Evaluation of Automated Securing Web Applications: A Survey

Rajeeb Saha
Master of Software Systems
University of British Columbia
rajeeb05@mss.icics.ubc.ca

ABSTRACT
With enormous increasing of e-business another thing is dramatically increasing, that is web application scamming. Therefore, it became a significant challenge for web application developer maintaining the confidentiality and integrity of the data they manipulate. Several research groups are working to secure web application end-to-end through partitioning application code (Swift, Links, Hop, UML-based Hilda), taking template-based approach (FlyingTemplate), abstracting security-critical code, building automated object oriented programing language (BAL) or specifying application-level data flow assertions (RESIN). This paper discusses construction, achievements, performance, limitations of these diverse procedures as well as different types of common web application-level attacks and vulnerabilities.

Categories and Subject Descriptors
C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks – *Internet*, D.1.m [**Programming Techniques**]: Miscellaneous, D.2.3[**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming*, D.2.11 [**Software Engineering**]: Software Architectures – *Data abstraction*, D.3.2 [**Programming Languages**]: Language Classifications – *Specialized Application Languages* , D.4.6 [**Operating Systems**]: Security and Protection – *Information flow controls*, D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks*, H.3.4 [**Information Storage And Retrieval**]: Systems and Software – *Distributed systems*, H.4.m [**Information Systems Applications**]: Information Systems – *Miscellaneous*, I.2.2 [**Artificial Intelligence**]: Automatic Programming – *Program transformation*

General Terms
Security, Languages, Design

Keywords
Information flow, security policies, Client-Server Partitioning, compilers, Swift, Links, Hilda, Hop, BAL, RESIN, Declarative Language, Security Policy Description Language, Application-Level Web Security, Data Driven Application, Template engines, Web applications, Web 2.0, JavaScript

1. INTRODUCTION
Now-a-days, an open question is how developers should construct web applications that accurately enforce robust security policies for data confidentiality and integrity. Information flow policies are an end-to-end requirement of web application security, unlike (discretionary) access control, which does not track information propagation. [16] Web site programmers often have strategy for precise data flow within their web application to escape SQL injection or cross-site scripting susceptibilities. Now-a-days, unluckily, these strategies are fulfilled indirectly where they try to introduce code in all the applicable places to ensure accurate flow, but it is possible and often miss some those can lead to web application exploits. For example, one well known web application, phpMyAdmin [19], involves sanitizing user input in 1,409 places. Not unexpectedly, developers forgot some of these calls and phpMyAdmin has suffered 60 vulnerabilities.[3]

Recently, different research groups proposed different automatic solutions to solve this and other types of web application's vulnerabilities. Automatic secure program partitioning [[14], [21]] has been recommended as a way to solve web vulnerabilities. Swift uses the Jif/split compiler for automatically partitions high-level, non-distributed

code into server-client subprograms that execute securely on a group of host machines that are trusted to varying degrees by the contributing principals. A partitioning treats as secure if the security of a principal can be affected only by the trusted hosts. As a result, the partitioning of the source code is driven by high-level trusted security policy specifications.

With the aims of implementation transparency, proficiency, security, and standards agreement in mind, Tastsubori and Suzumura developed FlyingTemplate [18]. Main two design principles behind FlyingTemplate are effective browser cache usage, and sensible negotiations which confine the template usage patterns and relax the security policies marginally but in a controllable way. This method permits typical template-based Web applications to run effectively with FlyingTemplate. As an experiment, they tested the SPECweb2005 banking application using Flying-Template without any other alterations and saw throughput enhancements from 1.6x to 2.0x in its best mode. Moreover, FlyingTemplate can implement compliance with a modest security policy, thus addressing the security glitches of client-server partitioning in the Web application environment.

Alexander, Xi, Nickolai and M. Frans [3] took data flow assertion approach to make web application more secure. Their approach knows as RESIN. In this methodology they made a system which allowed developers to create their design for precise data flow explicit using data flow assertions. Developers could write a data flow assertion in a place to capture the application's high-level data flow invariant, and RESIN checked the assertion in all relevant places, even places where the developers might have elapsed to check. And this way RESIN makes web applications more secure than conventional web applications. Main design goal of RESIN is provide developers to gain assurance in the accuracy of their application not to grip malicious code. RESIN faces some challenges to verify a data assertion, which are describes in details in section 3 of this paper.

