DeRef: A Privacy-Preserving Defense Mechanism Against Request Forgery Attacks

FUNG, Siu Yuen

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

September 2011

Thesis/Assessment Committee

Professor LUI Chi Shing (Chair)

Professor LEE Pak Ching (Thesis Supervisor)

Professor LEE Moon Chuen (Committee Member)

Professor YAU David King Yeung (External Examiner)

This page is intentionally left blank.

Abstract of thesis entitled:

DeRef: A Privacy-Preserving Defense Mechanism Against Request Forgery Attacks

Submitted by FUNG, Siu Yuen

for the degree of Master of Philosophy

at The Chinese University of Hong Kong in September 2011

One top vulnerability in today's web applications is request forgery, in which an attacker triggers an unintentional request from a client browser to a target website and exploits the client's privileges on the website. To defend against a general class of cross-site and same-site request forgery attacks, we propose *DeRef*, a practical defense mechanism that allows a website to apply fine-grained access control on the scopes within which the

client's authentication credentials can be embedded in requests. One key feature of DeRef is to enable *privacy-preserving checking*, such that the website does not know where the browser initiates requests, while the browser cannot infer the scopes being configured by the website. DeRef achieves this by using two-phase checking, which leverages hashing and blind signature to make a trade-off between performance and privacy protection. We implement a proof-of-concept prototype of DeRef on FireFox and WordPress 2.0. We also evaluate our DeRef prototype and justify its performance overhead in various deployment scenarios.

在現今的網絡世界，其中一種最常被黑客利用的漏洞是僞造請求(Request Forgery)。當中的攻擊者於受害者瀏覽器裏觸發一個HTTP 請求到目標網站，而受害者對於這個請求是全不知情的。當目標網站收到這個請求時，會以受害者的權限去處理，因此攻擊者能以受害者的權限在目標網站上作一些惡意的操作。針對著不同種類的僞造請求攻擊，我們設計了 *DeRef*。*DeRef*能切實讓網站管理員精細地控制他們網站的存取權，同時亦能保障網站及其遊客的私隱。

爲了達到這一個目的，*DeRef*應用了兩階段的檢查機制，當中包括 Hashing 及 Blind Signature 的檢測。最後我們爲 *DeRef*在 Firefox 及 WordPress 上製作了一個原型，以證明 *DeRef* 的成效。

Acknowledgement

First of all, I would like to express my greatest gratitude to my supervisor, Prof. Patrick Lee. During my graduate studies, he works with me shoulder-to-shoulder and spends so much time to improve our work again and again. Although my research road is bumpy, with the guidance from my supervisor, I can keep moving on the right track. The guidance and support from Patrick are not limited to my research. In the beginning of my last semester, I had made one of the toughest decision in my life, in which, Patrick gave me insightful suggestions as usual and acted in concert with my final decision. Therefore, thank you, Patrick. You are my role model.

I am also grateful for the help from my group, Advanced

Networking and System Research Laboratory, especially thanks to Prof. John Lui and Dr. Mole Wong.

Finally, I would like to thank my parents. My mother May Fung for taking care of me and providing me with such a warm family. My father Monty Fung for introducing me to computer science.

Contents

List of Figures

List of Tables

List of Publication

- Ben S.Y. Fung. A Fine-Grained Defense Mechanism Against General Request Forgery Attacks. In Proc. of IEEE/IFIP DSN Student Forum, 2011.

- Ben S.Y. Fung and Patrick P. C. Lee. A Privacy-Preserving Defense Mechanism Against Request Forgery Attacks. In Proc. of IEEE TrustCom, November 2011

Chapter 1

Introduction

Session state management [16] is a critical component in modern web applications. It augments stateless HTTP and embeds *authentication credentials* of web clients into HTTP messages (e.g., in the form of cookies or the HTTP authentication header), so that a website can identify different clients and determine their privileges. However, HTTP session state management is subject to various security vulnerabilities [22]. One such vulnerability is Cross-Site Request Forgery (CSRF), in which an attacker's website triggers a client's browser to send an HTTP request to

a target website. If the HTTP request carries the client's credentials, then the attacker can perform actions on the website using the client's privileges, without the client being notified. There are different variants of CSRF, such as Clickjacking [9] and Login CSRF [3].

There have been extensive studies on how to defend against CSRF (e.g., [3, 7, 13, 15, 17, 21]). One approach is `Referer` checking, in which the target website can determine the complete URL from which the request is initiated. However, the URL information can reveal the access history of the client [3]. A more robust approach is *token validation* (e.g., see [24]), in which the target website embeds secret tokens in HTTP responses, so that the browser can include those tokens in HTTP requests to authorize the request initiations. These tokens are inaccessible by third-party websites due to the same origin policy (SOP) [23]. However, such protection fails if both target and malicious websites have the same origin but are owned by dif-

ferent parties (e.g., `http://www.foo.com/~alice/` and `http://www.foo.com/~trudy/`), as the malicious party can steal the tokens from another same-origin website and trigger forged requests. We call this attack the same-site request forgery (SSRF) attack.

To effectively defend against both CSRF and SSRF attacks, we consider an approach based on *fine-grained access control of scopes*. A *scope* defines a combination of the protocol, domain, and path (see Chapter 3.2). The intuition is that a website can configure, in a policy file, the scopes that are legitimate to initiate or receive sensitive requests that contain authentication credentials. The browser can download the policy file from the website to check the validity of each of its initiated requests, and exclude sensitive credentials from any requests that are considered to be forged. This fine-grained access control is also considered in previous studies (e.g., [7, 21]). However, one shortcoming of this approach is that the policy file carries sensi-

tive scope information in plain format that is accessible by every browser to check against its initiated requests. Users can find out from the policy file how a website designs its access control policy and its trust relationships with other websites. Thus, our goal is to allow the browser and the website to exchange sensitive scope information while they may not need to fully trust each other.

In this thesis, we propose DeRef, a practical defense mechanism against cross-site and same-site request forgery attacks using privacy-preserving fine-grained access control. By privacy-preserving, we mean to not only protect a browser from revealing the URLs from which it initiates requests, but also protect a website from revealing how it configures the legitimate scopes, except for those that have been visited by the browser. The main idea of DeRef is to employ *two-phase checking.* First, the website configures (i) the scopes that are permitted to initiate sensitive requests and (ii) the scopes on the website that are

protected by DeRef. Then the website sends the hash values of the scopes to the browser, where the hash values are incomplete and reveal only *partial* scope information. In the first phase, the browser checks to see if its initiated requests potentially fall within the configured scopes, and eliminate those that are known to be not configured by the website. In the second phase, the browser sends the *blinded* scopes of its initiated requests to confirm if these scopes actually match the configured scopes. In a nutshell, DeRef uses two-phase checking to make a trade-off between performance and privacy protection in real deployment. To our knowledge, this is the first work that aims to build a practical system that addresses the defense against request forgery attacks, while achieving the privacy-preserving property.

