An automatic CSRF protection tool

Iwan G. Flameling

Januari 12, 2015, 48 pages

Supervisor: prof. dr. Jurgen Vinju
Host organisation: Universiteit van Amsterdam

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Chapter 1

Acknowledgement

This thesis marks the end of my enrolment for the Master of Science in Software Engineering programme at the University of Amsterdam. The majority of the programme was conducted, besides my job as a software engineer, on a part-time basis. I certainly had my struggles and challenges during this programme, but I also learned many things and met great people. My surrounding environment has been affected by my decision to take part in the Master Software Engineering programme and I would like to seize the opportunity to thank many of the people surrounding me:

My mother showed unconditional support from the moment I considered to enrol for scientific education. Thank you so much mom, it meant the world to me. I have had my dad's full support from the moment I enrolled for my pre-master courses. Even thou the past can't be changed, dad, I'm very grateful for all your support. Both my parents have been extremely supportive, especially at the end, during the last months of my thesis.

I thank Maja Valstar for fuelling that spark of curiosity towards the education programmes of research universities. I thank Renate Roze for taking the sacrifices, we made for my study, during our relationship. I thank Jasper Timmer for our collaboration during the pre-master programme. Many thanks to Cees Brouwer for giving last-minute feedback on this document. Special thanks to Johanneke Lamberink for her coaching, love and support. You've become a great friend. I would like to thank Jorryt-Jan Dijkstra for his substantive contribution during our talks. I hope we'll be friends for a long time to come. I'm thankful to the complete MSE-staff for being such knowledgable, sometimes admirable, researchers and teachers.

But most of all I would like to thank the person who I will remember as a role figure in life. Someone who has a remarkable sense for how I need to be coached, and he did it with a positive vibe that made me smile many times. It has been an honour that prof. dr. J. Vinju has been my supervisor. Jurgen, thank you so much, your way of coaching has made all the difference.

In memoriam of Jan Lankamp...

Chapter 2

Introduction

This chapter will focus on funnelling towards subject of this document, as is visually depicted in figure 2.1. Context is given in the first two sections. First we[1] begin with the position of clients in software security in section 2.1. Second, an introduction to retrofitting and how it can be used to improve software security, is given in section 2.2. In section 2.3 our research, on the use of Aspect Oriented Programming to retrofit applications with protection against a specific software security attack, is introduced. The chapter is concluded with a problem statement and questions for research in section 2.4.

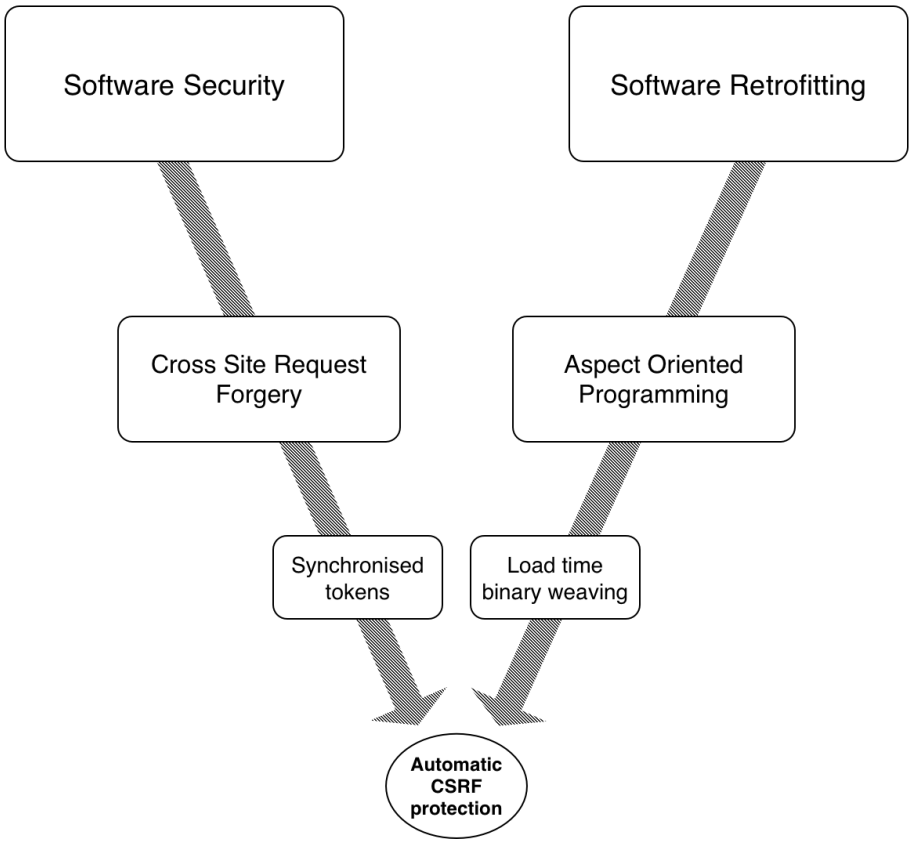


Figure 2.1: Funnelling towards this research' subject.

[1]Despite that this document and the accompanying research has been written by a single author, plural verbs are used to indicate that the document and research have been supervised.

2.1 Detrimental position of clients in software security

There is a dichotomy in the capabilities and motivation of software producers and consumers. Software producers are most likely in the best position to mitigate vulnerabilities, since they have access to source code, production tool chain and developers. Security mechanisms can be applied at the point where the most semantic knowledge of the code and the program are available. Consumers, however, have to deal with the risk and consequences of compromised software that can be addressed to software vulnerabilities and are, therefore, the most motivated to mitigate newly discovered vulnerabilities [1]. O'Sullivan points out that even if the consumers are security-conscious, they are not in a good position to address their software's security risks; "Even security-conscious software consumers often cannot properly evaluate the risks they face because they do not know what security mechanisms, if any, a producer has used in their development process and tool-chain".

2.2 Improve software security through retrofitting

Retrofitting applications, as in providing software with parts not in existence or available at the time of original manufacture[2], has been subject of research [1] [2] [3]:

- Within the field of software re-engineering NASA[3] reports on COTS applications that are retrofitted with new features, called 'hybrid re-engineering' [2].
- In the field of software transformation Necula created a transformation system that retrofits legacy software written in C with type-safety mechanisms. In order to address the impractical exercise to make pervasive changes in large existing programs, a pointer-kind inference algorithm is used to discover the best pointer qualifier for each pointer in the program [3].
- O'Sullivan presents a practical tool, and uses an advanced binary rewriter, for inserting security features against low-level software attacks into third-party, proprietary or otherwise binary-only software. They claim to be the first to demonstrate the use of such mechanisms in the absence of source code availability [1].

Our research focusses on the latter, adding security features to software without the need for its source code. Being able to add features to an already existing product that is used by software consumers, might help software consumers to secure systems when software producers' security measures are regarded as insufficient.

2.3 Retrofit security using Aspect Oriented Programming

Motivation for this research to specifically use Aspect Oriented Programming to retrofit applications with improved security is twofold and explained in section 2.3.1 and section 2.3.2.

2.3.1 Binary weaving to retrofit applications

In the work of Meridith, to retrofit networked applications with autonomic re-configuration features, it is suggested that "Future work may consider retrofitting applications for which the source code is not available" [4]. This future work, of retrofitting applications for which the source code is not available, is the context of our work.

Load-time binary weaving is a method that merges an application with additional code (aspects) when an application is loaded. This enables us to alter applications, with Aspect Oriented Programming (AOP), for which source code is not available. We've used this method to retrofit applications with existing mitigation techniques against Cross-Site Request Forgery (CSRF) vulnerabilities.

[2] http://www.thefreedictionary.com/retrofit

[3] For the sake of brevity, this document uses various abbreviations. Full names of these abbreviations can be found in Appendix 10.1

2.3.2 Aspect Oriented Programming to improve security

Two web application security vulnerabilities that are known to be exploited are:

- Code-injection attacks - in which data is provided by the user by including it in a SQL query - in such a way that part of the user's input is treated as SQL code, is referred as SQL injection [6].
- A special case of command injection flaw, whereby user input containing scripting content (JavaScript or non-JavaScript vector) is placed into the output HTML without being checked for HTML code or scripting code, is known as Cross Site Scripting or XSS [7].

It has already been shown that Aspect Oriented Programming (see section 3.3) can be used to mitigate these SQL Injection and Cross-Site Scripting vulnerabilities in web application security [5] [8]. Our research is focused on CSRF (see section 3.1), a vulnerability that has been left to further research [9], and is suggested to be solved by means of AOP [10].

2.4 Problem statement

Software consumers who lack knowledge, skills, source code or support of the original software developers (producers) to protect their systems from security vulnerabilities, is the challenge we face to address with our research.

The Open Web Application Security Project (OWASP)[4] has listed CSRF vulnerability within its top ten most critical web application security risks[5]. A disturbing effect of this vulnerability was showcased in the ING Direct incident that allowed an attacker to open additional accounts on behalf of a user and transfer funds from a user's account to the attacker's account [11]. Zeller and Felten also reported on other effects of CSRF that applied to the website of the New York Times and Youtube [11].

Within the context of enabling software consumers to retrofit (possible third party) software we are trying to answer the following question:

- Can AOP be used to retrofit multiple, possibly third party, applications with a solution that protects against CSRF attacks?

In order for the retrofitted solution to enable software consumers in protecting their software, the software consumer might assume that the solution does not change or 'break' the original application's functioning (besides being vulnerable to CSRF attacks). Functioning can be effected by unintended behaviour, therefore we try to answer our main question by answering the following sub-questions:

- Can the solution successfully block a CSRF attack from executing its desired effect?
- What effects, besides protecting against a CSRF attack, does the solution have on the application's functionality?

Our research focusses on tricking a victim's browser into issuing a request that the victim did not intend to send by means of Cross Site Request Forgery. Other means for infringement, like malicious browser plugins and Man-In-The-Middle attacks are deliberately left out of scope. These methods are not specifically related to Cross Site Request Forgery and in many cases they negate an attacker's need for Cross Site Request Forgery vulnerabilities.

[4] https://www.owasp.org/index.php/Main_Page
[5] https://www.owasp.org/index.php/Top_10_2013-Top_10

Chapter 3

Background

3.1 Cross Site Request Forgery in web applications

Many web applications use the HTTP protocol to exchange data between the client and the server-side part of the application. HTTP is a stateless protocol and cannot link requests that belong to the same user. This poses a problem for web application authentication, since the protocol cannot distinguish whether the client has already been successfully logged in or not. One way to address this is by using an identification token that is send as a reply to the client's browser after a successful login, a session identifier is set in the `Set-Cookie`[1] HTTP header. This session id is stored by the browser and send along with subsequent HTTP request messages in the `Cookie` header.
In order to have some sort of application state, which can not directly be modified by the client's environment and doesn't have to be transferred with every request, the server can maintain a server-sided session which is bound to a particular client by using the session's identifier [12].

A web application is vulnerable to CSRF attacks because it trusts the session, which the server sided part of the web application has with the client, and does not further validate individual requests made by that client. This enables an attacker to trick the unknowing user in sending a malicious request to the server, which is trusted by the server, since the client is authenticated and trusted within the session. An example is given in figure 3.1 which shows how a visit to evil.com leads to transferring 1500 euro's at bank.org, without the user knowing. This example is described in the following three steps:

- pre-condition: The user's web browser, the client, is in an authenticated and trusted session with the bank's application server and uses the same web browser to visit other web pages.

1. The user requests a page at evil.com, not knowing it is a site with bad intentions.
2. Evil.com responds to the request with a page that encapsulates a CSRF attack vector, in this case the `src` attribute of the HTML `img` element.
3. Under the pretext of requesting an image, the user's client actually requests the bank to do a money transfer from his account to the attacker's account.

[1] A different font is used to emphasise specific references to code or data constructs. Such as classes or objects, variables and operations. For example: `Person`, `age` and `getAge()`.

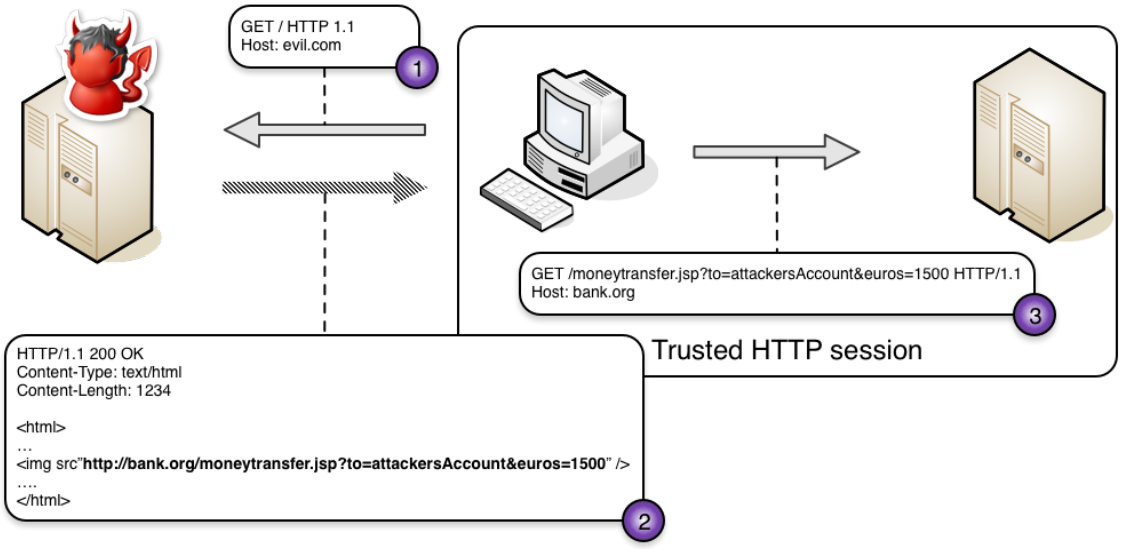


Figure 3.1: Use CSRF to transfer money

A short background on CSRF and an example of how it can be exploited have been given in this section. Illustrations of other examples can be found in appendix 9.1. Next, in section 3.2, a popular protection method will be illuminated.

3.2 Token based synchronisation

A very popular – advertised by OWASP[2] – protection against CSRF attacks is the requirement of a valid secret token parameter, before sensitive operations are performed. The reasoning is, when the server generates a token that is required for a possible subsequent request, the subsequent request originates from the (authenticated) user who uses the interface that is generated by the web application. In contrast to a user who is tricked in executing a request that has been forged by an attacker, while the user's browser has an open session with the targeted application.

