

1. We do not have a protection model for enforcing policies on the request processing behavior of a web application at the server side. A web application is constructed to process intended sequences of HTTP requests, and the application's integrity can be compromised if the actual requests do not follow the sequence. However, we lack protection models for enforcing these intended sequences. Therefore, attackers can manipulate request sequences in vulnerable web applications, facilitating attacks such as workflow attacks.

2. Web applications are not adequately isolated from one another inside the web browser. Moreover, web browsers do not distinguish the authority of two web sites at the required granularity. As a consequence, web browsers are vulnerable to confused-deputy attacks, in which one web application's resources can be put at risk by the actions of another web application. Cross-site-request forgeries (CSRF) are an example of such confused-deputy attacks.

3. Web browsers do not factor in the trustworthiness of JavaScript programs that execute in a web page. As a result, all JavaScript programs inside a web page have uniform access to the resources of a web applications irrespective of their trustworthiness. This has resulted in cross-site scripting (XSS) attacks, wherein semi trusted and untrusted JavaScript programs manipulate more trustworthy resources. XSS attacks are one of the widely reported attacks.

Because of the lack of adequate protection models for web applications, developers have to implement security from scratch. This task, at a high level, requires answering the following two questions.

1. What are the security concerns?

2. Are all the security concerns addressed in the implementation?

The first question relates to design. Some of the security concerns may be very specific to the application. For example, an online-retailing application may require that HTTP requests that comprise of a checkout transaction should follow a particular sequence. The second question relates to verification. Answering both these questions requires security expertise. However, an average developer is not a security expert. Therefore, both these questions are not properly answered in the implementation of most web applications. Most organizations hire an independent penetration testing team to help them answer the verification question after the application has been built. This process does not provide complete answers because the penetration testers are blindly trying to answer the verification question, without knowing about the security concerns or the security-relevant portions of the applications.

There is a clear need for designing better protection models for web applications for meeting their security needs. With the help of such models, developers can implement a significant portion of the security implementation in their web applications by appropriately specifying a policy. Moreover, both developers and publishers will have better guarantees that their applications will conform to their intended security policy. Furthermore, because the protection models will be a part of a trusted code base, corporations needs not spend additional resources on specialized security testing.

There is also a need to support developers with systematic design practices that can help them enumerate the security objectives early on during the design. Such design practices support the use of protection models. Good design will facilitate the use of the right protection models because the security objectives are clear.

Moreover, it may provide a supportive additional layer of defense. Finally, there may be a transition period before which new web application protection models will be uniformly available in all platforms. During this time, application designers need to address their security needs through well-informed design practices.

1.1 Thesis and Contributions

This dissertation's thesis is that

> ***The security of web applications can be improved by designing adequate protection models supported by good design practices.***

In support of this thesis, this dissertation describes the following contributions:

1. **BAYAWAK**: A new server-side approach for enforcing request integrity in web applications, and its implementation in a tool called BAYAWAK [53]. Web applications are constructed to process certain intended sequences of HTTP requests. Failing to enforce these intended sequences can lead to request-integrity attacks, wherein an attacker forces an application into processing unintended request sequences. Under the proposed approach, a publisher can specify the intended request sequences as a security policy, and BAYAWAK would strictly enforce this policy transparently without requiring any changes in the application's source code.

2. **ESCUDO**: A new web-browser protection model called ESCUDO [52]. Web applications are no longer simple hyperlinked documents; web pages combine content from several sources with non-uniform trustworthiness and also use significant portions of client-side code. The prevailing protection model,

the same-origin policy, cannot manage security consequences of this additional complexity. ESCUDO is a new protection model designed based on established principles of mandatory access control. Web publishers can use ESCUDO for enforcing policies on the behavior of all web application principals inside the browser based on their trustworthiness.

3. **The Web DFA Model**: A systematic methodology for designing web applications to strictly enforce intended request-response behavior in web applications by construction [54]. In the proposed methodology, a developer would first model the intended request-response behavior using the Web DFA model, and then apply four design patterns to produce a blueprint to guide the implementation.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides a survey of related work. The next several chapters discuss each of the contributions in this dissertation. Chapter 3 describes the design and implementation of BAYAWAK. Chapter 4 describes the design and implementation of ESCUDO. Chapter 5 describes the Web DFA-based methodology for designing and implementing new web applications. Chapter 6 provides concluding remarks and discusses future work.

Chapter 2

Related Work

The chapter presents work related to the thesis and contributions of this dissertation. First, this chapter discusses foundational systems work and concepts that are the basis for improving the security of web applications. Second, this chapter presents a survey of research in web application security that is related to the contributions of this dissertation.

2.1 Foundational Systems Work

This section presents foundational systems work related to the research presented in this dissertation. The results of these research work provide good starting points for improving the security and integrity of web applications.

2.1.1 Classic Security Principles

The classic paper of Saltzer and Schroeder [87] describes the following eight fundamental principles of protection:

1. Economy of mechanism: This principle requires that the system design be as simple as possible. A simple design will facilitate easier manual inspection of software and hardware to make sure there are no unwanted access paths.

2. Fail-safe defaults: According to this principle, a system should not permit any access by default, and should permit an access if and only if a protection scheme determines that the access should be permitted.

3. Complete mediation: According to this principle, a system should check each access to an object and determine if the principal in question has the required authority.

4. Open design: The security of the system should not depend on the design being a secret. It is impractical to guard the design of a system that is widely distributed. Therefore, designers should publish their design. Moreover, an open design would help a user ascertain if the design meets his security needs.

5. Separation of privilege: Whenever possible, the permissions to access a sensitive object should be split into two or more permissions and assigned to different users such that the object can be accessed only if all the users agree. For example, a protection scheme that requires two keys belonging to two different users to unlock is more robust when compared to a scheme that requires a single key. That way, a single compromise will not affect the system.

6. Least privilege: The system should enforce that each program and user should be bestowed only the minimum and necessary privileges for doing their job.

7. Least common mechanism: Reduce the mechanisms that are common to all users. Such shared mechanisms are a potential information path between users, i.e., covert channels. Moreover, errors in such mechanisms will affect all users.

8. Psychological acceptability: According to this principle, the system should feature appropriately designed user interfaces to help the user correctly configure the protection mechanisms. To the extent possible, there should be no gap between the typical mental model users have about their protection goals and the protection mechanism they are required to use. By following this principle, mistakes will be minimized.

All these eight principles continue to be relevant to design of protection mechanisms. The work of Reis [80] makes the observation that many of these concepts are relevant to web browsers and web content.

2.1.2 Classical Security Policies

This section presents a review of classic security policies that are relevant to the thesis statement. Textbooks such as Bishop [13], and Chin and Older [19] provide a detailed exposition of classic security policies. This section provides a brief summary of security policies related to the dissertation.

There are two types of security policies based on who has control over the policies, namely *discretionary access control (DAC)* and *mandatory access control (MAC* policies. In DAC, the owner of an object is responsible for specifying which principals have access to the object. The owner of an object may always modify the policy. File-based permissions in the Linux operating system are an example of DAC. In MAC, a system administrator is responsible for specifying which principals have access to an object. A MAC policy is static, i.e., it is never changed once it has been set. System-level mechanisms strictly enforce the policy at all times. An example of a MAC policy is the memory isolation enforced in operating systems using segment descriptors. Each program has its own segment descriptors that determine the portions of memory that it can access. The segment descriptors are setup by the system, and can neither be read nor modified by user programs. As a result, one program cannot alter the memory of another program.

Chapter 4 proposes a hierarchical multi-level protection model for managing a web application's resources in web browsers. Therefore, this section describes three classic multi-level protection models.

Bell-La Padula. The Bell-La Padula model is focused on preserving the confidentiality of data [10]. Under this model, all principals and objects are assigned labels that determine their level of trust and sensitivity respectively. For example, in the case of military systems the four labels in the order of increasing sensitivity are the following:

1. Unclassified

2. Confidential

3. Secret

4. Top secret

In the case of principals, the label is called the clearance level and determines the level of trust given to the principal. In the case of objects, the label is called the classification level and determines the object's confidentiality.

Figure 2.1 depicts the access-control policy in the Bell-La Padula model. All accesses of principals to objects are governed by the following two rules:

- *Simple security property:* A principal is allowed to read an object if and only if the principal's clearance level is greater than or equal to the object's classification level, and the principal is allowed discretionary read access to the object. Informally, this rule is referred to as the "no read up" rule.

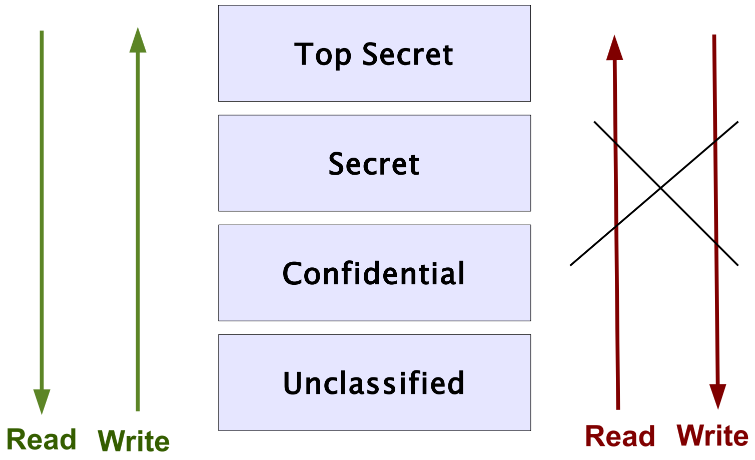


Figure 2.1: The Bell-La Padula Model

- **-Property:* A principal is allowed to write to an object if and only if the object's classification level is greater than or equal to the principal's clearance level, and the principal is allowed discretionary write access to the object. Informally, this rule is referred to as the "no write down" rule.

As a result of using these two rules, principals with lower clearance levels will never have access to information in higher classification levels, and principals cannot "declassify" information by copying them to objects in lower classification levels. In effect, information can flow only in the upward direction, i.e., from lower classification levels to higher classification levels.

Biba. The Biba model is focused on preserving the integrity of the system, and confidentiality of information is outside the scope of this model [12]. Similar to the Bell-La Padula model, principals and objects are assigned labels. For the purpose of exposition, this description uses four labels, namely "Level 0", "Level 1", "Level 2", and "Level 3". "Level 0" is assigned to principals and objects that have the highest level of integrity, while "Level 3" is assigned to principals and objects that

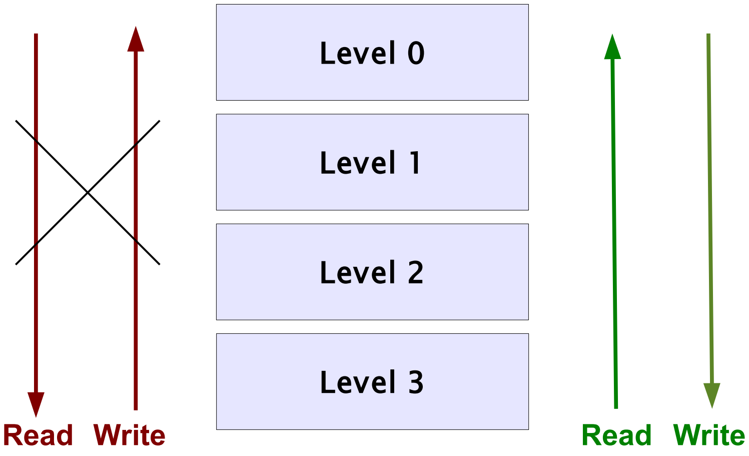


Figure 2.2: The Biba Model

have the lowest level of integrity. The objective of the Biba model is to prevent data that is deemed to have a lower integrity level from corrupting data at a higher integrity level. The permitted direction of information flow is reversed in the Biba model compared to the Bell-La Padula model, i.e., information can flow only from a higher classification level to a lower classification level.

Figure 2.2 depicts the access-control policy in the Biba model. All accesses of principals to objects are governed by the following two rules:

- *Simple integrity property:* A principal is allowed to read an object if and only if the principal's clearance level is lower than or equal to the object's classification level, and the principal is allowed discretionary read access to the object. Informally, this rule is referred to as the "no read down" rule.

- **-Property:* A principal is allowed to write to an object if and only if the object's classification level is lower than or equal to the principal's clearance level, and the principal is allowed discretionary write access to the object. Informally, this rule is referred to as the "no write up" rule.

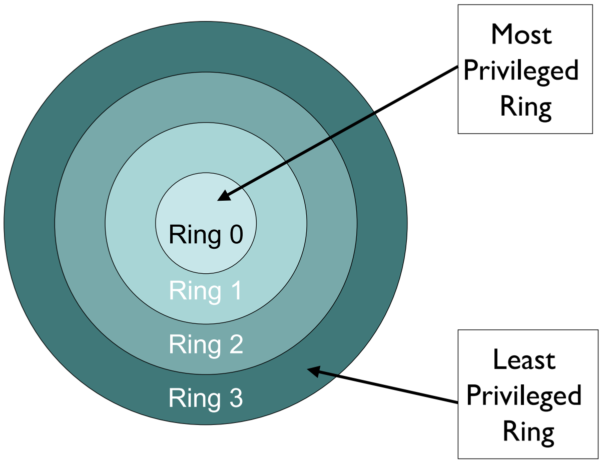


Figure 2.3: Hierarchical protection rings (HPR)

Hierarchical Protection Rings (HPR). HPR was first introduced in the Multics operating system [89]. Multics organizes the access permissions into hierarchical rings, which are the trust classes in the HPR model, numbered from 0 to n (Figure 2.3). Ring 0 is the most privileged ring and ring n is the least privileged ring. The access permissions in a ring x are a subset of access permissions in ring y, whenever $x > y$. A program in a particular ring is entitled to use the permissions available in its own ring and outer rings, but cannot use the permissions in the inner ring. There are special gates between rings to allow a process from an outer ring to request some resources from an inner ring in a controlled fashion. To isolate the memory address spaces of user programs, Multics uses segment descriptors. Organizing the programs in rings provides separation of privilege, and memory isolation enforced via segment descriptors further increases the granularity of protection offered by rings and enforces the principle of least privilege.

	file$_1$	**file**$_2$	**file**$_3$	**file**$_4$
Alice	read	read	read	write
Bob	read	write	write	read
Mallory	-	-	-	-

Table 2.1: Access-control matrix

2.1.3 Access-Control Lists and Capabilities

Access-control lists and capabilities are two common methods for enforcing access-control. Lampson [66] and Saltzer and Schroeder [87] describe the underlying concepts behind access-control lists and capabilities.

Let us consider a simple system comprising three users Alice, Bob, and Mallory, and four files file$_1$, file$_2$, file$_3$, and file$_4$. The permitted accesses of all the principals on the four files can be represented in the form of an access-control matrix (Table 2.1). In this access matrix, the rows represent principals, columns represent objects, and each cell contains the accesses that a principal has on an object. For example, in access-control matrix depicted in Table 2.1, Alice has read access on files file$_1$, file$_2$, and file$_3$, and write access on file file$_4$. A reference monitor is an entity in the system that is responsible for enforcing the access matrix. Access-control lists and capabilities are two methods for implementing the access-control matrix.

In the case of access-control lists, each object has an access-control list. The access-control list specifies the principals and the access permissions they have on the object. For example, in access-control matrix depicted in Table 2.1, Alice and Bob have read access on file$_1$. Therefore, the access-control list of file$_1$ has two entries, namely $\langle Alice, read \rangle$ and $\langle Bob, read \rangle$. Whenever a principal requests to access an object, the reference monitor permits the access if and only if the access-control list contains an entry permitting the access.

In the case of capabilities, each principal has a capability for each object that it can access. The capability describes the permitted accesses on the object. For example, in the access-matrix depicted in Table 2.1, Alice has read access on files file_1, file_2, and file_3, and write access on file file_4. Therefore, Alice carries four capabilities for the files. Whenever a principal requests to access an object, the reference monitor checks the principal's capabilities to determine if the access should be allowed.

2.1.4 System Security

Research in system security has led to several methods that could be used for the purposes of monitoring and enforcing policies on program behavior, and also eliminate broader classes of injection attacks. This section discusses run-time monitoring techniques, interpositioning and program rewriting techniques, and synthetic diversity techniques. All these three techniques provide starting points for exploring similar methods for web application security.

Run-time Monitoring. Monitoring sequences of system calls for the purpose of detecting abnormal or malicious program behavior is an active area of research. The basic idea is to learn a normal profile of a program in terms of the system calls it invokes by observing several normal executions, and to detect deviations from this normal profile at run time. Forrest et al. [34] describes the first seminal work in this research, and Forrest et al. [33] describes the evolution of this research since the first original research.

Several research proposals have explored the use of static analysis of either the program source code or the binary to derive the normal behavior of the program [9, 37, 38, 97, 104]. This is to avoid the limitations of learning the normal profile at run time. Moreover, the models derived from the program source code are relatively more deterministic and complete as they try to capture the programmers intent.

Program Rewriting and Interposition. Program rewriting and interposition techniques are also used for the purpose of enforcing policies on program behavior. Program shepherding uses an efficient machine-code interpreter for enforcing a wide range of policies on program behavior such as restrictions on code origins, restrictions on controls flows, etc. For example, in the case of restricting control flow, the interpreter checks whether the control-flow conforms to the policy each time the program executes a control-flow instruction.

Abadi et al. [2] describes the work on control flow integrity. In this work, a control-flow graph derived from a program's binary is enforced by using a binary rewriting technique. In this method, all instructions to jump to an address are instrumented with additional checks to make sure that the jump conforms to the intended control-flow graph of the program.

Several research proposals have explored rewriting the program source code to enforce security policies. This approach is attaractive because it does not require any changes in the underlying platform. Erlingsson and Schneider [30] describe a program rewriting approach for enforcing Java's stack inspection policy. There is also work on rewriting at the network level. *Shield* [99] is a network proxy for filtering malicious traffic based on signatures of known vulnerabilities. *BrowserShield* [81] is a similar system for rewriting webpages into safe equivalents based on known vulnerabilities.

Synthetic Diversity Several research proposals have demonstrated the use of synthetic diversity for eliminating broader classes of injection attacks. Forrest et al. [35] discusses the security benefits of building diverse computer systems. The basic idea in this research is to diversify a program appropriately such that an attacker cannot learn any new information from observing a program's execution for the purpose of exploiting it. Instruction-set randomization and address-space randomization are two well-known techniques for eliminating code injection attacks.

The basic idea in instruction-set randomization is to vary the instruction set each time a program runs [5, 59]. Typically, the system uses a secret key to randomize the instructions. As long as the secret key is protected and sufficiently long, an attacker cannot predict the valid instructions that can execute. Therefore, if an attacker injects instructions into the running program, then such injected instructions will be treated as invalid instructions.

The basic idea in address-space randomization is to vary the address space of a process each time a program runs [11, 103]. As a result, each time a program runs, it runs in a different virtual address space. Therefore, an attacker cannot predict the addresses of various vulnerable locations, making buffer overflows significantly harder. Operating systems such as Linux natively support address-space randomization.

N-Variant Systems is another architectural framework based on synthetic diversity for eliminating larger classes of attacks [24]. The N-Variant system concurrently executes diversified program variants on the same input and monitors their behavior to detect divergences. The variants are constructed to have disjoint exploitation sets with respect to an attack class. Therefore, an attack will be successful only if the attacker compromises all the variants. For example, to avoid code injection attacks, each variant may use a different randomized instruction set. An attacker can inject code that uses only one of the instruction set. As a result, the injected code will compromise one variant and be considered invalid in other variants, causing the variants to diverge.

2.2 Web Application Security

This section summarizes research efforts to improve the security and robustness of web applications. Several research proposals have considered alternate web browser architectures. These proposals are complementary to this research. Work on language-based information flow is also complementary to this research. There are several mashup solutions, mitigation methods, and anomaly detection methods that address the consequences of not having appropriate protection models. In contrast to these solutions, this research proposes new protection models directly targetting the fundamental problem.

2.2.1 New Browser Architectures

Several research proposals have explored alternative microkernel-like architectures for web browsers. Prior to these proposals, web browsers had a monolithic architecture, in which a single process handled the rendering of all the web sites that a user visits. Therefore, an error in one web site may cause the entire process to crash, terminating the ongoing sessions with other web sites. The new proposals vary in how they isolate the web sites. The *OP* web browser isolates each web page instance and various browser components using OS processes [39], and interposes itself in all inter-process communications. *Tahoma* isolates each instance of a web application inside the browser using separate virtual machines (VM) [25]. The policy for identifying program boundaries and the permissible characteristics, such as which URL may be visited in each VM, are specified in a manifest. *Chromium* [8, 82] and *Gazelle* [98] bifurcate the browser into two portions, namely kernel and applications, similar to operating systems, and both these browsers isolate web applications from one another using operating system processes.

Although these architectures are an important way forward, the access-control model in these architectures is still based on the same-origin policy (SOP). The SOP labels all principals and objects within a web application with their domains, i.e., the unique combination of protocol, domain, and port in the application's URL. JavaScript programs from one domain are not allowed to access objects belonging to a different domain. The SOP does not enforce the separation of privilege and principle of least privilege for modern web applications, and is not adequate for protecting them. Chapter 4 describes the shortcomings of SOP and proposes an alternate web browser protection model. That being said, these new architectures are complementary to the development of protection models because they provide better isolation compared to monolithic architectures.

Conscript is browser-side system for enforcing policies on the execution of JavaScript programs [72]. Examples of policies enforced in this system are disabling JavaScript access to cookies, disallowing dynamic generation of JavaScript code, etc. This system is useful for JavaScript programs, but we still need a general web browser protection model for all the principals and objects within a web application including JavaScript programs.

2.2.2 Language-based Information Flow

There is also work on using language-based information flow [86] for web security. SIF (Servlet Information Flow) [20] is an information-flow framework for building high assurance web applications. In this framework, a developer would provide confidentiality and integrity policies for data and variables inside the application, and the framework prevents the flows of confidential information to clients and low-integrity information from clients. Language-based information flow methods do not obviate the need for better protection models in the web browser and the web server. For example, let us consider the case of a JavaScript program and a portion of web page that have the same level of confidentiality. A publisher may want to allow the JavaScript program to read the web page portion, but does not want to allow a write in that web page portion. Such restrictions can only be enforced by a protection model.

2.2.3 Mashup Solutions

Mashups applications integrate content from several applications from differing origins into one web page. A key security concern in such applications is isolating the resources of each application from one another. Several frame-based design proposals for mashups have contributed new primitives and communication methods with minimal or no changes to the browser [26, 28, 47, 51]. Still, these proposals have a coarse-grained privileged model because they are based on the same-origin policy. Mashups provide additional motivation for designing better and fine-grained protection models for web applications. Such models would facilitate rich mashup applications and also meet the security needs of each of the applications being integrated into the mashup.

2.2.4 Mitigation Methods

Current work has proposed several mitigation methods for addressing specific web security issues. Several of these problems are a side effect of not having appropriate protection models. This section presents a survey of such mitigation methods

SOP Extensions. Current work has proposed several extensions to the same-origin policy (SOP) to address specific problems. Jackson et al. [50] extends the

SOP to browser cache content and visited link information to protect user privacy. Livshits and Ulfar [67] extends the SOP to additionally account for the principal names added to tag groups for neutralizing code-injection attacks. Karlof et al. [58] extends the SOP to account for certificate errors in the origin to distinguish resources in the authentic domain from a spoofed domain to detect dynamic-pharming attacks. While each of these proposals addresses a specific shortcoming in the SOP, they do not address the general gap between the fundamental model and the security requirements of modern web applications.

Cross-site-request forgeries (CSRF). In CSRF attacks, a malicious web site interferes with a victim user's ongoing session with a trusted website. The malicious web site tricks the web browser into attaching a trusted site's authentication credentials to malicious requests targetting the trusted site.

CSRF is a side effect of two access-control problems. First, we lack server-side protection models that help the web application clearly determine the validity of incoming requests. Second, web browsers do not manage application resources such as authentication credentials according to publishers intent. A publisher needs the facility to attach a policy that describes who can access the application resources such as authentication credentials.

Current work has proposed several methods for preventing CSRF. Typically, web applications defend against CSRF by making sure that only the trusted web page is making the request. This can be done by adding fresh nonces to each page to accompany all genuine requests. *NoForge* is a server-side proxy for automating the inclusion of such fresh nonces in the web page [57]. Alternatively, web applications can verify if the request originated from a trusted web page by checking either the HTTP referrer header [60] or the HTTP Origin header [6, 7] in incoming requests. *RequestRodeo* [56] and *BEAP* [68] are browser-side solutions for CSRF that work by removing authentication credentials from HTTP requests deemed as suspicious based on some heuristics.

Current work has also proposed mitigation methods for variants of CSRF such as *Login CSRF* [7] and *Clickjacking* [4]. Huang et al. [48] describes the *cross-origin css* attack and a solution for it. Cross-origin css attack can also be considered a CSRF variant because the vulnerability arises due to a malicious web site retrieving a web page belonging to the ongoing session between a user and the trusted web site.

Each of these solutions focuses on a specific variant of CSRF. Also, none of these solutions incorporates the publisher's intent in managing application resources. In contrast, Chapter 3 describes a new approach that eliminates several variants of CSRF and helps the browser make a determination of the validity of incoming requests. Also, Chapter 4 describes a new web browser protection model that incorporates the publisher's intent.

Cross-site scripting (XSS). In XSS attacks, an attacker injects a malicious JavaScript program into a trusted site's web page. If a victim user visits the affected web page, then the web browser executes the malicious JavaScript program as though it came from the trusted site. A web application is vulnerable to XSS if it does not adequately validate semi-trusted JavaScript programs or untrusted user-supplied input before adding them to its web page. The problem is magnified in the case of modern web applications, which embed JavaScript programs of non-uniform trustworthiness in their web pages. There are several mitigation methods proposed for XSS, and they can be categorized into server-side, client-side, and hybrid client-server solutions.

XSS-Guard is a server-side solution that compares JavaScript programs in each dynamically generated web page with those in a "shadow response" generated using benign inputs to identify malicious JavaScript programs [14]. There are also server-side solutions based on taint analysis. These solutions track the flow of user-supplied input to the web page, and either raise an alarm and do additional validation on "tainted" data before adding it to the web page [74, 78, 105].