In this paper I discuss about these different approaches in details. In section 2, I provide an overview about web vulnerabilities, why we need to secure web application, examples of web application attack. In section 3, I describe about evaluation of diverse research works to secure the web applications. In first part of the section 3, I provide details about secure web application code partitioning by java security annotations (Jif & SWIFT). In next part of the section 3, I discuss about automatic web application partitioning by UML and relational data model (Hilda). In subsequent part of same section, I go through about template engine approach (FlyingTemplate) to automated offloading from server to client. In flowing part of that section, I talk more details about RESIN which improves application security with data flow assertions. In last part of the section, I describe component based object oriented programing language (BAL), others similar approaches in brief and provide a comparison of these approaches. I discuss about conclusion and future works in section 4.

2. WEB VULNERABILITIES:

There are several web application vulnerabilities. SQL injection, Cross-site scripting, Denial of service, Buffer overflow, Directory traversal, Server-side script injection are major vulnerabilities. In this section, I describe these vulnerabilities briefly.

Table 1: Top CVE security vulnerabilities of 2008[4]

Vulnerability	Count	Percentage
SQL injection	1176	20.4%
Cross-site scripting	805	14.0%
Denial of service	661	11.5%
Buffer overflow	550	9.5%
Directory traversal	379	6.6%
Server-side script injection	287	5.0%
Missing access checks	263	4.6%
Other vulnerabilities	1647	28.6%
Total	5768	100%

2.1. SQL Injection and Cross-Site Scripting

In the last few of years, attacks against the Web application layer have required increased attention from security professionals. This is because no

matter how strong firewall rule sets are or how diligent patching mechanism may be, if Web application programmers have not kept an eye on secure coding practices, attackers will walk right into one's systems through port 80. The two main attack methods which have been used extensively are SQL Injection and Cross Site Scripting attacks [14]. From Table 1 it is noticeable that according to CVE [28] in 2008 top two web security vulnerabilities were these two techniques. Together these two techniques created ¼ of web application vulnerabilities. SQL Injection (SQLI) and cross site scripting (XSS) attacks are widespread methods of outbreak where the web attacker trades the input to the application to access or transform user data and perform malicious code. In the best severe attacks (known as second-order, or persistent, XSS), an attacker can corrupt a database which cause subsequent users to perform malicious code.

SQL Injection denotes to the method of injecting SQL meta-characters and instructions into Web-based input fields in order to manipulate the execution of the back-end SQL queries. According to Web Application Security Consortium Glossary, the definition of SQL Injection is

"*An attack technique used to exploit web sites by altering backend SQL statements through manipulating application input*."[26]

SQL Injection happens when a web application accepts user input that is straight placed into a SQL Statement and does not appropriately filter out unsafe characters. This can permit an attacker to not only snip data from the affected database, but also modify or delete data from database. Certain SQL Servers such as Microsoft SQL Server contain Stored and Extended Procedures (database server functions). It may be possible to compromise the entire system if an attacker can acquire access to these Procedures. [27] These are attacks focused largely against another organization's Web server.

According to Web Application Security Consortium Glossary, the definition of Cross Site Scripting attack is

"(Acronym – XSS) An attack technique that forces a web site to echo client-supplied data, which execute in a user's web browser. When a user is Cross-Site Scripted, the attacker will have access to all web browser content (cookies, history, application version, etc)" [26]

Cross Site Scripting attacks take place by embedding script tags in URLs and tempting unsuspicious users to click on them, ensuring that the malicious JavaScript gets performed on the victim's system. These attacks influence the confidence between the user and the server and become successful because server has no input/output validation to reject JavaScript characters.

2.2.Denial of Service

Denial of service is another common vulnerability that accounts for 11.5% of the vulnerabilities in Table 1. It is well known as DoS attack. According to Web Application Security Consortium Glossary Directory traversal is

"An attack technique that consumes all of a web site's available resources with the intent of rendering legitimate use impossible. Resources include CPU time, memory utilization, bandwidth, disk space, etc. When any of these resources reach full capacity, the system will normally be inaccessible to normal user activity."[26]

Let's give an example. What would happen if one person and some of his friends and relatives called the same pizza shop again and again and ordered pizza, but they did not really want? They would create obstruction the phone lines and devastate the kitchen same time. Therefore, the pizza shop could not proceed to any more new orders.

That is what takes place to Web servers while web attackers knock them with denial-of-service attacks. Web servers were cracked offline by too many unwanted requests from computers controlled by the attackers.