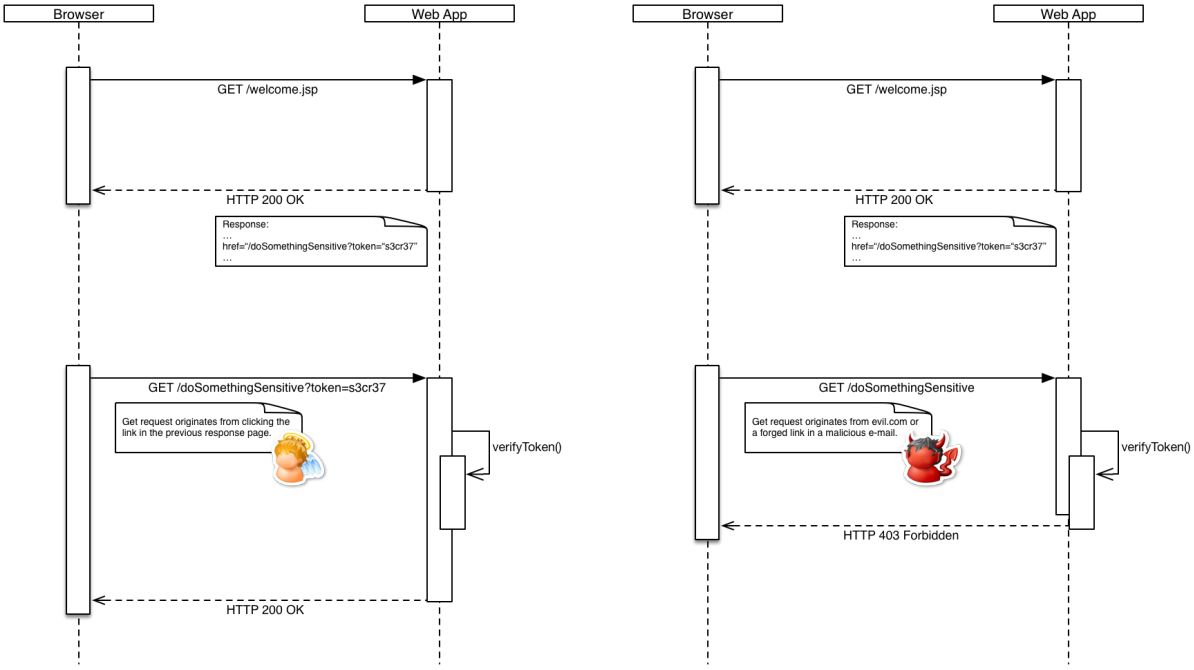
3.2.1 How it works

Figure 3.2 shows how token based synchronisation obstructs successful execution of CSRF. Figure 3.2a shows the proces of handling a regular, genuine, request. While figure 3.2b shows processing of a malicious request that has been forged by an attacker.

- The user's client, the web browser, requests a 'landing page' (a page request that does not require a token). In figure 3.2 this is indicated with a request for "welcome.jsp".
- The response to this request is a webpage that has added tokens (as an extra parameter) to URLs that the application has to protected against CSRF. Up until this point, figure 3.2a and figure 3.2b have been the same, but their process will start to differ in the next few steps.
- When a browser uses such an URL to perform its next request, the token is send along with the rest of the request, as can be seen in figure 3.2a. In the attacker's case of fig 3.2b, the attacker forged the request, but is unable to guess the token to send it along with the request.
- After verification of the token in figure 3.2a, the web application will regard the request as genuine usage of the application and continues regular processing. In the case of a malicious

[2] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

request, the web application can choose a different execution path. Most likely it is not to continue regular processing of the request, but to log, redirect or terminate with an error message instead. In figure 3.2b it terminates with a HTTP 403 Forbidden response message.



(a) genuine, regular request

(b) malicious, forged request

Figure 3.2: Token based synchronisation

To have a better understanding of the proposed solution in chapter 4, four aspects of AOP are explained next in section 3.3.

3.3 Aspect Oriented Programming

This section consists of four sections, highlighting the following: the relation of AOP with software security through the understanding of 'cross cutting concerns' and 'aspects', the notion of 'pointcuts', an explanation of an 'advice' and the technique of load-time weaving.

3.3.1 Aspects to address cross cutting concerns

In object oriented programming a program is segregated into pieces, called objects. These objects represent data and operations that are related to each other. For example, a `Person` object containing data like name or date of birth and operations like `getAge()`. In a course enrolment program a `Person` object is part of its 'business'. Other application concerns like logging, persistence and security are concerns that might not directly relate to the application's business purpose, but do occur across the various objects the application consists of. These so called cross-cutting concerns are isolated and addressed in Aspect Oriented Programming(AOP) as 'aspects' [13].

3.3.2 Cutting into the application

The isolated code that addresses the cross-cutting concern can be joined with the application business code at so called 'join points'. Join points may differ for various AOP implementation frameworks. A

set of join points are specified and expressed as pointcuts. Pointcuts create the possibility to define where in the application's code the crosscutting code should be joined [13].

3.3.3 Advising code with pointcuts using AspectJ

In the AspectJ implementation of AOP[3] the isolated code is referred to as an 'advice'. The applications code that is affected at a join point, as defined by a pointcut expression, is 'advised'. As soon as a location or multiple locations have been expressed with pointcuts, it can be used to direct an advice before, after or around those join points [13].

3.3.4 Weaving: combining the cross cutting concern with the application

Aspects can be weaved with application code at source code level, before its compiled to machine code. Java bytecode, as an interpreted language system, allows AspectJ code to be weaved after it's compiled. AspectJ comes with a Java Virtual Machine (JVM) Agent that is able to weave-in our security aspect at load-time. In our proposal in chapter 4, the bytecode is transformed after JVM initialisation using a weaver as JVM agent. Java uses the JAVA_TOOL_OPTIONS environment variable to predefine parameters for the JVM. In the context of our research this enables us to automatically weave a security aspect with a compiled Java program [13].

Two technical concepts which lay the foundation for the proposed solution in chapter 4 have been discussed and the next section, section 3.4, summarises related scientific work that puts the proposed solution of chapter 4 into context.

3.4 Related work

Mece and Codra were able to mitigate attacks by advising the Java Servlet API [14]. Hermosillo has shown that AOP load-time weaving can be used to mitigate web application security vulnerabilities [5] and Simic and Walden report on using AOP to protect multiple applications without specific application knowledge [8]. Mahjoubi et. al. has highlighted that CSRF is not yet solved [15], while Zeller and Felten have shown that CSRF is a real world vulnerability [11]. Finifter reports on experiment findings in which manual source code reviews were better able to find CSRF vulnerabilities than black-box testing was. He also found that automatic CSRF prevention mechanisms correlate with an increased mitigation of CSRF vulnerabilities [16]. In order to redirect processing for analysis of malicious behaviour Chew et. al. report on using AOP to weave in anti-CSRF tokens [10]. Jovanovic et. al. show that they were able to apply token based CSRF protection to various third party PHP applications without adjusting or recompiling any of its source code [12].

When automated protection can be beneficial to security

Finifter had various developers program an implementation with different tools, languages and/or frameworks from the same application specification. From the experiments it seems the presence of automatic CSRF protection correlates with the security against CSRF attacks, while manual protection support did not seem to correlate [16]. Such findings might support the presence of automatic CSRF prevention mechanisms. Finifter also found that manual source code review finds more CSRF vulnerabilities than automatic black-box testing does. Yet in another study that was done by Finifter, it was found that only 17% of their test group of security assessors found the planted CSRF vulnerabilities [16]. A framework or tool that could fully guarantee CSRF protection could possibly neglect this problem.

3.4.1 Security with Aspect Oriented Programming

Shanmughaneethi et. al. uses AOP to intercept dynamically generated SQL queries from an SQL sink and submits it to a web service. The experiment uses SQL grammar to create a XML structure

[3] http://www.eclipse.org/aspectj/

that separates the SQL keywords from non-keywords. Based on this XML transformation SQLI attacks by means of tautology, illegal/logically incorrect queries and piggybacking are mitigated. Shanmughaneethi claims this method is dialect independent. While the increase of response time is over 50%, it is claimed that the performance penalty in the test is negligible when compared to the consequence of an injection attack [17]. Because an already generated SQL query is used as input for the security code instead of user input only, the solution might be used in non-web applications as well as in web applications.

Load time weaving

Hermossilo reports on successfully implementing methods to mitigate XSS and SQLI using AOP for one specific, self made application. It has been shown that load-time binary weaving can be used on the JBOSS application server, so recompilation of source code is not needed[5].

Simic and Walden did a more extensive research in which they use a source code analyzer first to identify SQLI and XSS vulnerabilities. Based on the XML output of the analyzer, the security aspects are created. The aspects are based on pointcut and advise templates. The OWASP ESAPI library is used to create the counter measures for the vulnerabilities. After the aspect is generated, the application executes the protective aspect code at runtime to mitigate its security issues [8]. Their research shows protection of multiple, third party Java web applications against XSS and SQLI by using AOP without significant performance penalties.

Advising Java Servlets

Mece and Codra created security aspects that only advises pointcuts of de Java Servlet API with the intent to sanitize user input from request parameters. It uses a syntactic validator first to check for characters that might be dangerous in the context of XSS and SQLI. If the string is syntactically tainted an encoder is used before the string is passed to a semantic validator. The string is validated by comparing it against predefined patterns. The authors were unable to execute an attack successfully with the aspect enabled on the Webgoat project with an average performance overhead of 2.11% [14].

3.4.2 CSRF related research

Chew et. al. focuses on the creation of behavioral profiles by creating n-grams from a webserver's access log file. N-grams that are similar are grouped together and their frequency is measured to identify behavior. Low interaction honeypots are used to lure attackers from a real system to a fake system (high interaction honeypot). The interaction in the honeypot is used to create behavioral profiles from the anomalous requests. The authors suggest that analyzing these behavioral profiles can lead to finding discriminants that make anomalous requests detectable [10]. Chew et. al. report they successfully use anti-CSRF tokens woven into their system with AOP as low interaction honeypots to detect a malicious request and redirect the client to a high interaction honeypot.

Mahjoubi extended the work of Hermosillo by using AOP to solve broken authentication and session management. Majoubi et. al. highlights CSRF as a vulnerability that is not yet solved by AOP approaches [15].

Zeller and Felten report on four serious CSRF vulnerabilities in real world applications of The New York Times, MetaFilter, YouTube and ING Direct which are revealed and explained. The vulnerability in ING Direct allowed to transfer funds. They suggest a server side protection mechanism that takes the following precautions:

1. Only allow GET requests to retrieve data.
2. Require all POST request to include a pseudo-random value.
3. The pseudo-random value should be independent to the user's account.

An implementation that automatically includes tokens in HTML Forms for POST requests against CSRF is created by Zeller and Felten for the PHP Code Igniter framework. A client-side protection was implemented as a firefox plugin that intercepts HTTP requests:

1. Non-POST requests are allowed.
2. POST requests with a request and target domain that is within the same-origin policy of the browser are allowed.
3. POST requests with a different request and post domain are allowed if they are within the Adobe's cross-domain policy.

Zeller and Felten show that CSRF is a serious, real-world vulnerability with possible high impact consequences. They expect CSRF and related attacks to become more prevalent unless defenses are adopted [11].

Retrofitting applications with anti-CSRF tokens in PHP

Jovanic et al. give a short explanation on CSRF and discuss that referrer based mitigation techniques are less suited than shared secret (token) based techniques, because of the vulnerability in the referrer header itself to capture the previous request and the problem of dealing with empty referrer headers. The authors suggest a proxy solution in between the web server and the target application. The proxy processes a request to validate possible tokens and processes a reply to insert tokens for request triggers in an application non-specific manner, but only if a session exists or a session ID is present on the request. Their solution is, of course, nullified when the target application is vulnerable for XSS attacks, since the secret token can be stolen by an attacker. The suggested solution is implemented for PHP applications by using PHP, Java, a sed script and Apache alias configurations. The implemented solution is tested against 7 popular PHP applications with 5 applications being vulnerable to CSRF attacks. The implementation proved to be effective without breaking functionality of the test applications. Validations were manually executed and verified [12]. Jovanovic et. al. showcase a solution that doesn't require target applications to be adjusted and is expected to work with various applications. While it is a software engineering solution, no software engineers are required to install the security measures. The authors expect a configuration time of 5 minutes per application. This research and experiment is implemented differently and concerns other implementation techniques, but share the same goal as our research of creating a simple deploy-able solution that works for various applications without adjusting those applications.

Adjusting HTTP to mitigate CSRF vulnerability

Barth et al. introduces the notion of "Login-CSRF". Many mitigation techniques are bound to the session and won't work against login CSRF, since a session might not exist at login-time. They argue that the secret validation token mitigation technique is not well implemented, among which they refer to Jovanovic's solution not being able to protect against Login-CSRF. Instead, the Strict Referrer solution for HTTPS connections is preferred. Since the Referrer header is absent in a significant percentage of requests over HTTP, Barth et. al. suggest the implementation of the 'Origin' header which is less privacy intrusive than the Referrer header and is therefore expected not to be blocked by network devices and is preferred for HTTP traffic as opposed to secret tokens that might get leaked [18].

Chapter 4

Proposed solution

4.1 Token based solutions

This section describes our proposed solution in two parts. The first part describes how the token validation is applied to the applications that need to be protected using the AOP implementation of AspectJ. The second part describes proposed token injection methods.

4.1.1 Token validation

Concept

Token based synchronisation (see section 3.2) is recommended in the context of a developer who identifies sensitive operations when developing the web application. Within the context that no developer executes these identification tasks, a system could simply require secret tokens for every request. This context is applicable when a third party application, without available source code, has to be protected. A requirement for such a rigorous implementation, is that every possible request defined by the web application - that's intended to be handled by the web application itself - is equipped with the secret token.

If the application is vulnerable for Cross Site Scripting (XSS) injections, the token can be stolen and used by an attacker in a subsequent CSRF attack. In order for the token based solution to work, the application mus not allow XSS attacks.

The initial request for the web application will not carry a token, because the application hasn't yet provided the client with a token. Therefore certain pages, such as welcome pages, don't require a token. These requests are referred to as 'landing pages' and are free of token validation. Pages that need to be referenced from outside the application can be defined as landing page as well.

The following list supports the image in figure 4.1 that shows the execution process of this token validation concept.

- The user's client, the web browser, requests a 'landing page' (a page request that does not require a token). In figure 4.1 this is indicated with a request for "welcome.jsp".
- The security aspect allows the request to be processed, since the request path is a landing page and does not require a token.
- The response to this request is a webpage that has added tokens (as an extra parameter) to all URLs.
- When the browser uses such any URL to perform its next request, the token is send along with the rest of the request, as can be seen after the second request in figure 4.1.
- After successful verification of a token, the web application will regard the request as genuine usage of the application and continues regular processing.

If no valid token is present as extra parameter and the request is not directed at a landing page, the aspect terminates processing and returns a HTTP 403 Forbidden message. This has not been displayed in the process, but a similar example has been given in figure 3.2b.