Noxes is browser-side personal firewall that interacts with the user to build a database of suspicious JavaScript programs on a per-site basis and blocks the execution of such JavaScript programs. *NoScript* is a similar system available as an extension to the Firefox web browser [69]. Vogt et al. [95] describes a browser-side solution that tracks the flow of sensitive information using taint analysis, and prompts the user whenever sensitive information is about to be transferred to a third party.

There are also hybrid client-sever solutions for XSS. In *BEEP* [55], each web page in the application communicates a whitelist of JavaScript programs, and the web browser only executes the programs in the whitelist. In *Noncespaces* [42], the web application randomizes all trusted HTML tags in the web page and communicates the randomization key to the web browser. As long as an attacker cannot predict the randomization key, the web browser can distinguish between trusted and untrusted content in the web page, and avoid the execution of untrusted JavaScript programs. In *Document Structure Integrity* [73], a server-side method uses taint-tracking to identify portions of the web page that are heavily influenced by user-supplied input and delineates them using special random markers. The web browser treats those portions as not containing any HTML tags.

All these solutions try to categorize a JavaScript program as either trusted or untrusted. However, this binary decision is not adequate for modern web applications that use JavaScript programs with varying gradations of trustworthiness (eg. trusted, semi trusted, and untrusted). The fundamental problem is the lack of an access-control model, wherein a publisher can communicate a policy that determines the trustworthiness of various webpage components, and the web browser would execute the programs with permissions based on the trustworthiness. In summary, existing solutions focus on the symptoms of this fundamental problem. Chapter 4 describes a web browser access-control model that addresses the actual fundamental problem.

Malicious clients. There are research proposals for detecting malicious client behavior in web applications. Rich clients are a common feature of several modern web applications. These applications delegate some computational tasks to clients, typically client-side JavaScript programs, and these tasks may determine the application's integrity. For example, in an online game, a client-side JavaScript program may determine if a user's next move is legal based on the current state in the game. A malicious user may circumvent such checks on the client-side. *Ripley* detects such malicious behavior by redundantly executing the client-side program at the server-side, and also sending all the user's actions (mouse click's, keystrokes, etc.) from the client to the server [94]. If the behavior of the redundant program differs from the client-side program, Ripley disconnects the client. This dissertation is focused on general protection models for web applications, so detecting malicious clients is outside the scope of this research.

Web application firewalls. A web application firewall (WAF) is a reverse proxy for a web application. All user requests and responses to the application are intercepted by the WAF. A WAF protects a web application from common web-based attacks by detecting and eliminating malicious user requests from reaching the web application. In general, WAFs are not application aware and apply generic rules that are broadly applicable to all web applications. However, a WAF that is application aware can eliminate broader classes of attack. Chapter 3 describes a new approach that can eliminate broader classes of attacks compared to the current WAFs.

2.2.5 Anomaly Detection

Detecting malicious web requests using anomaly detection approaches is another active area of research. In these approaches, the system creates a statistical model of benign requests using a training data set. The training data set is typically derived from past behavior. For example, a system can build a model of benign requests by either characterizing the contents of HTTP requests [49, 65] or the application's internal session variables [23]. The system then checks if each incoming web request conforms to this model, and rejects all non-conforming requests. A key shortcoming of this approach is that there could be errors in the detection because the models are not deterministic. Moreover, the system has to go through an initial bootstrap phase of training before becoming operational. The training may have to be repeated each time there are major changes to the application.

Guha et al. [41] describes a intrusion-detection proxy for detecting unexpected requests from JavaScript programs in AJAX applications. The intrusion-detection proxy has a model of expected client requests and keeps track of all the requests issued by the client and processed by the application. Whenever a new request arrives, the intrusion-detection proxy checks if the incoming request forms a valid request sequence when conjoined with previous requests issued by the client. This approach would work only if the users are prohibited from simultaneously viewing multiple web pages in a application. If a user visits multiple web pages from the application, such an usage would constitute multiple ongoing request sequences. In these cases, the intrusion-detection proxy would generate a lot of false positives.

Chapter 3

Enforcing Request Integrity

A web application is constructed to process certain intended sequences of HTTP requests. Web applications must strictly enforce these intended sequences to preserve the correctness and integrity of the application. This chapter refers to the enforcement of intended request sequences as the enforcement of request integrity. The lack of such an enforcement can lead to a compromise of application integrity and user privacy.

Request integrity (RI) attacks are defined as attacks that are a consequence of failing to enforce intended request sequences in a web application. In RI attacks, an attacker takes advantage of vulnerabilities in web applications to trick them into processing an incorrect request sequence. Cross-site-request forgeries (CSRF) [57] and workflow attacks (WF) [23] are examples of RI attacks, but both these attacks have been treated as unrelated attacks in existing literature. Both these attacks are among the top 10 web-based attacks and are commonly found in web applications [1, 102].

One approach is to enforce request integrity in the application's implementation. However, an average developer is not a security expert. Moreover, certain weaknesses that make RI attacks possible are rooted at the very nature of web applications and web browsers. For example, the structure of a web application does not significantly change over time. Section describes how this lack of diversity can be abused to gather knowledge about application's structure and then construct seemingly valid request sequences. Therefore, developers cannot be expected to enforce request integrity in the implementation unless they are trained and equipped with appropriate design practices.

The research work described in this chapter is focused on developing a protection model for enforcing request integrity. In this case, an application takes advantage of methods available in a trusted code base such as operating systems or a security framework to implement security. For example, a significant portion of security and access control in desktop applications is controlled by operating system (OS)-level policy settings. Similarly, address-space diversity is a security concern for desktop applications. If the address space of processes do not vary between executions, then attackers can determine the addresses of various locations in the process address space and use them to design and inject malicious code. To address this problem, researchers have proposed OS-level methods for randomizing the process address space, and these methods are commonly available in all mainstream operating systems [11, 103]. Similarly, framework-level methods, i.e., web server or application framework methods, can be effective for enforcing request integrity.

This chapter describes a new approach for enforcing request integrity in web applications, and the design and implementation of a tool called BAYAWAK that is based on this approach. Under the proposed approach, the valid request sequences for a web application are specified as security policy, a web server method transparently and strictly enforces the valid request sequences, eliminating attacks that trick the application into processing an invalid request sequence. The proposed approach does not require any changes in the source code of the web application.Therefore, publishers need to only verify the security policy to be sure that valid request sequences are strictly enforced.

BAYAWAK accepts an abstract description of valid request sequences in the form of a request-flow graph (RFG) [1] as input. BAYAWAK enforces valid request sequences using the following three steps. First, BAYAWAK performs a behavior-preserving diversification of the RFG for each session. Second, BAYAWAK modifies the web pages produced by the application to be compatible with the varied per-session RFG. Third, BAYAWAK validates each incoming request against the per session RFG before forwarding to the application for further processing. These these three steps, together, eliminate the underlying root causes that facilitate RI attacks.

This chapter also describes the evaluation of BAYAWAK using nine open source web applications. The evaluation exercise identified several RI attacks in each of the applications. After configuring BAYAWAK instances for each of the application, all the attacks were eliminated. Furthermore, the BAYAWAK instances incurred negligible overhead.

The effectiveness of BAYAWAK depends on the correctness of the RFG. There are several methods for obtaining the RFG for a web application. The RFG could be derived from the specification of the application. In the case of legacy web applications, the RFG could be derived from the source code using reverse engineering. The reverse engineering methods vary in their sophistication ranging from simple web spiders to advanced program analysis methods such as WAMse [43] and Tansuo [100]. The experimental evaluation used both methods on the open source web applications depending on the application complexity.

The key contributions described in this chapter can be summarized as follows:

1. An approach for enforcing request integrity in web applications that moves

[1] We will define the term in section 3.1

the enforcement from the application into a security framework.

2. The proposed approach eliminates both classes of RI attacks, namely CSRF and workflow violations, which were previously considered unrelated attacks.

3. Implementation of the proposed approach for the Apache web server in a tool called BAYAWAK, and evaluation using nine open-source web applications.

3.1 Anatomy of Web Applications

This section will provide background information on web applications and define the terminology we will use in the remainder of the chapter. A web application comprises components such as server-side scripts, databases, and resources such as images and JavaScript programs, and is accessed over the Internet using the HTTP protocol. Typically, a user accesses a web application using a web browser. The web browser constructs HTTP requests in response to the user interacting with hyperlinks and forms in a web page, forwards them to the application, and displays the web page received in the response. Web applications receive and process incoming requests using their *interfaces* [43]. An interface receives an HTTP request and returns a web page in the response. Each HTTP request contains a target URL and several arguments in the form of name-value pairs that are either part of the URL (known as a query string) or the message body. The target URL and the name-value pairs in the request identify the target interface and we will refer to them as *interface names*.

Usually, web applications need to group incoming requests into sessions. For example, in an online shopping application, a user may add products to his shopping cart in one request, and then initiate a purchase transaction in another request. The shopping cart application should be able to group these requests into a single session and also associate the contents of the shopping cart with the correct user's session. However, the HTTP protocol was designed to be stateless so that hosts do not have to retain information about users between requests [32]. Therefore, web applications use cookies to group requests into sessions. Whenever a new session is created, web applications create a cookie in the web browser using the *set-cookie* HTTP header in the response. Web browsers attach all the cookies created by the application to all subsequent requests, helping the application associate each incoming request with its session.

Enforcing Intended Request Sequences: Each web application is designed to process certain intended sequences of requests. For example, in an online-shopping application, a request to initiate a purchase transaction is expected only after a user is signed in, and a request to finalize the purchase is expected only after a user provides valid payment information. Similarly, some web applications display the URL for the administrative interface in a web page only if the user is logged in as an administrator. These rules reflecting the intended request sequence can be abstractly represented using a graph; each node corresponds to an interface and edges correspond to HTTP requests. If a directed edge connects node$_1$ and node$_2$, then the web page created by the interface corresponding to node$_1$ contains a hyperlink or form targeting the interface corresponding to node$_2$. This chapter refers to this graph as the *request flow graph (RFG)* of a web application.

In a typical intended access model, users access the web application starting from a *session-initializing* interface (SII), which creates a new user session that will be shared by all subsequent requests from the user until the session terminates. In most applications, all requests targeted to the domain name of the application

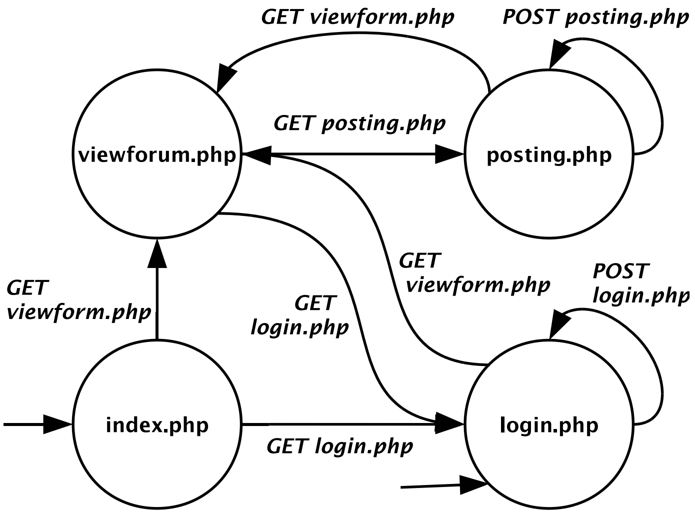


Figure 3.1: RFG for an online message board

are redirected to a SII. Also, in the absence of a session, all requests to non-SII are redirected to a SII. After the session is initialized, the browser issues all subsequent requests based on the user interaction with the hyperlinks and forms in the web page.

Example: The online message board application in Figure 3.1 has four interfaces and 10 interlinks between the interfaces. For the sake of illustration, we explain one node and its edges in the RFG. The message board application contains two SII, namely *index.php* and *login.php*. The node *login.php* has two outgoing edges. The edge to itself corresponds to a form in the web page that constructs an HTTP POST request for *login.php* using the username and password supplied by the user. The other edge corresponds to a hyperlink in the web page for *viewforum.php*.

Developers typically enforce the intended request sequence using a combination of *interface hiding* and *validation*. Interface hiding aims to prevent users from performing an illegal action by not providing GUI that would be used to initiate the action. Web pages created by the application typically display only the necessary hyperlinks and forms in web pages that are required in the next interaction step. For example, the hyperlink or form for the next step in a transaction is displayed only if a prior step completed successfully. Similarly, the hyperlink for the administrative interface is only displayed if a user is logged in as an administrator. Validation refers to the process of embedding checks in the application in order to verify that the request in the previous step completed successfully by checking the application's state before processing the current request.

3.2 Request Integrity (RI) Attacks

Request integrity (RI) attacks violate the intended RFG of a web application by tricking interfaces into accepting and processing unintended requests[2]. RI attacks take advantage of the very nature of web applications and browsers and attack the underlying assumptions or weaknesses of prevailing methods used to enforce the intended request sequences. The root causes of RI attacks can be traced to three weaknesses. We will explain the three weaknesses and then present two classes of existing RI attacks.

First, the web pages created by a web application do not significantly vary between sessions because the interface names do not change. An attacker who understands the application and interface names can forge requests for the application. There are several opportunities for understanding an application. Because web applications are easily accessible to both users and attackers, attackers can understand the application by using it. Furthermore, the source code of some widely used web applications such as phpBB are publicly available.

Second, methods such as interface hiding and validation used by web applica-

[2] Unintended by the application designer; clearly these requests are intentional on the part of the attacker.

tions do not strictly enforce intended request sequences. Interface hiding enforces request sequences only if the application is accessed using its web pages. For example, many web applications send web pages containing login forms to users, and expect an HTTP POST request in response. However, those applications will often process a similar HTTP POST request containing login information even if the user has not retrieved a web page containing a login form. In this case, the applications naively assume that they are accessed only via their web pages and do not strictly enforce their implicit access restrictions. In the case of validation, the checks embedded inside the application have to be complete and there should be no way to bypass the checks in order to strictly enforce the request sequence. Attackers attack these underlying assumptions to identify a vulnerability in the application. Furthermore, both these methods are implemented by a developer. Therefore, the efficacy of the enforcement is dependent on the security knowledge of the developer.

Third, the prevailing access policy used by web browsers for managing cookies can be abused by malicious web sites to inject session requests—web browsers attach all the cookies associated with a web application to all requests targeting the application irrespective of the origin of a request. Therefore, if a web site A embeds a HTML form or hyperlink that invokes an interface of web site B, the browser automatically attaches all the cookies (which may include a session cookie) of web site B (if any) to the requests created by web site A. As a result, web site A can inject requests into a session that the user has with web site B without the user's knowledge or consent.

Cross-site-request forgeries (CSRF): In a CSRF attack [57], an attacker uses a malicious web site to forge a request for a trusted site as though it is coming from the victim user. In a typical scenario, a user unknowingly visits a malicious site while

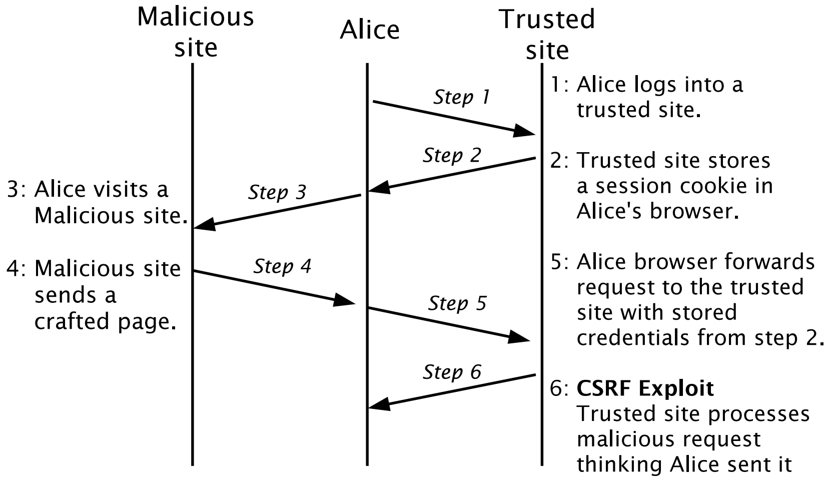

Malicious site
Alice
Trusted site
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
1: Alice logs into a trusted site.
2: Trusted site stores a session cookie in Alice's browser.
3: Alice visits a Malicious site.
4: Malicious site sends a crafted page.
5: Alice browser forwards request to the trusted site with stored credentials from step 2.
6: CSRF Exploit Trusted site processes malicious request thinking Alice sent it

Figure 3.2: Cross-site-request forgeries

having an active session with a trusted site. Figure 3.2 contains an example. Alice visits a trusted site and creates a new session (steps 1 and 2). Simultaneously, Alice also visits a malicious site (step 3), which sends a crafted page to Alice (step 4). Browsers do not have any restrictions on the URL that can be used in HTML tags such as *img*, *form*, *iframe*, etc. Using a crafted page, a malicious site can trick either the user or the browser into making a malicious request to the trusted site. When the web browser renders the crafted page, it forwards a request to the trusted site and also attaches all the cookies of the trusted site to the request (step 5). The trusted site processes the malicious request thinking it was created by Alice.

The *login CSRF* attack [7] is an interesting variation of CSRF that does not affect a user's active session. Rather, login CSRF creates a new session using the attacker's username and password. The attacker hosts a crafted page in his site that, when visited by the user, sends a login request for a trusted site using the attacker's credentials. This results in a session cookie associated with the attacker's credentials being stored in the user's browser. The attacker hopes that the user will later visit the trusted site; in such an event, all user activity will be attached to the attacker's session. An attacker could use this to monitor the activity of the user

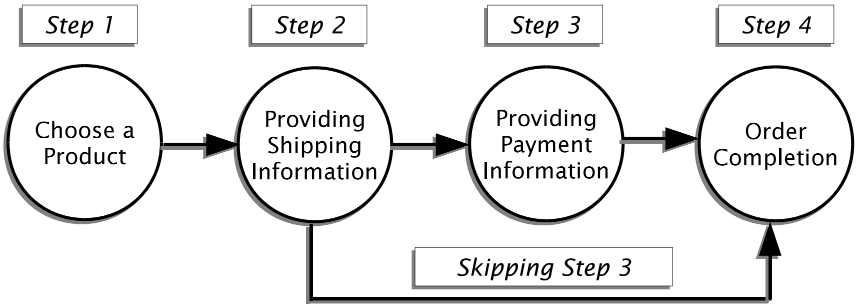


Figure 3.3: A workflow violation in a purchase transaction: Using a workflow attack, an attacker skips the third step and completes the order without paying.

on a trusted site or for other malicious purposes. For instance, an attacker may be able to track all the videos that a user views on `http://www.youtube.com`.

Workflow Attacks: A workflow is a specific sequence of interactions that a web application expects a user to perform to complete a transaction. Workflows range from simple two-step workflows to highly complicated workflows. An example of a simple workflow is a web application expecting an admin user to be signed in before accessing an administrative interface. An example of a slightly more complex workflow is a purchase transaction consisting of choosing a product, providing shipping information, providing payment information, and reviewing the order before final submission (Figure 3.3). Recall that interface hiding and validation are typically used to enforce the workflow. Workflow attackers exploit errors in these checks, or the lack of such checks, to bypass certain steps. In the simple workflow example, an attacker could directly visit the administrative interface using its URL while being logged in as a normal user. Similarly, in the a purchase workflow, an attacker may directly visit the page associated with the final step after submitting the shipping information, thereby submitting an order without payment.

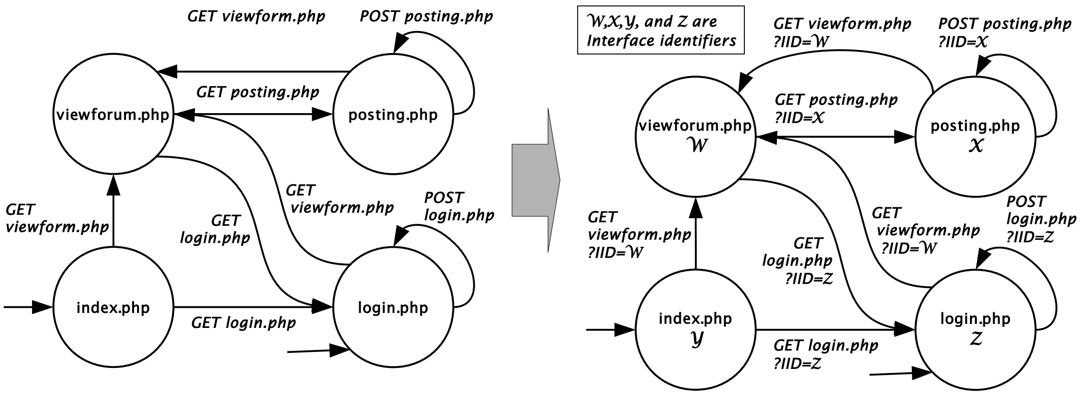


Figure 3.4: Behavior-preserving diversification of the request-flow graph

3.3 BAYAWAK

This section describes the behavior-preserving diversification approach that underlies BAYAWAK's effectiveness, the architecture and implementation of BAYAWAK, and finally how BAYAWAK eliminates RI attacks.

3.3.1 Behavior-preserving Diversification

BAYAWAK's approach is called behavior-preserving diversification and is based on the idea of building diverse computer systems [24, 35]. Instruction-set randomization (ISR) [5, 59] and address-space layout randomization (ASLR) [11, 103] are examples of methods that use this approach and are known to be effective in countering code-injection attacks. ISR methods vary the instructions that a program executes on a host machine, so that an attacker cannot determine the instruction set of the target machine. ASLR methods vary the address space of a process on each execution, so that an attacker cannot determine the address of various program locations. These methods have shown that it is possible to build practical and diversified computer systems that are resistant to a wide range of attacks.

The key idea behind BAYAWAK's approach is to diversify a web application for each user session, while preserving its behavior, such that it is impractical for both attackers and users to forge valid requests. The key component of an HTTP request is the interface name. BAYAWAK varies all the interface names in the RFG for each user session such that each of them additionally include a nonce. We will refer to each nonce as an interface identifier (IID). In effect, each user session has its own per-session RFG. There are no changes in the edges, i.e., there are no changes in the HTTP requests. The RFGs of the various sessions are isomorphic to the original RFG. Therefore, the request-response behavior is the same as behavior depicted in the original RFG. Each time the web application creates a web page, BAYAWAK modifies all the URL in the web page to include the IIDs; the IIDs are a secret shared only between the application and the web pages.

Figure 3.4 illustrates how the RFG of the message board application discussed in Section 3.1 is diversified. The four interfaces in the RFG are assigned IIDs; *viewforum.php*, *posting.php*, *index.php*, and *login.php* are assigned IIDs denoted $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ respectively. Symbols are used to denote the IIDs for easy exposition. In reality, the IIDs are sufficiently long random numbers. The strength of the session identifier can be measured using *guessing entrophy*. Guessing entrophy, in this context, is the amount of work that an attacker has to do to guess the session identifier. The guessing entrophy of a secret that has N equally probable values is given by $log_2(N)$ bits. The National Institute of Standards and Technology (NIST)'s electronic authentication guideline recommends the use of keys that are atleast 80-bits long for cryptographic keys. Similarly, IIDs should also be atleast 80 bits long.

An alternative and straightforward approach for enforcing intended request sequences is to track the valid requests issued by a user in a session, and accept an incoming request if and only if the request forms a valid request sequence when

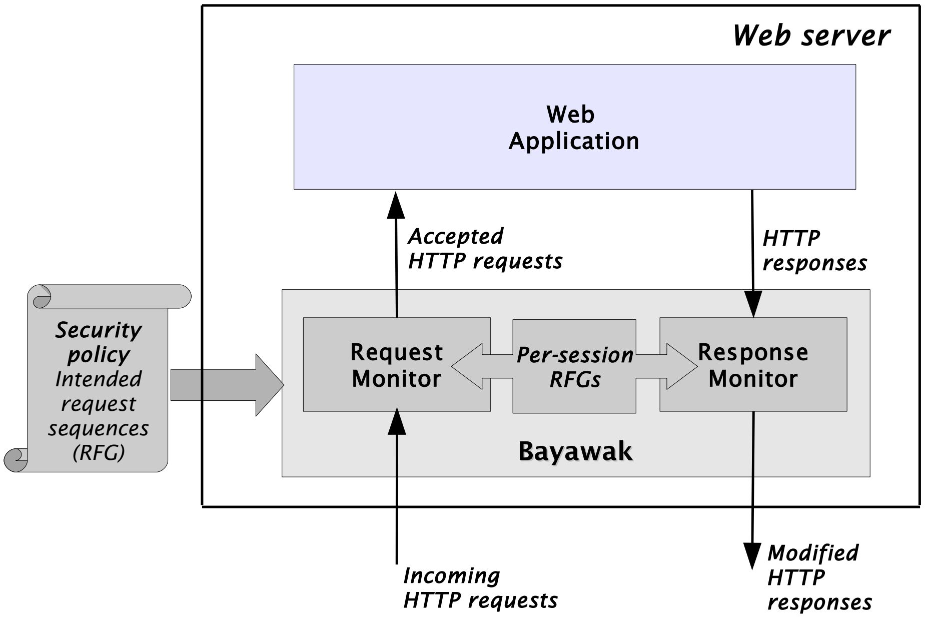


Figure 3.5: Bayawak Architecture

appended to the current request sequence. This approach is impractical because a user may have multiple ongoing request sequences, i.e., a user may be simultaneously interacting with multiple web pages in the same session by viewing them in multiple tabs of a web browser. In such cases, a server-side method cannot track the request sequences accurately. BAYAWAK's approach does suffer from this limitation.

3.3.2 Architecture

BAYAWAK is designed as a web server extension that monitors and controls a web application's execution, which comprises the HTTP requests and responses processed by the application. Figure 3.5 contains the architecture of BAYAWAK. The input to BAYAWAK is the security policy, which is the web application's intended request sequences specified in a configuration file. BAYAWAK reads this configuration file and creates a run-time monitor for the web application that comprises a request monitor and a response monitor. The run-time monitor creates the per-session RFGs, and the request and response monitors use them for their functioning. The functions of the run-time monitor can be broken in the following three steps:

1. Creating the per-session RFGs, i.e., diversifying the interface names in the application for each session.

2. Modifying the web pages created by the application to reference the correct interface names.

3. Verifying whether each incoming request carries the correct interfaces names and only forwarding conforming requests to the application.