To show that DeRef can be feasibly deployed in practice, we implement a proof-of-concept prototype of DeRef on FireFox [18] (as a browser plugin) and WordPress 2.0 [29]. We also address how the prototype is deployable in other web applications and

how it is backward compatible with the original client/server operations without DeRef. We evaluate our DeRef prototype, and show that its response time overhead can be reduced to within 20% by caching the already checked scopes.

The rest of the thesis proceeds as follows. Chapter 2 reviews the background on request forgery attacks and their defense mechanisms. Chapter 3 presents the design and implementation details of DeRef. Chapter 4 discusses several deployment case studies for DeRef and its security effectiveness. Chapter 5 evaluates the performance and scalability of DeRef. Finally, Chapter 6 concludes.

□ **End of chapter.**

Chapter 2

Background and Related Work

2.1 Request Forgery Attacks

A *request forgery attack* is to trigger a forged HTTP request from a victim client browser to a target website without the knowledge of the client. A forged request may carry the client's authentication credentials that an attacker can exploit to perform malicious actions on the website using the client's privileges. In the following, we describe different variants of request forgery attacks.

Cross Site Request Forgery (CSRF) [3, 13, 15, 17]. In

CSRF, an attacker uses an external website to trigger an HTTP request from a client to a target website. Suppose that a client currently has an active session with a target website A and then visits a malicious website B. The attacker can put a malicious URL on website B that triggers the client's browser to send an HTTP request to website A using the currently active session. Then the credentials associated with website A will be attached to the triggered HTTP request, and website A will process the request using the client's privileges.

There are two variants of the CSRF attack, namely *Clickjacking* [9] and *Login CSRF* [3]. Clickjacking puts an invisible frame of a target website on a malicious website. When a client clicks on the invisible frame, a forged HTTP request can be triggered to the target website without the client being notified. Login CSRF is an attack that can be launched even before a session starts. It triggers the client's browser to send a request that contains the attacker's login credentials to the target website.

This allows the attacker to later access the client's information such as the client's activity history.

Same Site Request Forgery (SSRF) [20, 21]. Different websites may have the same *origin* [23] (i.e., same protocol, hostname, and port number), while these websites correspond to different owners. For example, Alice (target) and Trudy (attacker) may individually own websites on the URLs `http://www.foo.com/~alice/` and `http://www.foo.com/~trudy/`. Suppose that a client currently has an active session with Alice's website and then visits Trudy's website. In this case, Trudy's malicious page can read the content in Alice's website, which is permitted under the *same-origin policy* [23]. This is referred to as an SSRF attack. Note that the attack still works even though Alice uses token validation [24], which can effectively defend against CSRF attacks.

2.2 Current Defense Approaches

There are various defense approaches against request forgery attacks. Most of them target cross-site attacks. We also describe the approaches that are based on fine-grained access control, such that they can be extended to defend against SSRF attacks as well.

Header checking. A simple approach is to let the website check the `Referer` header and determine where the request is initiated. However, this approach has privacy concerns, as the `Referer` header reveals the last visited URL of a client from which the request is initiated. To protect a client's privacy, the *origin header* approach [3] introduces the `Origin` header, which is similar to the `Referer` header except that it only contains the origin information with the path details removed. However, checking only the origin information cannot protect against SSRF attacks, in which both target and malicious websites are

hosted under the same origin but on different paths.

Token validation (e.g., [24]). Token validation is now widely deployed to defend against CSRF. The website generates a secret token in a client session, and validates the token when the client initiates requests to perform privileged actions. The token is protected from other websites by the same-origin policy. However, token validation is difficult to implement due to the possibility of leaking the token value [3]. *NoForge* [15] is a server-side proxy that associates secure tokens with active sessions. However, as addressed in Chapter 2.1, token validation cannot defend against SSRF attacks.

Client-side defense. Unlike the above approaches, some studies consider client-side approaches that do not require server-side participation, thereby making deployment easier. *RequestRodeo* [13] is a client-side proxy that strips credentials from a request whose URL has a different origin from the originating webpage. Since it is proxy-based, it cannot examine HTTPS traffic. *BEAP*

[17] is implemented as a browser plugin so that it can examine HTTP and HTTPS traffic. It focuses on inferring the intentions of clients in generating cross-site requests. However, it does not address how to defend against SSRF attacks.

Fine-grained access control. Fine-grained defense approaches allow website owners configure the access scopes from which requests can be initiated. *SOMA* [21] requires a website to set up the policy files that specify the external websites with which the website can communicate, and requires external websites to allow the interactions. The browser can use the policy files to enforce protection. *Csfire* [7] is a browser plugin that parses a fine-grained policy file that specifies which third-party sites can initiate cross-site requests. Other studies, such as MashupOS [10], Subspace [11], and OMash [6], consider more fine-grained access control for cross-site communications in mashup applications. W3C [26] also drafts a specification that states how websites can configure the objects that can be shared across ori-

gins. Although the above approaches focus on protecting against cross-site attacks, we can extend them by configuring the access scopes within the same site to defend against SSRF attacks.

2.3 Lessons Learned

In this thesis, we consider how to use fine-grained access control to defend against both CSRF and SSRF attacks. Similar to SOMA [21], we allow a website to configure a policy file that describes how requests can be initiated and received between a browser and the website. Then the browser uses the policy file to enforce access control. Although this approach is sound, one major concern is that clients can access the policy file and easily determine how a website designs its access control policy and its trust relationships with other websites. Thus, our goal is to design a privacy-preserving approach that can protect the policy information from outsiders, while still effectively defending against both CSRF and SSRF attacks. There are extensive

studies on privacy-preserving mechanisms in different aspects, such as in data mining (e.g., see [1]) and two-party communication (e.g., see [4, 19]). To our knowledge, this is the first work that aims to design a practical system that defends against request forgery attacks from a privacy-preserving perspective.

□ **End of chapter.**

Chapter 3

Design of DeRef

DeRef is designed as a privacy-preserving, fine-grained defense mechanism against request forgery attacks. In summary, DeRef aims for the following design goals.

- *Detecting forged requests.* DeRef seeks to defend against general request forgery attacks, including both cross-site and same-site (see Chapter 3.1).

- *Fine-grained access control.* DeRef enables a website owner to configure the scopes that are under protection, so as to eliminate stringent checking on all incoming requests (see

Chapter 3.2).