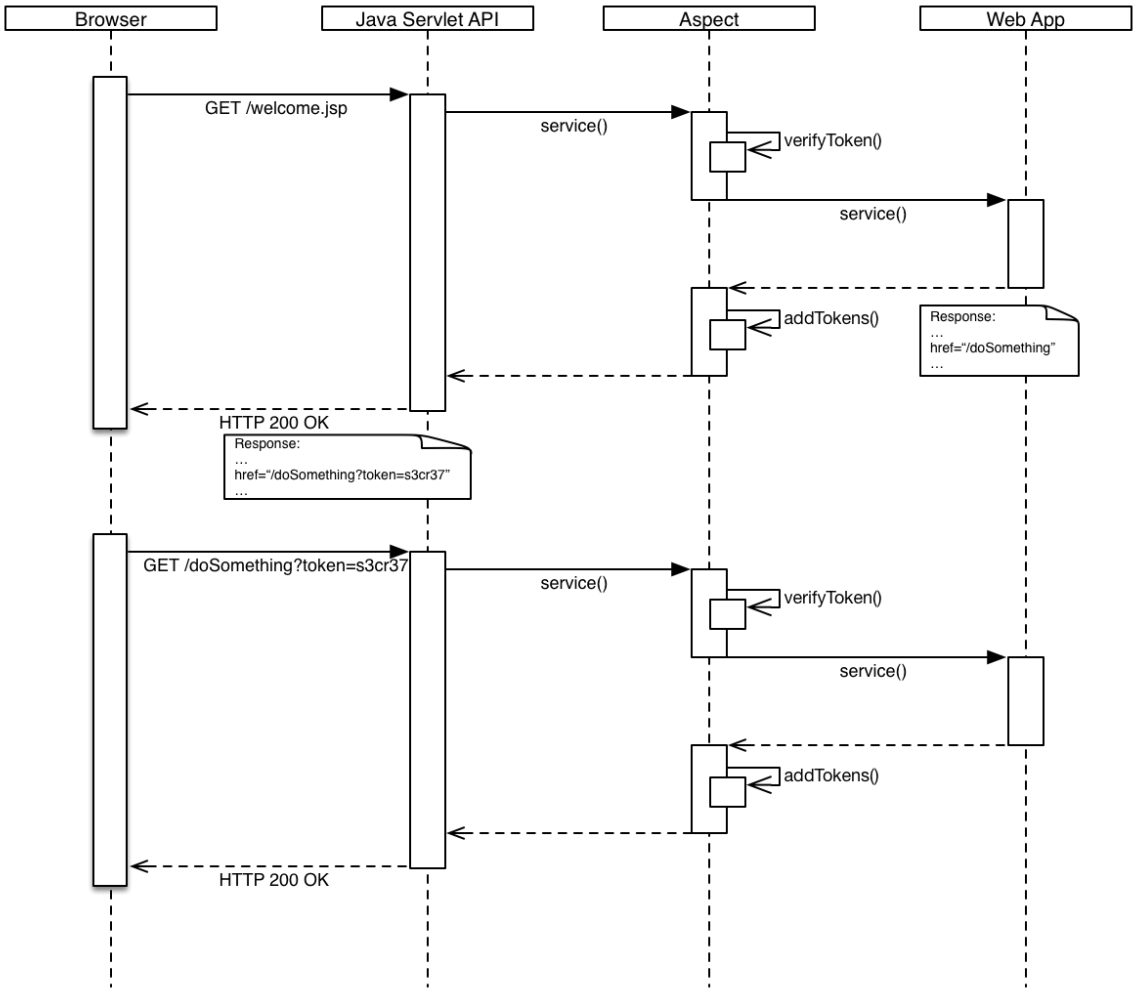


Figure 4.1: UML2 sequence diagram of automatically validating tokens using AOP

Implementation

Requests need to be supplied with a valid secret token parameter. Therefore these tokens are generated and injected in the response that is send to the client. The incoming request parameters (containing both POST and GET parameters) are searched for the anti-CSRF token and looked up in a set of generated tokens that has been cached on the server-sided session. If the token value is not present in the cached set, an exception is thrown which eventually results in a HTTP 403 Forbidden response message and the aspect-advice doesn't let the program proceed in handling the request.

A pointcut on every subclass of the Java Servlet's `Filter` interface addresses an around advice for it's `doFilter(ServletRequest, ServletResponse, FilterChain)` method. The same holds for the `Servlet` (sub)class' `service(HttpServletRequest, HttpServletResponse)` method and the `RequestDispatcher`'s `forward(ServletRequest, ServletResponse)` method. If applicable, the `ServletRequest` and `ServletResponse` are cast to their http subclass (`HttpServletRequest` and `HttpServletResponse` respectively). The session reference from `HttpServletRequest` is stored in a thread local variable to be used in case of possible redirects (see 4.1.1). Because every `doFilter`-method is advised, a flag is set that

is referred by the request in order to know whether the token was already generated and stored for that request.

The incoming request parameters (containing both POST and GET parameters) is searched for the anti-CSRF token and looked up in a set of generated tokens that has been cached on the server-sided session. If the the token value is not present in the cached set, an exception is thrown which eventually results in a HTTP 403 Forbidden response message and the advice doesn't let the program proceed in handling the request. If the token is present in the cached set, a new token for subsequent requests is generated.

For situations in which an application sends a HTTP 302 redirect, the `HttpServlet`'s `sendRedirect(String, ServletResponse)` is intercepted with an around advice. A token is generated and added to the already available cache or newly created cache which is referred by the server-sided session. Since the `sendRedirect(String ServletResponse)` method is not called with a(n indirect) reference to the session, the security aspect holds a thread local variable that refers to the applicable session. This variable has been set by an advice at the moment of request processing in either the `Servlet` or `Filter` and is bound to the thread that is serving the client.

4.1.2 Token injection

Response wrapping

A token is injected when the application does a method call that writes an URL in its response. Injecting tokens using this method is based on Apache Tomcat's CsrfPreventionFilter[1]. The token in the URL is send as a parameter when the client sends out the resulting request. Only URLs for references within its own domain are in need of the token. Tokens that are added to urls for any other domain cause leakage of the secret token.

The Java Servlet API describes a `Response` object that contains response data that is send back to the client. In many situations when a Servlet creates character text for its response that contains an URL, the `Response`'s `encodeURL(String)` method should be used[2].

Figure 4.2 shows a similar process as in figure 4.1. The difference resides in the specific method of token injection. The security aspect does not alter the return value directly, but changes a variable that is used when the web application is called during request processing. This variable now references another piece of code that wraps the original piece of code (the `HttpServletResponse` object) and adds a token to the return value of the `encodeURL()` method as soon as it is called by the web application directly or by the JSP-engine[3] when it resolves the JSTL `url` tag. With this method the security aspect does not change the web application's response directly, but alters some values during the request phase in order to affect the response.

[1] http://tomcat.apache.org/tomcat-6.0-doc/config/filter.html

[2] http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html#encodeURL(java.lang.String)

[3] An engine that transforms JSP code into a Java Servlet (the Jasper 2 JSP Engine for example[4])

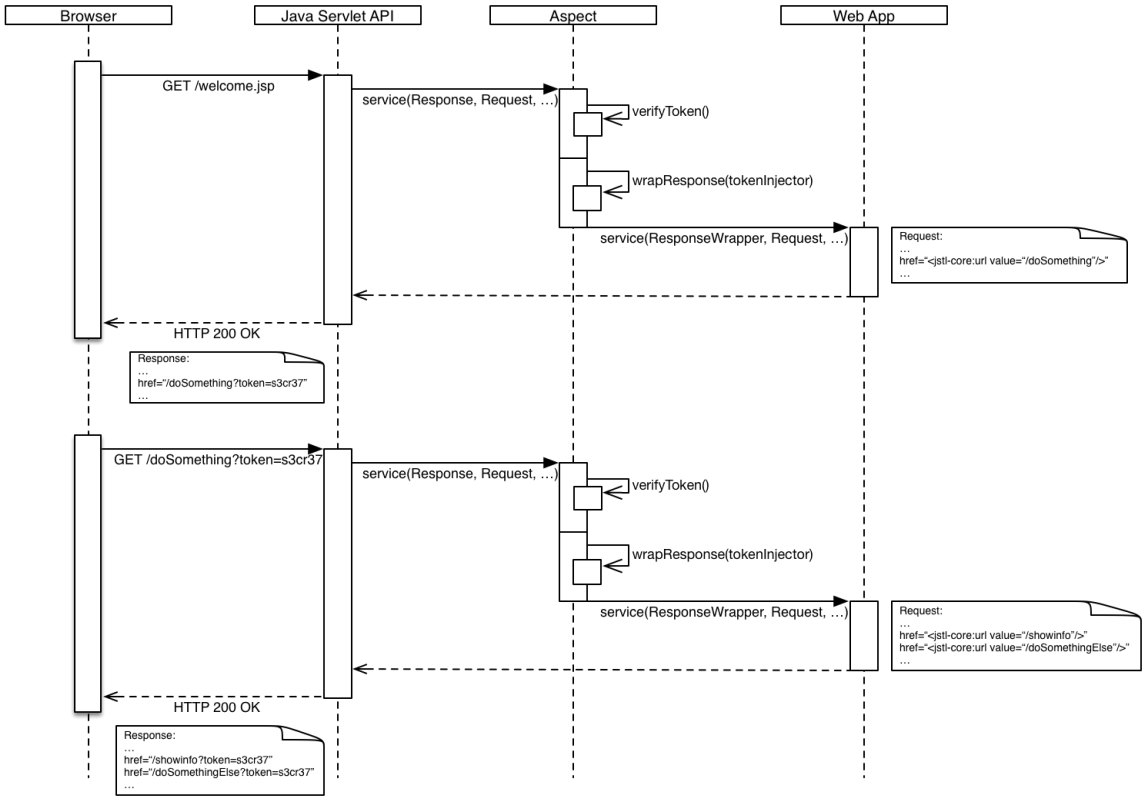


Figure 4.2: inject tokens using a ResponseWrapper

For applications that use Java Server Pages, the JSP Standard Tag Library `url` tag implicitly calls `encodeURL(String)`. AspectJ is used to create a pointcut that intercepts the Servlet's request handling with an around-advice. The `HttpServletResponse` is replaced by a `HttpServletResponseWrapper` that adds these tokens to urls that result from a call to `HttpServletResponse`'s `encodeURL(String)` method.

4.2 Automatic aspect weaving

Our AOP-CSRF defence system consists of a system environment in which unsecured applications are automatically secured by means of using AOP. In order to automatically register the AspectJ Load-Time Weaver Agent as soon as a JVM is launched when the `java` program is called, we use the system environment variable JAVA_TOOL_OPTIONS to supply the `-javaagent` parameter. The actual security aspect and AspectJ runtime environment are loaded by the Extension Classloader when the contents of Java's extension folder are initialised, in which our security-aspect JAR[5] and AspectJ runtime JAR are located[6]. This process of automatic aspect weaving have been visualised in figure 4.3. A description of this visualisation and more information on setting up the system environment can be found in Appendix 9.2.2 and Appendix 9.2.1 respectively.

[5] JAR stands for 'Java Archive'. It is a package file format that contains Java byte-code and/or other resources for a Java program.

[6] https://docs.oracle.com/javase/tutorial/ext/basics/load.html

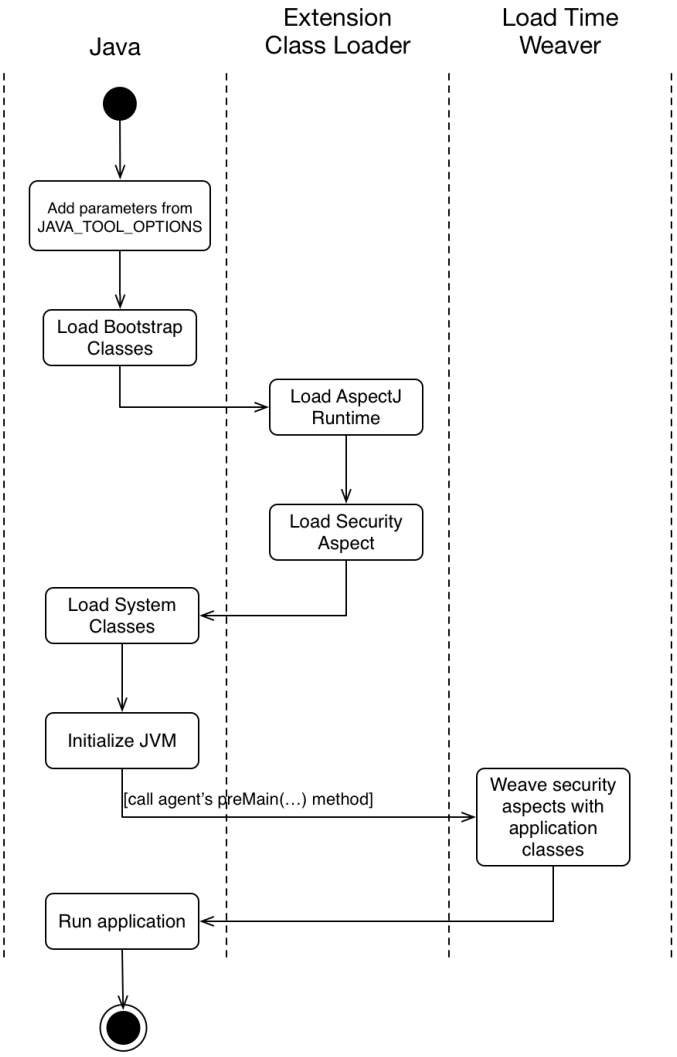


Figure 4.3: Automatically weave aspects with Java application

This chapter elaborated on the research object that has been proposed. The research perspective from which the research object will be observed is described next, in chapter 5.

Chapter 5

Research Method

The proposed solution in section 4 is implemented in order to create a research object. In search of answers to our research questions in section 2.4, we evaluate this research object using two test applications.

5.1 Test applications

The experiments are executed using two test applications that need protection against CSRF:

- OpenKM 5.1.7
- Apache Archiva 1.3.1

These applications suffer from known CSRF vulnerabilities for which reports and source code are freely available online. This can aid us in reasoning about the effects of our proposed solution. These applications are implemented using Java Servlets and are therefore subject to the Java implementation as described in our proposed solution in section 4.

In the domain of software security, an exploit is a method (code, data or command sequence) to cause unintended behaviour or behaviour that has not been anticipated in a software system. Exploits can abuse (exploit) a vulnerability to achieve its goal, like gaining access or rights in a software system or to put the system out of service. We obtained a set of eligible test applications by selecting exploit reports from The Exploits Database[1]and selected applications with reported exploits that specifically exploit CSRF vulnerabilities.