2.3.Buffer Overflow

Next top web vulnerability that according to CVE [28] in 2008 was buffer overflow which accounts for almost 1/10 of web application vulnerability. According to Web Application Security Consortium Glossary buffer overflow is

"An exploitation technique that alters the flow of an application by overwriting parts of memory. Buffer Overflows are a common cause of malfunctioning software. If the data written into a buffer exceeds its size, adjacent memory space will

be corrupted and normally produce a fault. An attacker may be able to utilize a buffer overflow situation to alter an application's process flow. Overfilling the buffer and rewriting memory-stack pointers could be used to execute arbitrary operating-system commands."[26]

Though it may happen unintentionally through programming errors, buffer overflow is a gradually well-known type of security attack on data integrity. In these types of attacks, the extra data may enclose codes intended to trigger specific actions. That could be happen by sending new instructions to the attacked computer. For example, damage the user's files, change data, or disclose confidential information can be outcome of buffer overflow further attacks.

2.4.Directory Traversal

Directory traversal is well-known another web application vulnerability. According to Web Application Security Consortium Glossary, the definition of Directory traversal is

"A technique used to exploit web sites by accessing files and commands beyond the document root directory. Most web sites restrict user access to a specific portion of the file-system, typically called the document root directory or CGI root directory. These directories contain the files and executables intended for public use. In most cases, a user should not be able to access any files beyond this point."[26]

From Table 1, it is noticeable that according to CVE [28] in 2008, Directory traversal is another well-known weakness that accounts for 6.6% of the web application vulnerabilities. To exploit this weakness, an attacker usually inserts the ".." string as part of the file name that permits the attacker to gain illegal access to read, or write files in the victim's file system. These abuses can be observed as faulty data flows. The file's data is incorrectly flowing to the attacker if he reads a file without the proper authorization. If the attacker writes to a file without the appropriate authorization, the attacker can cause an unacceptable flow into the file.[4]

2.5.Server-Side Script Injection

Server-side script injection is responsible for 5% of the exposures reported by CVE in Table 1. It is also known a Server-side information injection or SSI Injection. According to Web Application Security Consortium Glossary SSI Injection is

"A server-side exploit technique that allows an attacker to send code into a web application, which will be executed by the web server."[26]

Many applications allow uploading images or attachments. An attacker can abuse this by uploading a file with the desired code onto the server and then providing the name of that file as the theme to load.[4]

From previous discussion we know about different types of web application vulnerabilities. There can more types of vulnerabilities which are related to web application's confidentiality and client-server data integrity. Stephen Chong et. al. [21] gave a nice example - "suppose we want to implement a simple web application in which the user has three chances to guess a number between one and ten, and wins if a guess is correct. Even this simple application has subtleties. There is a confidentiality requirement: the user should not learn the true number until after the guesses are complete. There are integrity requirements, too: the match between the guess and the true number should be computed in a trustworthy way, and the guesses taken must also be counted correctly.

The guessing application could be implemented almost entirely as client-side JavaScript code, which would make the user interface very responsive and would offload the most work from the server. But it would be insecure: a client with a modified browser could peek at the true number, take extra guesses, or simply lie about whether a guess was correct. On the other hand, suppose guesses that are not valid numbers between one and ten do not count against the user. Then it is secure and indeed preferable to perform the bounds check on the client side. Currently, web application developers lack principled ways to make decisions about where code and data can be securely placed."

Therefore, to rid of these vulnerabilities web application developers need some automated tools which can figure out that web application have any exposure or not. Following section discuss about some of these tools and procedures.

3. EVALUATION OF AUTOMATED SECURING WEB APPLICATIONS:

For simplicity, I divided this section discussion into five major sub-sections. Frist two sub-sections I discuss about securing web application by portioning source code. Next section focuses on template base approach. Forth section is related to data flow assertions and in last section I discuss about other similar approaches.

3.1.Code Partitioning by Java Security Annotations:

Cornell University developed the Jif [14] programming language targeting to use replication and partitioning to develop secure distributed systems. They notice that the Java language has enhanced annotations in order to define access rights on each variable declaration. They used this into the Jif compiler to enforce the declared access rights for all usages of the annotated variables and splitter partitions the code in two parts to maintain consistency by automatically introducing state synchronize messages while also imposing the rights between the tiers. Their technique is well known as Jif/split system.

Later they introduced the Swift [21]. In Swift a web server is server side tier and client side tier is implemented though a web browser. Swift is a new principled methodology to developing web applications which are secure by construction. The system presents an intermediate language which known as WebIL. It allows the partitioning of the web application into client and server while considering data placement restrictions. A servlet implementation provided for the server communications while the Google Web Toolkit (GWT) [25] was used for the client.