- *Privacy-preserving checking.* DeRef can identify forged requests without requiring both the browser and the website to disclose private information to the other side (see Chapter 3.3).

- *Feasible deployment.* DeRef can be feasibly deployed in today's browsers and websites (see Chapters 3.4 and 3.5).

3.1 Threat Model

DeRef seeks to defend against the cross-site and same-site request forgery attacks (i.e., CSRF, Clickjacking, Login CSRF, and SSRF) described in Chapter 2. Specifically, DeRef enables a browser to identify "forged" requests and strip any *authentication credentials* from these requests or their corresponding responses before relaying them.

In this thesis, we focus on two types of authentication cre-

dentials: (i) cookies and (ii) HTTP authentication (i.e., the `Authorization` header). Although authentication credentials can also appear in the query strings of GET requests or in the data in POST requests, their definitions and formats are application-specific and it is difficult to distinguish the credentials from application data. The identification of application-specific credentials will be posed as future work.

To determine if a request is forged, we need to first determine how the request is triggered and where the request is destined for. We define the *initiating URLs* as the set of URLs that can directly or indirectly initiate the request. They include (i) the `Referer` URL and (ii) the URLs of the current active iframe's ancestors in the iframe hierarchy [3]. Also, we define the *target URL* as the destination URL of the request. Figure 3.1 depicts an example of how the initiating URLs and target URLs are defined. We allow a website owner to configure a set of target URLs on the website that are to be protected, as well as a set

of initiating URLs that are "approved" to initiate requests that carry authentication credentials to the protected target URLs (see Chapter 3.2 for details). If a request is sent to a protected target URL from any non-approved initiating URL, then we say that the request is *forged.* For example, in Figure 3.1, if the URL of the target link is protected, then all three initiating URLs (i.e., the URLs of iframe1, iframe2, and the browser tab) must be approved by the website in order for a request to be able to carry authentication credentials; otherwise, the credentials will be removed from the request. Here, we assume that the permitted initiating URLs are benign and no request forgery attacks are launched from there.

3.2 Fine-Grained Access Control

DeRef is built on two access control lists (ACLs), namely *T-ACL* and *I-ACL*, to enable fine-grained defense against request forgery attacks. T-ACL stores the target URLs on the website

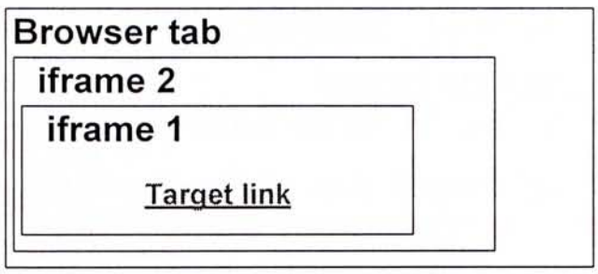


Figure 3.1: Initiating URLs and target URL. Suppose that the target link is clicked. The `Referer` header will have the URL of iframe1. The *target URL* will be the URL of the target link, and there are three *initiating URLs*, including the URLs of iframe1, iframe2, and the browser tab.

those are to be protected. The stored URLs generally correspond to the sensitive web objects that need to respond to the authentication credentials inside the requests, and hence they need protection against forged requests. Other non-sensitive web objects that are not stored in T-ACL will remain unaffected. Thus, a main purpose of T-ACL is to eliminate stringent checking on the non-sensitive web objects.

I-ACL stores the initiating URLs that are trusted to initiate requests to the target URLs configured in T-ACL. A main purpose of I-ACL is to configure the URLs that have different

origins while being trusted (i.e., the same origin policy cannot be directly applicable). One real-life example would be the websites `www.asiamiles.com` and `www.cathaypacific.com`. While they have different origins, they are mutually trusted as they deploy the Single Sign-On (SSO) mechanism [25]. Thus, I-ACL is used to customize the trusted initiating URLs that may have the same or different origins. If *any* initiating URL of a request is not configured in I-ACL, while the request is destined for the target URL that is configured in T-ACL, then the request is considered to be forged.

Scope. Before deploying DeRef, the website on the server side first configures the ACLs with a set of *scopes*. A scope is defined based on the *same origin policy for cookies* [31], and it specifies the range of URLs using `scheme://domain/path`, where (i) the `scheme` corresponds to the protocol of the request (e.g., `http` or `https`), (ii) the `domain` includes the domain itself, its subdomains, and its underlying hosts, and (iii) the `path` includes

the path itself and its path suffixes. To show how a scope is used, let us configure a scope `http://.foo.com/dir/`. Then examples of URLs that match our configured scope are `http://www.foo.com/dir/` and `http://www1.foo.com/dir/sub/`. On the other hand, examples of URLs that do not match our configured scope are `http://www.abc.com/dir/` and `http://www.foo.com/`, since they have a different domain and different path, respectively. Note that a scope can be simply an individual URL.

Creating privacy-preserving lists. The website should keep the ACLs private to browsers to avoid revealing its defense strategy. Instead, it releases the privacy-preserving lists of scopes derived from the configurations in the ACLs, so that the lists will be used in our two-phase checking approach (see Chapter 3.3). The lists will be stored in a *policy file* that is accessible by client browsers.

Publicizing the policy file. The website owner specifies the

base URL, which states the exact hostname and path of the website under which the policy file will be stored. We assume that only the website owner has the write permission to store the policy file under the specified base URL. The base URL will be included in a response message to let the browser know where to download the policy file. Note that a browser may have downloaded multiple policy files from different websites. To choose the policy file for a given request, we use the *longest prefix match* based on the target URL of the request. For example, if the target URL is `http://www.foo.com/~alice/login.php` and there are two policy files with base URLs `http://www.foo.com/` and `http://www.foo.com/~alice/`, then according to the longest prefix match, the browser chooses the policy file with the base URL `http://www.foo.com/~alice/`.

Checking. For each request to be sent to the website, the browser checks the initiating URLs and the target URL associated with the request against the scopes configured in the policy

file. Since a scope may not state the complete URL, we apply *incremental checking* for each URL. The main idea is to check all possible scopes associated with each URL, including all levels of domains starting from the top-level domain, as well as all levels of paths starting from the root path. To illustrate, suppose that we are given a URL `http://foo.com/a/b.html`. Then there are six possible scopes to check: including (1) `http://.com/`, (2) `http://.com/a/`, (3) `http://.com/a/b.html`, (4) `http://foo.com/`, (5) `http://foo.com/a/`, and (6) `http://foo.com/a/b.html`. We then apply two-phase checking on all derived scopes (see Chapter 3.3).