Step 1: Behavior-preserving Diversification

BAYAWAK creates a per-session RFG whenever the web application creates a new session. A web application creates a new session in two steps. First, the application initializes a new session and assigns a session identifier. Second, the application instructs the web browser to create a session cookie using the *set-cookie* header in the response. BAYAWAK tracks the *set-cookie* header in all the responses created by the application to detect the creation of a new session. On detecting a new session, BAYAWAK generates a set of random numbers to act as the IIDs for the interfaces.

The IIDs for the interfaces are a server-side secret associated with each session. BAYAWAK stores the mapping between the interfaces and their IIDs for each session in an in-memory map. The IIDs should be sufficiently long, so that it is nearly impossible to guess them. By default, the IIDs are 256 bit numbers, but can be configured to be larger numbers.

BAYAWAK refreshes the IIDs for workflow interfaces on a per-transaction basis. Therefore, the set of IIDs that identify the interfaces involved in a workflow are unique to each transaction. All other interfaces are issued per-session IIDs, which expire only at the end of the session. BAYAWAK refreshes a per-transaction IID whenever it accepts a valid request targetting the interface corresponding to the per-transaction IID.

Step 2: Modifying the Web Pages

The web pages that the web application creates are not compatible with the per-session RFG because the URLs do not have the correct IIDs. Therefore, BAYAWAK adds the correct IIDs to the URLs in each web page created by the web application. The IIDs can be incorporated in the URLs as a parameter; Figure 3.4 shows how a URL carries the IID as a parameter. In all the URLs, the IID is added as a value for a parameter with name IID.

Whenever the web application creates a web page, the response monitor parses the web page to look for all the HTML tags that carry a URL. The following HTML tags can specify a URL as an attribute and instruct the browser to create an HTTP request for the application:

1. *href* attribute of *a*, *style*, and *link* tags.

2. *action* attribute of *form* tags.

3. *src* attribute of *frame*, *iframe*.

4. *onclick* attributes of *button* tags.

5. *refresh* attribute of *button* and *meta* tags.

6. *url* attribute of *refresh* meta tags.

BAYAWAK modifies the URL in all these tags to incorporate the IID. Whenever the URL references an interface name in the web application, BAYAWAK incorporates the correct IID in the URL.

HTTP redirects are a special case of responses that should be handled separately. Sometimes web applications may redirect users to URL2 in response to requests for URL1. The target for redirection, URL2, is specified using the *Location* header in an HTTP 302 response. The browser issues a request for the target URL2 when it receives the redirect response. BAYAWAK intercepts redirect responses and adds the IID to the URL specified in the *Location* header.

Because the IID are specific for each session and are only contained in the web pages, users can access the application only using the web pages. Essentially, the web pages become a capability required to access the application. Without the capability, users cannot access the application.

Step 3: Validating Requests

BAYAWAK validates each incoming request before allowing the web application to process the request. Figure 3.6 describes how BAYAWAK validates an incoming HTTP request.

There are two type of requests, namely session and non-session requests. Session requests are part of an on-going session and carry a session identifier. Non-session requests are not part of a session and typically target a SII and the application creates a new session when processing the requests. Non-session requests are directly forwarded to the application if they target an SII. Otherwise, the non-session request is redirected to an SII. Similarly, requests that do not carry a valid session identifier (SID) are also redirected to an SII after removing the SID.

All valid session requests are expected to carry the correct IID required to in-

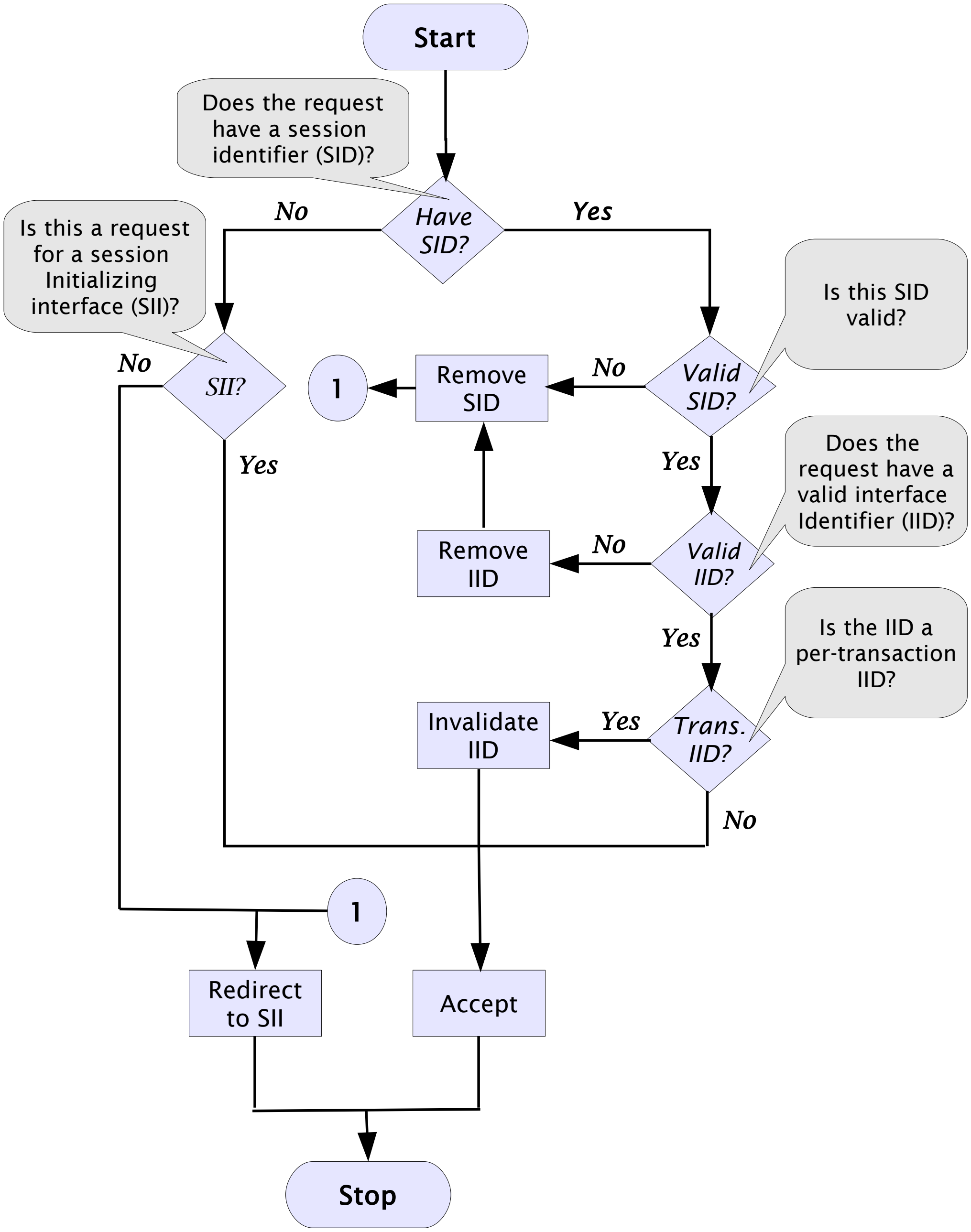

Start
Does the request have a session identifier (SID)?
Have SID?
No
Yes
Is this a request for a session Initializing interface (SII)?
SII?
No
Yes
Is this SID valid?
Valid SID?
No
Remove SID
1
Yes
Does the request have a valid interface Identifier (IID)?
Valid IID?
No
Remove IID
Yes
Is the IID a per-transaction IID?
Trans. IID?
Yes
Invalidate IID
No
1
Redirect to SII
Accept
Stop

Figure 3.6: Flowchart for request processing in BAYAWAK

voke the interface. If a session request does not contain the correct IID, then the request is treated as a non-session request and redirected to an SII, after invalidating the session. The request is processed further only if the request carries the correct IID. In the next step, BAYAWAK checks to see if the request is targetting an interface participating in a workflow. If the interface does not participate in any workflows, then BAYAWAK accepts the request. If the interface is participating in a workflow, then the associated IID is a per-transaction IID. Therefore, BAYAWAK refreshes the IID for interface and accepts the request.

Tool implementation. BAYAWAK is available in two forms—an Apache module written using mod_perl and a Java class implementing a Servlet API filter. The Apache module extends the request-response processing pipeline to implement BAYAWAK. The Java filter is essentially a hook into the Servlet interpreter for manipulating the requests and responses processed by the application. Both the implementations are functionally equivalent.

Configuring BAYAWAK

The configuration file for BAYAWAK contains the list of interfaces, interfaces participating in workflows, and name of the session cookie. The interfaces participating in workflows and the name of the session cookie are obtained directly. The methods for obtaining the list of interfaces vary depending on the type and complexity of the application.

In the case of construction frameworks such as Apache Struts[3], the list of interfaces is available as part of the application source code. Apache Struts is a model-view-controller (MVC) framework, which separate the concerns in the source code into three components, namely model, view, and controller. The model is the

[3] `http://struts.apache.org/`

database, the views are the page design code, and the controllers are the navigational code and correspond to the interfaces. In these frameworks, developers are expected to explicitly provide a mapping between the URLs in the application and their respective controllers. For example, in the Struts framework, developers are expected to provide such a mapping in an XML file. The list of interfaces can be obtained from this file directly in the case of struts.

In the case of some simple applications, where each interface corresponds to a single server-side script, the list of interfaces is essentially the list of server-side scripts in the deployment directory. Web spiders could also be used for the purpose. In more complex applications, each server-side script may implement several interfaces, each of which are distinguished by the parameters in the URL. For such applications, we need more sophisticated program analysis methods such as WAMse [43] or Tansuo [100]. WAMse uses an analysis technique based on symbolic execution for precisely identifying the interfaces in web applications.

3.3.3 Avoiding RI Attacks

BAYAWAK addresses the root causes of RI attacks as follows:

1. The web pages and the RFG are varied per session, so any information obtained from using one application instance or reading the source code is not adequate for forging requests to the application. This is because the IID required for making a request vary with session and are sufficiently long to thwart brute-force attacks.

2. Users are forced to access the application using the web pages because only the web pages carry the correct IID required to invoke the interfaces. The web pages force users to access the application in the intended way and all

the intended request sequences are strictly enforced irrespective of the completeness of the validation checks or the integrity of the session variables used in the interfaces.

3. Malicious web sites cannot access the IID necessary to invoke an interface. The IID are only embedded inside the web pages of the application. The *same-origin policy* prohibits web applications belonging to one domain from accessing the contents of the web page belonging to other domains [85].

We now describe how our approach avoids each of the RI attacks we described in section 3.2.

Cross-site-request Forgeries (CSRF): A malicious site cannot access the IID required to invoke an interface. Therefore, the malicious site can only create a request without the IID. Such requests are treated as non-session requests and are redirected to an SII, thwarting the attacks.

A Login CSRF attack forges a login request for the application. Depending on how an application creates a new session, a login request may or may not be a session request, but BAYAWAK avoids the attack in either case. In general, applications use one of two methods for creating new sessions. First, applications may create a new session in response to a non-session request. In this case, the user is not authenticated when the session is created and is expected to login only after the session is created. As a result, all login requests are session requests and are expected to have the IID that the attacker cannot access. Hence, the attack is thwarted. Second, applications may create a new session only after user authentication. In this case, the login request is a non-session request, which is forwarded to the application. Therefore, an attacker may be able to forge a login request for the application using the attacker's credentials. However, the session initializing processes creates a new session and a session RFG and the victim user does not have access to the IID compatible with session RFG. Therefore, when the user initiates a new request to the application independently using the browser, it will be associated with the session but will not have the appropriate IID. Recall that such requests invalidate the session, and are redirected to a SII, forcing the user to authenticate.

Workflow Attacks: Workflow attacks are eliminated in two ways. First, the web pages would only carry the IID for the interfaces they reference. Therefore, users cannot access interfaces that are not referenced by the web pages, thwarting arbitrary URL accesses. Second, the IID for the workflow interfaces are unique for each transaction. Typically, the web pages display the hyperlinks for the workflow steps in the intended sequence. The hyperlink for a step is displayed only on successful completion of the previous step. Therefore, the user is forced to step through the workflow only in the intended way. Moreover, because the IID for workflow interfaces expire at the end of each transaction, IID collected from completing a prior transaction in the session cannot be used to directly invoke the interface associated with the final step in a subsequent transaction.

3.4 Experimental Evaluation

The evaluation experiment used open source web applications to ascertain the following:

1. BAYAWAK's resistance to RI Attacks.

2. Run-time overhead of using BAYAWAK instances for protecting applications.

Experimental Setup. We installed and configured nine web applications, namely phpBB [77], punBB [79], Scarf [88], osCommerce [75], WebCalendar [101], Bookstore [16], Classifieds [22], Employees [29], and Events [31] on a web server configured with Intel Pentium-4 933MHz processor, 1GB RAM, Ubuntu Linux 8.04, MySQL 5.0, and the Apache 2 web server. phpBB and punBB are discussion board applications, Scarf is a conference management system, WebCalendar is a multi-user calendar application, osCommerce is an e-retailer application complete with a shopping cart, Bookstore is an online bookstore application, Events is a multi-user group-ware application, Classifieds is an online classifieds management application, and Employees is an online employee directory. Each application was installed as specified for use with a MySQL database. Web clients accessed the applications over a 100Mb Ethernet connection to measure their performance. phpBB, punBB, Scarf, osCommerce, and WebCalendar are built using PHP, so we used BAYAWAK available in the form of a mod_perl module. Bookstore, Classifieds, Employees, and Events are built using JSP, so we used BAYAWAK available in the form of a Servlet API filter.

Collecting Interface Names: We collected interface names for the various web application using two methods. For phpBB, punBB, Scarf, osCommerce, and WebCalendar, we used a simple web spider. For Bookstore, Classifieds, Employees, and Events, we used the WAMse tool to extract the list of interface names.

3.4.1 Resistance to RI Attacks.

We identified several RI attacks for all the nine web applications. Table 3.1 provides a summary of attacks.

We found several CSRF vulnerabilities in all the applications. In the discussion board applications, phpBB and punBB, the vulnerabilities allow an attacker

Web Application	**Attack Type**	**Attacks**	**Attacks Eliminated**
osCommerce	CSRF	7	7
phpBB	CSRF Workflow attacks	5 1	5 1
punBB	CSRF	6	6
Scarf	CSRF Workflow attacks	5 1	5 1
WebCalendar	CSRF	5	5
Bookstore	CSRF	4	4
Employees	CSRF Workflow attacks	3 1	3 1
Classifieds	CSRF Workflow attacks	6 1	6 1
Events	CSRF	3	3

Table 3.1: RI attacks on example applications

to forge new messages or delete existing ones. In osCommerce, we identified attacks that can add, modify, or delete products in the shopping cart and submit forged product reviews. In Scarf, the identified attacks can add or delete papers to sessions in a conference. In WebCalendar, the attacker can add or delete entries in the calendar and add or delete users from the calendar. In Classifieds, an attacker may add add, update, or delete the classified category headings or advertisements. In Events, an attacker may add, update, or delete events, user records, or category headings for events. In Employees, an attacker may add, update, or delete employee records or department names. In Bookstore, an attacker may add items to the shopping cart or add artificially high or low ratings for a book.

Scarf and Classified applications contained illegal URL access attacks. In Scarf, a server-side script that processes the site-wide configuration settings does not check whether the user has administrator privileges before making changes. The URL for the configuration page is only displayed in the web page if an administrator logs in. However, users can directly visit the URL associated with the configuration page and make changes. Similarly, in Classifieds, a server-side script that updates or deletes the category headings does not check whether the user has administrative privileges before making changes. Therefore, normal users can directly visit the URL associated with the server-side script and make changes. We created a illegal URL access vulnerability in phpBB. By default, the application displays the URL for the administrative interface only when the administrator logs in and the administrative script additionally checks the permission of the user. We disabled the permission checks to create an illegal URL access vulnerability.

For the purpose of evaluation, we created a workflow vulnerability in the osCommerce application. The checkout workflow comprises adding items to the shopping cart, entering shipping information, payment information, and final submission of the order. We created a vulnerability so that users could skip the payment step and directly proceed to the final submission step by visiting the URL directly.

All the attacks failed when we configured BAYAWAK instances for the web applications.

3.4.2 Performance Overhead

We measured the performance overhead of BAYAWAK by comparing the average response times for typical use cases of the applications with and without BAYAWAK instances. For each application multiple use cases were repeated with different users and content, providing at least 100 request-response pairs per application. Table 3.2 summarizes the average overhead of using BAYAWAK for all nine applications. The performance overhead significantly varied between the two forms of BAYAWAK. While the Apache mod_perl module incurred an overhead of 55ms, the Java-based Servlet API filter incurred an overhead of 8ms. The overhead of the Apache module could be reduced by implementing using C instead of Perl.

BAYAWAK's absolute overhead is related to the HTML document length, not ap-

Web Application	**Application Response (msec)**	**Avg. BAYAWAK Overhead (msec)**	**Percent Overhead (%)**
phpBB	278	55	19%
punBB	106	29	27%
Scarf	65	62	94%
WebCalendar	295	31	10%
osCommerce	325	96	29%
Bookstore	136	7	5%
Employees	121	4	3.5%
Classifieds	165	15	9%
Events	119	6	5%

Table 3.2: Performance overhead from using BAYAWAK instances

plication complexity. BAYAWAK detects the *set-cookie* header, creates a new RFG if necessary, but then must parse the HTML and rewrite URL. Therefore, the relative slowdown incurred by BAYAWAK will be the smallest for applications with non-trivial logic and relatively simple output. Conversely, simple applications with verbose output will have a higher relative overhead. Scarf is an example of a simple application with minimal server-side processing. Hence, its relative slowdown was the highest of all tested applications and is misleading as it represents the worst-case scenario for BAYAWAK deployment. All the other tested applications feature non-trivial logic and had significantly smaller relative overhead. In all cases BAYAWAK's overhead was imperceptible to end users. Moreover, our relative overhead estimates are conservative because the network latency in our test environment is likely to be much smaller compared to real deployments.

3.5 Discussion

In this section, we describe how BAYAWAK does not significantly interfere with widely used usability features and some potential attacks against BAYAWAK itself.

3.5.1 Usability Considerations

BAYAWAK does not significantly interfere with usability features added by both web browsers and web applications. Back button, multi-page access, and bookmarks are usability features commonly provided by web browsers for all web applications. Some web applications provide the auto-login feature to keep the users continuously signed, if they use the same machine.

Back Button: The Back button prompts the web browser to render a previously visited web page. Since many browsers implement some form of a caching mechanism, the Back button can work in one of two possible ways: the browser can fetch the page from its cache, or it can re-download the page from the web site where it originated.

If the page comes from the browser's cache, it will still contain state identifiers that were embedded by BAYAWAK, therefore any further requests generated by the page will be considered valid. Similarly, if the browser decides to re-download the web page, it will access a URL from its history. The previously accessed URL will contain an appropriate state identifier and since the state identifier is still valid, the page will be accessed and displayed as expected.

BAYAWAK can potentially disrupt a Back button navigation if the requested page is a part of a sensitive workflow and its state identifiers are expired. Since the workflow states are especially sensitive states that should only be traversed sequentially, web applications often disallow free traversal of such states, although for reasons more concerned with transaction correctness than with application security.

Multi-page Access: Web browsers generally allow users to open several browser windows or tabs that share a single web application session. From the standpoint of BAYAWAK, a user that opens several pages belonging to the same application will appear to be in several states of the DFA at same time. Because BAYAWAK tracks HTTP requests based on their target states, opening several pages does not interfere with BAYAWAK's or web application functionality.

Similarly as with Back button functionality, simultaneous multi-page access to sensitive workflow states might be restricted, either by the application itself protecting workflow correctness or by BAYAWAK expiring sensitive state identifiers.

Bookmarks: To enable users to easily visit some frequently visited web sites, browsers provide the bookmark feature. A user may store a particular URL as a bookmark and later visit the web site using the bookmark instead of typing the URL in the address bar. For web applications protected by BAYAWAK, the URL additionally contains the session-specific state identifier. If a user visits a BAYAWAK-protected application using a bookmark saved from an earlier active session, such a request will have an expired per-session state identifier, but will not contain a session cookie. Such requests would be treated as non-session requests. The Request Checker can be configured to allow non-session requests. Most web applications create a fresh session in response to such non-session requests.

Auto login: Web applications such as `www.amazon.com` use the auto-login feature to keep a user continuously signed in. However, such web applications force an additional authentication step when the user initiates a transaction such as a checkout transaction. The additional authentication step creates an additional session cookie for all the subsequent transactions after the authentication step. BAYAWAK can be configured to work with the new session cookie created as a result of the forced authentication step. All other requests can be treated as non-session requests. Therefore, BAYAWAK does not destroy the auto-login feature.

3.5.2 Attacks Against BAYAWAK

Any additional mechanism added to an existing system could potentially open new attack vectors. In this section we analyze potential attacks against BAYAWAK itself.

Mimicry Attacks against BAYAWAK: Mimicry attacks against BAYAWAK are actually cross-site scripting (XSS) attacks. XSS vulnerabilities in a web application may be used to inject a malicious JavaScript program into the web page delivered from a trusted site. The program may use the XMLHttpRequest API to forge HTTP requests to the application. Moreover, because such a program originates from the attacked web page, it can extract the target-state identifiers from the page without violating the same origin policy of the browser. The program can then forge requests including the required state identifiers and can effectively mimic a normal user visiting the web pages by using the application. For example, the MySpace *Samy* worm used an XSS vulnerability to inject a malicious JavaScript program into the attacker's profile page. Whenever a user viewed the profile of the attacker, the JavaScript program forged sensitive requests to manipulate the victim user's friends list and heroes list. Defending against this kind of a mimicry attack requires some protection against XSS attacks and is outside the scope of this work. There are several server side [14, 74, 78, 105], client side [63, 95], and hybrid solutions [55] that can be used for protecting from XSS attacks. BAYAWAK is compatible with such solutions.

Denial-of-service Attacks Using BAYAWAK: An attacker might attempt to abuse BAYAWAK to deny a user access to the protected web application. The denial-of-service attack would be launched by including some content from the target web site into the attacker's site similar to a Simple CSRF attack. When the victim accesses the malicious site and triggers the CSRF, BAYAWAK would detect the attempted attack and redirect the request to a safe landing page effectively destroying the current session that the victim has with the protected application. If the attacker crafts his malicious web page in a way that would continuously re-attempt the CSRF attack, BAYAWAK would continue to invalidate current user sessions thus preventing the user from using the protected web application. The attack however can last only as long as the malicious web page is open in the victim user's browser. To help the user realize that there might be an ongoing attack, the safe landing page that BAYAWAK redirects to can explain the disruptions and ask the user to close other browser windows.

3.6 Summary

This chapter described an approach for enforcing request integrity in web applications, and its implementation in a tool called BAYAWAK. BAYAWAK moves the request integrity enforcement mechanism from the application code into a security framework. Under this approach, the application's intended request sequences, or the request-flow graph (RFG), are specified as a security policy and BAYAWAK transparently enforces the intended request sequences, without requiring any changes in the application's source code. Our approach is based on applying a form of behavior-preserving diversification on the RFG. When evaluating BAYAWAK using nine open source web applications vulnerable to RI attacks, BAYAWAK eliminated all these attacks and incurred negligible performance overhead.

Chapter 4

A Protection Model for Web Browsers

Web applications have progressively become more sophisticated. Initially, web applications comprised a set of documents that mostly contained text to be rendered and hyperlinks to other documents, had little or no client-side code, and all the content originated from a single, trusted source. In contrast, modern web applications are highly interactive applications that execute on both the server and client. Web pages in these applications are no longer simple documents–they comprise highly dynamic contents that interact with each other. In some sense, a web page has now become a "system"–the dynamic contents are programs running in the system, and they interact with users, access other contents both on the web page and in the hosting browser, invoke browser APIs, and interact with programs on the server side.

Moreover, these web pages draw contents from several sources with varying levels of trustworthiness. Contents may be included by the application itself, derived from user-supplied text, or from partially trusted third parties. During parsing, rendering, and execution inside the browser, the dynamic and static contents of web pages can both act and be acted upon by other entities—in classic security parlance, they can be instantiated as both principals and objects. These principals and objects are only as trustworthy as the sources from which they originate.

The security of a web application is primarily dependent on the integrity and confidentiality of its resources inside the web browser. For example, session identifiers in cookies need to be protected against access by untrusted principals; code from untrusted sources must be authorized before it is allowed to modify any trusted content on a web page. Without appropriate access control in web applications, we cannot preserve the trustworthiness of contents, and security could be compromised. If we consider each web page as a "system," we need an adequate protection model in browsers to mediate the interactions within such a system.

The prevailing web browser protection model, the *same-origin policy*, has not adequately evolved to manage the security consequences of the additional complexity in modern web pages. It cannot distinguish gradations in trustworthiness, nor does it provide sufficient isolation between web browser objects to ensure proper access control. As a result, web applications have become attractive targets of exploitation. Both cross-site-scripting attacks and cross-site-request forgery attacks are examples of untrusted principals exercising control over trusted objects inside the web browser. The root cause of the problem is a failure of access control. The same-origin policy clearly violates two important principles of access control, namely separation of privilege and principle of least privilege [87].

Because of the inadequacy of web browser protection model, web applications that embed third party content in their web page cannot restrict the permissions of the third party code. For example, a blog publisher may sell a small portion of his web page to an advertising network. The advertising network, in turn, accepts JavaScript ads from its clients and displays them on the publisher's web page. The publisher has no further control over what appears in that ad space—he trusts the network to have verified all content. An attacker posing as an advertiser could compromise the integrity of the publishers web application using a malicious JavaScript program [92]. JavaScript verifiers such as ADsafe [27] could be used by an advertisement network to verify a JavaScript program, but that does not change the publisher's position: he is relying on a third-party to vouch for the trustworthiness of JavaScript programs that will run in his own web pages.