A significant aspect of Swift is that it provides security by construction: the developers states security specifications, and the system converts the application to guarantee that these specifications are met.

The Jif/split system as well takes Jif as a source language and alters applications by placing code and data onto sets of hosts in agreement with the labels in the source code. Jif/split finds the general problem of distributed computation in a system integrating mutual distrust and random host dependence relationships.

Swift varies in discovering the challenges and prospects of web applications. Web applications have a specified trust model. Therefore specified construction methods are used to exploit this dependence relationship. In particular, replication is used by Jif/split to increase integrity, whereas Swift uses replication to increase performance and responsiveness. In addition, Swift uses a more sophisticated algorithm to control the placement and replication of code and data to the existing hosts. Swift applications support dynamic user interfaces and control the information flows. No Jif/split applications contain data structures or control flow of comparable complexity. Jif's label parameterization is needed to reason about information flow in complex data structures, but Jif/split lacks the necessary support for label parameters.[22]

Swift is an acronym of "**S**plitting **W**ebapps via **I**nformation **F**low **T**ypes". In this system web applications are written in high-level programming language – Java and information security specifications are unambiguously visible as declarative annotations. The Swift compiler selects where code and data in the web application can be placed securely by these security annotations. Code and data are partitioned at fine granularity, at the level of individual expressions and object fields. Building web applications in this way guarantees that the resultant distributed program protects the integrity and confidentiality of information flow between web server and client.

3.1.1. The Swift Architecture:

The system starts with annotated Java source code at the top of the diagram. Proceeding from top to bottom, a series of program alterations changes the code into a partitioned form shown at the bottom (Figure 1), with Java code running on the web server and JavaScript code running on the client web browser. Roughly the Swift architecture is as follows:

3.1.1.1. Jif source code:

Through the use of Jif – a programing language for information flow control and access control, Swift enforces security by construction. Jif extends Java programing language. For Swift, Stephen et al. [22] assume that the web server can be trusted, but the

client machine and browser may be buggy or malicious. Consequently, Swift must convert program code so that the application runs securely, even though it runs partly on the untrusted client.

3.1.1.2. WebIL intermediate code:

In this part Jif language program transforms into an intermediate language called WebIL. This language used to determine which part of the code should be placed on the server and which part of the code placed on the client.

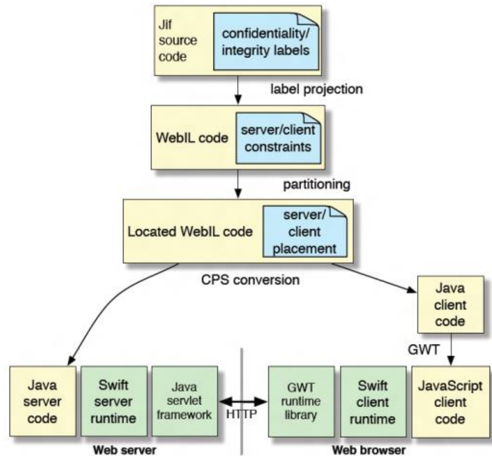

Figure 1: The Swift Architecture [21]

3.1.1.3. WebIL optimization:

In this phase initial WebIL code in optimize by compiling into a form such that it minimizes partition cost of the placement, in particular by avoiding unnecessary network messages between the client and server based on the specifications made earlier. The minimization of the partitioning cost is expressed as an integer programming (IP) problem, and maximum flow methods are then used to find a good partitioning.

3.1.1.4. Splitting code:

Then, optimized WebIL code is converted into actual Java programs with two parts - one for the server-side computation and the other for the client-side computation. This is a fine-grained transformation.

3.1.1.5. JavaScript output:

Since as a client I do not want to execute the client-side Java code on my web browser. Therefore, in this phase Swift translate the client side Java program into JavaScript code. On the client, this code then uses the Google Web Toolkit (GWT) run-time library and Swift's own run-time support. On the server, the Java application code links against Swift's server-side run-time library, which in turn sits on top of the standard Java servlet framework.[21]

From the browser's perspective, the application runs as a single web page, with most user actions (e.g., clicking on buttons) handled by JavaScript code. This approach seems to be the current trend in web application design, replacing the older model in which a web application is associated with many different URLs. One result of the change is that the browser "back" and "forward" buttons no longer have the originally intended effect on the web application, though this can be largely hidden, as is done in the GWT. [22]

3.1.1.6. Partitioning and replication:

Compiling a Swift application places some code and data onto the client. Code and data that implement the user interface clearly must be located on the client. Other code and data are placed on the client to avoid the latency of communicating with the server. With this tactic, the web application can have a rich, highly responsive user interface that waits for server replies only when security demands then the server involved. The Swift runtime support is responsible for handling synchronization and communication among the different segments used to perform the system.