Caching. If a URL has been checked, then the browser can cache the URLs in memory to eliminate checking on the subsequent requests for those URLs. We note that using caching can significantly improve the performance, as shown in Chapter 5.

3.3 Two-Phase Privacy-Preserving Checking

We now present our *two-phase checking* approach that acts as a building block in DeRef. It allows the browser and the website to exchange information in a privacy-preserving manner. It is mainly composed of two phases: *hash checking* and *blind checking.*

Before we describe how our two-phase checking works, let us assume that the website configures L legitimate scopes in an ACL (either T-ACL or I-ACL), denoted by x_i, where $i = 1, 2, \cdots, L$. Now, if the browser initiates a request to the website from URL y, then it checks if y belongs to any of the x_i's, so as to decide whether the request is within the configured scopes. To do this, the browser derives all possible scopes for a given URL y (see Chapter 3.2) into $y_1, y_2, \cdots, y_m$, where m is the number of scopes that are derived from y. Then the browser checks if any y_j (j =1, 2, $\cdots$, m) equals any x_i ($i = 1, 2, \cdots,$

L). Our privacy-preserving goals are: (1) the browser does not reveal y to the website and (2) the browser does not know the x_i's configured by the website, unless a scope of y matches any of these.

Hash checking. In hash checking, the website sends the browser a list of k-bit hashes of the configured scopes, i.e., $h(s, x_1)$, $h(s, x_2)$, $\cdots$, $h(s, x_L)$, where $h(.)$ is a function derived by the first k bits of some one-way hash function, and s is a random salt [28] that is sent alongside the hash list. When the browser initiates a request from URL y, it computes $h(s, y_j)$ (j = 1, 2, $\cdots$, m) and checks if it matches any $h(s, x_i)$ (i = 1, 2, $\cdots$, m). Note that the checking process does not reveal y to the website (i.e., goal (1) is achieved).

The value of k determines the degree of privacy that the website reveals its configured scopes. If k is large (e.g., k = 128 bits as in MD5) and $h(.)$ is collision resistant, then we claim

that it is unlikely for two URLs to have the same hash value[1]. However, having a large k is susceptible to the *dictionary attack*. For example, after downloading the hash list, an attacker can use the popular URLs (e.g., the frequently visited URLs) and the salt s as inputs, and see if the resulting hash values equal any $h(s, x_i)$.

On the other hand, if k is small, then the browser cannot surely tell if a x_i is being configured since there are many false positives that create "noise" to prevent x_i from being fully revealed. For example, if $k = 4$, then there are $2^4 = 16$ possible values of $h(.)$. If $h(.)$ is uniformly distributed, then on average 1/16 of URLs in the entire web can potentially match a $h(s, x_i)$. However, we need to eliminate the false positives through blind checking (see below) to see if URL y is actually within a configured scope.

[1]As of December 2010, the number of indexed webpages in the web space is about 22 billion (less than 2^{35}) [30], which is significantly less than the MD5 space size.

Blind checking. Blind checking is built on the *privacy-preserving matching protocol* [19], which uses Chaum's RSA-based blind signature [5]. We adapt the matching protocol to allow the browser to query the website in a privacy-preserving manner. Specifically, we use the potentially matched scopes returned by hash checking as inputs, and conduct blind checking as follow:

- **Initialization.** The website prepares a RSA public-private key pair (e, d) with modulus n. The public key (n, e) will be sent to the browser. Also, the website sends the list to the browser: $H'(x_i, H(s, x_i)^d \bmod n)$ for i=1, 2, $\cdots$, L, where $H(.)$ and $H'(.)$ are some one-way hash functions and s is the salt value (which is also sent to the browser). We assume that $H(.)$ and $H'(.)$ return a long-enough hash (e.g., 128 bits in MD5) so that it is unlikely for two inputs to return the same hash.

- **Step 1.** For each scope y_j (for j = 1, 2, $\cdots$, m) that

matches any $h(s, x_i)$ in the first phase, it generates a random value r_j and sends the *blinded* hash $r_j^e H(s, y_j) \bmod n$ to the website.

- **Step 2.** The website signs and returns $r_j H(s, y_j)^d \bmod n$ to the browser, which removes r_j and retrieves $H(s, y_j)^d \bmod n$. It then computes and checks if $H'(y_j, H(s, y_j)^d \bmod n)$ equals any signed hashes $H'(x_i, H(s, x_i)^d \bmod n)$.

Since the browser sends only blinded hashes to the website, it does not reveal y to the website (i.e., goal (1) is achieved). Also, an attacker cannot feasibly launch the dictionary attack offline as in hash checking, since it is computationally infeasible to generate the signature of the website for a given input y without knowing the website's private key. Although the attacker can launch the dictionary attack *online* by querying the website with different values of y_j, the attack becomes more difficult than the offline one as it can easily alert the website if the querying rate

is too high. By limiting the query rate of a browser, the privacy of the configured x_i's of the website is also protected (i.e., goal (2) is achieved).

We emphasize that using blind checking alone can still achieve our privacy-preserving goals. A key drawback is that there will be significant process overhead. In blind checking, the browser needs to take a round trip to send every potentially matched scope to the website and have the website sign the scope. Also, each signing consists of an expensive asymmetric cryptographic computation. Thus, we introduce hash checking to ignore any scopes that are guaranteed to be not configured, so as to reduce the overhead of blind checking.

Figure 3.2 summarizes the idea of two-phase checking.

3.4 Putting It All Together

DeRef is implemented on both client and server sides to examine the communication between the browser and the website. We

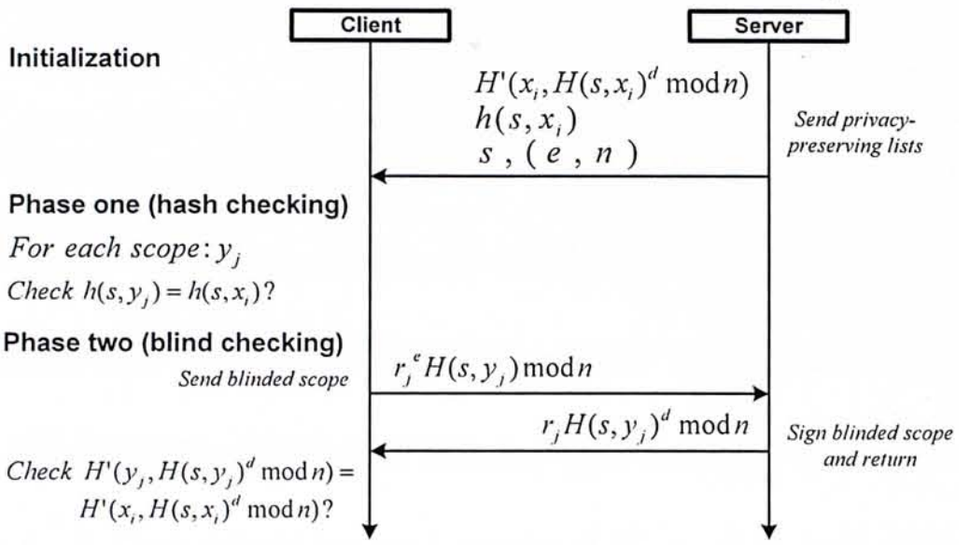


Figure 3.2: Main idea of two-phase checking.

now explain the flow of DeRef and how it enforces protection. Figure 3.3 shows the flow of DeRef.