- Archiva 1.3.1[2]
- JForum 2.1.8[3]
- Jira 4.4.3[4]
- OpenFire 3.6.4[5]
- OpenKM 5.1.7[6]
- Tomcat Manager 5.5.25[7]

[1] http://www.exploit-db.com/
[2] http://www.exploit-db.com/exploits/15710/
[3] http://www.exploit-db.com/exploits/13754/
[4] http://www.exploit-db.com/exploits/21052/
[5] http://www.exploit-db.com/exploits/15918/
[6] http://www.exploit-db.com/exploits/18888/
[7] http://www.exploit-db.com/exploits/29435/

The Archiva and OpenKM exploits showed to be severe; gaining administrator access and executing shell commands. Archiva uses a popular J2EE implementation, Apache Tomcat. OpenKM uses another well known J2EE implementation, JBOSS[8]. Since our proposed solution does not advice the implementation itself, but the Java Servlet API as specified in the J2EE specification, we preferred test applications with different implementations of the Java Servlet API. Therefore we choose OpenKM and Archiva as test applications.

In the next two subsections, section 5.1.1 and section 5.1.2, the exploits for the OpenKM admin-module and Apache Archiva are described. The description includes an attack procedure that elaborates on the steps to take in order to execute the attack vector of the exploit.

5.1.1 OpenKM 5.1.3

OpenKM allows administrative users (having the AdminRole) to run shell scripts. An attacker could lure an OpenKM administrator to a malicious web page that causes arbitrary OS commands being run in the administrators OpenKM session context. This is possible because OpenKM does not implement access control mechanisms, besides a valid HTTP session, to validate requests and is therefore susceptible to CSRF attacks. The commands can be executed without the administrator being notified. In the end, this allows an attacker to run OS commands with the privileges of the process owner of the application server[9]. The following attack procedure allows an attacker to place a (malicious) file on the file-system of the application server OpenKM runs on:

Attack procedure

- precondition: running OpenKM 5.1.3

1. Open the OpenKM web application's admin page in a web browser.
2. Login as administrator.
3. Send a request that doesn't originate from the web application, by opening the OpenKM attack page in a new browser tab or window within the same browser instance[10].

- post-condition: a file named "poc" has been created in the filesystem's root /tmp folder.

5.1.2 Archiva 1.3.1

Apache Archiva uses Apache Redback[11] for its user authentication system. It suffers from a CSRF vulnerability, because the admin-user's password can be changed by sending a new password as a POST-request and the only validation is an authenticated session. An attacker can redirect the victim's browser to a page that sends the POST-request, changing the password and gaining admin access to Archiva[12]. The following attack procedure allows an attacker to seize the administrator's account by changing the administrator's password:

Attack procedure

- precondition: running Archiva 1.3.1

1. Open the Archiva web application's landing page in a web browser.
2. Navigate to the login page.
3. Login as administrator with the default username 'admin'.

[8] http://jbossas.jboss.org/
[9] http://www.exploit-db.com/exploits/18888/
[10] The source code listing of the OpenKM Attack Page can be found in Appendix 9.2.
[11] http://archiva.apache.org/redback/
[12] http://www.exploit-db.com/exploits/15710/

4. Send a request that doesn't originate from the web application, by opening the Archiva Attack Page in a new browser tab or window within the same browser instance[13]. Fill in a new password and submit the form.
5. Logout from the web application's user interface.
6. Login again using the password that has been used in step 4.

- post-condition: successfully logged in as administrator.

In order to verify effectiveness of the protection, the post-conditions in the attack procedures (in Archiva the administrator's account is seized by the attacker. In OpenKM an attacker's file has been written to the file-system), are conditions that should not occur if our proposed solution of chapter 4 is enabled. This evaluation of the proposed solution is discussed next, in section 5.2.

5.2 Evaluation

The research object is evaluated on criteria that relate to the research questions in section 2.4 as shown in table 5.1.

Research question	Evaluation criterium
Can the solution successfully block a CSRF attack from executing its desired effect?	Effectiveness
What effects, besides protecting against a CSRF attack, does the solution have on the application's functionality?	Correctness

Table 5.1: Questions related to evaluation criteria

5.2.1 Effectiveness

This criterium is binary, the security measure is effective or it is ineffective. Every test application has a sequence of actions that need to be performed in order to execute the CSRF attack. For every test application a description of these steps is documented in the attack procedures in section 5.1. The attack is executed in a regular, unprotected, environment and is repeated again in the, AOP based, protected environment.

- If the postcondition of the attack procedure in the protected environment is the same as in the regular environment, the protection has been ineffective.
- If the attack procedure's postcondition differs from the postcondition of the attack procedure in the unprotected environment, in such a way that security has not been breached, then the protection is effective.

5.2.2 Correctness

The protection should not break other application functionality. The application needs to yield the same results as it does without the protection. The response send from application server to the browser is seen as application output from its web applications. This output is interpreted by a web browser to be displayed to the user, such as the HTML source code. Therefore we expect to gain relevant information concerning correctness by comparing HTML source code from a web application that is not secured by our security aspect, with HTML source code from the same web application that is secured by our security aspect.

[13]The source code listing of the Archiva Attack Page can be found in Appendix 9.3.

Automated source code harvesting

We created an automated tool to capture the source code. We choose to create an automated tool, because:

- We believe it saves us time and repetitive work.
- We believe that we would miss (subtile) differences in HTML source code, when compared manually without the aid of a comparison tool.
- An automated tool makes it easier to reproduce a test run.

The tool uses a crawler (Crawljax[14]) to request webpages from the application. The crawler can be configured to click hyperlinks in order to retrieve HTML source code text from various DOM states in the crawled application (see figure 5.1 and appendix 9.4).

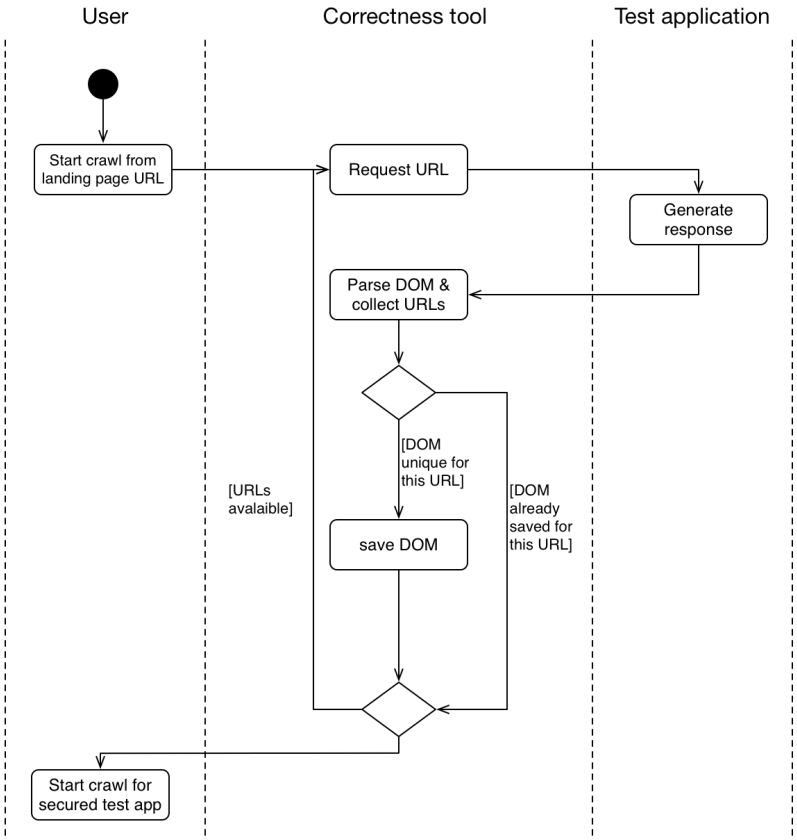


Figure 5.1: Accumulate DOM's from test application

When the test application runs in the AOP based secured environment, the result is manipulated by injecting the response with secret tokens that are not added when the test application runs in the regular, unsecured, environment. This results in an extra challenge for evaluating correctness, since these tokens need to be removed before the source code can be compared without yielding false positives for source code that has been changed directly due to anti-CSRF token injection. False positives can also occur because the harvesting of the DOM states occur at different moments in time for the original and secured application. This can cause elements, that show time, to be of different value. Therefore (see figure 5.2),

[14] http://crawljax.com/

- The tool can remove HTML elements that are indicated to cause false positives, like differing values of timestamps, using the jsoup HTML parser[15].
- In order to prevent false positives, the tool can remove tokens from HTML attributes in source code it saves to disk by using the jsoup HTML parser[16].
- Because failing token injection will not be seen as a change in the DOM source text, it will not be highlighted as changed code. Therefore a missing token identifier is added to identify the failure as well as to count it as a source change. A hyperlink without token won't function, anyway.
- The tool uses GoogleCode's Diff-Match-Patch project[17] to compare two sets of HTML source code text and highlight the differences.

[15] http://jsoup.org/
[16] http://jsoup.org/
[17] https://code.google.com/p/google-diff-match-patch/

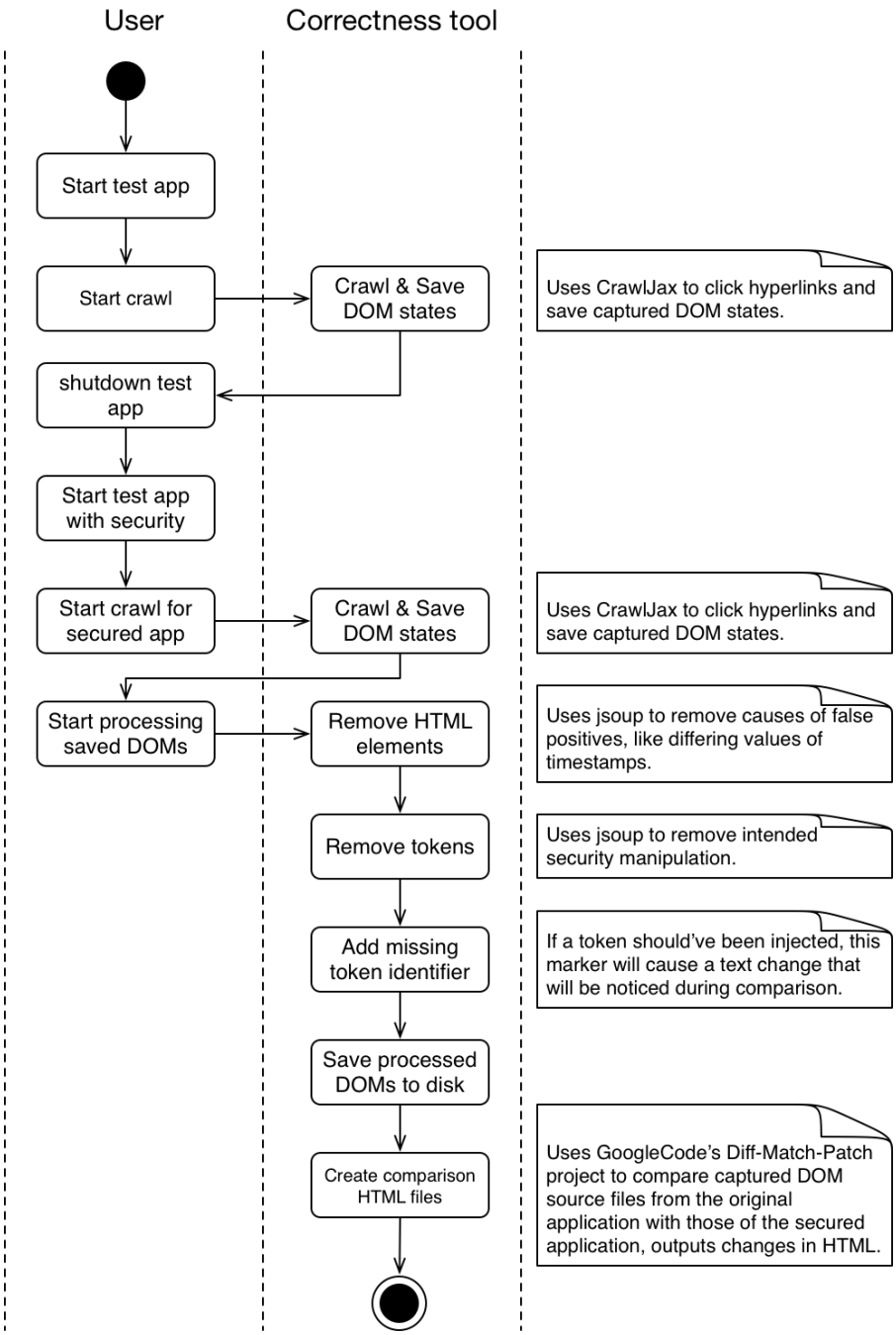


Figure 5.2: Validating correctness using our automated tool

Chapter 6

Results

This chapter reports on the results from the evaluation as described in chapter 5 and ends with a short result analysis in section 6.2.

6.1 Observations

This section is divided in two sections and the research questions from section 2.4 will be answered with result observations at the end of section 6.1.1 and section 6.1.2. Screenshot groups, accompanying result descriptions, consist of the following image types:

- "Original output" is a screenshot from the application running without the proposed security solution of chapter 4. The screenshot content is irrelevant, but does become relevant in comparison to the next type;
- "Secured output" is a screenshot from the same situation as in the "unsecured application", but in this situation the security solutionhas been enabled. Relevance is found in noticeable differences when compared to the original output of the unsecured application.
- "Changed output" is only applicable to correctness evaluation and highlights the differences in the response of the unsecured and secured application at the DOM code level. A screenshot group may consist of multiple images showing changed output.

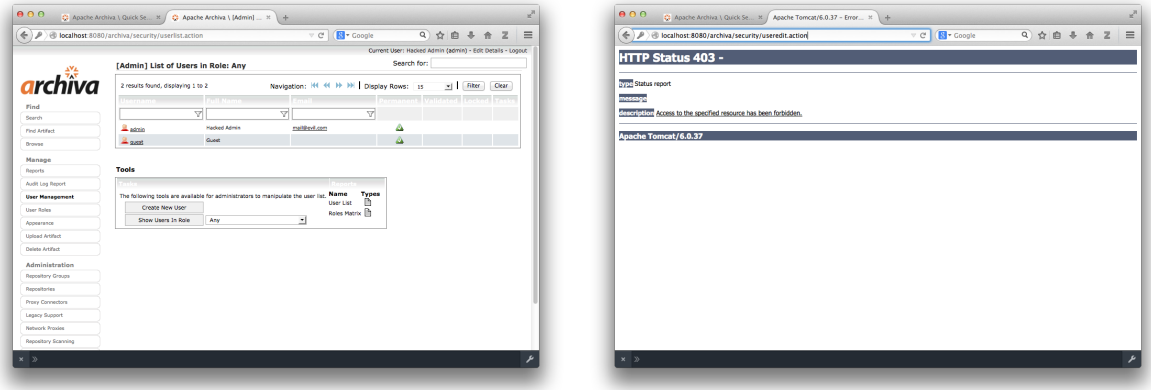
6.1.1 Effectiveness

The effectiveness was evaluated by penetrating the original application and an attempt to reproduce the same penetration by taking the same actions on the application running in it's AOP secured environment. Both our test applications, Apache Archiva 1.3.1 and OpenKM 5.1.3, were effectively protected against our attack by running them with our anti-CSRF-aspect weaved at load-time.