There have been other approaches for dealing with the inadequate access-control model. Web applications, as a first line of defense, employ input validation and content filtering at the server when generating the web page. The objective of this step is preventing known attacks from instantiating an untrustworthy principal inside a web page. For example, to defeat cross-site scripting attacks, we can filter out all the code from contents originating from untrusted sources. This first-line of defense has proven to be difficult to implement properly; many vulnerabilities are because of the errors in such a process [40, 44]. Second, there are browser patches that address specific attacks [50]. In general, all these approaches address the symptoms of specific problems without addressing the fundamental root cause—the lack of a robust protection model suitable for modern web applications.

This chapter describes the ESCUDO, a fine-grained web browser protection model designed based on vetted access-control principles. Under the ESCUDO model, publishers provide an ESCUDO policy for the application. The policy identifies the principals and objects inside the application and their trustworthiness. The web browser strictly enforces access restrictions on application resources based on the policy.

ESCUDO's design derives its motivation from operating systems. The protection requirements of web applications are similar to operating system programs; operating systems execute programs with non-uniform trustworthiness (kernel vs user programs). HPR model provides both separation of privilege and principle of least privilege. ESCUDO is an adaptation of HPR tailored to meet the protection requirements of web applications.

Redesigning the access-control model for web browsers involves four challenges. First, the access-control model should be able to identify principals and objects at required granularity. Second, the access-control model should use an appropriate policy to secure content with varying levels of trustworthiness. Third, a challenge unique to web applications is distributed enforcement–the applications at the server are aware of trustworthiness, but the actual interactions that have to be restricted happen at the browser. Finally, the new model should be backward compatible with the same-origin policy to facilitate incremental deployment. ESCUDO addresses each of these four challenges.

This chapter also describes the implementation of a prototype of ESCUDO for the Lobo browser, and its evaluation. The evaluation results of ESCUDO results show that ESCUDO incurs around 5% overhead. The evaluation also features two case studies that illustrate the experience of building building web applications for ESCUDO. The key contributions described in this chapter can be summarized as follows:

- ESCUDO is a new fine-grained web browser protection model to meet the protection requirements of modern applications.

- A backward-compatible configuration method that web applications can use to identify the principals, objects, and their trustworthiness in order to use ESCUDO.

- A prototype implementation of ESCUDO on the Lobo web browser.

- Case studies illustrating the experience of building web applications for ESCUDO.

Principals	Objects
HTTP-request issuing principals	**Document object model (DOM)**
- HTML Form	
- HTML Anchor	**Cookies**
- HTML Img	
- HTML Iframe	**Native Code API**
- HTML Emded	-XMLHttpRequest API
	-DOM API
Script-invoking principals	
- JavaScript Programs	**Browser State**
- UI event Handlers	- History
	- Visited link information
Plugins (Cannot be controlled by web applications)	

Table 4.1: Principals and objects inside the web browser.

4.1 Protection Requirements for Web Applications

The section provides an analysis of principals and objects in web applications and their protection requirements. This section also describes the prevailing web-browser protection mode, the same-origin policy, and its inadequacies.

4.1.1 Principals and Objects

In a web application, principals are action-inducing HTML excerpts such as JavaScript programs, and objects are application resources such as the web page contents and cookies that are targets of actions. Some HTML excerpts, such as JavaScript programs, may act as both principals and objects. Table 4.1 enumerates the principals and objects inside a web application, and the remainder of this section explains each of them.

HTTP-request Issuing Principals. HTTP-request issuing principals are HTML tags such as *a*, *img*, *form*, *embed*, and *iframe* that instruct the web browser to issue an HTTP request.

Script-invoking Principals. Script-invoking principals are HTML constructs such as *script* and the CSS *expression* that can invoke the JavaScript interpreter. Additionally, web applications can specify user-interface (UI) event handlers to be invoked for specific events using attributes such as *onload*, *onmouseover*, etc.

Plugins. Plugins are content-specific run-time environments for certain types of web contents such as Flash, Silverlight, and PDF. Additionally, browsers such as Firefox provide a framework for creating extensions, enabling users to extend the functionality of the browser. Plugins and extensions have their own security models and may or may not be controlled by the web applications. Therefore, plugins and extensions are outside the scope of this work because this research focuses only on the principals that can be controlled by the web applications.

Document object model (DOM). The DOM is a hierarchical data structure that web browsers use to manage the web page contents. Each HTML tag in the web page becomes a DOM element; JavaScript programs can read and update the HTML tags using the DOM API. Because DOM elements are targets of modification via the DOM API, all DOM elements are objects. Some DOM elements can additionally act as principals depending on the type of HTML tags they represent. For example, DOM elements representing script-invoking principals or HTTP-request initiating principals act as principals momentarily when they are instantiated.

Cookies. Cookies are web application-specific data stored in the form of key-value pairs in the web browser. Web applications create these cookies, and they typically contain data used to track sessions. Cookies are accessed in two ways. First, after a web application creates a cookie, web browsers attach the cookie in all subsequent HTTP requests back to the web application. In this case, the web browser accesses the cookie on behalf of the principal initiating the request. Second, JavaScript programs can access cookies using the DOM API.

Native Code. Native browser code is exposed to JavaScript programs via an API. For example, the `XMLHttpRequest` API is an example of native code that helps JavaScript programs to interact with server-side programs. Similarly, the DOM API is used by JavaScript programs to access and modify the web page. Web applications may not want to expose these API to untrusted code. Therefore, the ability to invoke such API must be a controllable privilege.

Browser State. Web browsers maintain browsing history and visited link information for each browsing session with a web site. This information is part of the state of a browsing session and is accessible to JavaScript programs through the DOM API. Browsing history is a log of recently visited URL and users may use this information to instruct the web browser to render a previously visited web page. Visited link information is used by web browsers to differentiate recently visited from unvisited URL—typically, web browsers use differing colors to display visited and unvisited links.

The principals inside modern web applications have evolved significantly because of the following two features:

1. **Increasing Use of Client-side Code.** With the introduction of client-side scripting languages such as JavaScript, web applications could additionally execute in the browser to provide some interactive features. For example, JavaScript programs are commonly used to display drop-down menus by dynamically updating the contents of the web page. Furthermore, AJAX enables JavaScript programs to communicate with the application at the server.

For example, an instant-messaging application might use an AJAX-based JavaScript program for communicating with the server and updating the chat window. As a result, these web applications feature several browser-side principals.

2. **Principals with Varying Levels of Trustworthiness.** In modern applications, the content inside web pages is derived from multiple sources with nonuniform trustworthiness. For example, blogs and wikis enable users to provide arbitrary text that will be part of the web pages. Because the text is supplied by the user, it should not be trusted. There are also several examples of applications including semi-trusted content. A social networking application may allow users to add applications, essentially JavaScript programs, in their profile to extend the functionality of their profile pages. Online advertising is another example, wherein a publisher may lease a portion of his web page to an advertiser, who uses a JavaScript program in the leased portion to display an advertisement. Because these contents comprise HTML tags, they can lead to principals of nonuniform trustworthiness inside the browser.

As a direct consequence of these two characteristics, we have principals of varying trustworthiness inside the web page. Currently, these principals access or modify content in the web page, invoke native API, and communicate with the application at the server, irrespective of their trustworthiness.

4.1.2 The Same-Origin Policy

The same-origin policy (SOP) identifies an application's origin as a unique combination of ⟨*protocol, domain, port*⟩. For instance, `http://www.amazon.com/index.php` and `http://www.amazon.com/search.php` belong to the same origin, but http://www.gmail.com and http://www.amazon.com do not belong to the same origin because they have differing domains. Similarly, http://www.gmail.com and https://www.gmail.com do not belong to the same origin because they use different protocols. Web browsers mark an application's objects such as cookies and document object model (DOM) with their origin, and the SOP prevents JavaScript programs from one origin from accessing application objects belonging to other origins.

An analysis of the SOP with respect to classic design principles can help in identifying its inadequacies. Saltzer and Schroeder [87] describes eight design principles for building protection mechanisms: economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability. Of the eight guidelines, the same-origin policy clearly violates two principles, namely least privilege and separation of privilege, but has done a fairly good job with respect to the other characteristics.

Under the SOP, all principals inside the web application are associated with a single principal identified by the origin and are associated with all the privileges irrespective of their trustworthiness. In addition, principals and resources across applications are not appropriately isolated from one another. As a result, the SOP violates both the principle of least privilege and separation of privilege. Both cross-site-scripting (XSS) attacks and cross-site-request forgery (CSRF) attacks are a side effect of these inadequacies.

In XSS attacks, an attacker, by exploiting weaknesses in the application, deftly constructs input data for an application that is interpreted as a JavaScript program by the web browser. Because this program belongs to application's origin, it executes with all the privileges of a principal from the same origin. . Ideally, the

JavaScript program should execute with limited or no privileges because it was derived from untrusted web content.

In CSRF attacks, a malicious site forges and injects a request into a victim user's active session with a trusted site. Some HTML tags such as *a*, *img*, and *iframes* can initiate an HTTP request. There are no restrictions on the URL that can be used in these HTML tags. In addition, web browsers automatically attach a target site's cookies to the HTTP request, irrespective of who is making the request. A malicious site abuses this weakness to forge a request for a trusted site. Ideally, principals and objects across applications should be isolated from these types of unintended interferences.

4.1.3 Protection Needs

Based on the analysis of modern web applications, the same-origin policy, and the vetted design principles, a web browser protection model requires the following two characteristics:

1. **Separation of Privilege:** Separation of privilege indicates that, if possible, privileges in a system should be divided into less powerful privileges, such that no single accident, deception, or breach of trust is sufficient to compromise the protected information. In the context of web applications, the privileges for accessing web application objects should be appropriately partitioned into less powerful sets of privileges.

2. **Principle of Least Privilege:** The protection model should be able to limit the interactions of principals based on their trustworthiness. Essentially, a principal should not have more privileges to access information or resources than required for its legitimate purpose. Therefore, origins alone are insuffi-

cient for determining the authority of a principal at the required granularity. In addition, a principal should not be able to elevate its privilege in an uncontrolled manner.

4.2 The ESCUDO Access-Control Model

A hierarchical multi-level mandatory access-control (MAC) model can clearly address the protection needs of web applications. In such models, a system organizes the principals and objects into hierarchical trust classes that each carry privileges based on the trustworthiness of the principals and objects it contains. Access decisions are primarily based on the trust class of the principals and objects. Mainstream operating systems such as SELinux and Windows Vista use such MAC models to enforce restrictions on programs based on their trustworthiness.

ESCUDO's access-control model is designed based on hierarchical protection rings (HPR) [89], which is the protection model used by operating systems. Biba [12], Bell-LaPadula [10], and HPR are three well-known hierarchical multilevel MAC models in the current literature. Of these three models, HPR is the most appropriate model for meeting the protection requirements of web application because of the similarities between the protection needs of web applications in web browsers and those of programs in operating systems. Both client-side JavaScript programs and programs in operating systems (OS) have non-uniform trustworthiness and require separation of privilege; user-level programs must be isolated from kernel-level programs in operating systems. Similarly, programs in operating systems also require the principle of least privilege; the memory address spaces of user programs should be isolated from one another. HPR meets both these requirements.

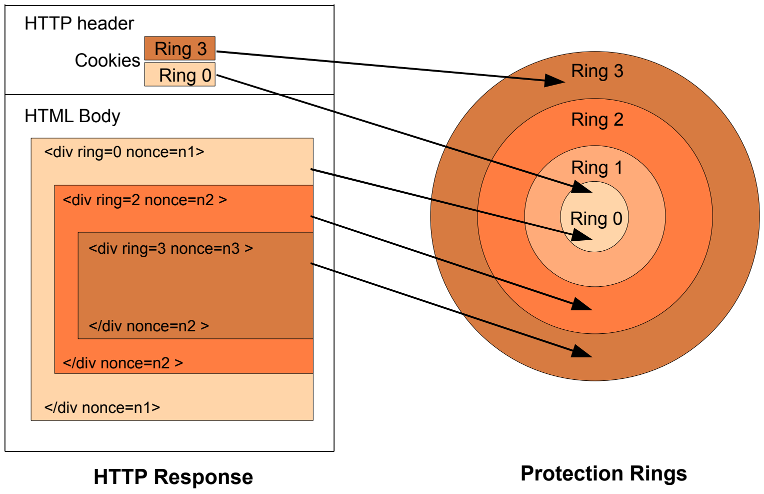


Figure 4.1: ESCUDO protection model

ESCUDO also uses hierarchical rings to organize access permissions similar to HPR. In the ESCUDO model, developers attach a security policy for the application by appropriately configuring it. The configuration reflects an ESCUDO security policy by identifying the principals and objects inside the application, and their policy settings such as ring assignments (Figure 4.1). Web applications communicate this configuration to the web browser. In the web browser, ESCUDO enforces access decisions based on the policy settings. ESCUDO comprises the following four core components:

1. *Rings:* ESCUDO provides a static set of per-page protection rings for each web application. ESCUDO treats each web page as a system and places all the principals and objects in the web page in the per-page rings. Unlike in operating systems, where there is only one set of rings, a browser can simultaneously host multiple systems or web pages, and the set of rings for each web page is independent from the others. However, the rings of web pages belonging to the same origin are compatible with each other. JavaScript programs belong-

ing to differing rings are completely isolated from one another, i.e., JavaScript programs belonging to various rings cannot share date between them.

2. *Ring Assignments:* A web application should assign all the principals and objects to rings based on their trustworthiness. The ring assignment method varies depending on the type of element and is discussed in section 4.2.1. This step is called "configuration," analogous to a system administrator configuring a system. The ESCUDO configuration method provides fine-grained granularity in specification.

3. *Access Control List (ACL):* ESCUDO allows objects to additionally use an ACL to improve the granularity of protection provided by the rings. Section 4.2.1 describes how an object can configure its ACL.

4. ESCUDO *MAC Policy:* ESCUDO enforces a MAC policy, wherein the access decisions are strictly based on the policy settings provided by the application. Section 4.2.2 describes the rules in the ESCUDO MAC policy.

The design meets the requirements summarized in Section 4.1.3. Rings and ACL facilitate the division of privileges in the system into several pieces; these pieces are organized into hierarchical trust classes, making them easy to use. The presence of multiple rings also helps the web browser identify the trustworthiness of principals at the required granularity. Therefore, ESCUDO facilitates the enforcement of both separation of privilege and principle of least privilege.

4.2.1 Rings, Ring Assignment, and ACL

ESCUDO allows web developers to organize their application's principals and objects in a set of rings. The set of rings for one web application is independent from that of others; therefore, other than defining the relationships among different rings, ESCUDO does not define what each ring means, nor does it stipulate the total number of rings. The definitions are up to the web application designers. Designers can choose the total number of rings that fit their application needs; they can make their own ring assignment, independent of other applications.

Rings in ESCUDO are labeled 0, 1, $\ldots$, N, where N is application dependent. Similar to the HPR model, higher numbered rings have lesser privileges compared to lower numbered rings. Ring 0 is the highest-privileged ring, and ring N is the least-privileged ring. All examples in this paper, for the sake of simplicity in illustration, use $N = 3$. This is a large enough number to demonstrate interaction between rings without being cumbersome; other than that, 3 is arbitrary. The remainder of this section describes how various principals and objects in the web application are assigned to rings. Web applications can communicate the ring assignment to ESCUDO either using HTML tags or optional HTTP headers, depending on the type of the object.

DOM Elements. ESCUDO requires the use of HTML *div* tags to provide the ESCUDO policy for HTML tags. HTML *div* tags were originally introduced to specify style information for a group of HTML tags; recently they have been extended for other purposes [91]. ESCUDO introduces a new attribute called the *ring* attribute for the *Div* tag. This *Div* tag attribute assigns a ring label to all the HTML tags within the *Div* tag's scope, which is the region between by the *Div* and */Div* tags (Figure 4.2). Recall that HTML tags can act as both principals (e.g., script-invoking constructs) and objects (e.g., HTML excerpts). When a HTML tag such as the *Script* tag is instantiated as a principal, the ring specified in it's *Div* tag will be the principal's ring. On the other hand, when a JavaScript program tries to access the DOM element for a HTML tag, the ring specified in the *Div* tag will be the object's ring.

```
<div ring=2 r=1 w=0 x=2>
  ...
  <div ring=3 r=2 w=0 x=2>
  ...
  </div>
</div>
```

Figure 4.2: Ring assignment using div tags.

HTML allows hierarchical *div* scopes, i.e., a *div* scope can be enclosed entirely within another *div* scope. Therefore, ring assignments can also be hierarchical. To maintain the integrity of the ring assignment, ring numbers in the inner scope must be equal to or higher than the ring numbers in the outer scope (i.e., fewer privileges). Figure 4.2 gives an example of ring assignment. Special attention must be taken to ensure the integrity of the ring assignment. In section 4.4, we will describe specific mechanisms to thwart attempts to compromise the integrity of ring assignment.

When a DOM element acts as an object, ESCUDO allows web applications to additionally specify an Access Control List (ACL). Each ACL consists of three items: permissions for `read`, `write`, and `use` operations. The meanings for `read` and `write` operations are straightforward; the `use` operation needs more explanation. In some scenarios, web browsers implicitly access objects on behalf of principals, even though the principal does not explicitly request the access. For example, whenever an HTTP request is generated for a target URL, web browsers automatically attach the cookies belonging to the target site to the HTTP request. However, the principal who initiated the request does not explicitly reference the cookies. Another example is delivering a UI event to a DOM element using a JavaScript program. We call these implicit accesses the `use` operation.

An ACL is specified using a list of attributes (`r, w, x`) in the *div* tag, where `r`, `w`, `x` refer to the `read`, `write`, and `use` operations respectively. The value of each

```
1 Set-Cookie: SID=A4P230482348903843GTE34;Domain=.google.com;Path=/;
2             Expires=Wed,13-Jan-2021 22:23:01 GMT;Secure;
3 SID: 0; r=0; w=0; x=0; // Assigning cookie named SID to ring 0.
```

Figure 4.3: Assigning cookies to rings

attribute identifies the outermost ring required for the operation. For example, in Figure 4.2, the outermost *Div* tag maps the objects inside its scope to ring 2 ("ring=2"). However, only principals in ring 0 can modify any DOM elements embedded inside the outermost *AC* tags ("w=0").

Cookies. Web applications instruct the web browser to store a cookie in the browser using the *set-cookie* HTTP header. Whenever an application creates a cookie, Escudo requires the application to use an optional header to provide the Escudo policy for the cookie. Figure 4.3 contains an example. In lines 1 & 2, the *set-cookie* header creates a cookie named SID. In line 3, the application uses an optional HTTP header with name SID to specify both the ring assignment and the ACL for the SID cookie. Cookies that contain sensitive data such as session identifiers should be mapped to a higher-privileged ring. Other cookies could be mapped to lesser-privileged rings. If ring mappings are omitted from the HTTP header, by default, all cookies are assigned to ring 0.

An alternative method is to provide the configuration as part of the *set-cookie* header. However, such a design would hinder adoption for two reasons. First, server-side frameworks should modify their cookie-handling API to support this change. Second, the modification would destroy backward compatibility with legacy applications that do not provide the configuration.

Native Code API. Web applications can optionally use an optional HTTP header to specify the ring mappings for native code APIs such as XMLHttpRequest. By default, ESCUDO assigns native code API such as XMLHttpRequest to the highest-privileged ring 0, conforming to the fail-safe defaults guideline. Web applications may assign the native code APIs to different rings.

Browser State. ESCUDO assigns internal browser state such as cache and browsing history to ring 0. In our current model, the ring assignment of browser state is not configurable. The web browser could manipulate or use the state information. However, JavaScript programs in the applications cannot manipulate the state, unless they belong to ring 0. This is because there are well-known attacks that abuse this information for tracking users [50].

4.2.2 The ESCUDO MAC Policy

ESCUDO defines a MAC policy based on rings and ACLs, and this policy controls how principals in a web page can access the objects.

For expository purposes, the remainder of this subsection uses the following notation for describing the policy: $\langle P \triangleright O \rangle$ denotes a principal P trying to perform an operation $\triangleright$ on object O. $\mathcal{R}(P)$ and $\mathcal{R}(O)$ denote the rings of the principal and object respectively. $\mathcal{O}(P)$ and $\mathcal{O}(O)$ denote the origin of the principal and object respectively. We use $\sqcap(O, \triangleright)$ to denote the least-privileged ring that is allowed to conduct the operation $\triangleright$ on the object O. An access request $\langle P \triangleright O \rangle$ is permitted if and only if the access is permitted by all the following three rules:

1. **The Origin Rule:** $\mathcal{O}(P) = \mathcal{O}(O)$

   Origin is the unique combination of $\langle protocol, domain, port \rangle$ in the URL of the web application that instantiates the principal or object. The origin rule requires the principal and object to belong to the same origin. However, unlike the SOP, this is not the only basis for access-control decisions.

2. **The Ring Rule:** $\mathcal{R}(P) \leq \mathcal{R}(O)$

   The ring rule factors the trustworthiness of the principals and objects into the model. The ring rule requires that the principal's ring should be of equal of greater privilege than the object's ring.

3. **The ACL Rule:** $\mathcal{R}(P) \leq \sqcap(O, \rhd)$

   The ACL specifies a ring for all the three operations (r, w, and x), and the ACL rule requires that a principal's ring be at least as privileged as the ring specified for a specific operation.

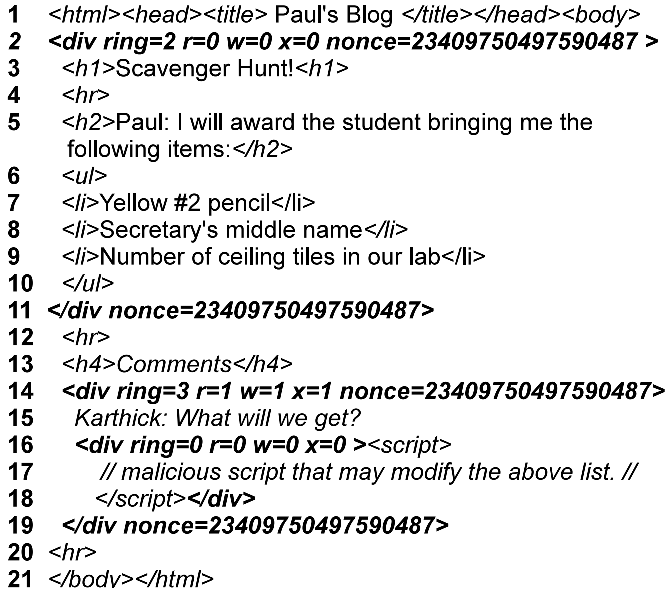
   The ACL rule further limits the access requests on objects, providing additional granularity. For example, a web application may want to avoid a JavaScript program in ring 2 from modifying other *Script* tags in the same ring. This policy can be enforced by attaching a more restrictive ACL, such as "(w=1)', to all the *Script* tags in ring 2.

   It should be noted that web applications cannot associate an ACL with an object that is less restrictive than the object's ring. For example, an object assigned to ring n cannot have an ACL that permits a principal belonging to n', where $n' > n$, to access the object. Even if the ACL is set incorrectly, the ACL will be ineffective because the Ring Rule prevents such an access.

4.2.3 An Example

Figure 4.4 contains a HTML page with ESCUDO configuration. This is an example of a blog application. In Line 2, the original blog post (Lines 2-11) is assigned to ring 2 as a principal, and its ACL indicates that only ring 0 has the permission to `read/write/use` it [1]. The user comment (Lines 14-19) is assigned to ring 3, so

[1] Please temporarily ignore the number in the `nonce` attribute. We will explain the purpose of that attribute in Section 4.4.



```
<html><head><title> Paul's Blog </title></head><body>
<div ring=2 r=0 w=0 x=0 nonce=23409750497590487 >
  <h1>Scavenger Hunt!<h1>
  <hr>
  <h2>Paul: I will award the student bringing me the
   following items:</h2>
  <ul>
  <li>Yellow #2 pencil</li>
  <li>Secretary's middle name</li>
  <li>Number of ceiling tiles in our lab</li>
  </ul>
</div nonce=23409750497590487>
  <hr>
  <h4>Comments</h4>
  <div ring=3 r=1 w=1 x=1 nonce=23409750497590487>
    Karthick: What will we get?
    <div ring=0 r=0 w=0 x=0 ><script>
       // malicious script that may modify the above list. //
       </script></div>
  </div nonce=23409750497590487>
<hr>
</body></html>
```

Figure 4.4: Example: Assigning DOM elements to rings.

even if there is a malicious script in the user comment, the script cannot access anything in the original blog post. If a ring specification is missing, ESCUDO assumes a safe default value, i.e., the ring attribute will be set to the least-privileged ring, and the ACL will be set to `r=0, w=0, x=0`, allowing only the principals in ring 0 to access it.

4.3 Implementation

This section describes a prototype implementation of ESCUDO for the Lobo web browser [90], an extensible Java-based web browser. Implementing ESCUDO on Lobo requires around 1000 lines of code. These changes are required for extracting the ESCUDO policy for the various principals and objects, tracking them inside the browser, and enforcing the policy. The remainder of this section describes Lobo's architecture and ESCUDO's implementation.

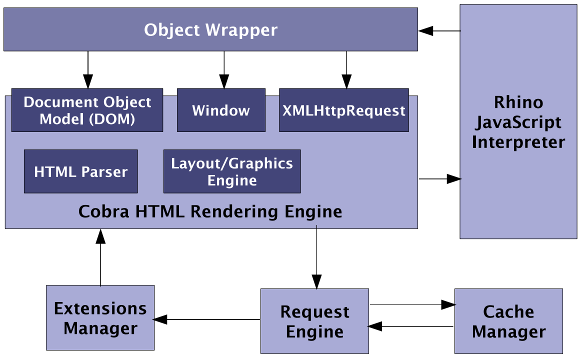


Figure 4.5: Architecture of the Lobo web browser

4.3.1 Lobo Architecture

Lobo is intended to be a platform for building new client-side web languages. Therefore, the browser architecture is designed to be easily extensible (Figure 4.5). The important components in the architecture are the *Request Engine*, *Extensions Manager*, and *Cache Manager*. The *Request Engine* forwards user requests to the servers and uses the *Extensions Manager* to choose an appropriate extension to render the response. For web pages, the extensions manager uses the Cobra rendering engine. The *Cache Manager* is responsible for caching responses based on the instructions specified in the HTTP cache-control header. The *Request Engine* interacts with the Cache Manager before issuing a network request, and serves the response form the cache if possible.