Start-up. When a user signs in a website, it initiates a login request with valid authentication credentials. Then the website replies a login response, in which the server-side DeRef includes a new header `Protection-Policy`, whose syntax is `Protection-Policy: Last Update Time=[ Time stamp ]; Expiry Time=[ Time stamp ]; Base URL=[ Base URL ]`. This header serves two purposes: to indicate DeRef is implemented in this

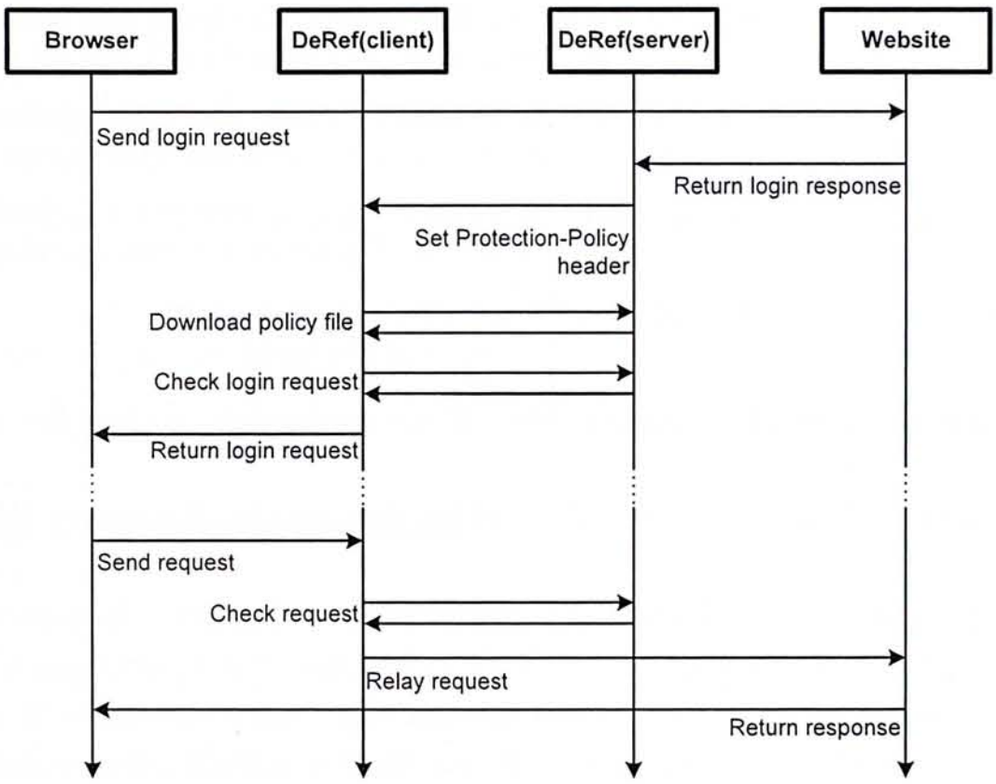


Figure 3.3: DeRef workflow.

website and to state the base URL in which the policy file is stored. Also, the header includes the last update time and the expiry time of the policy file. If the policy file with the same base URL has been downloaded before, while the last update time remains the same and the expiry time is not yet reached, then the client-side DeRef will not download it again.

Downloading the policy file. If no up-to-date policy file is available, then the client-side DeRef downloads the policy file as specified in the base URL and stores it locally. However, an attacker may intercept and modify the policy file when it is being downloaded, for example, by deleting some of the entries in the policy file. To prevent the policy file from being modified, we propose to have it transmitted through HTTPS, which authenticates all message transmissions. Since the policy file is downloaded during the login process, we expect that HTTPS has been enabled by default.

Checking Process. The client-side DeRef performs the two

phase checking on the login request that is previously relayed before returning the login response to the browser, so as to defend against any possible login CSRF attack. For subsequent requests originated from the browser, the client-side DeRef checks the target URLs and the initiating URLs against the policy file. It strips any authentication credentials (i.e., cookies and HTTP authentication headers) from the requests and the corresponding responses if the requests are considered to be forged.

3.5 Implementation

We implement a prototype of DeRef to justify its practicality in deployment. DeRef is built on the components residing on both server and client sides. We now explain in detail the implementation on both sides, and address the deployment issues if only one side enables DeRef.

Server side implementation. The server-side DeRef is implemented in PHP, and hence is applicable in any PHP-enabled

websites. There is a PHP program `genPolicy.php`, which generates the policy file with respect to the URLs defined by the website owner. Here, we use MD5 for hash operations and 1024-bit RSA for blind checking. In addition, we use the `header` function of PHP to specify a new custom HTTP header `Protection-Policy` to indicate the base URL that specifies the locations of the policy file. The browser can retrieve the policy file by visiting `genPolicy.php`. In Chapter 4, we explain via examples how DeRef can be deployed in various real-life server-side web applications.

Client side implementation. We implement a Firefox browser plugin compatible with Firefox versions 3 and 4. It retrieves the policy file from the base URL stated by the server-side DeRef, and inspects any outgoing requests for any forged requests. Our plugin intercepts requests and responses by listening to the events `http-on-modify-request` and `http-on-examine-response`, respectively, both of which are available in the Firefox imple-

mentation. Our implementation of the plugin consists of about 1000 lines of code.

Incremental deployment. DeRef requires the supports of both the client and server sides. If only one side has DeRef enabled, then our implementation is *backward compatible* with the normal operations without DeRef. To elaborate, if the client side implementation is absent, then the browser simply ignores the custom header `Protection-Policy` defined by the server side and will not download any policy file. On the other hand, if the server side implementation is absent, then the browser plugin will find that the custom header `Protection-Policy` is absent and will simply forward all outgoing requests.