3.1.2. Advantage of Swift:

1. Web developers get a tool which ensures that the resulting distributed application protects the confidentiality and integrity of the information based on given security annotations.
2. By the general enforcement of information integrity, Swift also guards against common top two web application vulnerabilities – SQL injection and cross-site scripting.
3. Swift applications are also easier to write because control and data do not need to be explicitly transferred between client and server through the awkward extra-linguistic mechanism of HTTP request.

4. In current practice, the developer has no help planning the protocol or interfaces by which client and server code communicate. With Swift, the compiler automatically produces secure, effective interfaces for communication.
5. Swift also optimizes the server – client communication which saves bandwidth.
6. Moreover, Swift replicates some codes to client side which makes the web application more responsive.
7. Swift does not introduce new programming language or platform to secure the web application. It uses well known Java language with security annotations.

3.1.3. Weakness of Swift:

1. Main problem of Swift is bandwidth due to it puts some extra code in client side to make web application more responsive.
2. Next weakness of Swift introduces when developers try to trace bug in the system, because Swift consist of too many modules – initial code needs to be compiled, transformed and optimized at least three times to get actual application code.
3. When Google Web Toolkit translates client side codes to JavaScript, it creates extra code which is bandwidth inefficient.
4. It only supports Java programming language. Therefore, widely used web language (php, python) developers need to learn Java. That means longer learning curve.

Besides these weaknesses, I personally think that Swift would deliver a more secure system and it will save both time and money for companies who are looking to upgrade their current not so secure web application.

3.2.Data Driven Web Application Partitioning:

To secure web application, another effort was made with the introduction of the programming language Links [8] at the University of Edinburgh. It follows the functional programming paradigm and integrates an SQL based database. OCaml was used to develop the compiler and the run time environment consists of JavaScript code for the browser and SQL commands at the server. The state is maintained completely on the client.[20]

Recently, Hilda was introduced. In Hilda, as in Links, the language delivers for data definition as well. The language is declarative and the notion of separating the user interface from the business logic is introduced.[20]

Data-driven web applications are usually structured in three tiers with different programming models at each tier.

1. Lowest Tier: A database system which supplies persistent data
2. Middle Tier: An application server which holds most of the application logic
3. Top Tier: The client web browser that encloses some client-specific program logic and presentation

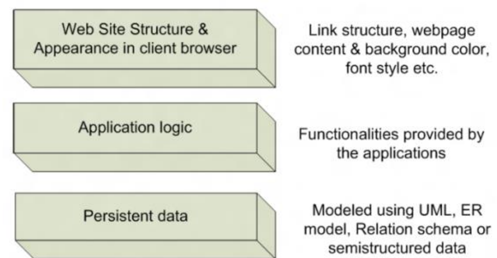

Figure 2: Tiers in a Data-Driven Web Application [29]

This division forces Programmers to manually partition program functionality across the different tiers which results complex logic, suboptimal partitioning, and expensive repartitioning of applications.

Fan Yang et al. [9] introduce a unified platform for automatic partitioning of data-driven web applications. Their approach is based on Hilda, a high-level declarative programming language with a unified data and programming model for all the layers of the application. Based on run-time properties of the application, Hilda's run time system automatically partitions the application between the tiers to improve response time while adhering to memory and/or processing constraints at the clients.

3.2.1. The Hilda Architecture:

First, Hilda is based on UML, a well-accepted modeling framework. Hilda delivers an application building block called an AUnit (for Application Unit), analogous to a UML class. AUnits support encapsulation like a regular UML class, but the formation and manipulation of AUnits is stated declaratively and delivers natural support for conflict discovery in the face of concurrent application updates. AUnits are single-entry and single-exit, which enables structured programming. The main dissimilarity from the traditional use of UML is that the object formation and operations are stated declaratively, which enables the Hilda compiler to automatically execute various optimizations without burdening the user with performance issues.

Second, Hilda uses a single data model – the relational model – to denote the state of all parts of the application, as well as the database, application logic and the client.