The Cobra rendering engine is responsible for parsing and rendering HTML content. Internally, Cobra comprises the HTML parser, layout/graphics engine, document object model (DOM), window, and XMLHttpRequest objects. Cobra uses the HTML parser to parse the web page and construct a DOM object corresponding to the page. Each web page is assigned a distinct DOM and a window, which is an abstraction of the window in which the web page is displayed. The XMLHttpRequest object is used by JavaScript programs to send HTTP requests.

Cobra uses the Rhino JavaScript interpreter for executing JavaScript programs. The DOM, window, and XMLHttpRequest objects are accessible to the JavaScript interpreter via the object wrapper. All requests to the three objects are mediated by the object wrapper.

4.3.2 ESCUDO Implementation

ESCUDO maintains a security context for all the principals and objects inside the application. The security context is derived from the application's ESCUDO policy. The information in the security context of each principal and object comprises the ring assignment, domain, and an ACL. Whenever a principal makes a request to access an object, ESCUDO tracks and identifies the security contexts of the principal and the object, and the ESCUDO reference monitor (ERM) determines if the access should be allowed based on the security contexts. ESCUDO's implementation can be categorized into three parts: extracting the security contexts, tracking the security contexts, and enforcing the access control policy.

Extracting the Security Contexts. The method for extracting the security context varies depending on the type of the principal and the object.

All the principals and objects inside the web page are HTML tags. Recall that the special *Div* tags specify the ESCUDO policy for HTML tags. Whenever the Cobra HTML parser parses the web page and constructs the DOM object, ESCUDO extracts the security context from the *Div* tags and stores it in the DOM elements for the respective HTML tags. ESCUDO extends the DOM object with additional properties to store the security context for each DOM element. Also, ESCUDO prohibits JavaScript programs from accessing these additional DOM properties that contain the security context. Therefore, a JavaScript program cannot modify the security contexts after ESCUDO establishes them initially.

In the case of cookies and XMLHttpRequest API, ESCUDO extracts the security context from an optional HTTP header. Lobo stores all the cookies and their attributes in a text file. ESCUDO stores the security context for each cookie along with its other attributes, and uses an in-memory table to store the security context of the XMLHttpRequest API for each web site. ESCUDO enforces its restrictions on the use of XMLHttpRequest API by controlling who can create an XMLHttpRequest object; A JavaScript program must create an XMLHttpRequest object to be able to use the API.

In the case of browser state such as history, recall that ESCUDO assigns them to ring 0 by default. ESCUDO's current implementation deals only with the history object, and ESCUDO extends the history object with a property to denote the ring assignment. In Lobo, each window has its own history object to keep track of recently visited URLs. All these history objects are marked as belonging to ring 0 at creation time.

Tracking the Security Contexts. ESCUDO tracks the security contexts of principals based on the execution of three threads, namely parsing and rendering, UI-event handling, and XMLHttpRequest callback processing. All the principals inside the web page execute in one of the three threads. ESCUDO maintains a webpage-specific table to maintain the security contexts of the principals currently executing in the three threads. Each time one of the three threads switches to a new principal, ESCUDO updates the security-context table accordingly. All accesses made by principals executing in these threads are constrained by the security contexts referenced in the webpage-specific table.

The parsing-and-rendering thread processes each HTML tag in the web page in the order of their appearance. The common processing work for all HTML tags is creating DOM elements and adding them to the DOM tree. Processing some HTML tags, such as *script*, *img*, *iframe*, etc., can momentarily create HTTP-request issuing or script-invoking principals. Before instantiating the principal, ESCUDO retrieves the principal's security context from the DOM and updates the web-page specific table and then instantiates the principals.

The UI event handling thread processes UI events. Inside a web page, an UI event targets a DOM element. For example, a text field inside the web page could be a target for an *onmouseover* event, i.e., the text field may require to run a JavaScript program whenever a user's mouse moves over the text field. In this case, a JavaScript program is referenced in the HTML tag of the text field using a HTML attribute called *onmouseover*; the browser executes this JavaScript program whenever the *onmouseover* event is delivered to the text field. In the presence of ESCUDO, this JavaScript program should execute with the security context of the text field. Therefore, whenever a UI event is delivered to a DOM element, ESCUDO retrieves the security context from the target DOM element and updates the webpage-specific table to associate the context with the UI-Event-handling thread.

Controlling the JavaScript program's access to XMLHttpRequest API is straightforward, but further tracking is necessary for enforcing the application policy during XMLHttpRequest callback processing as follows. XMLHttpRequest has two types of callback mechanisms: synchronous and asynchronous. In the case of synchronous callbacks, a JavaScript program issues a XMLHttpRequest and also waits until the response arrives to process it. Because the JavaScript program creating the request is already tracked, no additional tracking is necessary for the callback. In asynchronous callbacks, a JavaScript program creates a request and registers a callback function that will process a response to the request whenever it arrives. The callback function should execute in it's creator's security context.

Therefore, ESCUDO stores the creator's security context in XMLHttpRequest object, which updates the security-context table of the callback function's thread with the creator's privileges before the callback function is invoked.

Enforcing the Access Control Policy. ESCUDO's enforcement comprises three parts. First, ESCUDO isolates all JavaScript programs belonging to the various rings, i.e., a JavaScript program in one ring cannot share code or data with a JavaScript program in a different ring. Second, ESCUDO mediates all accesses to the web application's objects and enforces restrictions based on the application's ESCUDO policy.

ESCUDO creates a JavaScript context for each ring. A JavaScript context comprises native JavaScript objects and custom JavaScript objects. Native objects include objects such as Date, String, etc. The JavaScript interpreter natively supports these objects. Custom objects are objects that are defined by the JavaScript programs. Each time the Cobra parser invokes the JavaScript interpreter to execute a JavaScript program, it passes the JavaScript context corresponding to the program's ring. As a result, JavaScript programs belonging to a ring can access only the custom and native objects that reside in the JavaScript context belonging to the ring. This isolation is necessary to prohibit a JavaScript program in one ring from interfering with the execution of programs in other rings.

Figure 4.6 depicts the high level architecture of the ESCUDO reference monitor (ERM). The ERM enforces access restrictions on the DOM object, XMLHttpRequest API, cookies, and browser state as follows:

- *Document Object Model:* ESCUDO mediates all accesses to DOM elements in the object wrapper. Whenever a JavaScript program tries to access a DOM element, ESCUDO intercepts the access inside the object wrapper. A JavaScript

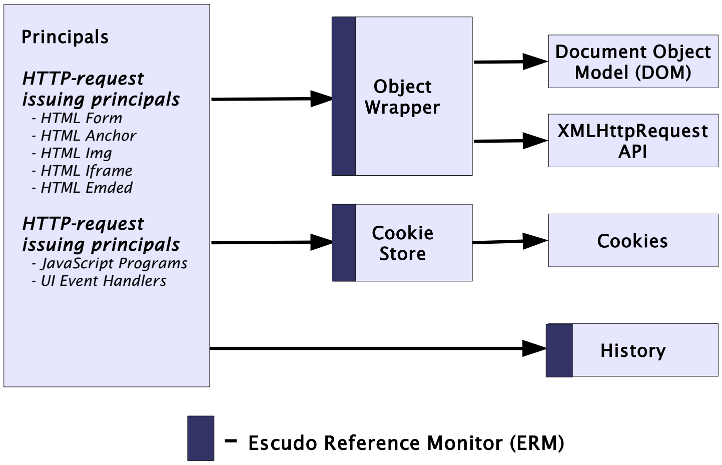


Figure 4.6: Escudo reference monitor (ERM)

```
var scriptArray = document.getElementsByTagName('script');

for (var i=0; i¡scriptArray.length; i++)
  scriptArray.item(i).src = “http://evil.com/evil.js”;
```

Figure 4.7: A JavaScript program that accesses a list of DOM elements

program may either access a single DOM element or a list of DOM elements. Depending on the case, ESCUDO retrieves the security context of both the principal and object(s) involved, and determines if the access should be allowed.

The case of a JavaScript program accessing a single element is straightforward. Figure 4.7 contains a JavaScript programs that accesses a list of DOM elements. In this example, the JavaScript program obtains a list of DOM elements corresponding to the HTML *script* tag in the web page. In response to this request, the object wrapper creates a list of DOM elements to return to the program, and ESCUDO inspects this list and removes DOM elements that the principal is not entitled to access.

A JavaScript program may create a new DOM element and add it to the DOM object. In this case, ESCUDO copies the security context of the principal to the DOM element, so the new DOM element will have the same ring assignment and ACL as its creator.

- *XMLHttpRequest:* ESCUDO mediates the creation of XMLHttpRequest object inside the object wrapper. Whenever a JavaScript program creates an XMLHttpRequest object, ESCUDO intercepts the call inside the object wrapper, retrieves the security context of principal, and either permits or blocks the request based on the policy.

- *Cookie Access:* ESCUDO mediates all accesses to cookies in the cookie store. Cookies are either accessed using DOM APIs or automatically attached to an HTTP request for the same target domain. In either case, the cookie store mediates all cookie accesses. ESCUDO intercepts each access, retrieves the security contexts of the principal and the cookies, and returns only those cookies that can be accessed by the principal. A JavaScript program may modify an existing cookie using the *setCookie* DOM method. However, ESCUDO prohibits changes to the security context using this method.

- *Browser State:* Whenever a JavaScript programs invokes a history method such as *history.back()*, the request is translated to a function call on the history object of the respective window. In all such functions, ESCUDO retrieves the principal's security context and permits the function call if and only if the principal belongs to ring 0.

There are two types of HTTP-request invoking principals that are not part of the web application, namely user-initiated requests such as bookmarks and cross-site requests. ESCUDO enforces the following restrictions for them.

1. *User-initiated requests:* There are two types of HTTP requests that a user may initiate without interacting with a web page. First, a user may directly type a URL in the address bar. Second, the user choose to visit a URL in his bookmark. In these cases, ESCUDO treats these requests as originating from ring 0. These interactions are a bootstrap process for accessing the web application. Therefore, these principals can may have access to cookies belonging to their target domain. When the target domain responds with the web page, ESCUDO establishes the policy for the web application's objects and its ESCUDO policy, and further interactions with the web page are constrained by the policy.

2. *Cross-site requests:* A web application can create a HTTP request for a different domain. In these cases, ESCUDO does not attach the cookies owned by the target domain to the request because the principal belongs to a different domain. ESCUDO could be extended to facilitate publishers to provide

a cross-domain policy, which could specify if a cross-site request should be allowed to reference the cookies.

4.4 Enforcing Policy Integrity in Escudo

The key to ESCUDO's enforcement is the safety and integrity of the policy provided by the publisher. This section outlines the malicious manipulations threats to an ESCUDO policy and ESCUDO's countermeasures to avoid them.

4.4.1 Policy Manipulation Threats

This section presents a categorization of ESCUDO policy manipulation threads for HTML tags. When a web browser receives a web page, it parses and constructs a DOM object for the web page, and then renders the page. An attacker may have opportunities to modify the policy either before the DOM creation, during the DOM construction, and after DOM creation. The remainder of this section describes such attack vector. However, none of these attacks work in ESCUDO. Section 4.4.2 describes the measures that ESCUDO implements to ensure the integrity of ESCUDO policy.

Before DOM Construction

When an application is creating a web page to be sent to the user's browser, vulnerabilities in the application can help an attacker to trick the application into creating a malformed web page, in which certain portions of the web page are disabled. This could be done by confusing the parser either with respect to the boundary of a HTML comment or a HTML attribute.

Figure 4.8 contains an example where an attacker injects HTML comments into

```
1 <div ring=0 r=0 w=0 x=0>
2   ..
3   <!--
4   <div ring=2 r=1 w=1 x=0>
5     ...
6   </div>
7   -->
8   ...
9 </div>
```

Figure 4.8: Tag boundary confusion

```
1 <div ring=0 r=0 w=0 x=0>
2   ..
3   <img src=``http://evil.com?params=
4   <div ring=2 r=1 w=1 x=0>
5     ...
6   </div>
7   '' />
8   ...
9 </div>
```

Figure 4.9: Attribute boundary confusion

select portions of the web page to disable certain web page contents. In this example, the attacker has the opportunity to inject arbitrary text in lines 3 and line 7, and the attacker injects a beginning and ending HTML comment tag in those lines. As a consequence, the Div tag and its enclosing contents between lines 4-7 are disabled. We call this boundary confusion because the parser is confused about the boundary of the HTML comment tag.

Figure 4.9 contains an example where an attacker injects an Img tag into a portion of the web page to disable certain web page contents. The vulnerability highlighted in this example is the same as Figure 4.9, i.e an attacker can inject arbitrary tect in lines 3 and 7. In this case, the attacker injects an Img tag, whose Src attribute begins in line 3 and both the attribute and the tag are closed in line 7. As a consequence, the Div tag in lines 4-7 is enclosed within the img tag as an attribute.

```
1  <div ring=2 r=2 w=0 x=0>
2    ...
3    <div ring=3 r=3 w=2 x=2>
4      <script>
5        document.write(’</div> <div ring=2 r=2 w=0 x=0>’);
6        document.write(’<script type="text/javascript"
7           src="http://someadagency.com/somead?params">’);
8        document.write(’</script>’);
9        document.write(’</div>’);
10     </script>
11   </div>
12 ...
13 </div>
```

Figure 4.10: JavaScript program that modifies the scope of ESCUDO policies

During DOM Construction

During the DOM construction, malicious JavaScript programs inside the web page may attempt to alter the scope of the *Div* tags that assign a group of tags to a ring. Figure 4.10 contains an example of such a program. In this example, contents of web pages between lines 2-12 are mapped to ring 0 and contents of web pages between lines 4-10 are mapped to ring 3. However, the JavaScript program enclosed in lines 5-9 alters the scope of the ESCUDO policy in the Div tag in line 3, by prematurely terminating the Div tag in line 5 and starting a new Div tag belonging to ring 2.

After DOM Construction

After the DOM is constructed, the only way for an attacker to manipulate the policy is using a client-side JavaScript program. Recall that the ESCUDO policy for each HTML tag is maintained as a property of the corresponding DOM element. Therefore, unless the browser enforces special constraints, JavaScript programs can access and modify the policy of existing DOM elements and also create new DOM elements and attach ESCUDO policies to them.

```
1   <script>
2   var alldivtags = document.getElementsByTagName('Div');
3
4   for(var i=0; i<alldivtags.length; i++)
5   {
6      var tmptag = alldivtags[i];
7
8      if(tmptag.getAttribute('ring') == 3)
9         tmptag.setAttribute('ring',0);
10  }
11  </script>
```

Figure 4.11: JavaScript program that modifies the ESCUDO policy of existing DOM elements

```
1   <script>
2   var evil_script = document.createElement('script') ;
3   evil_script.setAttribute('src','http://www.evil.com/evil.js');
4
5   var bodytag = document.getElementsByTagName('body');
6   bodytag.appendChild(ring0_div);
7   </script>
```

Figure 4.12: JavaScript program that adds new DOM elements in ring 0

Figure 4.11 contains a JavaScript program that modifies the ESCUDO policy of existing DOM elements. This program elevates all DOM elements in ring 3 to ring 0 by modifying their ring attribute. In line 2, the program obtains a reference to a list of all "Div" tags in the web page. In lines 4-10, the program reads each "Div" tag in the list and sets its ring attribute to 0, whenever its ring attribute is 3.

Figure 4.12 contains a JavaScript program that creates new DOM elements and tries to add it a web page region assigned to a higher privileged ring. In lines 2 and 3, the program creates a script tag that references a JavaScript program from a malicious web site. In lines 5 and 6, the program adds the new script tag as a child of the body tag, which in this example is assigned to ring 0. As a result, the new script tag will also belong to ring 0.

```
<div ring=0 r=0 w=0 x=0 nonce=1ACE237490832747 >
  ..
  ...
</div nonce=1ACE237490832747 >
```

Figure 4.13: Div tag with nounces

4.4.2 ESCUDO Countermeasures

1. **ESCUDO policy is neither accessible nor changeable to DOM API for existing DOM elements.**

   ESCUDO prohibits read or write access to the DOM properties that specify the ESCUDO policy. In effect, the ESCUDO policy for a DOM element can not be changed once it has been set. As a consequence, the attack in Figure 4.11 cannot succeed.

2. **Newly created DOM elements inherit the creator's policy, and additions to the DOM are required to conform to the scope rule.**

   A JavaScript program may add new DOM elements to the web page using DOM API. The new DOM elements inherit the policy from the JavaScript program that creates it. When a JavaScript program tries to add the new DOM element to the DOM, such additions are strictly subject to the scope rule. Therefore, a DOM element cannot have a higher privilege compared to the parent tag to which it is added. As a consequence, the attack in Figure 4.12 cannot succeed.

3. **All Div tags that carry the ESCUDO policy use nounces to authenticate themselves to the browser.**

   Each Div tag that carries an ESCUDO policy has a unique nonce in the beginning tag and the ending tag (Figure 4.13). The nonce is unique to the web

```
1   ..
2   <!--
3   <!-- -->
4   <div ring=2 r=1 w=1 x=0>
5     ...
6   </div>
7   <!-- -->
8   ''-- >
9   ...
10  <img src=``http://evil.com?params=
11  <!-- -->
12  <div ring=2 r=1 w=1 x=0>
13    ...
14  </div>
15  <!-- -->
16  ''-- >
```

Figure 4.14: Div regions with empty comment tags

page. The nonce to expect in each of the Div tags is specified in the HTTP header. Browser interprets a Div tag carrying an ESCUDO policy only if it carries the correct nonce. As a consequence, attackers cannot inject any ESCUDO policy carrying Div regions without accurately guessing the nonce. Therefore, the attack in Figure 4.10 cannot succeed.

4. **All Div tags that carry the ESCUDO policy use a empty comment tag before and after the Div region**

   Figure 4.14 contains an example, where we have added empty HTML comment tags before and after the div regions. In the figure, the empty comments tags are show in italics. We illustrate how these empty comment tags avoid the attacks in Figure 4.8 and 4.9.

   Let us consider the attacker injected HTML comment beginning and ending in lines 2 and 8. The presence of the empty HTML comment in line 3 causes the parser to terminate the HTML comment in line 2. As a result, the Div region in line 3 is interpreted as expected.

Let us consider the attacker injected HTML Img tag in line 10. The presence of the empty HTML comment in line 11 causes the parser to terminate the Img tag beginning in line 10. Therefore, the Div region beginning in line 12 is interpreted as expected.

By preceding the Div region with an empty HTML comment, ESCUDO terminates any open HTML tags or attributes prior to the Div region.

4.5 Evaluation

This section describes ESCUDO's experimental evaluation that assessed the following: (1) how web applications can take advantage of ESCUDO (2) compatibility with legacy web applications, (3) resistance to common XSS and CSRF attacks, and (4) performance overhead.

4.5.1 Building ESCUDO-based Web Applications

To illustrate how to build ESCUDO applications, two open-source web applications, phpBB and PHP-Calendar, were analyzed and ESCUDO policies were created for meeting their security requirements. phpBB (http://www.phpbb.com/) is a multi-user message board application and PHP-Calendar (http://www.php-calendar.com/) is a multi-user online calendar application. We analyzed the principals and objects in these web applications and understood their security requirements. It did not take more than a day for modifying either application to use ESCUDO. A developer who knows the application better would be able to make the changes faster.

phpBB

phpBB is primarily used to create an online community, in which users may interact with one another by posting new topics for discussion, responding to existing discussion threads, or sending private messages to other users. The key security concern in phpBB is appropriately limiting the capabilities of messages posted by users. Table 4.2 describes the security requirements. Application contents, such as trusted JavaScript programs and HTML forms included into the web page by the application, require access to the messages, cookies, and the XMLHttpRequest object. Topics, replies, and private messages, however, do not require such privileges. Furthermore, user-provided topics, replies, and private messages are not expected to manipulate the contents of the web page. An ESCUDO policy was created to meet these requirements.

Principal	**Modify Messages (DOM)**	**Access Cookies**	**Access XMLHttpRequest**
Application contents	Yes	Yes	Yes
Topics and replies	No	No	No
Private messages	No	No	No

Table 4.2: Security requirements for phpBB web application.

The ESCUDO policy for phpBB is described in Table 4.3. The head portion of the page contains style information and some trusted JavaScript programs. These are all assigned to ring 0 and can be manipulated only from ring 0. The content enclosed between the *body* and */body* tags is a mix of application provided content and

Configuration	**Cookies**	**XMLHttpRequest**	**Application contents**	**Topics & Replies**	**Private Messages**
Ring	1	1	1	3	3
Access-control List					
Read access	≤ 1	≤ 1	≤ 1	≤ 2	≤ 2
Write access	≤ 1	≤ 1	≤ 1	≤ 2	≤ 2

Table 4.3: ESCUDO policy configuration for phpBB.

user-provided topics, replies, and private messages. The body tags are assigned to ring 1 and can only be manipulated by principals in rings 0 and 1. Topics, replies, and private messages appearing inside the body are assigned to ring 3, but their ACL is configured so that they can be manipulated only by principals in ring 0, 1, and 2. Therefore, content provided by one user is completely isolated from content provided by another. There are two cookies in the web application, namely *phpbb2mysql_data* and *phpbb2mysql_sid*. Both cookies are assigned to ring 1. The cookies are attached only to HTTP requests generated by principals belonging to rings 0 and 1.

phpBB uses a template engine similar to *Smarty* to provide a separation between the HTML layout and the content that will be placed in it. In this template system, a web application defines a template for each web page, and a PHP program inserts the content/data in this template to create a web page. Because the ESCUDO policy will not change each time a web page is produced, it can be added to the template. Therefore, the ESCUDO policy for HTML tags was added to the template for each web page.

phpBB creates two session cookies and sends them to the browser using the *set-cookie* header. There were two places in the source code that create the cookies. In these places, the PHP source code was modified to add the optional HTTP header using the PHP *header* function for specifying the ESCUDO policy for the cookies.

PHP-Calendar

PHP-Calendar is meant to facilitate a group's collaborative creating and tracking of events. An event in PHP-Calendar consists of a text message describing the event, time, and date of the event. The key security concern in PHP-Calendar is appropriately limiting the capabilities of events inside the web application. Table

4.4 describes the security requirements for PHP-Calendar. Application content requires privileges to modify events, session cookies, and use the XMLHttpRequest object. However, events should be prohibited from modifying other events via the DOM API and are not expected to manipulate cookies or use the XMLHttpRequest object. The security requirements for the PHP-Calendar application are very similar to phpBB.

Principal	**Modify Messages (DOM)**	**Access Cookies**	**Access XMLHttpRequest**
Application content	Yes	Yes	Yes
Calendar events	No	No	No

Table 4.4: Security requirements for PHP-Calendar.

Configuration	**Cookies**	**XMLHttpRequest**	**Application content**	**Calendar events**
Ring	1	1	1	3
Access-control List				
Read access	≤ 1	≤ 1	≤ 1	≤ 2
Write access	≤ 1	≤ 1	≤ 1	≤ 2

Table 4.5: ESCUDO policy for PHP-Calendar.

We created an ESCUDO policy for meeting the PHP-Calendar security requirements. Table 4.5 describes the ESCUDO policy for PHP-Calendar. In all the web pages inside PHP-Calendar, the body of the web page is a mix of application content and user created events. The content enclosed between the body tags is mapped to ring 1 and its ACL is configured to permit manipulation only by rings 0 and 1. However, as allowed by the scoping rule, the individual calendar events that appear within the body are assigned to ring 3 and configured to allow manipulation by rings 0, 1, and 2. Therefore, the various calendar events are isolated from one another. All the session cookies in the application are assigned to ring 1, along with the XMLHttpRequest object.

PHP-Calendar has created an HTML type system using PHP classes for separating the HTML layout from the internal processing required for producing content for the web page. This organization made it easier to modify the layout to incorporate the ESCUDO policy for HTML tags. The ESCUDO policy was added to the web page by extending the PHP classes for the HTML tags. For cookies, the ESCUDO policy was added using the same method as phpBB.

Framework Support for ESCUDO Configuration

Creating ESCUDO configurations for static web pages is very straightforward because the configuration can be directly embedded in the web page and is not expected to change. In the case of web applications with significant portions of dynamic code, we need more systematic methods for specifying the configurations. Otherwise, specifying the configuration will be cumbersome.

HTML template engines provide a structured method for isolating the view elements from the business logic. The view elements are specified in a template and data computed at run-time is plugged into the template to create the web page. The ESCUDO policy can be specified in the template, isolating the configuration from dynamic data. Sophisticated template engines such as StringTemplate [76] provide a stricter separation between view and model, making it easy to manage ESCUDO policies for large-scale web applications.

Language-based information flow could also be used to create ESCUDO configurations. The SIF framework is an extension of the Java Servlet framework to enforce confidentiality and integrity policies at run-time using language-based information flow [21]. In SIF, developer provides annotations in the source code to mark the confidentiality and integrity policies. These policies are then enforced at run-time when the program executes at the server. The confidentiality and integrity policies on the data can be used to automatically derive the ESCUDO configuration for the web page, when the web page is created.

4.5.2 Compatibility with Legacy Applications

There are two types of compatibility concerns with respect to ESCUDO: (1) compatibility of ESCUDO-configured applications with non-ESCUDO browsers, and (2) compatibility of ESCUDO-based browsers with non-ESCUDO applications.

ESCUDO-configured applications are compatible with non-ESCUDO browsers. The only aspect that distinguishes an ESCUDO-based application is the availability of ring mappings for cookies, the XMLHttpRequest API, and DOM objects. For DOM objects, ring mappings are specified using *AC* tags, which are additional attributes in the `div` tag. Non-ESCUDO browsers would simply ignore these attributes. For cookies and the XMLHttpRequest API, ring mappings are specified using an optional HTTP header; they also will be ignored by non-ESCUDO browsers.