□ **End of chapter.**

Chapter 4

Deployment Case Studies

DeRef needs both client-side and server-side deployments. On the client side, DeRef is deployed as a browser plugin, which can be readily included in a browser. On the other hand, the deployment on the server side needs modifications in web applications. It is important that the modifications are minimal to make DeRef deployable. In this chapter, we show via examples that DeRef can be feasibly deployed in today's web applications. We explain how to deploy DeRef in three top open-source content management systems [27], including WordPress

[29], Joomla! [14], and Drupal [8].

4.1 WordPress

We first study the deployment of DeRef on WordPress 2.0. We choose WordPress 2.0 as it has a known CSRF vulnerability [12], which allows us to test the security effectiveness of DeRef in defending against request forgery attacks. Note that we also verify that the modification we make in this version is applicable to the latest WordPress versions as well.

Suppose that Alice wants to host WordPress 2.0 on her personal website `http://www.foo.com/~alice/`, on which she deploys DeRef. First, Alice needs to first configure T-ACL to specify the target URLs to be protected. Here, we include three scopes in T-ACL for WordPress, including:

- `http://www.foo.com/~alice/wp-admin/`,
- `http://www.foo.com/~alice/wp-login.php`, and

- `http://www.foo.com/~alice/wp-comments-post.php`.

The folder `wp-admin/` contains the webpages that manage all WordPress operations, and hence needs to be protected. We include `wp-login.php` so as to defend against the Login CSRF attack by restricting all login actions to be initiated from authorized URLs only. We also include `wp-comments-post.php`, which handles the comments posted by visitors.

Alice also needs to configure the valid initiating URLs in I-ACL to specify where the requests can be triggered to the protected scopes. Here, we assume that Alice includes `http://www.foo.com/~alice/`, meaning that all requests must be initiated from within Alice's website.

Both T-ACL and I-ACL are transformed into a privacy-preserving policy file (see Chapter 3.2). Alice can store the policy file on `http://www.foo.com/~alice/`, from which different browsers can retrieve.

In the following, we use WordPress 2.0 as a case study and

present how each of the request forgery attacks described in Chapter 2.1 is feasible. We then justify why DeRef can defend against these attacks.

CSRF. The CSRF attack is possible in WordPress 2.0 [12], by exploiting the vulnerability that WordPress 2.0 does not validate the origin of the requests. An attacker can host a malicious webpage on, say, `http://www.attack.com/csrf.html`, and trigger forged requests to Alice's WordPress. If DeRef is used, then the client-side DeRef browser plugin will strips all cookies of the requests that are initiated from `http://www.attack.com` as it is not within the scope of `http://www.foo.com/~alice/`. Thus, any forged request will not be processed by WordPress, and the CSRF attack is avoided.

Clickjacking. In the original WordPress 2.0, the Clickjacking attack can work as follows. An attacker hosts a malicious webpage on `http://www.attack.com/clickjacking.html`, which embeds Alice's website `http://www.foo.com/~alice/` as an in-

visible frame. The malicious webpage `clickjacking.html` can instruct Alice to click on different buttons to trigger forged requests to her WordPress. If DeRef is deployed, then the DeRef browser plugin will find that each request contains three initiating URLs, including the `Referer` URL, the URL of the invisible iframe, and the URL of the browser tab. Both the `Referer` URL and the URL of the invisible iframe are `http://www.foo.com/~alice/`. However, the URL of the browser tab is `http://www.attack.com/clickjacking.html`, which is not configured in I-ACL. Thus, DeRef can defend against Clickjacking.

Login CSRF. Login CSRF is possible in the original WordPress 2.0. An attacker can host a malicious webpage on `http://www.attack.com/logincsrf.html`, which triggers a login request to the login page of Alice's WordPress on `http://www.foo.com/~alice/wp-login.php`. The login request includes the login credentials of the attacker. When Alice visits Word-

Press afterwards, she would have been signed in as the attacker. If DeRef is deployed, then before relaying the login response back to the browser (see Figure 3.3), the client-side DeRef inspects that the target URL of the request is `http://www.foo.com/~alice/wp-login.php`, while the initiating URL is `http://www.attack.com/logincsrf.html`, which is not defined in I-ACL. Thus, the attacker's login becomes unsuccessful.

SSRF. The SSRF attack is similar to the CSRF attack, except that an attacker hosts a malicious webpage on `http://www.foo.com/~trudy/ssrf.html`. Although both Alice's website and the malicious webpage are hosted on `http://www.foo.com/`, DeRef can still defend against the SSRF attack because the initiating URLs are restricted by the policy file but not the same-origin policy. Specifically, DeRef can determine that `http://www.foo.com/~trudy/ssrf.html` is not configured within the scope of I-ACL (which includes only `http://www.foo.com/~alice/`). Thus, DeRef will strip off any authentication credentials of the

requests that are initiated from `http://www.foo.com/~trudy/ssrf.html`.

4.2 Joomla! and Drupal

To deploy DeRef in Joomla! and Drupal, we need to address some implementation subtleties that (slightly) complicate the server-side deployment of DeRef, as explained below. Our discussion is based on Joomla! 1.6.3 and Drupal 7.0.

Joomla! The deployment of DeRef requires the scope configurations of T-ACL and I-ACL. In particular, T-ACL specifies the sensitive web objects being protected (see Chapter 3.2). However, in Joomla!, the same URL may correspond to either a sensitive or an insensitive web object, depending on the query strings in the URL. For example, the webpage `index.php` itself simply lists the index page and is considered insensitive. However, the webpage `index.php?task=article.save` may correspond to the article editing function and is considered a sensi-

tive web object. To differentiate between sensitive and insensitive web objects defined by query strings, one can create a new sensitive web object (e.g., `protected/article.save.php`) and redirect the request for `index.php?task=article.save` to `protected/article.save.php`. Then the URL for `protected/article.save.php` can be included in T-ACL.

Drupal. By default, Drupal uses query strings to access web objects. We use the "Clean URLs" function in Drupal to make all web objects accessible without using query strings. For example, the administration page is originally accessed by `/?q=admin`. After enabling "Clean URLs", the relative URL becomes `/admin/`.

□ **End of chapter.**

Chapter 5

Evaluation

We now evaluate our implemented DeRef prototype in real network settings. The client-side DeRef is deployed as a plugin in Firefox 4.0, where the browser is deployed in a desktop PC with CPU 2.4GHz. The server-side DeRef is included in WordPress 2.0, with the same configurations as stated in Chapter 4.1. There are three different entities: a client browser (Firefox), a target website (WordPress), and a malicious website. We deploy all entities in the same local area network of a university department, so as to minimize the overhead of network transmission.

This allows us to focus on evaluating the performance overhead of DeRef.