Archiva

Executing Archiva's attack procedure from section 5.1.2 and submitting the request from the Archiva Attack Page in listing 9.3 in an unsecured execution environment, returned the response in Figure 6.1a. Subsequently it was possible to use the attacker's password to log in as Administrator.

Submitting the same request from the Archiva Attack Page in listing 9.3 within the AOP secured environment, resulted in a 403 Forbidden response as shown in Figure 6.1b. Subsequently we weren't able to use the attacker's password to log in, while the original's Administrator's password did grant access to log in to the system.

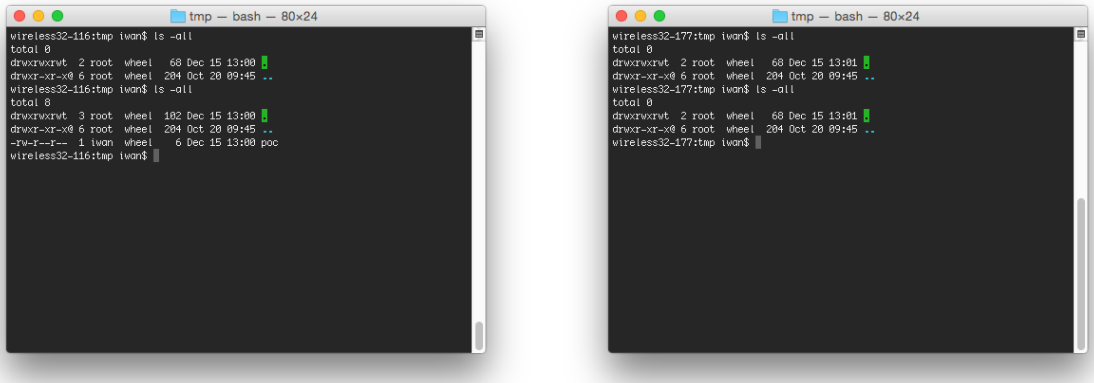


(a) unsecured application (b) secured application

Figure 6.1: Web response after submitting CSRF request

OpenKM

Listing the files in the system's `/tmp` folder before and after opening the OpenKM Attack Page in listing 9.2, resulted in file listings as shown in figure 6.2a. Note the added item "poc" after executing the second list - or `ls` - command, a file has been created on the filesystem. Listing the files in the same fashion as in figure 6.2a and opening the OpenKM Attack Page, but running OpenKM in the AOP secured execution environment, resulted in file listings shown in figure 6.2b. No file with the name "poc" is added to the file listing of this system's `/tmp` directory[1].



(a) unsecured application (b) secured application

Figure 6.2: File system listing during pre- and post-condition of the OpenKM attack procedure in section 5.1.1

[1]The OpenKM Attack Page automatically sends the forged request, without need for the victim to submit it. In Appendix 9.4 a manual attack is showcased using a browser.

Reflecting on the research question

Recalling the predicates to evaluate effectiveness of the CSRF mitigation in section 5.2.1, the following can be observed:

- The result of the attack in the protected environment is not the same as in the regular environment.
- The result in the protected environment differs from the result in the unprotected environment, in such a way that security has not been breached.

Therefore the result for effectiveness of the security measure is regarded as effective. The solution can successfully obstruct a CSRF attack and will not be analysed any further in section 6.2, considering the binary nature of this evaluation outcome (as has been noted in section 5.2.1).

6.1.2 Correctness

The correctness was evaluated by crawling the DOM output of the unsecured test application and comparing it with the DOM output of the secured test application. Ideally, the DOM should remain the same.

- The secured set AST's are traversed to alter `src`, `href` and `action` attributes to remove possible inserted tokens. If no token is found in the value of these attributes, a missing token message is concatenated to the attribute's value to indicate that a token was expected to be found but was found absent instead. If any `<input>` element is found with the element's `name` attribute containing the AOP-CSRF token key, that element is removed from the AST. The element is removed, because of reasoning that such an element, containing the AOP-CSRF token key as `name` attribute, is most likely created because of the AOP-CSRF token injection and therefor needs to be removed.
- Data of `<script>` elements that contain "javascript" within it's `type` attribute value is unescaped for html codes (like `&lt` and `&gt` representing `<` and `>` respectively). Html comments surrounded with `<!-- -->` are removed as well. The remaining Javascript code is parsed using Mozilla's Rhino parser[2]. The resulting AST is inspected for string literals. Any string literal that contains text similar to an URL's query path is searched for the AOP-CSRF token key and, if found, the token key-value combination is removed from the string.

For quantitative measurements we looked at the number of changed lines of code. If our validator expects a token in a url, but none has been found, it modifies the url to indicate the missing token. The resulting line change is a considered a valid change, since a url without token would not function in the protected environment.

In Table 6.1 the results for the quantitative metrics are shown for the tested applications. More detailed, qualitative results are shown using screenshots from the validator output and secured/unsecured output of the test applications in figures 6.3, 6.4 and 6.5.

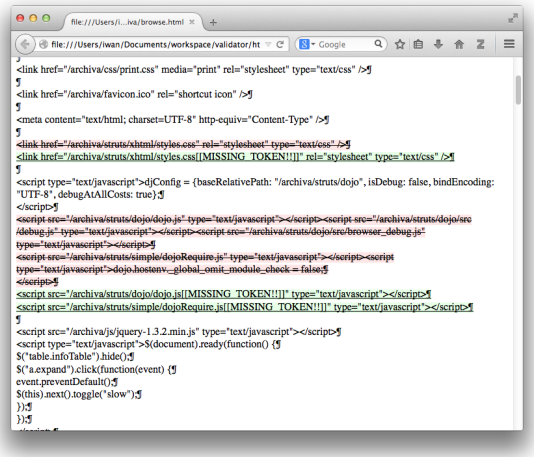
Test application	changed loc's	missing tokens	unsecured states / secured states
Apache Archiva	175	109	24/24
OpenKM admin	16937	50	39/2

Table 6.1: Quantitative results

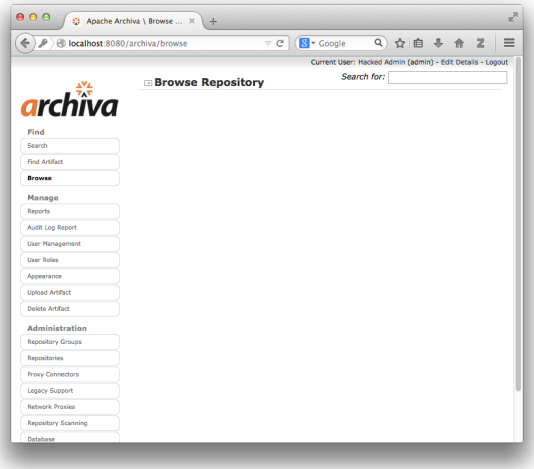
[2] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

Archiva

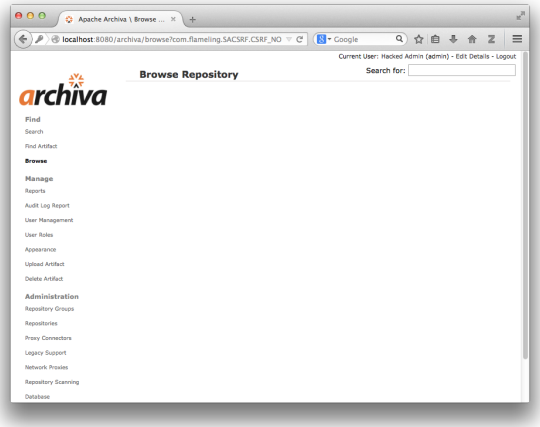
Figure 6.3 shows three screenshots that are related to a request to browse Archiva's artifact repository. The original response is a page that lists the repository contents (in our case the repository is empty), as shown in figure 6.3b. Figure 6.3c shows the response from a similar request that is executed by the validator during time that Archiva was running in the secured environment using tokens injected by the response wrapper. Visually there hardly seems any difference, but some details - like the missing lines around the menu-navigation items - can be noticed. Further inspection using the validator's output, in figure 6.3a, shows that a token is missing in the reference to one of Archiva's Cascading Style Sheets. These screenshots are made after the browser explicitly is commanded to reload it's resources, like the style sheet. Otherwise it is possible that Archiva's style doesn't appear broken, since the style sheet can be loaded from the browser's cache.
Another difference in output, which does not appear visually noticeable, are client-side DOM manipulations that seem to be attributed to the Dojo framework. In the regular environment, Dojo evaluates that it is necessary to change the DOM in order to incorporate debug versions of it's Javascript source. In the protected environment we do not see this change. We do see missing token notifications in references to Dojo resources, which could be the cause that the DOM manipulations do not occur.



(a) changed output



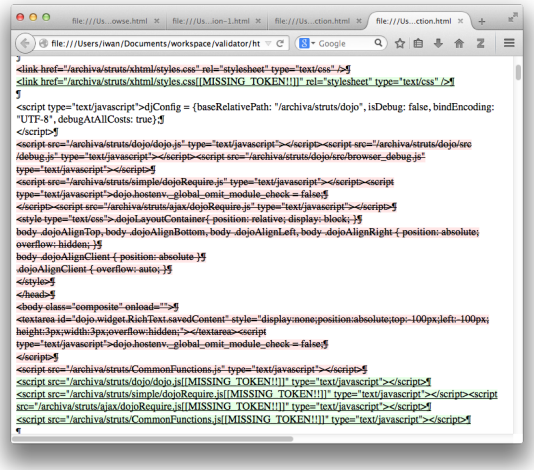
(b) original output



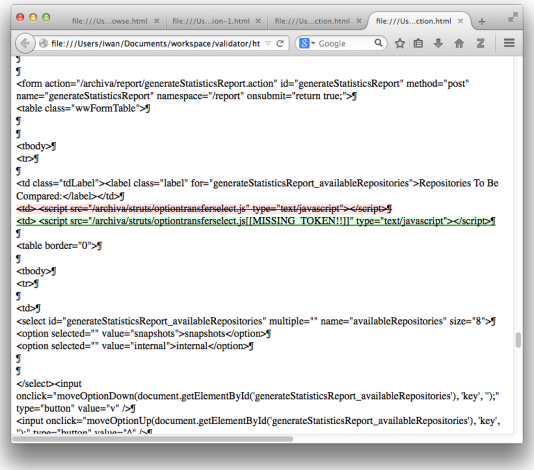
(c) secured output

Figure 6.3: Response for requesting the browse page

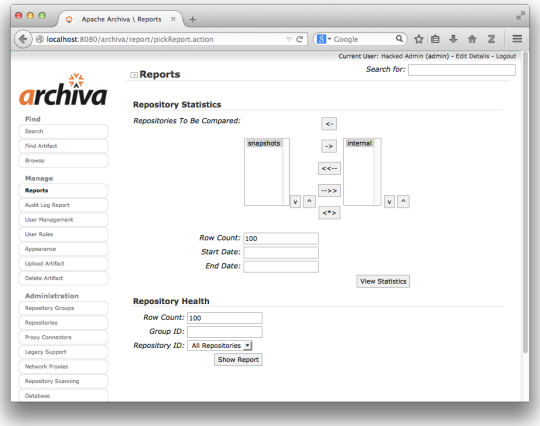
Figure 6.4 shows output that relates to requesting Archiva's reporting page. The validator's output in figure 6.4a and figure 6.4b show that a style sheet reference isn't equipped with a token, which can be noticed in visual differences in figure 6.4c and 6.4d. Besides missing line decoration around the menu-navigation items, differences in alignment and formatting of text are noticeable. More extensive DOM manipulation, that seems related to Dojo, and missing tokens in Javascript references are detected. Besides visual differences, UI functionality seems to be broken. The "option-transfer-select" function is not functioning as it does in the unsecured environment.



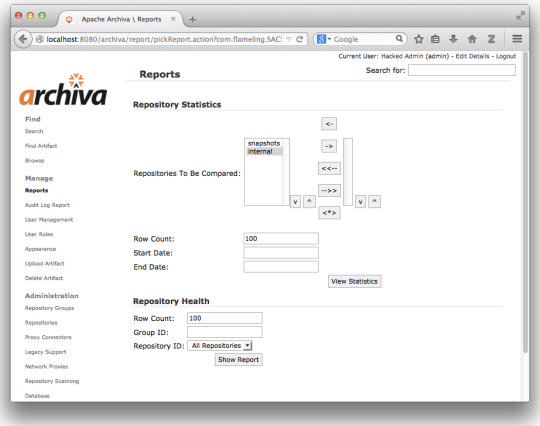
(a) changed output



(b) changed output



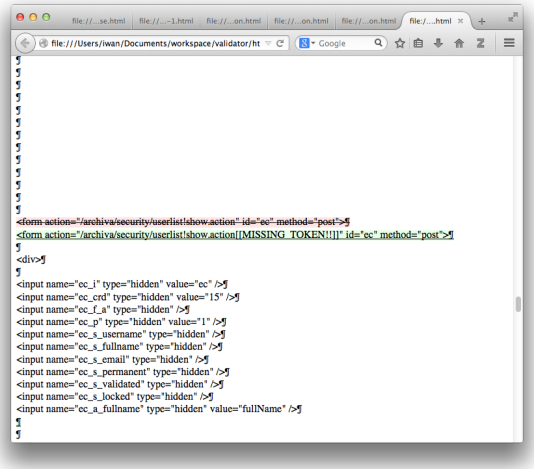
(c) original output



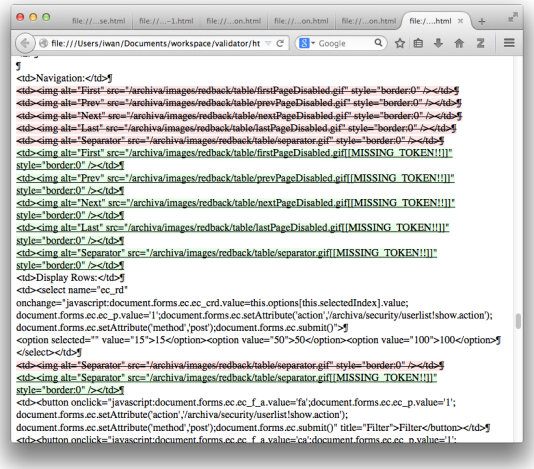
(d) secured output

Figure 6.4: Response for requesting the reporting page

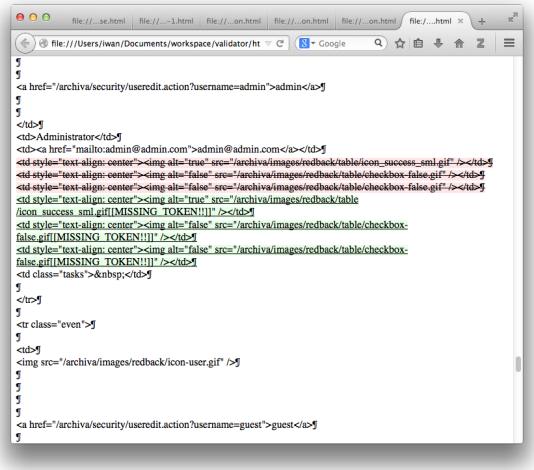
The response to requesting the page for user management is shown in figure 6.5. Missing image-icons for table-navigation in figure 6.5e relate to the line changes in figure 6.5a and it can be seen that Archiva's grey-like style sheet is not loaded. Missing image-icons for status indication seem to be attributed to missing tokens in figure 6.5c. Notable is the missing token occurrence in figure 6.5a, which breaks functionality for form submission.