ESCUDO-based browsers are also compatible with non-ESCUDO applications. Non-ESCUDO applications do not provide any ring mapping. Therefore, all principals and object inside the application are assigned to a single ring, effectively mimicking the same-origin policy.

4.5.3 Defense Effectiveness

Recall XSS and CSRF are side effects of using a inadequate protection model. Therefore, the evaluation work ascertained ESCUDO's effectiveness in avoiding common XSS and CSRF problems.

Four XSS attacks were created for each web applications. In phpBB, the XSS attacks could be used for posting new messages on behalf of victim users and for modifying existing messages. In PHP-Calendar, XSS attacks could be used for creating new events on behalf of victim users, and modifying existing events. All the attacks were neutralized in the presence of ESCUDO. This is because the

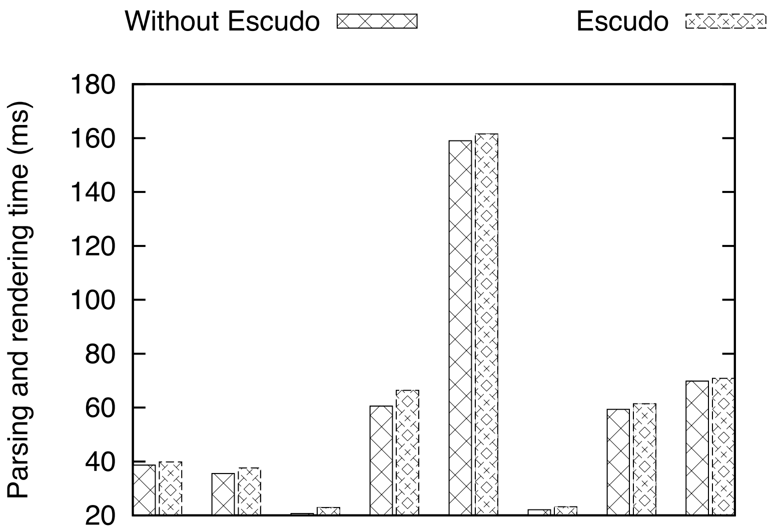


Figure 4.15: Performance overhead in parsing and rendering (in 8 different scenarios).

ESCUDO policy assigned user-influenced regions to belong to ring 3, and there principal arising out of these regions could not neither post messages nor modify existing messages.

Five CSRF attacks were created for each web applications. We set up a malicious web site that crafted cross-origin requests for the two web applications, when accessed by a user. When accessed using the ESCUDO-enabled Lobo browser, the malicious site still issued the requests for the two web applications. However, ESCUDO did not attach the session cookie automatically to the requests (because of the insufficient privileges of the principals), neutralizing the attacks.

4.5.4 Performance Overhead

ESCUDO's execution is invoked during both parsing and rendering of web pages and while responding to UI events. Therefore, the performance overhead was estimated by measuring the slowdown in both parsing and rendering activities. Lobo was instrumented to measure the amount of time taken to parse the web page and also to respond to UI events. In both cases, no noticeable overhead was observed in any of the activities. The experiment used 8 web pages with varying amounts of *Div* tags and dynamic content. The overhead was measured by comparing the average time taken for parsing and rendering the 8 pages for over 90 execution both in the presence and absence of ESCUDO (Figure 4.15). The average overhead was 5.09%. ESCUDO primarily does bookkeeping to keep track of the principals and this activity does not add any significant cost. Similarly, we did not notice any overhead for UI event handling.

4.5.5 Applications of ESCUDO

ESCUDO empowers a publisher to appropriately to restrict all browser-side interactions by providing a policy. Therefore, a publisher can meet its protection requirements by appropriately adding ESCUDO policy. The following list contains several examples of how ESCUDO can be used for meeting web application security needs:

1. *Constrain untrusted JavaScript programs:* Web applications such as blogs and wikis accept user-supplied input and add it to their web pages. Moreover, these applications allow users to use a set of HTML tags such as *b*, *i*, *a*, etc., in their text to improve its look and feel. To preserve the integrity, these applications need to filter the user-supplied text to make sure there are no malicious script-invoking constructs. Recall that such filtering is very difficult to get it right. With ESCUDO, an application can add an ESCUDO policy to the user-supplied portion to restrict the permissions of any principal originating from user-supplied text.

2. *Constrain third party JavaScript utilities:* Web applications commonly use third party supplied JavaScript libraries and programs in their web pages. For

example, several web applications use third party JavaScript programs for keeping track of web-site statistics. With ESCUDO, a publisher can restrict the permissions of such programs by adding an ESCUDO policy without having to peruse the source code.

3. *Online advertising:* In the case of online advertising, recall that a publisher may lease portions of his web page to an advertising network. With ESCUDO, a publisher can enforce restrictions on JavaScript programs from the advertising network by providing an ESCUDO policy for the leased space. For example, a publisher can map the leased portion of the web page to ring 3. As a result, a publisher need not place any trust on the advertising network's process for vetting its JavaScript programs – the advertising network's JavaScript programs are constrained by the ESCUDO policy inside the web browser.

4.6 Summary

There is a disconnection between the protection needs of modern web applications and the prevailing protection model–same-origin policy. This chapter described ESCUDO, a new protection model that is systematically designed to meet the protection needs of modern web applications. This chapter also described the implementation of a prototype of ESCUDO in the Lobo web browser, and illustrated how web applications can use ESCUDO to secure their resources using case studies. The evaluations results indicate that ESCUDO is a practical access-control model. In addition, ESCUDO can be incrementally deployed because it retains backward compatibility with legacy applications.

Chapter 5

Designing Secure Web Applications

Web applications continue to be attaractive targets of exploitation. Verizon Business' 2010 Data Breach Investigations Report (DBIR), a study conducted along with United States Secret Service, reports that 95% of data breach incidents were perpetrated by remote organized malicious groups hacking web servers and applications. According to WhiteHat Security's 2010 Web Security Statistics Report, most webapplications have between 0 and 42 vulnerabilities, and an average webapplication has nearly 13 serious vulnerabilities. These statistics show the wide prevalance of vulnerabilities in web applications.

Moreover, an attacker does not need any resources to initiate most web-based attacks. For example, in attacks such as cross-site request forgeries, an attacker needs to only post a malicious URL, and all victims users that visit the URL are affected. Therefore, attackers could use web-based attacks to affect a large number of users without requiring any resources. The wide-prevelance of web application vulnerabilities, the non-need for any significant resources to attack, and the large number of users using a web application together make web applications highly attaractive targets.

A key reason for the wide prevelance of web application vulnerabilities is that some of them are rooted in the very nature of web applications. Request integrity attacks are an example. As explained in Chapter 3, because of the very nature of web applications, attackers can understand the structure of a web application by using it, and construct seemingly valid HTTP request sequences for it. Using this oppurtunity, an attacker could potentially trick a web application into processing an incorrect sequence of HTTP requests, affecting its integrity and privacy.

For problems rooted in the nature of web applications, an important question is how we could design applications to eliminate classes of vulnerabilities. While tools such as BAYAWAK are useful for eliminating request integrity attacks, we should still investigate classes of vulnerabilities and identify what can be done to improve the design of web applications such that we can avoid these attacks. Such investigations may be useful for educating developers, creating better construction methods, and will also contribute to the development new construction frameworks.

Exploring better design methods is also important from the economic perspective of maintaining the software. In the case of request integrity attacks, the solution is essentially a change in the design of the request-response behavior of the application. Applying such design fixes for vulnerabilities after a software product is delivered is prohibitively costly [46]. This is because such changes require massive amount of regression testing for validating the correctness of the software and its impact on othe functionalities. Therefore, proactively designing application to avoid classes of integrity will reduce the cost of software maintenance.

There are existing defense mechanisms against request integrity attacks (e.g., [7]), but developers do not have any systematic design methodology to identify where to apply a countermeasure. The absence of a design methodology creates several problems. First, developers have to locate the places in the source code to apply the techniques. This process is both cumbersome and arbitrary, if done manually. Second, because these additions are not driven by systematic design, the solution is often either incomplete or incorrect. Therefore, we need a systematic design methodology that will guide the developers better to fix the vulnerabilities.

This chapter presents a methodology for designing web applications that are secure from request integrity attacks by construction. The methodology consists of two steps. The first step uses a deterministic finite state automaton, the Web DFA, to abstractly model the intended user-application interactions or request behavior of a web application. The second step is to augment the Web DFA using four design patterns. The augmented Web DFA produces a model that drives the implementation to strictly enforce the intended request-response behavior.

The chapter also describes the experience of using the methodology for both building a new application, and analyzing an existing application. A comparison of the Web DFA models of these application reveals why the Web DFA model successfully enforces the intended request-response behavior, but an arbitrary design methodology might fail to do so. The contributions discussed in this chapter can be summarized as follow:

1. A systematic methodology for designing web applications that strictly enforce the intended request-response behavior.

2. Four design patterns that prevent web request forgery attacks.

3. A case study that illustrates the experience of using this methodology to design and implement a web application. Also, the case study describes how the methodology can be used for analyzing an existing web application to uncover its vulnerabilities.

Organization. The remainder of the chapter is organized as follows. Section 5.1 describes the Web DFA model, and creating this model is the first step in the methodology. Section 5.2 describes the four design patterns that augment the Web DFA model in the second step. Section 5.3 describes the case study that illustrates the use of the methodology. Section contains a summary of the chapter.

5.1 The Web DFA Model

A web application's request-response behavior can be modeled using a deterministic finite-state automaton (DFA), a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ that is defined as follows:

1. A finite set of states Q: Each state corresponds to a type of an application response. An application response is an HTTP response that contains a web page, which determines what further requests a user may issue. For example, if a user issues an HTTP request to the URL of the application, then he receives an HTTP response containing the home page, and the home page becomes the current state. The links and forms inside the home page determine what further requests a user may issue subsequently.

   In highly dynamic web applications, the contents of a web page may vary even though they belong to the same application response type. For example, the contents of a home page may be updated for each request. However, all these pages belong to the same application response type and characterize the same state, so they are equivalent.

2. A finite set of input symbols Σ: The input to the application is a type of application request. An application request is an HTTP request that targets an interface name. Each application is designed to accept only certain types of application requests, and the application will return an error for unrecognized request types. For example, an online-message board may accept requests for viewing, posting, or modifying messages in a message board,

and may return an error for requests that try to add a product to its shopping cart.

3. A transition function $\delta : Q \times \Sigma \rightarrow Q$: The transition function describes the responses that the application will return for valid requests in each state. A web application's implementation defines the transition function.

4. A start state $q_0 \in Q$: Typically, a user begins browsing an application from its home page. Therefore, an application response carrying the home page is the start state.

5. A set of accept states $F \subseteq Q$: The accept states correspond to states wherein the user or the application chooses to terminate the application session. For example, a user may choose to logout of the application, and the application response type for logging out would be an accept state.

This dissertation refers to this DFA model representing the application's request-response behavior as the Web DFA model. The proposed methodology advocates the use of the Web DFA model for designing web applications. Developers may use the Web DFA model as a tool for analyzing the request-response behavior of the application.

The Web DFA model does not focus on reflecting actual browser interactions. For example, a user may be visiting several pages of the web application in separate browser windows. Alternatively, the web application may display several pages from the application inside a web page using HTML *frames*. Moreover, the user may also use back button for going back to a previously visited web page. These client-side interactions do not affect the way in which an application would process an incoming application request. Therefore, these interactions are outside the scope of the Web DFA.

The remainder of this section describes the process of creating and using a Web DFA model using the example of the online shopping-cart application. There are two step in the process. First, developers should create a Web DFA model based on the intended application request-response behavior. Second, developers may classify various states in the DFA based on its impact to the application. For expository purpose, we use a running example, an online shopping cart application, to explain the methodology.

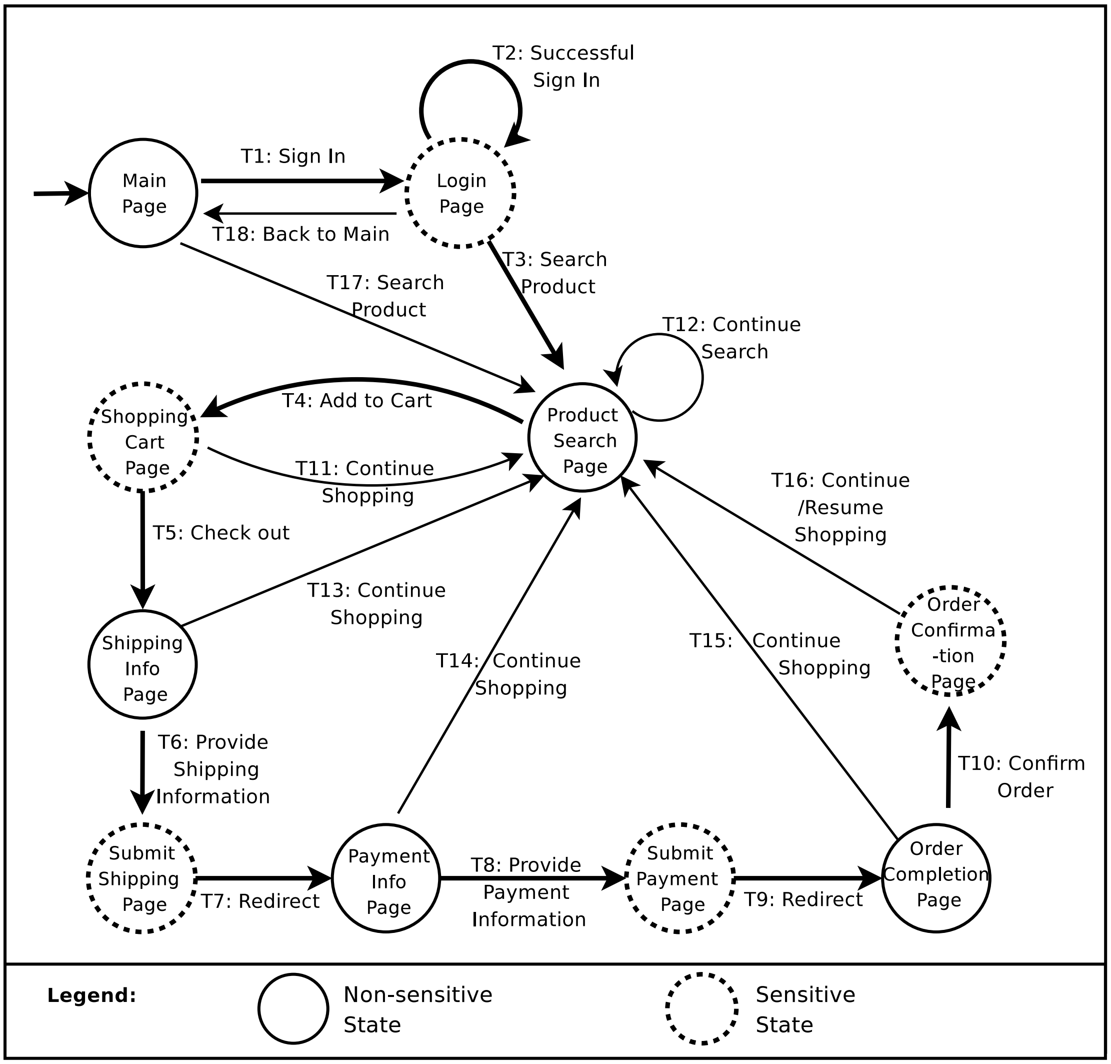


Figure 5.1: A Web DFA model for an online shopping cart application.

Creating the Web DFA. The Web DFA model for the shopping cart application has 10 states and its transition function defines 18 transitions. Figure 5.1 depicts the Web DFA model. The simplest interaction model of purchasing a single product is represented by transitions T1 through T10 (solid line transitions in figure 5.1) . In this interaction model, a user comes to the home page, goes to sign in (T1), successfully completes authentication (T2), searches for products (T3), adds a product to cart (T4), continues to checkout (T5), confirms shipping and payment information (T6-T9), and completes order (T10). Transitions T5–T10 form the transitions that are part of the checkout process.

Transitions T11–T15 illustrates various ways in which a user may resume shopping during the checkout process. The user may do this by choosing to go the product search page (T11, T13, T14, T15, and T16), and may continue to add products to his shopping cart (T12). Similarly, at any point the user may decide to go to the main page, or they might decide to logout and be moved to the main page. These transitions (originating at various states and ending at the main page state) are omitted. These omissions are for illustration purposes; they do not affect the final outcome.

Identifying Vulnerable Request Classes. After creating the initial Web DFA, the states in the DFA are classified into two categories. There are two types of states, non-sensitive and sensitive, depending on whether transition to the state will have side effects. This categorization would be useful in choosing the right HTTP request types.

A state is **non-sensitive** if the transitions that lead to it do not have any side effects. An application does not modify the session data or database when processing these requests. A side effect free request does not modify an application's state irrespective of the number of times that it is issued.

A state is **sensitive** if a transition that leads to it has side effects, i.e., it modifies application state. A transition with side effects could affect a user or the correctness of an application, if it is forged by an attacker. A web application needs higher guarantees to ensure that a user knowingly performed the transition and is not tricked into doing it. Transitions from non-sensitive to sensitive states and between two sensitive states should be protected from forgeries.

In Figure 5.1, the Web DFA model also depicts a classication of the 10 states of the shopping cart application. Of the 10 states, five are sensitive and the other five are not sensitive. The *Login Page* state is sensitive because the application validates the users password and marks the session as belonging to the user. On successful validation (T2), the user is redirected to the Login page, which is essentially the main page that displays the name of the user in the top right-hand corner of the web page. An HTTP redirect is used by the sensitive states to trigger loading the next state containing the next form in the workflow. Of the 7 states performing the checkout transaction, from *Shopping Cart Page* through *Order Confirmation Page*, 4 are sensitive. The remaining 3 states present a form to a user whose information is then sent to the sensitive states. When a user decides to check out (T5), he/she first goes to the non-sensitive *Shipping Info Page*. In this state, the user is just presented a static web page to add shipping info; there is no change in the application state. Then the user provides shipping information (T6). The application takes the user to *Submit Shipping Page*. This state is sensitive since the application state is updated with the shipping information. The user, however, does not see a separate webpage. Instead, a script updates the application state and redirects (T7) the user to the *Payment Info Page*.

In practice, a single server-side script could be used to implement the processing relating to several states. For example, we can implement a single server-side script for the entire checkout operation. Based on the incoming request type, the same script could either display a form for the user to fill out or process the data submitted in the request and redirect the user to the next step. However, this implementation detail does not merge the 7 checkout operation nodes into a single node. They are separate nodes in the Web DFA from the perspective of request-response behavior, different functions, and security requirements.

5.2 Design Principles

This section describes four design principles for enhancing the Web DFA models with additional specifications for strictly enforcing the intended request-response behavior. The four design principles are described in the form of design patterns. A design pattern is a general solution to a commonly recurring problem in software design. In the context of the proposed methodology, all these four patterns are general solutions for enforcing intended request-response behavior in web applications. Because developers are familiar with design patterns, it will be easy to communicate the ideas to them using the pattern format. The security patterns repository also presents several general security design principles in the form of design patterns.

These design patterns could be applied without the Web DFA model, but tying the design patterns with Web DFAs makes the design process systematic, complete, and less cumbersome. In this section, we will describe each of the patterns and illustrate how the patterns are applied using the running example of online shopping-cart application. Table 5.1 summarizes the patterns. The first three patterns protect a web application from CSRF attacks, while the fourth one protects from workflow attacks.

Pattern	Summary
Non-sensitive GET/ Sensitive POST	Choose the correct type for an HTTP request.
Secret-token Validation	Use a secret token whenever a sensitive request is made to distinguish between genuine and forged requests.
Intent Verification	Add an additional verification step to a request to verify whether a user intends to issue the request.
Guarded Workflow	Check preconditions and postconditions for each transition.

Table 5.1: Design patterns to prevent web request forgery attacks

Design patterns are usually follow a standard documentation format. This section uses a format that is dervided from Gamma et al. [36] and the security patterns repository [62]. Each pattern has the following sections:

1. Intent: This section describes the goal of using the pattern.

2. Forces: This section describes the motivations for using the pattern, and also the context in which the pattern can be used.

3. Solution: This section describes the solution and also describes how the solution resolves the various forces described in the Forces section.

4. Example: This section illustrates the application of the pattern to the online shopping-cart application.

5. Consequences: This section describes the results of applying the pattern, side effects if any, and the trade offs caused by using the pattern.

6. Known uses: This section provides examples of real applications of the pattern.

5.2.1 Non-sensitive GET/ Sensitive POST

Intent

HTTP is the cornerstone of the World Wide Web. HTTP/1.1 [32] defines eight request methods, each with its explicit recommended usage. HTTP methods for reading and updating content follow the Create, Read, Update and Delete (CRUD) [70] model of relation databases: PUT is used to create, GET to read content, POST to update content, and DELETE to delete content from a URL. GET and POST are more common in web applications, while PUT and DELETE are seldom used.

Despite the explicit specification of method roles, HTTP methods are often misused in web applications [3, 64]. For example, a developer who is considering whether to use HTTP GET, should follow these guidelines:

1. GET should be used when a request does not affect application state. The HTTP protocol defines GET as *safe* and *idempotent*: an HTTP GET request should not have any effect on an application's state and the effect of multiple requests should be identical to that of a single request [32].

2. For making sensitive requests, POST is favored over GET. It is harder for an attacker to forge a POST request, but GET requests are easily forged. This is because GET requests can be issued by putting URLs in the attribute header of many HTML tags (e.g., `img`, `iframe`, etc). When a user visits the page, GET requests are initiated without the user noticing them. On the other hand, forging a POST request requires either user interaction or JavaScript. To forge a POST request, an attacker has to coerce a user to submit a form. Alternatively, JavaScript programs can submit the form, but security setting in browsers prohibits untrusted JavaScript programs.

In practice, GET is often mistakenly used for making sensitive requests and modifying application state [15]: `Bloglines` sync API uses GET request to mark unread items as read, `Flickr` API previously used GET to delete a photo set, `del.icio.us` API uses GET to delete a post from the site, etc are some examples. Implementing a GET request in a web application is typically easier and results in less code than a POST request, possibly explaining their improper use.

Web application developers arbitrarily choose request methods, instead of considering which one is the most appropriate. The request type is treated as an implementation detail. Since the factors that influence the choice, such as whether the request is expected to have side effects or not, are known during design, it is best to determine the appropriate request type during design.

Forces

The following forces should be considered when choosing to use this pattern.

1. Web applications should choose the most appropriate HTTP request type for each request.

2. Choosing the wrong request type would facilitate request forgery.

3. Choosing the most appropriate request type is best done during design.

Solution

Identify the type of processing and side effects associated with each request during the design phase and use this information to choose the appropriate HTTP request method. Strictly use POST for any request of a sensitive state, i.e., it modifies database or a web application's session data. Use GET for non-sensitive requests that do no have side effects.

There are certain requests that may have side effects, but they may still be considered side-effect free. For example, a request to visit the index page of a web site may automatically update page visit statistics. However, these statistics may not be considered as part of the application state. Therefore, such requests may still be considered non-sensitive and implemented as GET requests.

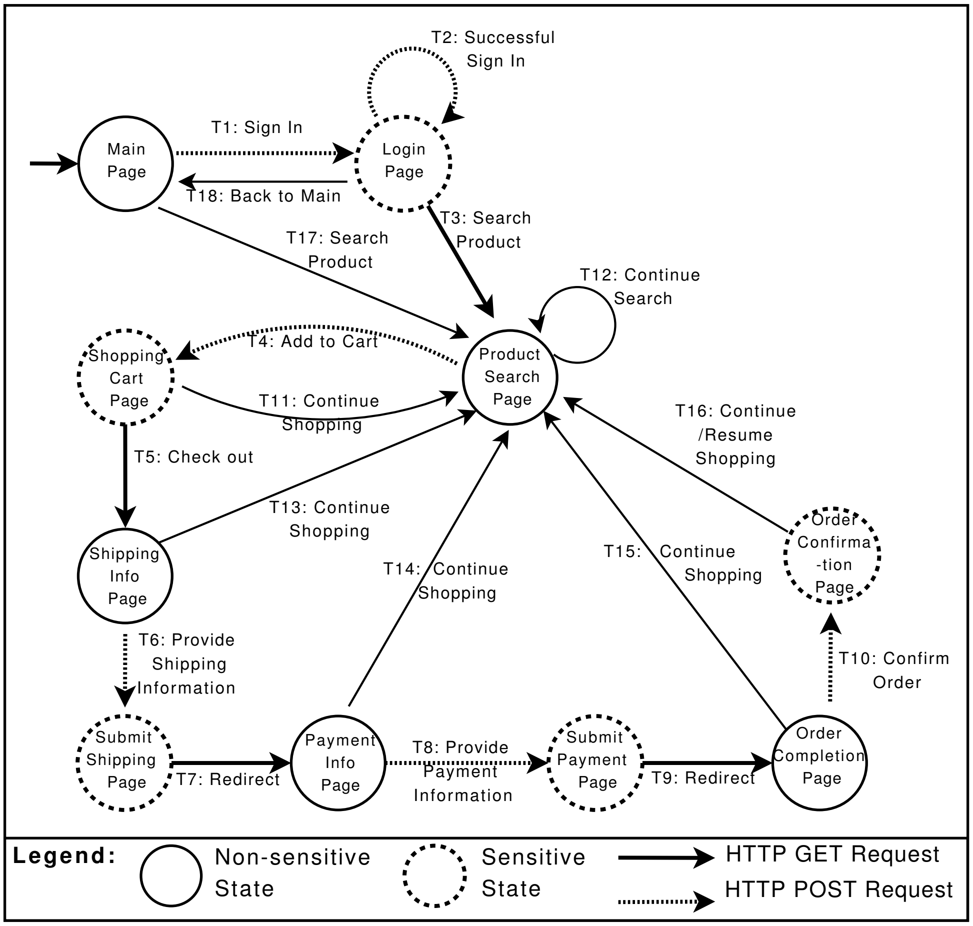


Figure 5.2: Web DFA for our shopping application with appropriate request type

Example

This section describes how the pattern is applied to augment the Web DFA model in figure 5.1. Figure 5.2 shows the modified Web DFA. States such as the *Login page*, *shopping cart page*, *submit shipping page*, *submit payment page*, and *order* *confirmation page* are non sensitive. Therefore, transitions T1, T2, T4, T6, T8, and T10 must be designed as HTTP POST requests. States such as the *Login page*, *shopping cart page*, *shipping info page*, *payment info page*, and *order completion page* are non sensitive. Therefore, transitions T3, T5, T7, T9, and T11-T18 can be implemented as HTTP GET requests.