5.1 Performance Overhead of DeRef in Real Deployment

We first evaluate the performance overhead of our DeRef prototype in real deployment using Firefox and WordPress. Our goal is to understand the overhead of DeRef in surfing different types of webpages. We also evaluate how the use of caching (see Chapter 3.2) on the client-side DeRef improves the performance.

Recall that DeRef uses two-phase checking. Here, we focus on the case where there is no false positive returned by hash checking by setting a large enough value of k (e.g., using $k =$ 128 bits as in MD5). In Chapter 5.2, we evaluate how different values of k affect the performance.

We measure the *response time*, i.e., from the time when the

	Exp. A.1	Exp. A.2		Exp. A.3	
	Index	**Admin**	**Login**	**CSRF**	**Login CSRF**
No DeRef	144.94ms	165.44ms	225.82ms	65.33ms	58.47ms
DeRef (no cache)	159.58ms (10%)	494.77ms (199%)	647.55ms (187%)	108.65ms (66%)	131.6ms (125%)
DeRef (with cache)	160.76ms (11%)	184.08ms (11%)	261.78ms (16%)	77.35ms (18%)	70.26ms (20%)

Table 5.1: Performance overhead of DeRef in different settings.

browser sends the first request until it receives all response messages from the WordPress website. Note that the response time also includes the processing time of performing two-phase checking between the browser and the website. The measurements are averaged over 100 runs. Table 5.1 summarizes the results of our experiments.

Experiment A.1 (Browsing insensitive webpages). We

first consider the case where the browser visits an insensitive webpage that is not under the protection of DeRef, i.e., the URL of the webpage is not configured in T-ACL. Here, we measure the response time when we visit the index page `index.php` on WordPress. Since the index page is insensitive, DeRef does not need to perform blind checking (provided that no false positive is returned in hash checking). Thus, we expect that DeRef incurs minimal overhead. Table 5.1 shows that the additional overhead of DeRef is around 10%, which conforms to our intuition. Note that the performance is similar with or without cache.

Experiment A.2 (Browsing sensitive webpages). We next consider the case when the browser visits a sensitive webpage. In this case, the DeRef browser plugin will perform both hash checking and blind checking, to confirm that the URL of the sensitive webpage is in T-ACL and the initiating URL is in I-ACL. Here, we measure the time when the browser visits `/wp-login.php` and `/wp-admin/` on WordPress from a legitimate initiating

URL.

Table 5.1 shows that both cases incur significant performance overhead, mainly due to the RSA blind signature computation in blind checking. If no caching is used, then the overheads are 199% and 187% for `/wp-login.php` and `/wp-admin/`, respectively. Nevertheless, we can mitigate the overhead via caching, which stores the URLs that are known to be configured in T-ACL and I-ACL. When we visit the two webpages again, the overheads decrease to 11% and 16% for `/wp-login.php` and `/wp-admin/`, respectively.

Experiment A.3 (Browsing malicious webpages). We now consider the case when we visit malicious webpages that trigger request forgery attacks to sensitive webpages. Here, we consider the CSRF and login CSRF attacks, in which forged requests are sent from our malicious website that we set up to the URLs `/wp-admin/` and `/wp-login.php`, respectively. Note that in both cases, the initiating URLs are not configured in I-ACL, so

DeRef only performs two-phase checking to confirm that the target URLs are configured in T-ACL. Thus, the number of URLs to be signed in blind checking is less than Experiment A.2. Overall, the additional overheads are 66% and 125% for CSRF and Login CSRF, respectively, when caching is disabled, and they reduce to 18% and 20%, respectively, when caching is used.

Compatibility study. Note that DeRef is backward compatible with existing websites that do not deploy DeRef (i.e., no server-side deployment of DeRef). To justify this, we enable the DeRef browser plugin and have it visit the top 50 websites as listed on Alexa [2]. We observe that the DeRef browser plugin does not have any incorrect behavior in those visits.

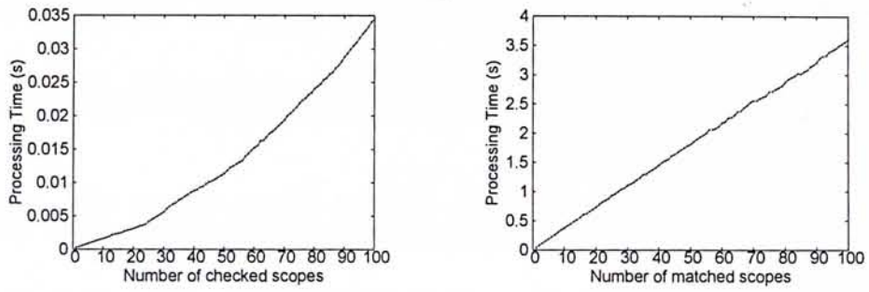
5.2 Performance Overhead of DeRef with Various Configurations

We now study how various configurations affect the performance of DeRef. In particular, we aim to show that DeRef can maintain acceptable performance even under complicated settings.

Experiment B.1 (Scalability study of two-phase checking). We evaluate the scalability of DeRef in performing a large number of checking steps during two-phase checking (see Chapter 3.2). We note that there are two potential performance bottlenecks in two-phase checking. First, we apply incremental checking for all possible scopes derived from a URL, and its performance depends on the number of checked scopes. Second, we conduct blind checking for all matched scopes found in hash checking, and its performance depends on the number of the matched scopes.

We modify our DeRef browser plugin to generate a random

number of scopes and measure the processing times of the two potential bottlenecks. Figure 5.1(a) shows the processing time of incremental checking versus the number of checked scopes. We observe that the processing time increases with the number of checked scopes, and it is within 35ms when the number reaches 100. We expect that this processing time has limited impact when compared to the overall performance in DeRef in real deployment (see Chapter 5.1), where the response time is on the order of 100ms. Figure 5.1(b) shows the processing time of blind checking (i.e., the time from the browser sending the blinded hashes for all matched scopes until the website returning the signed hashes) versus the number of matched scopes. We observe that the processing time increases linearly with the number of matched scopes, and it reaches 3.6 seconds when the number of matched scopes is 100. As shown in Chapter 5.1, the performance overhead can be significantly reduced by caching the already checked URLs.