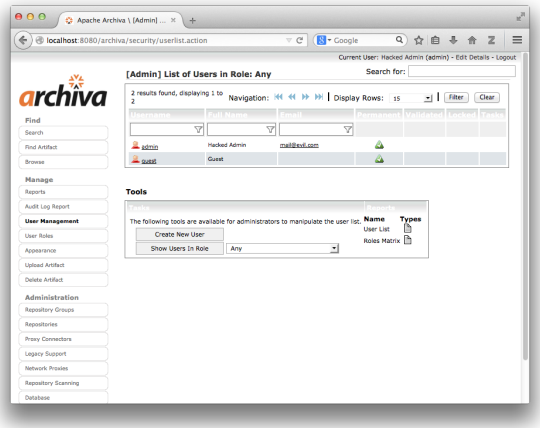
(a) changed output



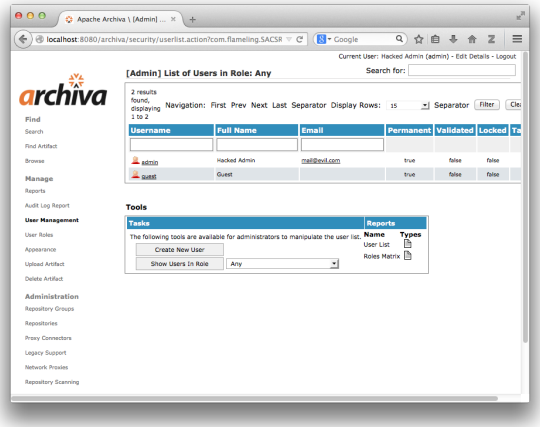
(b) changed output



(c) changed output



(d) original output



(e) secured output

Figure 6.5: Response for requesting the user management page

The Dojo and style sheet situations in figure 6.3a as well as the missing token for the validation file in 6.6, occur multiple times in various responses. For a large part this can be attributed to the use of a decorator that decorates every page with the style sheets and Dojo code.

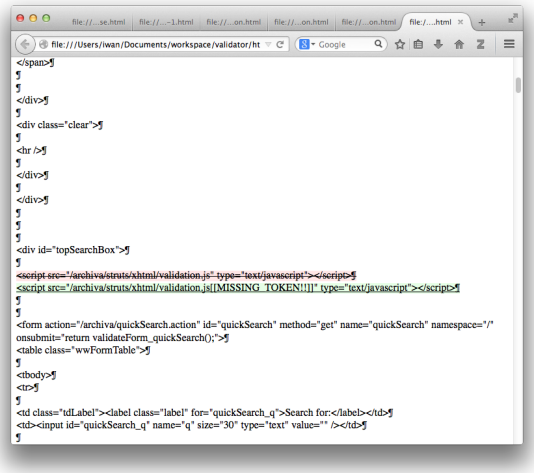
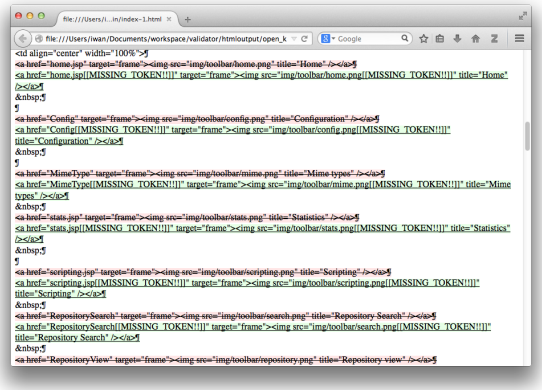


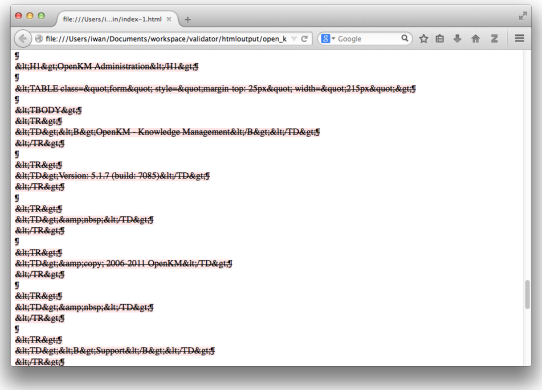
Figure 6.6: Common occurring breakage

OpenKM

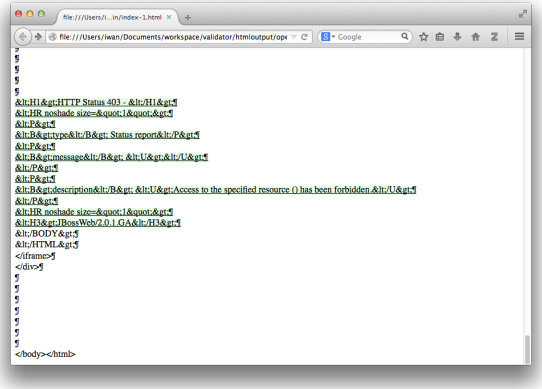
In OpenKM only 2 of the 39 states occur in the secured environment. Looking at the results in figure 6.7 it can be seen that almost all visual components in figure 6.7e are broken. Figure 6.7a only shows a part of all the missing token notifications for the navigation menu. The replacement of the original application's content with that of the HTTP 403 message is shown in 6.7b and 6.7c. Since the secured application in figure 6.7e lacks any navigation mechanism, the validator was unable to crawl the remainder of the application. This also leads to a very high number of changed lines of code in table 6.1.



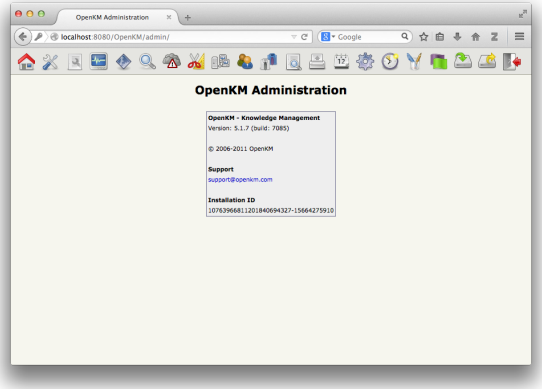
(a) changed output



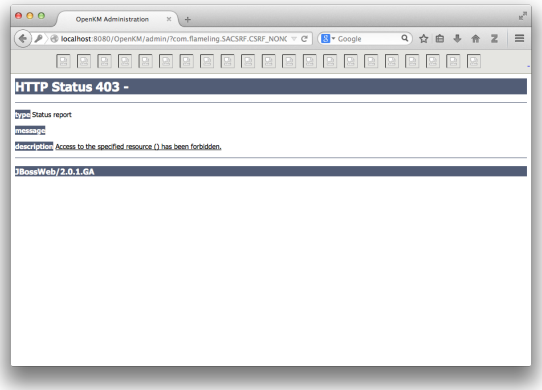
(b) changed output



(c) changed output



(d) original output



(e) secured output

Figure 6.7: Response for requesting the administration main page

Reflecting on the research question

Considering the research question "What effects, besides protecting against a CSRF attack, does the solution have on the application's functionality", the observations show the following effects:

- Visual defacing of webpages.

- Failing DOM manipulations
- Broken (Javascript) functionality.
- Broken (form) functionality.
- Complete visual and functional destruction

In the next section a short analysis will be given with regard to these effects.

6.2 Result analysis

6.2.1 Correctness

Visual defacing of webpages

Cascading Style Sheets do not call the `encodeURL` method, since in many cases it are static resources that are not generated by a Servlet. For example, the visual defacing of the Archiva menu in figure 6.3c is not attributed to the broken import of `styles.css`. It is attributed to a `background-image` reference in `maven-theme.css`. This reference is not equipped with a token and requesting the image without the token is not accepted by the defense-mechanism.

Failing DOM manipulation

The not-occurring DOM manipulations are expected to be related to missing tokens due to failing references of the Dojo framework. Archiva itself is using JSTL's `url` tag but the web application framework 'Struts 2', which is used by Archiva, does not explicitly or implicitly call `encodeURL`. Struts 2 uses FreeMarker templates for its JSP Tags that doesn't make use of the `encodeURL` method but does create links to refer to the Dojo framework.

Broken (Javascript) functionality

Figure 6.4 shows that the "option-transfer" is not functioning. The missing token in the script-URL that refers to "archiva/struts/optiontransferselect.js" might be the cause since this script will not be returned by the server, but dispatched as a possible CSRF-attack.

Broken (form) functionality

The broken form functionality in figure 6.5 can be related to the action url that misses the anti-CSRF token causing the request to be discarded by the server.

Complete visual and functional destruction

The OpenKM Admin module's JSP source files do use JSTL. However it references links statically, instead of using the JSTL's `url` tag. Causing the token not to be injected and breaking all navigation options, causing complete visual and functional destruction of the web application.

Chapter 7

Discussion

7.1 Threats

7.1.1 Flaws in the research method

Does every request need to be verified?

Requiring tokens for every request might generate unnecessary overhead for requests that do not need to be protected against CSRF attacks, because those requests might not result in any actions that could be harmful. Also, requiring a token for every request, renders the system prone to imperfect token injection mechanisms.

Does the security aspect need to be thread safe?

Our security aspect isn't thread-safe. Storing a session reference for a particular client in an aspect-scoped variable might cause requests from other clients to use that session object. Storing the reference in a thread local variable might eradicate the problem in case of guaranteed thread stickiness for request processing. Binding the session reference variable to a particular thread might cause null pointers or corrupt behaviour when threads are switched during the processing of a request. Apache Tomcat, for example, uses thread pooling, but is it guaranteed that all Java Servlet API implementations will also exclusively reserve a thread for request processing until that process has finished?

Are all possible subsequent requests detected?

Since requests need to be attributed with a token to be processed, subsequent requests need to be equipped with a token. This is implemented by attributing URLs, because URLs are suspected to lead to a request. Are all URLs on the server side detected and are therefore all requests, that are expected to originate from client side of the application, detected as well? Failing in doing so can lead to missing token injections.

Can the `encodeURL` method be trusted to inject a token to every possible subsequent request?

Programmers advice that has been given in the Javadoc-documentation does not imply a guarantee that the advice will be or has been used in applications or application components. Overriding the `encodeURL` method with a ResponseWrapper has AOP taking action at the beginning of the request processing only. Because, until the response is being send, various manipulations might occur; having AOP taking action at the end of the request/response-proces might lead to better token injection coverage. Since the ResponseWrapper token injection implementation requires `encodeURL` to be called explicitly (or implicitly using the JSTL `url`-tag), retrofitting anti-CSRF security to an unknown existing application is likely to fail because of missing token injections to URLs.

Does an environment variable sufficiently enforce aspect weaving?

The binding of the aspect weaving agent depends on an environment variable. Therefore it must be able for system administrators to enforce such an environment variable without applications being able to override it.

7.1.2 Threats to validity

Hidden variables

There could be a hidden variable that effects the post-condition of the attack procedure for the effectiveness evaluation. An example could be:

- A first run of the attack procedure, influencing the post-condition of the second run.

Is the Document Object Model representative for evaluating correctness?

- The output that is send to the browser may not be the only output of the web application. Other output may be calls to web services or sending an e-mail.
- The information concerning correctness that is expected to be gained from comparing HTML source code does not cover any server side execution flow of the process. Nor does it take server side state changes into account.
- The HTML source code might reference Javascript or CSS files, which are not evaluated for correctness, but can cause breakage of webpage functionality, behaviour or visuals.

Does the automated source code comparator evaluate correctness effectively?

- The captured output, compared to all possible output, determines the coverage of the correctness measurements. The coverage, therefore, influences to what extent the measurements give information concerning the actual application correctness.
- If a certain request is referenced by an URL in the unsecured environment and yields a response, while the secured environment does not yield a result for the same request than: all rows of the HTML text are counted as changed lines of code. The "changed lines evaluation metric" is therefore influenced by the extensiveness in lines of code in the response from the unsecured environment. If a request/response is absent in the secured environment than the lines in the response that is present in the unsecured environment determines the impact on the "changed lines evaluation metric".
- Bugs in the tools that are used by the automated source code comparator, as well as in its own source code, may invalid the measurements.
- Imperfect token removal leads to false positives.
  The automated source code comparator removes inserted tokens from `src`, `href` and `action` attributes. If tokens are added to urls in other attributes or elements, they will result in false positives for line changes that might decrease correctness grading.
- Arbitrary changes might occur because of changing DOM manipulation behaviour of client side code execution due to added or missing tokens.
- While the automated source code comparator removes tokens from complete urls in string literals that are detected in Javascript code, it does not remove tokens from urls that are present in strings that result from a concatenation of strings. This, again, might lead to false positives.

Especially because of false positives, in depth human qualitative evaluation must be done on the correctness measurements.

limitations of limited test applications

Only two test applications are used to gather evaluation data. This data has led to information regarding flaws and imperfections in the proposed solution. The data does not guarantee that the proposed solution will perform as bad, or as well, on other applications.

7.2 Questions for further research

Can we improve the correctness results with DOM manipulation techniques?