Consequences

Applying the pattern has the following consequences.

- *Easy to understand*. Applying this pattern would make a web application easy to understand. Each request becomes intention-revealing; its type gives a hint of the operation to be invoked.

- *Weak Defense*. Attackers can forge POST requests [83]. This pattern provides the first layer of defense; more mechanisms are necessary following the defense in depth [93] principle.

- *Increased complexity*. Applying this pattern could make a web application more complex. The simplest option for implementation is to use one HTTP request for all purposes; GET is the most suitable candidate. Having more than one HTTP request type would add more complexity; however, this is essential complexity [17] to make a web application secure.

Known Uses

`phpBB` and `punBB` are multi-user message board applications. `osCommerce` is an online shopping cart application. In all three web applications, HTTP GET requests are used only for non-sensitive requests, and POST is used otherwise.

5.2.2 Secret Token Validation

Intent

Strictly using POST to make sensitive requests provides a weak defense against request forgery. For example, let us consider the case of a CSRF attack. In this attack, recall that the objective of a malicious web site A is to forge a valid request for the trusted web site B. If the web site B uses HTTP POST requests for transitions to sensitive states, then it would be harder for the attacker to forge those requests. In this case, the only way for the attacker is to use a HTML *Form* tag with the correct parameters inside his web page, and trick the user into either submitting the form or use a JavaScript program to automatically submit the form.Therefore, replacing GET with POST makes it harder for an attacker to launch an attack, but it is not hard enough to prevent the attack.

The underlying problem that enables cross-site-request forgery is that a vulnerable web request can be repeatedly made. Typically, applications store session cookies in a web browser to customize each user's request, but session cookies are attached whenever a browser makes a request. Session cookies are static; the same cookie is presented for all requests made from a user. Hence, an application has no way of distinguishing a legitimate request from a request that a user has unsuspectingly made on behalf of an attacker.

A user and a web application should have a secret that an attacker cannot know. If the secret is part of a web request, an attacker cannot forge it.

Forces

The following forces should be considered when choosing to use this pattern.

- HTTP requests can be repeatedly made.

- Session cookies are used to customize each user's request, but they provide an insufficient mechanism to prevent forgery. This is because, for all requests to a domain, a browser automatically attaches that domain's cookie.

- Cryptographic mechanisms could be used to create a unique token between an application and a user; an attacker cannot guess the token.

Solution

Use a secret token whenever a sensitive request is made. Protect the secret token, so that an attacker can not know it. Verify each incoming request for a sensitive action to check that the secret token is present and correct.

In secret token validation, all HTML `form` tags that create HTTP requests include a random value as a hidden input field. This random value is passed to the server, and the server processes a request only after validating it. An attacker cannot access this random value since, 1) the value is available only in the web page given to the user, and 2) the security policy in web browsers prohibits the value to be shared.

Example

Consider the Web DFA model of the online shopping application in figure 5.2. All transitions to sensitive states are attractive targets for request forgeries. The processing of these requests should additionally incorporate a secret-token validation technique.

Consequences

Applying the pattern has the following consequences.

- *Strong Defense.* Secret tokens offer very strong protection with minimal computational overhead.

- *Need for Protecting Secret*. The session secret should be protected from attackers. One-time-use token values per form can be used, but they increase complexity and overhead.

Known Uses

This pattern is widely used for preventing cross-site request forgery attacks. `phpBB`, a message board application, adds a session identifier additionally as a hidden field to all web forms. The server-side scripts validate requests based on the session identifier. `phpMyAdmin` is a web application used for remotely administering MySql database. `phpMyAdmin` associates a random token for each session and adds the random token as a hidden field in forms.

5.2.3 Intent Verification

Intent

CSRF is a form of confused deputy attack [45]. The victim user, whose browser is making the request, does not know that he/she is being attacked. The user is tricked into submitting a request on behalf of the attacker. If a user is always asked before his/her browser sends a request, the user knows when he/she is about to be tricked by an attacker. Consequently, there will be no CSRF attacks. However, asking for consent at every step is impractical as users will find it annoying. There should be a lightweight approach that ensures usability of the application while assuring the integrity of each submitted request.

Many web applications use long expiration values for their browser cookies to keep a user continuously signed in. The cookies are used to track a session as well as to keep a user logged in so that users revisiting a site will not need to re-login. Applications which use long expiration values for their session cookies are highly vulnerable to CSRF.

Forces

The following forces should be considered when choosing to use this pattern.

- Users do not know when they are tricked by an attacker into a CSRF attack.
- Web applications should verify the intent of each submitted request.
- The intent verification reduces the usability of the application.

Solution

Inside a web application, certain sequences of requests together may form a transaction, whose security from request forgery is important for preserving the application's integrity. For example, in the shopping cart application, there are seven states that participate in the checkout operation. A web application can introduce an additional verification step in the beginning of each such transactions. The first request in the transaction can be considered as the stepping stone in the transaction. A web application should identify all such stepping stones, and use an additional verification step in processing those requests for making sure that the user is consciously initiating the transaction. The application need not verify the intent of the user for the subsequent requests in the transaction. As a result, the application could balance the security and usability of the application.

There are two methods for verifying the intent of the user. First, the application may employ an additional authentication step. In each of the stepping stones, the application may present the user with a login form and ask the user to supply his username and password, and the request processing will proceed further if and only if the password is validated successfully. Second, an application may use a CAPTCHA [96], which is an image challenge designed to distinguish between a human being and a computer. The user is presented with picture of word, whose characters are distorted, and will be asked to type the word in the picture. Human being innately excel in these image recognition tasks, but image processing algorithms have not advanced adequately. Therefore, a human being can easily recognize the word in such challenges, but a computer program may not be able to accurately recognize the word. By using a CAPTCHA, a web application may be able to distinguish a genuine user initiating the transaction from a computer program automatically submitting the transaction.

Example

In the Web DFA model (see figure 5.2) of our running example, the stepping stone to the checkout transaction is transition T4: adding a product to a cart. The web application should verify the intent of the user on this transition.

However, intent verification at T4 will hinder usability. A user, who is adding a lot of products to the shopping cart (following the T4–T11–T12–T4 cycle), has to verify for every product added (T4). Clearly, this is annoying. A better way is to check on transition T5 instead, when the products have been added to the cart and the user is opting for checkout. Real online shopping applications, such as `www.amazon.com`, verify a user at this step.

Consequences

Applying the pattern has the following consequences.

- *Informed User*. A victim user is informed when he is unsuspectingly initiating a sensitive request on behalf of an attacker.

- *Better detection of bots*. As a side effect of applying the pattern, web applications may distinguish Internet bots from real users.

- *Hindered Usability*. The verification step might be annoying for a user legitimately using the application.

Known Uses

Several web applications employing long login timeouts verify the user intent at the stepping stones of transactions. Both `www.ebay.com` and `www.amazon.com` allow users to search for products and add them to the shopping cart using long-term login. However, when a user tries to initiate a checkout transaction, the web application requests for a username and password. The checkout transaction is initiated only after correctly executing the verification step.

5.2.4 Guarded Workflow

Intent

A workflow is essentially one compound task composed of subtasks that have to be executed in a particular sequence. Each subtask expects its caller to meet some preconditions. In a web application, the preconditions are constraints on session variables or the application's contents in a database.

If a subtask does not strictly check that its preconditions have been met, an attacker can violate the conditions and invoke the task nevertheless. Workflow attacks attempt to create an unintended interaction, in which certain subtasks are skipped by an attacker.

Forces

The following forces should be considered when choosing to use this pattern.

- Subtasks in a workflow should be executed in a pre-defined order.

- Attackers want to manipulate the normal execution order.

- Subtasks have preconditions that a caller should satisfy before invocation.

Solution

Identify the preconditions for each subtask in a workflow during design. During implementation, add checks to verify that all the preconditions are satisfied when a caller calls a subtask, otherwise identify it as a workflow violation.

Each of the subtasks have a set of preconditions. After invocation, each subtask creates a set of postconditions, which becomes the set of preconditions for the next subtask in the sequence. The precondition of any subtask is the union of postconditions of all the preceding subtasks. For each $subtask_n$ that should strictly follow a sequence of subtasks $\{subtask_1, subtask_2, ..., subtask_{n-1}\}$,

$$postconditions_1 \cup postconditions_2 \cup .... \cup postconditions_{n-1} \subset preconditions_n$$

The design specification should outline an exception handling procedure for failing preconditions. The exception handler may either direct the caller to execute a preceding subtask or terminate the transaction.

Example

Consider the checkout transaction in the Web DFA model of figure 5.2. The transaction comprises of four steps: opting for check out (T5), submitting payment (T6), submitting shipping (T8) and confirming order (T10). Each transition has preconditions and postconditions (figure 5.3).

The postconditions in each transition are chained so that they become the preconditions of the subsequent transition. As a result, there is no way an attacker can skip intermediate steps in the checkout transaction.

Exception handling procedures can also be described for workflow violations. For example, if the pre conditions associated with *T8: Provide Payment Information*

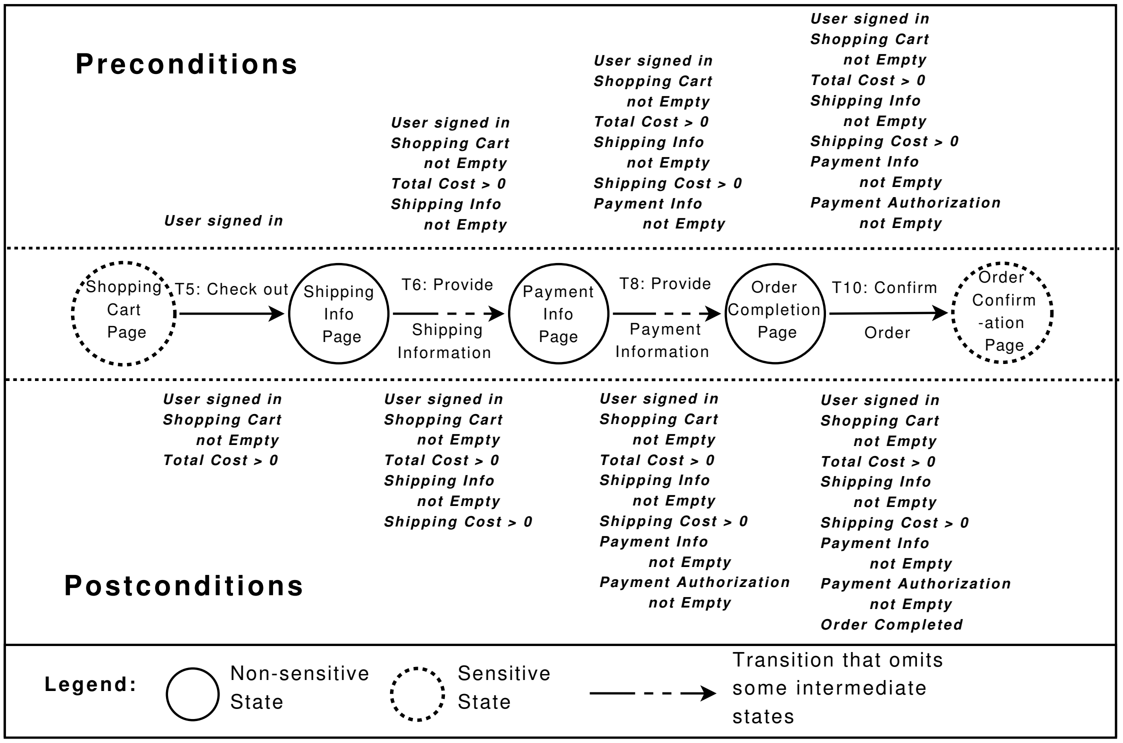


Figure 5.3: A checkout workflow annotated with required preconditions and postconditions.

are not satisfied when processing *T10: Confirm Order*, the application may direct the user to a web page to provide the payment information.

Consequences

Applying the pattern has the following consequences.

- *Design by Contract*. Each of the preconditions and postconditions are determined carefully during design. The implementation that follows checks the conditions. Hence, the chance of a workflow violation is minimized.

- *Hard to Determine Preconditions*. In practice, determining the appropriate preconditions might not be straightforward. There might still be workflow vulnerabilities after applying this pattern. However, careful design nearly eliminates the vulnerability.

Known Uses

Directed Session pattern [61] uses a different approach. An application using *Directed Session* exposes a single URL. All webpages are accessed via this single URL. A server, using session data, determines which page be serve to the client. This dynamic approach, however, does not support the functionality of a back button in a browser. The *Guarded Workflow* pattern combined with the Web DFA is a more systematic way of exploring preconditions; it also supports the back button of a browser.

Design by contract [71] is a software engineering theory that describes formal contracts among software entities. A contract is the set of preconditions that a caller must guarantee before calling a module and the set of postconditions that would hold after the call. This pattern is essentially an application of design by contract for modeling workflow transactions in web applications.

5.2.5 Defense in Depth

The catalog of four design patterns is an ideal example of defense in depth [93]. First, the Web DFA is augmented by applying *Non-sensitive GET/ Sensitive POST* pattern. It determines sensitive states where POST requests should be used. But even POST requests could be forged. Therefore, *Secret Token Validation* mechanism is added with each POST request. Another line of defense is to keep a user informed about his/her actions. Hence, *Intent Verification* pattern is used to introduce verification mechanism in the transitions between Web DFA states where the user is stepping from a non-sensitive action to the start of a sensitive transaction. Finally, *Guarded Workflow* pattern is applied to the Web DFA to enforce design by contract [71]. Together, these patterns create multiple layers of defense that successfully prevent request integrity attacks.

5.3 Case Study

This section describes a case study for illustrating the use of the proposed methodology. Section 5.3.1 describes how the proposed methodology was used in the design and implementation of INFBB, a bulletin-board application that is safe from web request integrity attacks by construction (section 5.3.1). We describe how our methodology guided us to pro-actively insert appropriate checks into INFBB during design. Section 5.3.2 describes the experience of using the using the Web DFA for modeling and understanding two existing open source web applications. The Web DFA model helps in identifying both the design flaws in these applications that lead to the vulnerabilities and also how they could be redesigned to fix the vulnerabilities.

5.3.1 Building INFBB

This section describes the design and implementation of INFBB using the Web DFA model. INFBB is a bulletin board system that supports infinite topic depth. It supports two types of users: non-paying users, and paying premium users. A premium user can use premium credits to post premium messages, i.e., messages that appear highlighted and on the top of a page. We have captured the developer-intended interactions for INFBB using a Web DFA model, and applied the four patterns to augment the model.

Web DFA model for INFBB**:** Figure 5.4 shows the Web DFA model for INFBB. It has 12 states and 22 transitions (only transitions relevant to this discussion are marked). Of the 12 states, 3 are non-sensitive, and 9 are sensitive. A user starts at *Content Page*. From there, he/she can add content, modify information, or add premium credits.

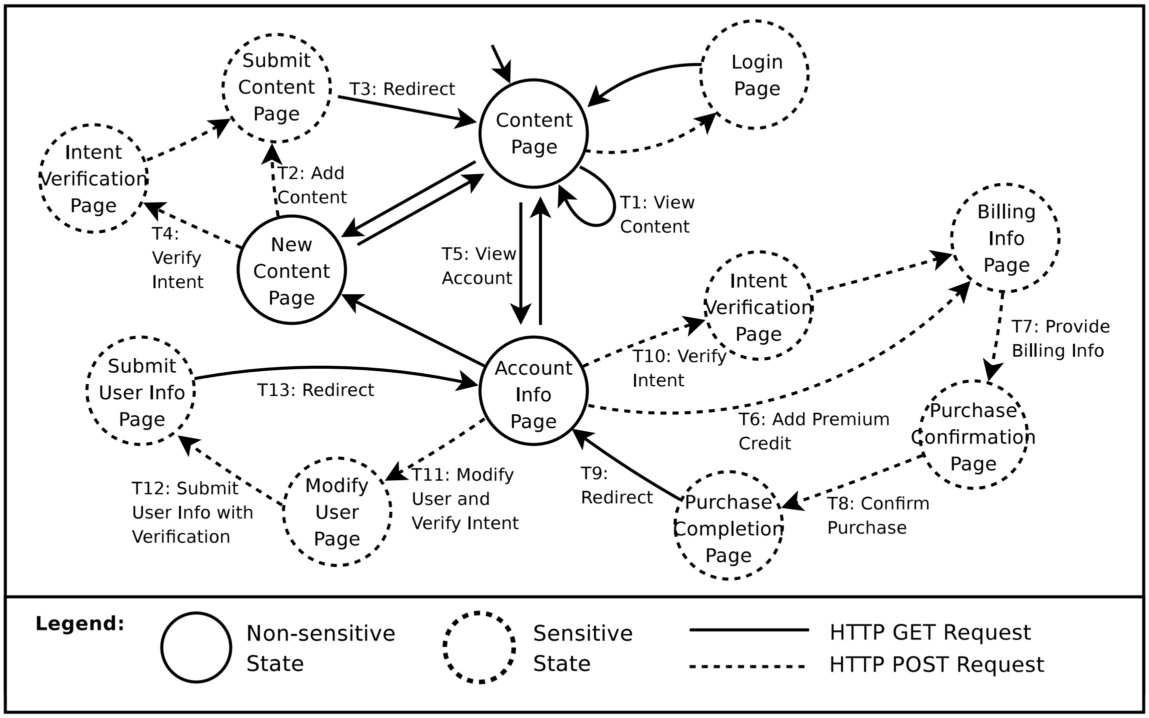


Figure 5.4: INFBB DFA labeled with sensitive/ non-sensitive states and proper request type

1. *Add New Content*. A user goes to *New Content Page*, which is a static web form. When the user adds content (T2), he is redirected (T3) back to *Content Page*.

2. *Modify Account*. A user goes to *Modify User Page* in order to modify account information (T11). Upon modification (T12), the user is redirected (T13) to *Account Info Page*.

3. *Add Premium Credits to Account*. A user may add message credits to his/her account. The user goes (T6) to a *Billing Info Page*, provides billing information (T7), then confirms purchase (T8). Finally, the is redirected to *Account Info Page*.

```
1  <?php
2  $key = getKey('account');
3  ?>
4
5  <h2>Please fill in the billing address of the credit card.</h2>
6  <form action="confirm_purchase.php" method="post">
7  <input type="hidden" name="key" value="<?php echo $key; ?>">
8  Street:<input type="text" name="street"><br>
9  Phone:<input type="text" name="phone"><br>
10 <input type="submit" value="Submit">
11 </form>
```

Figure 5.5: Implementing secret-token validation

Applying the patterns: The four patterns are applied to the model to add multiple layers of protection. Following the *Non-sensitive GET/ Sensitive POST* pattern, all transitions to sensitive states are designed as HTTP POST requests (see figure 5.4). In addition, the scripts responsible for processing requests for transitioning to sensitive states use the *Secret Token Validation* mechanism. Figure 5.5 shows an excerpt from a server-side script that prepares the *purchase confirmation page*. In line 2, the script creates a secret token by using the function *getKey* with a parameter "account". *getKey* creates the secret token, which is a pseudo-random number, and stores both the values "account" and the secret token as a key-value pair in the application's session. In line 7, the script creates a hidden-input field that carries the secret token. When this form is submitted in a subsequent request, the application will compare the incoming secret-token in the hidden field with the one stored in the session as a value for the key "account".

The Web DFA model shows the states where the *Intent Verification* pattern is applied. The intent verification step adds new states to the Web DFA model; these are denoted by two *Intent Verification Page* states. When a user is adding new content, or purchasing premium credit, he/she is verified (T4 and T10 correspondingly). The web application dynamically inserts intent verification step into the work-

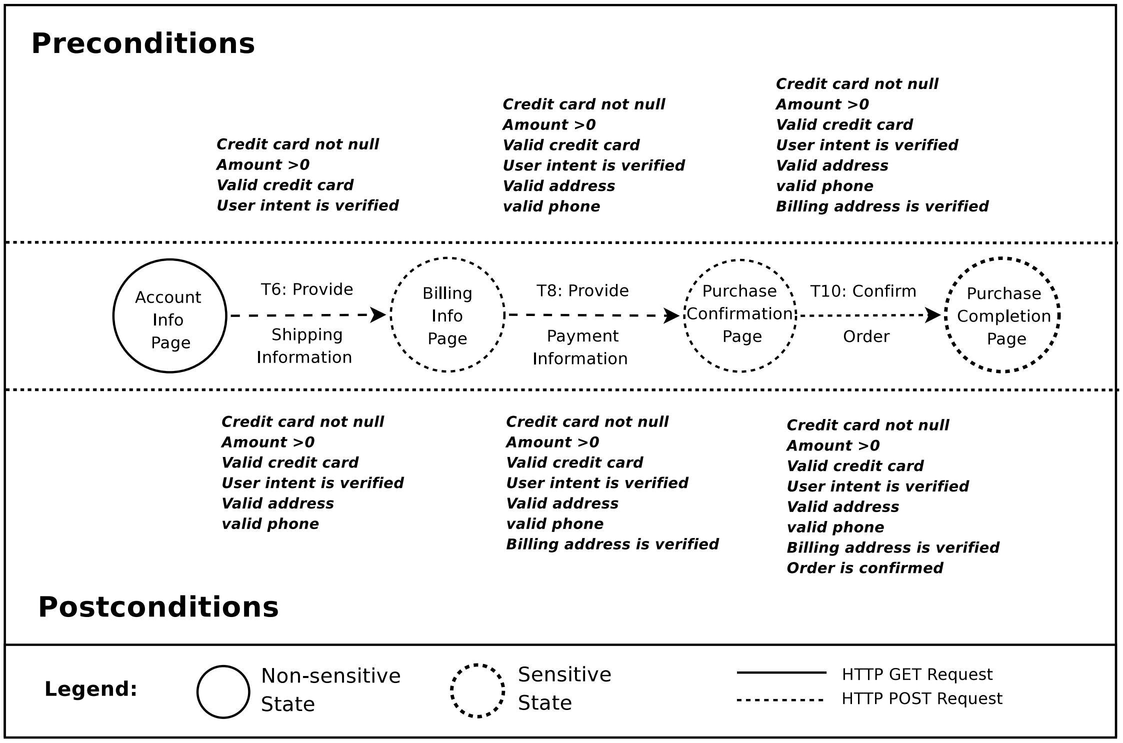


Figure 5.6: INFBB workflow annotated with preconditions and postconditions

flows based on the length of time the session has remained idle. The threshold for the length of time is designed to be configurable by an administrator. Therefore, the intent verification does not significantly degrade the usability. On the other hand, when a user is modifying account information, the intent verification step is mandatory (T11). In INFBB, it is part of the *Modify User Page* state. The user verifies intent whenever he/she is submitting modified account information (T12). This design decision is to ensure the following policy: any changes to account information would require higher guarantees of conscious user involvement.

All the workflows in the application employ the *Guarded Workflow* pattern to safeguard the integrity and correctness of the transaction at each step. This was implemented using a set of functions to check session variables representing preconditions in the workflow. When a form is submitted the values are sanitized and stored in session variables, followed by the checking of all preconditions necessary. If any precondition fails an error is thrown and the user is returned to the last valid state. Figure 5.6 depicts INFBB's premium-credit purchasing workflow that is annotated with pre and post conditions.

As a consequence of modeling the intended interactions using the Web DFA and applying the four design patterns, INFBB is sufficiently hardened against request integrity attacks by design.

5.3.2 Modeling Legacy Web Applications

This section describes the utility of Web DFA model for analyzing legacy application using `punBB`, a fast and lightweight PHP-powered discussion board, and `SCARF`, a conference management application. The Web DFA model reveals an undocumented CSRF vulnerability in `punBB` and how it could be redesigned to remove the vulnerability. `SCARF` has a well known workflow vulnerability [18]. Our Web DFA model for `SCARF` identifies the vulnerability and suggests how it could be redesigned.

Modeling `punBB`: The Web DFA modeled for *punBB* [79] has 28 states and 340 transitions. Our model reveals that `punBB` versions 1.2.11 and earlier is vulnerable to an undocumented CSRF vulnerability that allows a malicious web site to post arbitrary topics and messages through a victim user's account.

Figure 5.7 outlines the attack with the relevant portion of `punBB` Web DFA model. A user in the *viewform* or *viewtopic* state is capable of posting new topics or messages in the forum. After a successful post, the user is taken to the *post* state. The Web DFA model shows that the transition to the sensitive *post* state is correctly implemented as an HTTP POST (*Non-sensitive GET/Sensitive POST* pattern). However, `punBB` developers do not add another layer of defense by implementing the

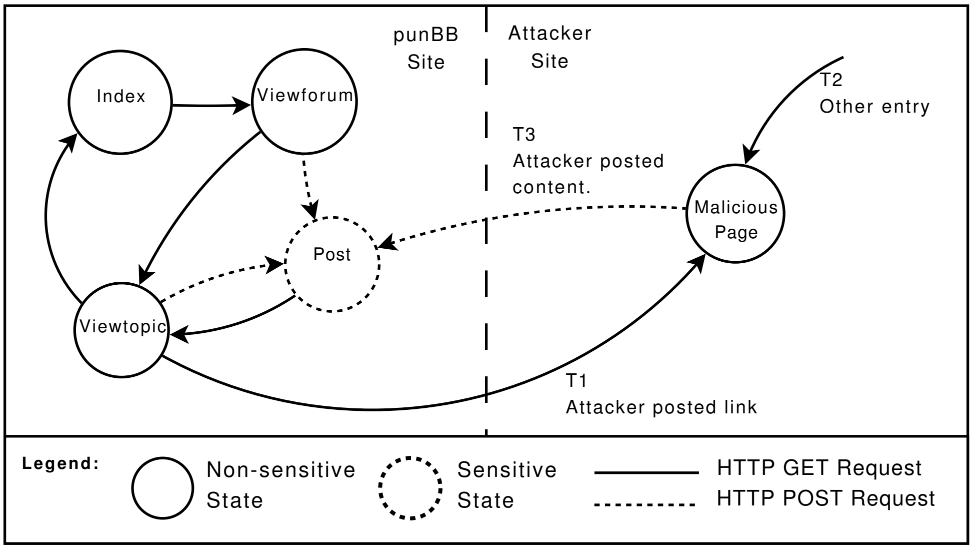


Figure 5.7: Possible CSRF attack on punBB bulletin board system

Secret Token Validation pattern. This makes punBB vulnerable to CSRF. Suppose, a victim user visits a *malicious page* hosted by an attacker while having an ongoing session in punBB. The victim user might be led to the *malicious page* by following some link from a punBB state (T1), or from some other web page (T2). The attacker can trick the user into posting arbitrary content (T3), and reach the sensitive *post* state in punBB.