Third, Hilda logically splits server and client state by separating persistent states and local states to enable highly concurrent execution.

Fourth, Hilda models the application logic and associated control flow as a hierarchy (Activation Tree) which captures the application logic of the system.

Finally, Hilda provides a HTML-based presentation construct. It ensures a clear separation of application logic from presentation. [29]

3.2.2. Advantage of Hilda

1. Hilda allows an exciting optimization opportunity where client-server partitioning can be done automatically and correctly by a compiler instead of having the developers write low-level and error prone code for the same purpose.
2. The developers only need to focus on developing the core logic of the system, and the Hilda compiler can automatically compile it into code depending on the abilities of the client and other influences such as bandwidth limitations and concurrent actions.

3.2.3. Drawback of Hilda:

1. Although Hilda logically separates server and client state, but it has ignored the security problems produced by porting some parts of the server-side logic of a web program to the untrusted clients.

Hilda, as same as Swift, also uses Java programing language. In Hilda, the statement will either be executed at the server or compiled into Java code and executed at the client based on the client capabilities. Therefore, if we can introduce Java security annotation in Hilda as like Swift, then it would be overcome the security drawbacks. Or, if it is possible to encrypt the ported server-side logic, then it can be overcome the security drawbacks. It would be great future work and hope more researchers come forward and take the challenge to make Hilda as a secure platform for web applications.

3.3.Template Engine Approach:

Template-based Web programming is widespread mostly because it splits the page representation, the “views”, from the business logic and data of an application, the “controls and models”. Such templates are presented as software libraries, as programming or modeling language features or as Web application frameworks. The benefits include encapsulating the look and feel of a website, clearly described views, a better division of labor between graphics designers and coders, component reuse for view designs, unified control over the evolution of the appearance, better maintainability of the runtime, interchangeable view artifacts for different development projects, and security compatible with end-user customizability.[18]

FlyingTemplate is a server-side template engine. It looks much like a regular server-side engine that does the template filling work on the server side, but actually lets the clients do that work. Michiaki and Toyotaro [18] used Smarty as a reference template engine and PHP as the underlying programming language, but the design of FlyingTemplate itself should be applicable to other template engines and programming languages.

The major design goals of FlyingTemplate are:

Efficiency – FlyingTemplate should perform better than existing template engines, at least in typical circumstances.

Standards compliance – The implementation should conform to Web standards.

Implementation Transparency – Existing applications should run correctly without modifications.

Server Security – Introducing FlyingTemplate should not create unexpected security vulnerabilities.

FlyingTemplate is a server-side template engine that automatically handovers more of the task of generating HTML documents to the client browsers. Instead of generating a fully-generated HTML page, the proposed template engine creates a skeletal script which contains only the dynamic values of the template parameters and the bootstrap code that executes on a Web browser at the client side. Michiaki and Toyotaro designed the architecture of the client-server partitioning for effective browser cache use with the implementation of a simple server security policy. The efficiency of the partitioned system like Swift depends on the complex analysis of the entire program, which may not always be quite optimum. The methodology of FlyingTemplate can be considered as greatly easing this kind of security annotation task and giving heuristics for efficient partitioning according to the convention of the template-based programming model. Therefore, if we can combine Swift and FlyingTemplate approach together, then it would form secure web application which can be easily developed.

3.4.Data Flow Assertions:

RESIN [4] is a new language runtime that supports avoiding security vulnerabilities, by allowing programmers to specify application-level data flow assertions. RESIN delivers policy objects, which developers use to specify assertion code and metadata; data tracking, which allows developers to associate assertions with application data and to keep track of assertions as the data flow through the application; and filter objects, which developers use to express data flow boundaries at which assertions are tested.

Using RESIN, Web application developers can avoid a range of glitches like SQL injection, cross-site scripting, accidental password disclosure and missing access control checks. Adding a RESIN assertion to a program needs few modifications to the existing program code, and an assertion can reuse existing code and data structures. For instance, 23 lines of code detect and prevent three previously-unknown missing access control vulnerabilities in phpBB, a popular Web forum application. Other assertions comprising tens of lines of code prevent a range of vulnerabilities in Python and PHP applications. A prototype of RESIN incurs a 33% CPU overhead running the HotCRP conference management application. [4]

3.4.1. Benefits of RESIN:

For implementing data flow assertions, RESIN provides three mechanisms:

1. Data tracking as data flows through an application,
2. Policy objects associated with data,
3. Filter objects that define data flow boundaries and control data movement.