Using HTTP headers to defend against a CSRF-attack could be an alternative to the synchroniser token pattern [18]. Do we need to abandon the DOM manipulation to inject tokens or can we improve the correctness results with DOM manipulation techniques? If we abandon DOM manipulation, do we also need to change the method of correctness evaluation?

Does a HTML parser yield a better result for injecting tokens?

The results on the correctness evaluation show that many changed lines of code are due to missing tokens. In the case of Archiva, we've seen that the components that miss these tokens are attributed to some of the frameworks or libraries that are used by Archiva and do not call the `encodeURL` method. If we can not trust on the `encodeURL` method being used, would a parser that transforms the server-side HTML result (excluding client side HTML transformations) perform better in adding the anti-csrf token to URLs? Would our evaluation method results show an increase on correctness?

Should we capture, parse and compare CSS and JavaScript files?

Since URLs can occur in referenced Javascript or CSS files, these URLs could potentially miss tokens or could be wrongly transformed by the security aspect. This effect can be seen in figure 6.3c. Should we, therefore, compare the CSS and Javascript code returned from the secured application with that of the unsecured application?

How much does CPU and memory usage increase with server side parsing?

Parsing might be be used to inject tokens to HTML, CSS or Javascript code. Such operations are expected to cause a higher strain on CPU and memory resources. Till what extend will these operations increase CPU usage? How much increase in memory footprint will occur? Till what extend does a possible increase render the system unusable for its intended purpose?

Can javascript inject all requests with tokens?

Results of our experiments show various situations where a token was expected, but wasn't found. Missing tokens seem to be a concern for our proposed solution. We think that parsing and transforming the HTTP Response might yield better results with token injection, but could also lead to significant increase in CPU usage and memory footprint. Can this task be executed by the client machine? Does delegating token injection to the client, result in an increased number of token-injection coverage without a higher CPU usage and memory footprint as expected with server-side token-injection by using parsers? Is it possible to inject tokens to all requests that originated from application usage by using Javascript?
If possible, can we automatically inject that javascript to all rendered pages with a performance impact that is lower than using server side parsing for injecting tokens?

Does a HTTP header solution yield better results as compared to token injection based solutions?

If the anti-CSRF implementation is created using HTTP headers only, there is no need to transform parts of the DOM in order for requests to carry a token and successfully pass token validation to be processed by the target application. Therefore, we expect that due to the absence of missing token

injections such a solution would yield better results for correctness. If the DOM does not need to be altered, parsing of the DOM isn't needed. When parsing is not needed, we expect significant smaller impact on CPU usage and memory footprint. If we would need to add some client side code to the HTTP response, we might still need parsing, which will have its effect on performance.

What protocol changes could be done to support generic CSRF protection for any Java Servlet application?

Adding headers or changing header values might require additional client side code that fulfils this task. What information would need to be present in a HTTP request in order to support CSRF protection?

How does a system, that automatically identifies CSRF weak spots first, score in our evaluation?

Simic and Walden use a static analyser to identify SQL and XSS Injection weak-spots before security measures are automatically implemented [8]. Can a similar approach be implemented for CSRF? We did find the requirement, that all valid requests carry a token, hard to satisfy if we are unable to inject tokens for every possible requests that will be generated by the application. Could Simic and Walden's approach work if only byte-code is available (as apposed to source-code)? Would it show better results in our evaluation, especially concerning our token injection challenges and would our evaluation method still be suitable for this implementation?

7.3 Contribution

7.3.1 Token injection is hard

We have not seen many calls to the `encodeURL` method directly in our test applications. We did find many more usages of the JSTL `url` tag, which is transformed to a call to `encodeURL` during JSP compilation. Requiring the `encodeURL` method to be used for token injection was not suitable for especially OpenKM. OpenKM did use JSTL, but the `url` tag was not used to define URLs for hyperlinks.

7.3.2 Requiring every request to carry a token, makes it even harder

In the Archiva test application the JSTL `url` tag was extensively used and many URLs were injected with our anti-CSRF token. This enabled us to protect many of Archiva's operations against CSRF, including the operation that changes the administrator's password. But our proposed solution requires every request to carry a token and our evaluation of Archiva did show various situations in which a token was expected, but not found. Archiva used various frameworks that generate code with URL's that are not formed with a call to `encodeURL`. Therefore, sometimes hard to notice, malfunctions occurred.

7.3.3 The challenge of retrofitting code with tokens

Given the requirement that every request must carry a token, a challenge lies in supplying these tokens to the requests. We expect that parsing the code, which is responded to the client, delivers a higher token injection coverage than a mandatory call to the `encodeURL` method does. We don't know the performance impact of this method, but expect it to slow down or clog up the server's processing or memory capacity. Even when there would be no problems with processing and memory capacity, parsing is not expected to fully solve the token injection challenge: in our evaluation we were unable to verify token injection for URL's that are not complete as a string literal. The same could be the case for token injection. The static strings can be transformed. While a parser could inject code that transforms runtime data, runtime data itself is not transformed by a (server-side) parser injection method.

7.3.4 Know what you measure during evaluation

The evaluation results we found are limited in what they mean to their intended goals. First of all we compare the DOM of various responses from the test application. Even if the DOM changes don't show side effects of the security measure, side effects or unwanted behaviour might still occur (elsewhere) in the test application. Using test applications as evaluation method might be practical from the perspective of the application of security, but shows merely the results for an instance of applying the security aspect. Any conclusion, based on these results only, can not be generalised. We did not measure correctness, we measured the number of changed lines between the secured and unsecured application. While this might give a hint or indication concerning correctness, it is important to know that it contains distortions (as explained in section 7.1.2) and it is left to debate till what extend these measurements represent true correctness.

7.3.5 Recommendation

During conception of this research we expected that our proposed AOP based solution could successfully be applied to arbitrary test applications. However, the results of our evaluation don't show challenges with the usage of AOP, but more specifically show challenges in retrofitting CSRF defences to unspecified applications. This is why we think it is less important for further research to focus on AOP and instead recommend emphasises on:

- Better token injection coverage.
- Improved evaluation methods.
- Smarter token validation mechanisms.
- Exploring other techniques than the synchroniser token pattern to retrofit applications against CSRF.

Chapter 8

Conclusion

8.1 Retrospect

Our motivation for this research was to address the problem of software consumers who lack knowledge, skills, source code or developer support to protect their systems from CSRF vulnerabilities. Because of this motivation we implemented a solution, based on previous research and the synchroniser token pattern anti-CSRF method, and performed an evaluation. Both the implementation and evaluation have been done to seek insight in whether Aspect Oriented Programming can be used to retrofit multiple, third party, applications with a solution that protects against CSRF attacks. In our evaluation, using two test applications, the implementation blocked a CSRF attack which was otherwise successful. This was the intended effect of the implementation, but we also observed non-intended side effects in our test applications consisting of:

- Visual defacing of webpages.
- Failing DOM manipulations.
- Broken (Javascript) functionality.
- Broken (form) functionality.
- Complete visual and functional destruction.

8.2 Judgement

Looking at these intended and non-intended effects - even thou it blocked a CSRF attack in both test applications - we conclude that the implementation of the solution we proposed can't be used to retrofit multiple, third party, applications with protection against CSRF attacks. Next we will explain how we came to this conclusion.

8.2.1 Motivation

The main question in section 2.4 has been the following:

> "Can AOP be used to retrofit multiple, possibly third party, applications with a solution that protects against CSRF attacks?"

If we base the answer to this question solely on the effectiveness evaluation of our test application we can give a positive answer; Yes, AOP can be used to retrofit multiple, third party, applications with a solution that protects against CSRF attacks. But if we take the correctness evaluation into account, a positive answer would accompany risks of - possibly severe - visual and functional broken webpages as can be seen with our test applications.

From our view, the side effects that occur, in just two applications we've tested with, are to significant. Where some small defacement of webpages might be acceptable for some software consumers, we find it unacceptable for a generic solution. Broken functionality is, in our perspective, also unacceptable as a side-effect of the solution. The OpenKM administration became unusable, entirely.

While - based on our results - we concluded a negative answer, we note that our reply is based on our specific implementation. The solution does not protect against CSRF without ?breaking? the application, but that can?t be generalised or said of any possible solution. The emphases of the challenges we encountered are in the field of retrofitting and application non-specific anti-CSRF solutions They do not seem to reside in the field of Aspect Oriented Programming itself. Other evaluation methods, better token injection coverage, smarter token validation mechanisms or using another technique than the synchroniser token pattern, might lead to other results and therefore to other conclusions. To do so, further research is needed.

Bibliography

[1] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting Security in COTS Software with Binary Rewriting," in *Future Challenges in Security and Privacy for Academia and Industry* (J. Camenisch, S. Fischer-Hübner, Y. Murayama, A. Portmann, and C. Rieder, eds.), no. 354 in IFIP Advances in Information and Communication Technology, pp. 154–172, Springer Berlin Heidelberg, Jan. 2011.

[2] L. H. Rosenberg and L. E. Hyatt, "Software re-engineering," in *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1996.

[3] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe Retrofitting of Legacy Software," *ACM Trans. Program. Lang. Syst.*, vol. 27, pp. 477–526, May 2005.

[4] M. G. Merideth and P. Narasimhan, "Retrofitting Networked Applications to Add Autonomic Reconfiguration," in *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, DEAS '05, (New York, NY, USA), pp. 1–7, ACM, 2005.

[5] G. Hermosillo, R. Gomez, L. Seinturier, and L. Duchien, "Using aspect programming to secure web applications," *Journal of Software*, vol. 2, no. 6, pp. 53–63, 2007.

[6] W. G. Halfond and A. Orso, "AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 174–183, ACM, 2005.

[7] J. Bugeja, "A pragmatic policy-driven xss protection framework," Master's thesis, Royal Holloway University of London, London, 2011.

[8] B. Simic and J. Walden, "Eliminating SQL Injection and Cross Site Scripting Using Aspect Oriented Programming," in *Engineering Secure Software and Systems* (J. Jürjens, B. Livshits, and R. Scandariato, eds.), no. 7781 in Lecture Notes in Computer Science, pp. 213–228, Springer Berlin Heidelberg, Jan. 2013.

[9] V. Shanmughaneethi, R. Y. Pravin, and S. Swamynathan, "XIVD: Runtime Detection of XPath Injection Vulnerabilities in XML Databases through Aspect Oriented Programming," in *Advances in Computing and Information Technology* (D. C. Wyld, M. Wozniak, N. Chaki, N. Meghanathan, and D. Nagamalai, eds.), no. 198 in Communications in Computer and Information Science, pp. 192–201, Springer Berlin Heidelberg, Jan. 2011.

[10] B. Chew, J. Wang, A. Ma, and J. Bigham, "Anomalous Usage in Web Applications," in *Networking and Electronic Commerce Research Conference 2008*, (Lake Garda), MPI-QMUL Information Systems Research Centre, 2008.

[11] W. Zeller and E. W. Felten, "Cross-Site Request Forgeries: Exploitation and Prevention," technical report, Princeton University, Princeton, Oct. 2008.

[12] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *Securecomm and Workshops, 2006*, pp. 1–10, IEEE, 2006.

[13] R. Laddad and R. Johnson, *AspectJ in Action: Enterprise AOP with Spring Applications*. Greenwich, CT: Manning Publications, Second Edition edition ed., Oct. 2009.

[14] E. K. Mece and L. Kodra, "Towards full protection of web applications based on Aspect Oriented Programming," *Global Journal of Computer Science and Technology*, vol. 12, no. 1, 2012.

[15] S. Mahjoubi and S. Ghoul, "Enhancing the Web Application Security Using Aspect-Oriented Programming," tech. rep., Applied Science Private University, Amman, Jordan.

[16] M. Finifter, *Towards Evidence-Based Assessment of Factors Contributing to the Introduction and Detection of Software Vulnerabilities*. PhD thesis, University of California at Berkeley, Berkeley, 2013.

[17] V. Shanmughaneethi, R. Y. Pravin, C. E. Shyni, and S. Swamynathan, "SQLIVD - AOP: Preventing SQL Injection Vulnerabilities Using Aspect Oriented Programming through Web Services," in *High Performance Architecture and Grid Computing* (A. Mantri, S. Nandi, G. Kumar, and S. Kumar, eds.), no. 169 in Communications in Computer and Information Science, pp. 327–337, Springer Berlin Heidelberg, Jan. 2011.

[18] 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, (New York, NY, USA), pp. 75–88, ACM, 2008.