Modeling SCARF: The Web DFA model for SCARF has 17 states and 90 transitions. SCARF has a workflow vulnerability [18] that allows an attacker to bypass authentication. SCARF developers want to enforce that only users with administrative privileges can access administrative pages. When a user logs in as an administrator, the application shows a URL for administrative pages. But the application does not check the privilege of a user when he/she attempts to access a protected page. Thus, unauthenticated attackers can access protected pages, if they know the URL for those pages. This is an example of a trivial workflow vulnerability as a result of developers' failure to apply the *Guarded Workflow* pattern.

The Web DFA model for `SCARF` shows where to enforce these checks.

5.4 Summary

This chapter described a novel method for designing web applications. The method uses a formal methodology, based on finite state automaton, in conjunction with design patterns to model and enforce intended user-application interactions in web applications. This chapter also described a case study that illustrates the use of this methodology for constructing a new web application and analysing an existing web application to uncover its vulnerabilities.

Chapter 6

Conclusion and Future Work

We have inorganically increased the use of web applications to the point of using them for almost everything and making them an essential part of our everyday lives. However, methods for ensuring the security and integrity of these applications are still lagging behind. Moreover, several web applications have a large volume of users and attackers do not need significant resources to abuse web application vulnerabilities. As a result, web applications are highly attractive targets for exploitation.

The research work described in this dissertation is based on the thesis that we can improve the security and integrity of web applications by providing better protection models and appropriately designing web applications. In support of this thesis, this dissertation described the following contributions:

- The design and implementation of BAYAWAK, which is a tool based on a new approach for enforcing request integrity in web applications [53]. Publishers can describe the intended sequences of HTTP requests as a security policy, and BAYAWAK enforces this policy strictly and transparently without requiring changes in the application.

- The design and implementation of ESCUDO, a web browser protection model designed based on established principles of mandatory access control [52]. ESCUDO factors in the trustworthiness of web application principals when making access-control decisions. Publishers provide a security policy for the application's behavior inside the web browser, and ESCUDO strictly enforces the policy.

- A systematic methodology for designing web applications such that they strictly follow the intended application request-response patterns [54]. Developers design the intended request-response behavior using the WEB DFA model and apply four design patterns to produce a blueprint for the implementation.

These contributions can help in improving the security and integrity of web applications. However, several opportunities still remain to improve the security of web applications in fundamental ways. The remainder of this chapter describes future work that are related or based on the work of this dissertation.

6.1 Future Work

This section describes future directions in request integrity and web-browser access control, and also describes some complementing research directions.

6.1.1 Future Directions in Request Integrity

Future research should consider developing methods to facilitate the authoring of request integrity policies. Such methods would make the maintenance and deployment activities easier. In addition, enforcing request integrity for web applications deployed in the Cloud, AJAX applications, and web services are also important research directions.

Support for Authoring Request Integrity Policies

In the case of BAYAWAK, recall that an application's request integrity policy is its list of interface names, interfaces participating in workflows, and the name of its session cookie. The correctness of this policy determines the application's request integrity. The research work presented in Chapter 3 has used reverse engineering methods for obtaining the policies, and these methods vary depending on the application's complexity. There are straightforward methods for simple applications, but sophisticated program-analysis methods are needed for more complex applications. However, this reverse engineering method is not sustainable in the long run because the correctness of the methods depends on the application's complexity.

Future research should explore new systematic methods for authoring request integrity policies. In particular, we need methods that provide correct request integrity policies for an application irrespective of its complexity. Design of new web application construction frameworks that automatically provide these policies is a promising direction.

Enforcing Request Integrity in the Cloud

Deploying web applications in the cloud is becoming an attractive option with the availability of commercial cloud services. There are several advantages to deploying applications in the cloud such as reduced cost and latency, and better availability. Enforcing request integrity in these deployments is more challenging. A key issue is that web application servers may be dynamically provisioned or deprovisioned in the cloud depending on the traffic on the application. As a result, requests that belong to the same user's application session may not be handled by the same server. Therefore, all servers need to be aware of the session-specific interface identifiers (IID).

Future research should design scalable methods for enforcing request integrity in the cloud. One approach is to store the session-specific interface identifiers in a database. However, this approach is not scalable because of the latency of database access. An alternate method is to use mathematical one-way functions to derive the IID for an interface in a session/transaction. For example, the IID could be derived using a secret-key known to all servers, the interface name, and the session identifier as input to an one-way hash function. We need a similar method to appropriately derive the per-transaction IIDs for workflows. These and similar methods could be explored for enforcing request integrity in the cloud.

Request Integrity for AJAX and Web Services

In the case of BAYAWAK, the web pages that the application sends to the browser control what further requests the user may issue to the application. Moreover, because web pages have a standard syntax, BAYAWAK could intercept and modify web pages. AJAX-based applications and web services differ from typical web applications in both these aspects. Because of these differences, the request integrity enforcement method for typical web applications cannot be supplanted for AJAX applications and web services.

The application at the server-side has minimal control or no control over the client-side component for AJAX applications and Web Services. In AJAX applications, the client-side component is a single JavaScript program called the AJAX client. This AJAX client is initially downloaded to the browser as a result of the first HTTP request, and is responsible for interacting with the user, issuing subsequent requests, and rendering web pages in the browser. In web services, the client-side component is under the control of a different party. For example, a bank may host a web service to enable electronic bank account management (EBAM). A corporation who is the bank's customer may design a client-side program to interact with this web service.

AJAX applications and web services also vary in how the client-side component communicates with the application at the server side. Typical web applications send web pages to the client-side component. AJAX applications and web services send messages to the client-side component, which in turn updates the web pages based on the messages. These messages may follow an application-specific format in contrast to web pages, which have a standard syntax.

Future research should design new methods for enforcing request integrity for AJAX applications and web services in light of these differences.

6.1.2 Future Directions in Web-browser Access Control

Future research in browser access-control should consider how to facilitate richer web applications while enforcing the principle of least privilege. In the future, web applications will feature richer and more interactive clients executing in the web browser. Web browsers will need more architectural improvements to enforce access control in these richer applications.

Systematic Identification of Principals and Objects

We need architectural improvements in the web browser to systematically identify principals and objects within web applications. Existing web browsers do not have a concrete way for identifying web application principals and objects. In the case of ESCUDO, the web browser was modified to recognize each of the principals and objects described in Table 4.1. However, this strategy is not viable in the long term because it will require changes to the web browser each time a new principal or object is introduced. The client-side storage proposal of HTML5 is an example of a new object. Similarly, new web application principals and objects will continue to emerge. Therefore, we need a browser architecture that can recognize principals and objects without needing any code changes.

Inter-origin Application Interaction

In the future, we will see a proliferation of client-side mashup applications. These are applications which integrate several web applications from differing domains in a single web page. In effect, these applications integrate services and content in the client, which is the web browser. An example is an application that integrates with an online map service on the client-side to display the location of hotels in the neighborhood.

Future research should consider enhancements to the ESCUDO model to facilitate cross-origin interaction without compromising the security intents of the application. For example, one approach is to have each web application in a mashup specify a policy that determines how the rings of a different application maps to its rings.

Fine-grained Privilege Management

Future research should design methods and API to facilitate JavaScript programs to manage their privileges. Such methods can further enhance the enforcement of the principle of least privilege.

JavaScript programs can be empowered to voluntarily relinquish privileges when performing certain operations. For example, a JavaScript program in ring 0 may temporarily downgrade itself to ring 3 when performing operations that do not need the ring 0 privilege. The UNIX operating system provides the *seteuid* and *setuid* system calls to provide a similar functionality. Similar methods can be designed for JavaScript programs.

Similarly, JavaScript programs in higher-numbered rings can be empowered to access more trustworthy resources in a controlled manner. In the HPR model, there are special gates between rings that allow a principal in the outer ring to access resources in the next inner ring in a controlled manner. Similar gate mechanisms can be designed for ESCUDO.

Formal Analysis of Browser Access-control Policies

There are significant variations in the implementation access-control policies in web browsers. For example, the same-origin policy specification omits several corner cases [106], and this ambiguity has resulted in a varying implementation in each browser. In addition, we do not have a way to evaluate the security consequences of adding new additional features to the web browser. To avoid these problems, we need an unambiguous interpretation of browser access-control policies and the underlying trust assumptions. Future research should consider the use of formal logic such as the access-control logic described in Chin and Older [19] for this purpose. Such an analysis would facilitate creating a holistic view of browser access-control policies and also provide a way to understand the consequences of adding new and experimental features to the browser.

6.1.3 Complementing Research Directions

In addition to protection models, future research should consider completing approaches to further improve the enforcement of access-control in web applications. This section describes two complementing research directions.

Strongly-Typed Web Content

Web applications create web content with the intent that they will be processed by the browser using an appropriate handler and will be rendered in a particular way. For example, a web application would expect its web pages to be processed only by a HTML parser and that the web browser and application will have the same interpretation of the web page. Similarly, a web application would expect its cascading-style-sheet (CSS) to be processed only by a CSS parser and that both the web browser and the application will have the same interpretation of the CSS file. However, existing web browsers cannot guarantee both these goals.

A majority of web pages in the Internet have malformed HTML. Each browser has its own way of correcting and rendering malformed HTML. As a result, web applications and browsers may not have a consistent interpretation of the web pages. Also, when choosing a renderer for an incoming content such as a web page or a CSS file, web browsers do not require a strict match between the content and the renderer. For example, it is possible to trick a CSS parser into interpreting a web page [48]. Both these problems facilitate attacks such as cross-site scripting and cross-origin CSS attacks.

Future research should design methods to guarantee that both the application and the browser will share a consistent interpretation of the web content. By doing so, web applications and browsers can eliminate a large class of security problems. There are type systems that guarantee a strict separation of structure and content in web pages [84] at the server-side. These methods provide a good starting point for the exploration.

Formal Analysis Tools

In the future, web applications may use a combination of several policy enforcement mechanisms. For example, an application may use a server-side request integrity policy, a browser policy, and a policy for interacting with applications from different domains. Therefore, administrators will require sophisticated policy analysis tools for the following purposes:

1. To have a holistic end-end view of all the policies

2. Verify the policies for correctness

Future research should explore the design of such sophisticated tools. One approach is to use model checkers. Modern model checkers feature rich logical and declarative interfaces that could facilitate the representation and reasoning of web application policies.

BIBLIOGRAPHY

[1] Whitehat website security statistic report. http://img.en25.com/Web/WhiteHatSecurityInc/WPstats_fall10_10th.pdf, 2010.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[3] P. Adamczyk, M. Hafiz, and R. Johnson. Non-compliant and proud: A case study of HTTP compliance. Technical Report UIUCDCS-R-2008-2935, UIUC, Jan 2008.

[4] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, New York, NY, USA, 2010. ACM.

[5] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[6] A. Barth, C. Jackson, and I. Hickson. The HTTP Origin Header (Work in progress). http://tools.ietf.org/html/draft-abarth-origin-09, November 2010.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[8] A. Barth, C. Jackson, and C. Reis. The security architecture of chromium browser. http://crypto.stanford.edu/websec/chromium/.

[9] S. Basu. Proxy-annotated control flow graphs: Deterministic context-sensitive monitoring for intrusion detection. In *In ICDCIT: Proceedings of the International Conference on Distributed Computing and Internet Technology*, pages 353–362, 2004.

[10] D. E. Bell and L. J. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation, 1976.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[12] K. J. Biba. Integrity Considerations for Secure Computer Systems, April 1977.

[13] M. A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[14] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 23–43, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Blog Entry. Misunderstanding REST: A look at the bloglines, del.icio.us and flickr APIs.
http://www.25hoursaday.com/weblog/PermaLink.aspx?guid=
7a2f3df2-83f7-471b-bbe6
-2d8462060263, Apr 2005. Blog Entry.

[16] Bookstore. `http://www.gotocode.com/apps.asp?app_id=3&/`.

[17] F. P. Brooks Jr. No silver bullet Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[18] Bugtraq ID 20934. Stanford conference and research forum authentication bypass vulnerability. http://www.securityfocus.com/bid/20934, November 2006.

[19] S.-K. Chin and S. Older. *Access Control, Security, and Trust : A Logical Approach*. Chapman and Hall/CRC, 1 edition, July 2011.

[20] S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1:1–1:16, Berkeley, CA, USA, 2007. USENIX Association.

[21] S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1:1–1:16, Berkeley, CA, USA, 2007. USENIX Association.

[22] Classifieds. `http://www.gotocode.com/apps.asp?app_id=5&/`.

[23] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: an approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 63–86, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework

for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[25] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 350–364, Washington, DC, USA, 2006. IEEE Computer Society.

[26] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 99–108, New York, NY, USA, 2008. ACM.

[27] D. Crockford. ADSafe. `http://www.adsafe.org`.

[28] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 535–544, New York, NY, USA, 2008. ACM.

[29] Employee. `http://www.gotocode.com/apps.asp?app_id=6&/`.

[30] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–, Washington, DC, USA, 2000. IEEE Computer Society.

[31] Events. `http://www.gotocode.com/apps.asp?app_id=7&/`.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[33] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 418–430, Washington, DC, USA, 2008. IEEE Computer Society.

[34] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society.

[35] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[37] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association.

[38] R. Gopalakrishna, E. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *Security and Privacy, 2005 IEEE Symposium on*, pages 18 – 31, May 2005.

[39] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[40] J. Grossman. Cross-site scripting worms and viruses. The impending threat and the best defense. http://www.whitehatsec.com/downloads/WHXSSThreats.pdf.

[41] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 561–570, New York, NY, USA, 2009. ACM.

[42] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, NDSS, San Diego, CA, February 2009.

[43] W. G. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 285–296, New York, NY, USA, 2009. ACM.

[44] R. Hansen. XSS cheat sheet. http://ha.ckers.org/xss.html.

[45] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4):36–38, Oct. 1988.

[46] C. Henderson. *Building Scalable Web Sites*. O'Reilly, 2006.

[47] J. Howell, C. Jackson, H. J. Wang, and X. Fan. Mashupos: operating system abstractions for client mashups. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 16:1–16:7, Berkeley, CA, USA, 2007. USENIX Association.

[48] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from cross-origin css attacks. In *Proceedings of the 17th ACM conference on*

Computer and communications security, CCS '10, pages 619–629, New York, NY, USA, 2010. ACM.

[49] K. L. Ingham, A. Somayaji, J. Burge, and S. Forrest. Learning DFA representations of HTTP for protecting web applications. *Computer Networks*, 51(5), 2007.

[50] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 737–744, New York, NY, USA, 2006. ACM.

[51] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 611–620, New York, NY, USA, 2007. ACM.

[52] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 231–240, Washington, DC, USA, 2010. IEEE Computer Society.

[53] K. Jayaraman, G. Lewandowski, P. G. Talaga, and S. J. Chapin. Enforcing request integrity in web applications. In *Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy*, DBSec'10, pages 225–240, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] K. Jayaraman, P. G. Talaga, G. Lewandowski, S. J. Chapin, and M. Hafiz. Modeling user interactions for (fun and) profit: preventing request forgery attacks on web applications. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, PLoP '09, pages 16:1–16:9, New York, NY, USA, 2010. ACM.

[55] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM.

[56] M. Johns and J. Winter. RequestRodeo: Client-side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference by Piessens, F. (ed.), Report CW448*, 2006.

[57] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1 –10, September 2006.

[58] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 58–71, New York, NY, USA, 2007. ACM.

[59] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[60] F. Kerschbaum. Simple cross-site attack prevention. In *Proceedings of 3rd International ICST Conference on Security and Privacy in Communication Networks*, SecureComm '07, pages 464 –472, September 2007.

[61] D. M. Kienzle and M. C. Elder. Security Patterns Repository Version 1.0. `www.scrypt.net/~celer/securitypatterns/repository.pdf`.

[62] D. M. KIENZLE and M. C. ELDER. Final Technical Report: Security Patterns

for Web Application Development. `http://www.scrypt.net/~celer/securitypatterns/finalreport.pdf`, 2002.

[63] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 330–337, New York, NY, USA, 2006. ACM.

[64] B. Krishnamurthy and M. Arlitt. PRO-COW: Protocol compliance on the Web — A longitudinal study. In USENIX, editor, *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01), March 26–28, 2001, San Francisco, California, USA*, pages ??–??, pub-USENIX:adr, 2001. USENIX.

[65] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 251–261, New York, NY, USA, 2003. ACM.

[66] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8:18–24, January 1974.

[67] B. Livshits and U. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS '07, pages 95–104, New York, NY, USA, 2007. ACM.

[68] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[69] G. Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox Experience. `http://www.noscript.net/`.

[70] J. Martin. *Managing the Data Base Environment*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1983.

[71] B. Meyer. Applying ”design by contract”. *Computer*, 25(10):40–51, 1992.

[72] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP ’10, pages 481–496, Washington, DC, USA, 2010. IEEE Computer Society.

[73] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, NDSS, San Diego, CA, February 2009.

[74] A. Nguyen-tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[75] osCommerce. `http://www.oscommerce.com/`.

[76] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, WWW ’04, pages 224–233, New York, NY, USA, 2004. ACM.

[77] phpBB Group. phpbb. `http://www.phpbb.com/`.

[78] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th international conference on Recent advances in intrusion detection*, RAID’05, Berlin, Heidelberg, September 2005. Springer-Verlag.

[79] PunBB. Punbb. http://punbb.informer.com/.

[80] C. Reis. *Web Browsers as Operating Systems: Supporting Robust and Secure Web Programs*. PhD thesis, University of Washington, Seattle, WA, June 2009.

[81] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[82] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 219–232, New York, NY, USA, 2009. ACM.

[83] Robert Auger. The Cross-Site Request Forgery (CSRF/XSRF) FAQ. http://www.cgisecurity.com/csrf-faq.html#post.

[84] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association.

[85] J. Ruderman. The Same origin policy. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

[86] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5 – 19, Jan. 2003.

[87] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[88] SCARF. http://scarf.sourceforge.net/.

[89] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3), 1972.

[90] J. Solorzano. The Lobo Project. `http://lobobrowser.org/`.

[91] M. Ter Louw, P. Bisht, and V. Venkatakrishnan. Analysis of hypertext isolation techniques for XSS prevention. In *Web 2.0 Security and Privacy 2008*, May 2008.

[92] A. Vance. Times web ads show security breach. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`.

[93] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems The Right Way*. Addison-Wesley, 2002.

[94] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 173–186, New York, NY, USA, 2009. ACM.

[95] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[96] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Communications of the ACM*, 47(2):56–60, Feb. 2004.

[97] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–, Washington, DC, USA, 2001. IEEE Computer Society.

[98] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association.

[99] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 193–204, New York, NY, USA, 2004. ACM.

[100] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *Proceedings of the 25th IEEE International Conference on Software Maintenance 2009*, ICSM'09, pages 211–220, Edmonton, AB, September 2009.

[101] WebCalendar. `http://sourceforge.net/projects/webcalendar/`.

[102] J. Williams and D. Wichers. OWASP top 10 - 2010. Technical report, The open web application security project (OWASP), 2010.

[103] H. Xu and S. J. Chapin. Address-space layout randomization using code islands. *Journal of Computer Security*, 17:331–362, August 2009.

[104] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, RAID'04, pages 21–38. Springer, 2004.

[105] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th*

conference on USENIX security symposium, SSYM'06, pages 121–136, Berkeley, CA, USA, 2006. USENIX Association.

[106] M. Zalevski. Part1 - browsersec - Browser Security Handbook, part 2. `http://code.google.com/p/browsersec/wiki/Part2`, March 16, 2009.

VITA

KARTHICK JAYARAMAN

EDUCATION

- PhD Computer Engineering, July 2011. GPA : 3.96/4
  Syracuse University, Syracuse, NY, U.S.A
  Thesis: Protection models for web applications

- Master of Engineering, Aug 2002 - Aug 2004. CGPA : 9.27/10
  Thiagarajar College of Engineering, Chennai, Tamilnadu, India

- Bachelor of Engineering. Sep 1997 - May 2001. Grade: 80.56/100
  Amrita University, Coimbatore, Tamilnadu, India

EXPERIENCE

Aug 2005 - Present, Graduate Assistant, **Syracuse University**.

- **ESCUDO**: Modern web applications combine content from several sources with varying levels of trustworthiness, and cannot be protected by same-origin policy—the prevailing browser access-control model. I designed and implemented ESCUDO, a novel web browser access-control model designed based on hierarchical protection rings, an established mandatory access-control principal. Our results are promising. Web applications can enforce certain types of content restrictions and also avoid common XSS and CSRF attacks. Also, ESCUDO incurs low overhead and is backward compatible with same-origin policy. *Published in ICDCS 2010*

- **BAYAWAK**: Web applications are commonly vulnerable to request integrity attacks (RIA) (eg. CSRF and workflow attacks). BAYAWAK is a server-side method for enforcing request integrity in web applications and avoiding common RIA. I designed and implemented BAYAWAK. We evaluated BAYAWAK on several open source web applications, and our results are promising. BAYAWAK incurs low overhead and eliminated several RIA. *Published in DBSec 2010*

- **Partner Key Management - *Collaborative project with JP Morgan Chase***: Partner key management (PKM) is a new protocol for managing digital credentials for high value transactions in wholesale banking. PKM resolves liability issues appropriately, reduces cost, and most importantly is compatible with wholesale banking business practices unlike public key infrastructure (PKI). I co-designed PKM and PKM is likely to be deployed for use inside JP-Morgan Chase. *Published in MMM ACNS 2010*

- **MOHAWK**: Current research has proposed novel and expressive access-control frameworks, but we do not have sophisticated policy analysis tools that can scale with the size and complexity of access-control policies. I designed and implemented MOHAWK, which is a tool for detecting errors in access-control policies. MOHAWK uses an abstraction-refinement based technique in conjunction with a bounded model checker. MOHAWK scales very well with the size and complexity of the input policies and is orders of magnitude faster than competing tools. *Available as MIT Technical Report TR-2010-022 (Under submission)*

- **Teaching Experience**: As a teaching assistant, I helped teach several undergraduate, graduate, and PhD-level courses.

May 2007 - Aug 2007, Research Intern, **Microsoft**, Redmond, WA, USA.

My internship work focused on methods for detecting XSS attacks on web applications. I proposed a collaborative sever-browser method for distinguishing maliciously injected code from non-malicious code. Our approach uses instruction-set randomization, and additional methods for ensuring backward compatibility with browsers to enable a smoother transition. I implemented our approach for ASP.NET and Internet Explorer.

Aug 2004 - Jul 2005, Software Engineer, **Sasken**, Bangalore, India.

My work focused on maintenance of a software application called Loader, which is a crucial part of a network device called the base station transceiver (BTS) used in cell phone networks. An example is a major maintenance activity I led to detect and fix memory leaks caused by the loader application not freeing unused memory. The customer was **Nortel Networks**. My additions are part of the software's latest release.

Jun 2003 - Dec 2003, Research Intern, **Globarena Web Technologies**, Hyderabad, India.

My work focused on implementing copy-protection techniques for e-learning material distributed in CD-ROMs.

Aug 2001 - Jul 2002, Software Engineer, **Info Intellisys**, Chennai, India.

I implemented MIS reports in a merchandise management application deployed for a leading retail chain called **RPG's Food World** in India. The reports I implemented are currently used by the company.

TECHNICAL SKILLS AND INTERESTS

- Programming languages: C, C++, Perl, and Java.
- Web programming: HTML, PHP, Servlets, and JavaScript.
- Databases: MySQL, PostgreSQL

PUBLICATIONS

1. Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin,
   Automatic Error Finding for Access Control Policies (Under submission).
   MIT Technical Report TR-2010-022

2. Glenn Benson, Sean Croston, Shiu-Kai Chin, Karthick Jayaraman, and Susan Older,
   Interoperable Credentials for High Value Transactions, , *In Proceedings of 5th International Conference on Mathematical Methods, Models, and Architectures for Computer Networks Security (MMM-ACNS) 2010.*

3. Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve Chapin,
   ESCUDO: A Fine-Grained Protection Model for Web Browsers, *In Proceedings of 30th International Conference on Distributed Computing Systems (ICDCS 2010)*, Genoa, Italy, June 21-25, 2010.

4. Karthick Jayaraman, Gregg Lewandowski, Paul Talaga, and Steve Chapin,
   Enforcing Request Integrity in Web Applications, *In 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2010)*, Rome, Italy, June 21-23, 2010.

5. Karthick Jayaraman, Gregg Lewandowski, Paul Talaga, Steve Chapin, and Munawar Hafiz,
   Modeling User Interactions for Fun (and Profit): Preventing Web Request Forgery Attacks in Web Applications, *In Proceedings of 16th Conference on Pattern Languages of Programs 2009*, Chicago, Illinois, USA, August 28 - 30, 2009.

6. Karthick Jayaraman, David Harvison, Vijay Ganesh, Adam Kiezun,
   jFuzz: a concolic whitebox fuzzer for Java, In NASA Formal Methods Symposium (NFM) 2009, Moffett Field, California, USA, April 6-8, 2009.

7. Adam Kiezun, Philip J. Guo, Karthick Jayaraman, Michael D. Ernst,
   Automatic Creation of SQL Injection and Cross-site Scripting Attacks, In IEEE International Conference on Software Engineering (ICSE) 2009, (Vancouver, Canada), May 16-24, 2009.

AWARDS AND HONORS

- Outstanding teaching assistant, Syracuse University, 2007
- Best student, Thiagarajar College of Engineering, 2004.