Alexander et al. [4] evaluated RESIN by adding data flow assertions to prevent security vulnerabilities in existing PHP and Python applications. Their results show that data flow assertions are effective at preventing a wide range of vulnerabilities like SQL Injection, Cross-Site Scripting, directory traversal, server-side script injection, access control, password disclosure, etc. These assertions are short and easy to write. Moreover, In RESIN, assertions can be added incrementally without having to restructure existing applications.

3.4.2. Limitation of RESIN:

RESIN currently has a number of limitations:

First, Alexander et al. [4] would like to provide better support for data integrity invariants. Instead of requiring programmers to specify what writes are allowed using filter objects, they predict using transactions to buffer database or file system changes, and checking a programmer-specified assertion before committing them.

Second, for example, an assertion could avoid clear-text passwords from flowing out of the software module that handles passwords. Attaching filter objects to function calls helps with these boundaries, but languages like PHP and Python allow code to read and write data in another module's scope as if they were global variables. An internal data flow boundary would need to address these data flow paths.

Table 2: Compare Swift, FlyingTemplate and RESIN

	Swift	**Hilda**	**FlyingTemplate**	**RESIN**
Technique	Partitioning by Java security annotation	Data driven partitioning	Template base	Data flow assertion
Language	Java	Java , UML	PHP	PHP, Python
Complexity	Complex	Complex	Simple	Simple
SQL Injection	Prevent	No	Unknown	Prevent
Cross-Site Scripting	Prevent	No	Unknown	Prevent
Bandwidth Efficiency	No	Yes	Yes	Yes
Support Dynamic Web Application	Yes	Yes	Yes	Yes

Finally, dynamic data tracking adds runtime overheads and presents challenges to tracking data through control flow paths. Developer would like to investigate whether static analysis or programmer annotations can help check RESIN-style data flow assertions at compile time.

3.5.Other Automated Approaches:

Alefragis and Chondros defined a new object oriented programming language called BAL [20] (Business Applications Language) supporting the usual constructs of class, property and method, with automatic memory management via garbage collection. It is enriched with domain specific commands in order to aid in the implementation of the partitioning logic by reducing the problem from its general form.

A different approach comes from INRIA where a new programming language called Hop [17] was introduced aiming to provide for interactive web applications. The approach is not aimed for database oriented applications but mostly deals with the user interface. Still, the target program is developed monolithically and the compiler splits it in a client and server part. The programming language is modeled closely to the HTML layout.[20]

4. CONCLUSIONS

Now-a-days, there are security concerns to address privacy, data confidentiality and integrity. In static partitioning like Swift, a module is permanent to run either at the client or at the server. If a module contains sensitive server-side data, the module should run at the server. In dynamic partitioning, web developers need to confirm that a module which contains the sensitive data of the web application does not run at client browser. This obligation brings up challenging privacy-preserving partitioning difficulties. There are approaches like Swift [22] that rely on developers annotations. One exciting research trend is to look at more automated methods of privacy preserving partitioning through static and dynamic application investigation. I have confidence in that automated dynamic partitioning of application code is a vital part of future secure web application development.

From previous discussion, we notice that Swift makes web application more secure. But in Swift, applications should be written in Java that creates language constrain. There are other well used web languages like php, python remaining unsecure and it is hard to convert these applications into java. So, we need some solution which can convert these unsecure web applications to secure web applications. It can be possible by combining Swift with FlyingTemplate or by overcome the limitation of RESIN. These would be good future research topic.

In this paper, I tried to give brief overview about some automatic tools which were used to make web applications more secure as well as reduced time to develop them. I also tried to figure out their benefits and limitation. Then I discussed some solution to overcome these limitations.

5. REFERENCES

[1] Kiezun, A., Guo, P.J., Jayaraman, K., Ernst, M.D. 2009, Automatic creation of SQL Injection and cross-site scripting attacks. In *Proceedings of ICSE*. 199-209.

[2] Myers, A.C. and Liskov, B. 2000. Protecting privacy using the decentralized label model. In *Journal of ACM Transactions on Software Engineering and Methodology (TOSEM)*. 9, 4 (October 2000), 410-442.

[3] Yip, A. S. 2009. *Improving web site security with Data Flow Management*. Doctoral Thesis. Massachusetts Institute of Technology. (September 2009).

[4] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M.F. 2009, Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*. 291-304.