Chapter 9

Appendix A

9.1 Alternate CSRF attack examples

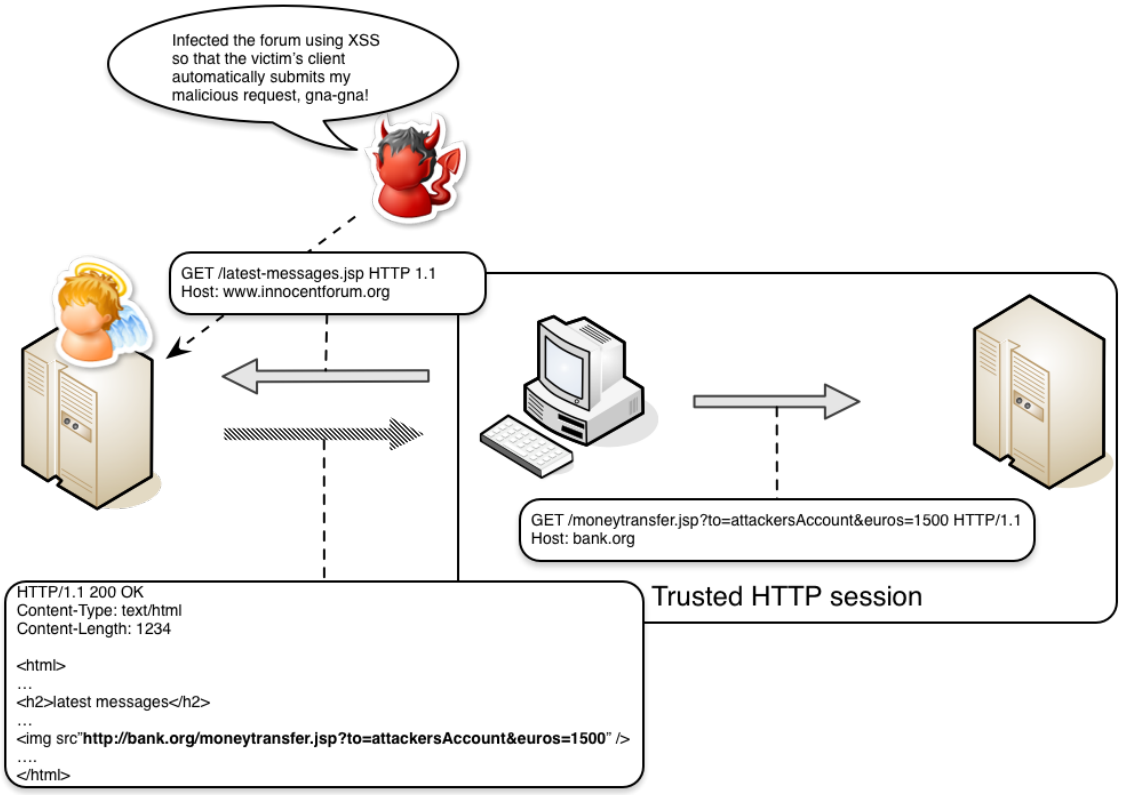


Figure 9.1: Use CSRF to transfer money

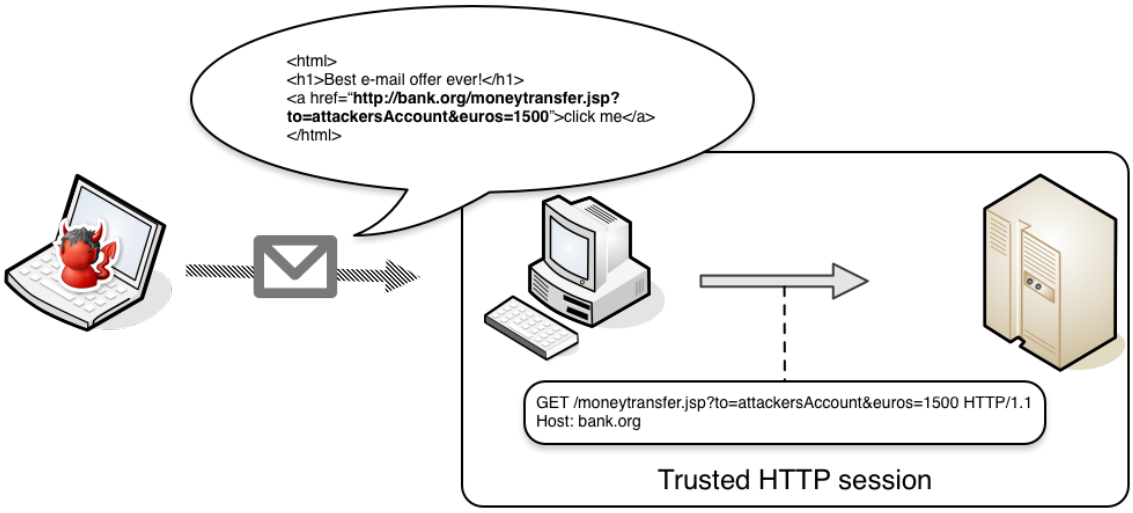


Figure 9.2: Use CSRF to transfer money

9.2 Automatic aspect weaving

9.2.1 Preparation

The following actions have to be taken in order to prepare automatic aspect weaving:

- Place the AspectJ Runtime JAR in Java's extension library directory (`path/to/jre/lib/ext`). When a Java program is started, Java's Extension Class Loader will load classes from the extension directory.
- Set the environment variable JAVA_TOOL_OPTIONS to contain a reference to the AspectJ Load-Time Weaver. How this is done depends on the execution environment. An example for the Born Again SHell(BASH)[1] is given in listing 9.1.

  Listing 9.1: setting environment variable in BASH to automatically bind the Weaver Agent to the JVM

```
1 export JAVA_TOOL_OPTIONS=-javaagent:path/to/aspectjweaver.jar
```

9.2.2 Java program execution

The actions from section 9.2.1 are expected to yield a result in executing a Java program as has been shown in figure 9.3:

1. When the `java` program execution starts it adds the value of JAVA_TOOL_OPTIONS as its parameter. In our case it's the `-javaagent:path/to/aspectjweaver.jar` parameter.
2. The native Bootstrap Classloader loads the Java Runtime.
3. The Java Extension Classloader loads classes from the Java extension directory (`path/to/jre/lib/ext`). This directory includes our security aspect JAR, as well as the AspectJ Runtime JAR.
4. The Java System Classloader loads classes, including classes in JAR files, on paths specified by the system property `java.class.path`. This property defaults to the current directory and

[1] http://www.gnu.org/software/bash/bash.html

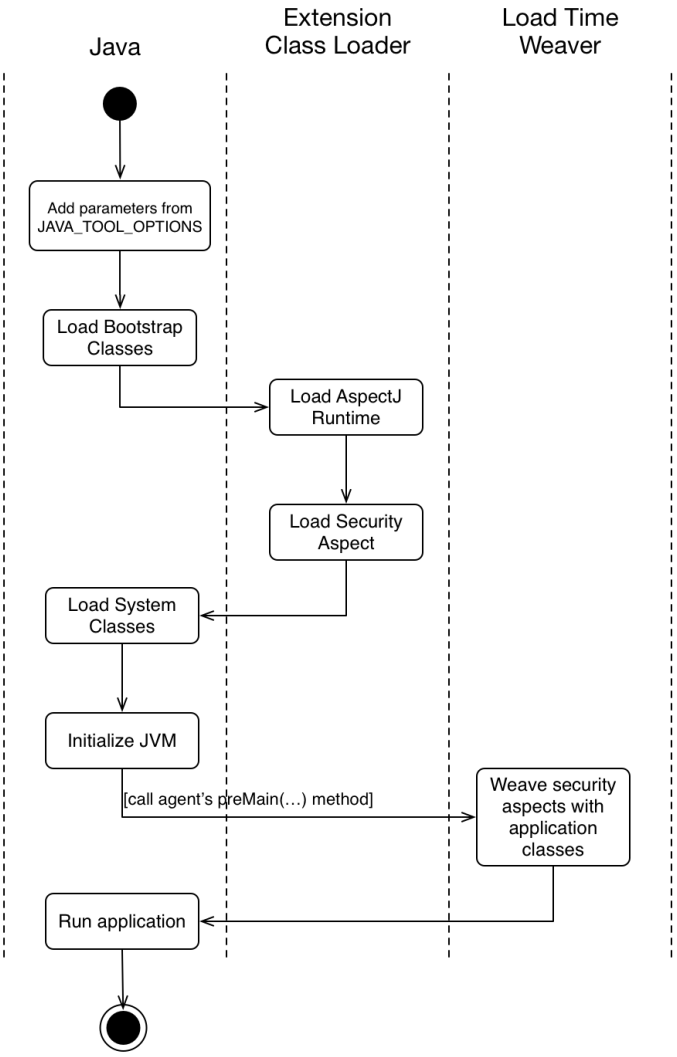


Figure 9.3: Automatically weave aspects with Java application

can be changed by setting the CLASSPATH environment variable[]. It also loads any agent JARs that were referenced by the `-javaagent` parameter. Which is, in our case, the AspectJ Load-Time Weaver.

5. When JVM initialisation has finished, all agents' `preMain(...)` method are called. This enables the AspectJ Weaver to weave the loaded security aspect's advices with the loaded application's classes as defined by the security aspect's pointcuts.

6. When the Java system calls the application's `main(...)` method, the security bytecode is weaved with the application's bytecode.

9.3 Attack Pages

9.3.1 OpenKM admin Attack Page

Listing 9.2: OpenKM Attack Page

```
<html>
<body>
<script>
img = new Image();
img.src="http://localhost:8080/OpenKM/admin/scripting.jsp?script=String%5B%5D+cmd
    +%3D%7B%22%2Fbin%2Fsh%22%2C+%22-c%22%2C+%22%2Fbin%2Fecho+pwned%3E+%2Ftmp%2Fpoc%22%7D%3B
    %0D%0ARuntime.getRuntime%28%29.exec%28cmd%29%3B"
</script>
</body>
</html>
```

9.3.2 Apache Archiva Attack Page

Listing 9.3: Archiva Attack Page

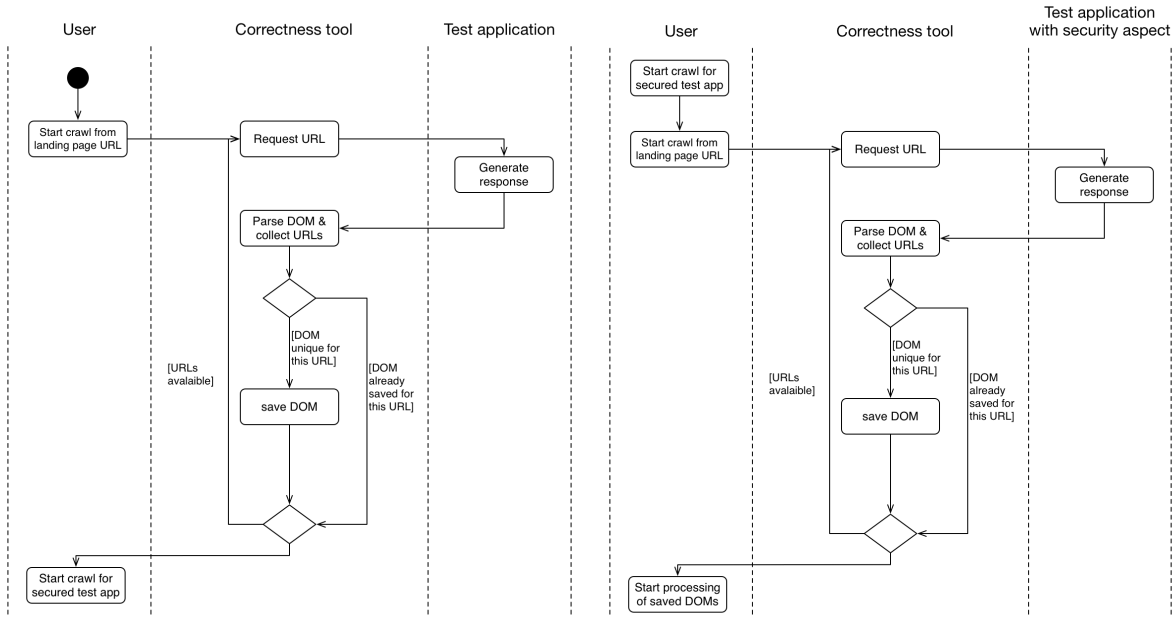
```
<html>
<head>
</head>
<body>
<form id="userEditForm" name="useredit" onsubmit="return true;"
    action="http://localhost:8080/archiva/security/useredit.action" method="post"><table>

<tr>
    <td><label for="userEditForm_user_username" class="label">Username:</label><td>
    <td>admin</td>
</tr>
    <input type="hidden" name="user.username" value="admin" id="userEditForm_user_username"/>
    <tr>
    <td>Full Name:</td>
    <td><input type="text" name="user.fullName" size="30" value="Hacked Admin"
        id="userEditForm_user_fullName"/></td>
</tr>
    <tr>
    <td>Email Address</td>
    <td><input type="text" name="user.email" size="50" value="mail@evil.com"
        id="userEditForm_user_email"/></td>
</tr>
      <tr>
    <td>Password<span class="required"></td>
    <td><input type="password" name="user.password" size="20"
        id="userEditForm_user_password"/></td>
</tr>
  <tr>
    <td>Confirm Password:</label></td>
    <td><input type="password" name="user.confirmPassword" size="20"
        id="userEditForm_user_confirmPassword"/></td>
</tr>
<tr>
       <td valign="top" align="right">
    </td>
    <td valign="top" align="left">

<input type="checkbox" name="user.locked" value="true" id="userEditForm_user_locked"/>
<input type="hidden" name="__checkbox_user.locked" value="true" />Locked User</td>
</tr>
     <tr>
       <td valign="top" align="right">
```

```
38      </td>
39      <td valign="top" align="left">
40  <input type="checkbox" name="user.passwordChangeRequired" value="true"
        id="userEditForm_user_passwordChangeRequired"/>
41  <input type="hidden" name="__checkbox_user.passwordChangeRequired" value="true" />Force User
        to Change Password</td>
42  </tr>
43        <input type="hidden" name="username" value="admin" id="userEditForm_username"/>
44        <tr>
45      <td colspan="2"><div align="right"><input type="submit" id="userEditForm__submit"
            name="method:submit" value="Update"/>
46  </div></td>
47  </tr>
48        <tr>
49      <td colspan="2"><div align="right"><input type="submit" id="userEditForm__cancel"
            name="method:cancel" value="Cancel"/>
50  </div></td>
51  </tr>
52  </table></form>
53  </body>
54  </html>
```

9.4 DOM state harvesting

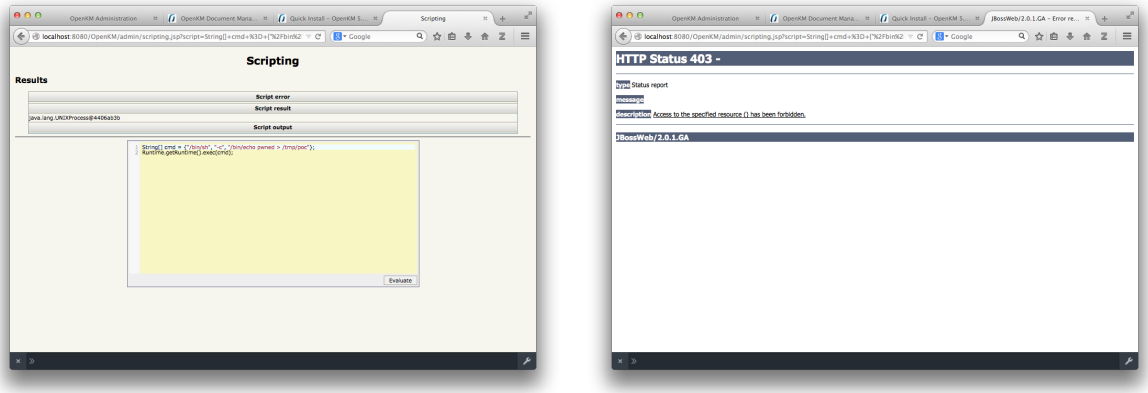


(a) Accumulate DOM's from unsecured, original, test application

(b) Accumulate DOM's from secured test application

9.5 Manual attack on OpenKM in a browser

The OpenKM Attack Page automatically sends the forged request, without need for the victim to submit it. In order to show the different webserver responses, the request is explicitly send using the browser's address bar in figure 9.4.



(c) unsecured application

(d) secured application

Figure 9.4: Web response after submitting CSRF request using the browser's address bar

Chapter 10

Appendix B

10.1 List of abbreviations

- AOP - Aspect Oriented Programming
- COTS - Commercial Of-The-Shelf
- SQL - Structured Query Language
- XSS - Cross Side Scripting
- NASA - National Aeronautics and Space Administration
- HTML - HyperText Markup Language
- HTTP - HyperText Transfer Protocol
- DOM - Document Object Model
- JSP - Java Server Pages
- JSTL - JSP Standard Tag Library
- SQLI - SQL Injection
- CSRF - Cross Side Request Forgery