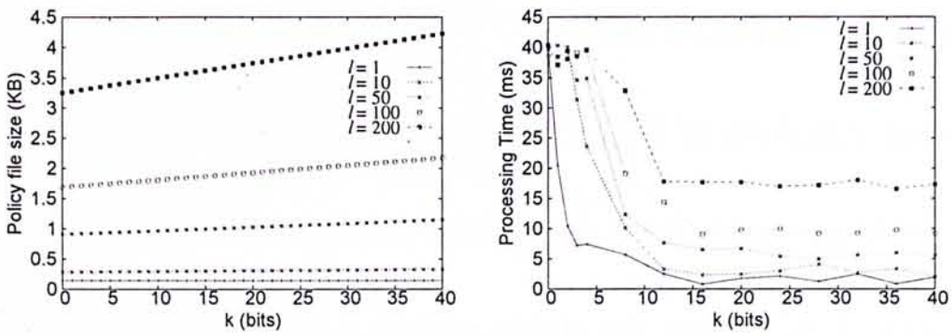


(a) Number of checked scopes in incremental checking

(b) Number of matched scopes in blind checking

Figure 5.1: Experiment B.1: Scalability study of two-phase checking.

Experiment B.2 (Trade-off between performance and privacy). Recall that the performance-privacy trade-off of two-phase checking is determined by the value of k (see Chapter 3.3), which decides how much information is revealed in hash checking. In this experiment, we evaluate the impact of k. We first collect the top 500 website URLs on Alexa [2]. We then configure the first l of the 500 URLs in I-ACL, where $l = 1$, 10, 50, 100, or 200. The configuration of T-ACL remains the same as in Chapter 4.1. We generate 500 requests from our DeRef browser plugin to the WordPress website that we set up, such that each request has its initiating URL hardcoded to each of the 500 col-



(a) Size of policy file (b) Processing time

Figure 5.2: Experiment B.2: Performance versus k for different numbers of URLs configured in I-ACL.

lected URLs. For different values of k, we measure the processing time for performing two-phase checking (i.e., hash checking, followed by blind checking if needed) on each initiating URL between the browser plugin and the WordPress website. We do not include the time of returning the response from WordPress, so the processing time of two-phase checking is less than the total response time that we measure in Chapter 5.1. Note that when $k = 0$, we assume that the browser directly conducts blind checking.

Figure 5.2(a) shows the size of the policy file versus k for

different numbers of URLs configured in I-ACL. The size of the policy file increases with k and the number of URLs being configured in I-ACL, but the size is within 4.5 KB in all cases. Note that the policy file is downloaded once at the start-up phase and is cached until it expires (see Chapter 3.4). Thus, we expect that the policy file itself introduces minimal overhead.

Figure 5.2(b) shows the processing time of two-phase checking. We observe that when k increases, the time used in two phase checking decreases, mainly because hash checking discovers most non-configured URLs and skips the second-phase blind checking. For example, if I-ACL contains only 10 URLs, then the processing time is reduced by 40% from $k = 0$ to $k = 4$. The trade-off is that more information of the configured scopes is revealed with a larger value of k. Another observation is that when the number of configured URLs (i.e., l) increases, the processing time is higher. The reason is that hash checking can only filter non-configured scopes. If more scopes are configured in an

ACL, then more scopes need to be verified by blind checking as well.

□ **End of chapter.**

Chapter 6

Conclusions

We present DeRef, a practical privacy-preserving approach to defending against cross-site and same-site request forgery attacks. DeRef uses fine-grained access control to allow a website owner to decide how requests should be sent and received within protection scopes, so as to prevent forged requests from being initiated outside the scopes. We use two-phase checking as a building block that allows the browser and the website to exchange configuration information in a privacy-preserving manner. We implement a proof-of-concept prototype of DeRef,

and demonstrate that it can successfully defend against request forgery attacks in real-life applications, while incurring justifiable performance overhead. We plan to publicize the source code of DeRef in the final version of this thesis.

□ **End of chapter.**

Bibliography

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. of ACM SIGMOD*, 2000.

[2] Alexa the Web Information Company. `http://www.alexa.com`.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proc. of ACM CCS*, 2008.

[4] S. M. Bellovin and W. R. Cheswick. Privacy-enhanced searches using encrypted bloom filters. Technical Report CUCS-034-07, Columbia University, 2007.

[5] D. Chaum. Blind Signature for Untraceable Payments. In *Advances in Crypto.: Proc. of Crypto*, 1983.

[6] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proc. of ACM CCS*, 2008.

[7] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Int. Symp. on Engineering Secure Software and Systems (ESSOS)*, 2010.

[8] Drupal. `http://drupal.org`.

[9] R. Hansen and J. Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, 2008.

[10] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: operating system abstractions for client mashups. In *Proc. of USENIX HOTOS*, 2007.

[11] C. Jackson and H. J. Wang. Subspace: Secure CrossDomain Communication for Web Mashups. In *Proc. of WWW*, 2007.

[12] M. Johns. Outdated advisory: Code injection via CSRF in Wordpress < 2.03. `http://shampoo.antville.org/stories/1540873`, 2007.

[13] M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In *Proc. of the OWASP Europe 2006*, 2006.

[14] Jommla! `http://www.joomla.org`.

[15] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *SecureComm*, 2006.

[16] D. Kristol and L. Montulli. HTTP State Management Mechanism, Oct 2000. RFC 2965.

[17] Z. Mao, N. Li, and I. Molloy. Defeating Cross-Site Request Forgery Attacks withBrowser-Enforced Authenticity

Protection. In *Financial Cryptography and Data Security*, 2009.

[18] Mozilla. About mozilla add-ons. https://addons.mozilla.org/en-US/firefox/about.

[19] R. Nojima and Y. Kadobayashi. Cryptographically Secure Bloom-Filters. *Transactions on Data Privacy*, 2:131–139, 2009.

[20] Nytro. Using XSS to bypass CSRF Protection. `http://packetstormsecurity.org/files/view/82676/Using_XSS_to_bypass_CSRF_protection.pdf`, Nov 2009.

[21] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *Proc. of ACM CCS*, 2008.

[22] OWASP. Owasp top 10 for 2010. `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[23] J. Ruderman. Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, 2010.

[24] E. Sheridan. OWASP CSRFGuard Project. `http://www.owasp.org/index.php/CSRF_Guard`, 2010.

[25] The Open Group. Introduction to Single Sign-On. `http://www.opengroup.org/security/sso/sso_intro.htm`.

[26] W3C. Cross-origin resource sharing. `http://www.w3.org/TR/cors/`, Jul 2010.

[27] Water&Stone. 2010 open source cms market share report. `http://www.waterandstone.com/sites/default/files/2010%20OSCMS%20Report.pdf`, 2010.

[28] C. Wille. Storing Passwords - done right! `http://www.aspheute.com/english/20040105.asp`, Jan 2004.

[29] WordPress. `http://en.wordpress.com/about/`.

[30] WorldWideWebSize.com. http://www.worldwidewebsize.com/.

[31] M. Zalewski. Browser security handbook, part 2. http://code.google.com/p/browsersec/wiki/Part2, 2010.