[5] Chun, Byung-Gon, & Maniatis, P. 2010. Dynamically Partitioning Applications between Weak Devices and Clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond (*MCS '10*)*. Retrieved February 01, 2012, from http://dl.acm.org/citation.cfm?id=1810938.

[6] Scott, D. and Sharp, R. 2002. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web (WWW '02)*. 396-407.

[7] Scott, D. and Sharp, R. 2002. Developing Secure Web Applications. In *Proceedings of IEEE Internet Computing*. 38-45.

[8] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. 2006. Links: Web Programming Without Tiers. In *Proceedings of FMCO*. 266-296.

[9] Yang, F., Gupta, N., Gerner, N., Qi, X., Demers, A.J., Gehrke, J., and Shanmugasundaram, J. 2007. A unified platform for data driven web applications with automatic client-server partitioning. In *Proceedings of the 16th international conference on World Wide Web (WWW '07)*. 341-350.

[10] Trouessin, G., Fabre, Jean-Charles, and Deswarte, Y. 1991. Improvement of data processing security by means of fault tolerance. *In14th National Computer Security Conference*, pages 295-304, Washington, USA.

[11] Bar-Gad, I., and Klein, A. 2002. *Developing Secure Web Applications*. *Sanctum Inc*. (June 2002) Retrieved February 01, 2012, from http://www.cgisecurity.com/lib/WhitePaper_DevelopingSecureWebApps.pdf

[12] Fabre, Jean-Charles., Deswarte, Y., and Randell, B. 1994. Designing secure and reliable applications using fragmentation-redundancy-scattering: an object-oriented approach. In *PDCS 2: Open Conference*, pages 343-362, Newcastle-upon-Tyne. Department of Computing Science, University of Newcastle, NE1 7RU, UK.

[13] Kuuskeri, J. and Mikkonen, T. 2009. Partitioning web applications between the server and the client. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*. 647-652. Retrieved February 01, 2012, from http://doi.acm.org/10.1145/1529282.1529416

[14] Mookhey, K. K., & Burghate, N. 2010. *Detection of SQL Injection and Cross-site Scripting Attacks*. (2010, November 02) Retrieved from symantec: http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks

[15] Princehouse, L. and Birman, K. 2009. Code-partitioning gossip. In *Proceedings of Operating Systems Review*. 40-44.

[16] Zheng, L., Chong, S., Myers, A.C., and Zdancewic, S. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of IEEE Symposium on Security and Privacy*. (May 2003) 236-250.

[17] Serrano, M., Gallesio, E., and Loitsch, F. 2006. Hop: a language for programming the web 2.0. In *Proceedings of OOPSLA Companion*. 975-985.

[18] Tatsubori, M. and Suzumura, T. 2009. HTML templates that fly: a template engine approach to automated offloading from server to client. In *Proceedings of WWW*. 951-960.

[19] phpMyAdmin. phpMyAdmin 3.5.0. http://www.phpmyadmin.net/.

[20] Alefragis, P. and Chondros, N. 2009. BAL: A Language for Component Based Distributed

Applications Development. In *Proceedings of EUROMICRO-SEAA*. 486-489.

[21] Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. 2007. Secure web application via automatic partitioning. In *Proceedings of SOSP*. 31-44.

[22] Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. 2009. Building secure web applications with automatic partitioning. *In Proceedings of Communication*. *ACM*. 79-87.

[23] Zdancewic, S., Zheng, L., Nystrom, N., and Myers, A.C. 2002. Secure program partitioning. In *Proceedings of ACM Trans*. Comput. Syst. 20, 3 (August 2002), 283-328.

[24] Garg, V., Stock, A. v., & Owen, K. OWASP Guide Project. Retrieved February 04, 2012 from *The Open Web Application Security Project:* https://www.owasp.org/index.php/OWASP_Guide_Project

[25] Google Web Toolkit, http://code.google.com/webtoolkit

[26] Web application security consortium. 2004. *Web Security Glossary*. (2004, February 23) Retrieved from http://www.webappsec.org/projects/glossary/

[27] What is SQL Injection?. Retrieved April 05, 2012, from http://www.cgisecurity.com/questions/sql.shtml

[28] The MITRE Corporation. *Common vulnerabilities and exposures (CVE) database*. Retrieved from http://cve.mitre.org/data/downloads/

[29] Gerner, N., Yang, F., Demers, A.J., Gehrke, J., Riedewald, M., Shanmugasundaram, J.: Automatic client-server partitioning of data-driven web applications. In SIGMOD Conference(2006)760-762