Simple Cross-Site Attack Prevention

Florian Kerschbaum
SAP Research
Karlsruhe, Germany
Email: florian.kerschbaum@sap.com

Abstract—**Many web applications are security critical, since they involve real-world monetary transactions, e.g. online auctions or online banking. Attackers have found new attacks to exploit vulnerabilities in these web applications. Among these attacks reflected cross-site scripting and request forgery attacks have received much attention in the recent scientific literature. There are client-side and server-side solutions which can complement each other in protecting against these cross-site attacks. Server-side solutions are currently limited to either cross-site request forgery attacks or cross-site scripting attacks which attack the integrity of the session (session theft). This paper presents a lightweight and efficient solution that prevents reflected cross-site scripting and request forgery attacks using a gateway at the server. It is so strikingly simple (yet solves this practically pressing problem), that it should be part of best practices for every web site operator. It ensures that input to a web site originated in the user's browser and has not been forged by an attacker by following a link. We show the correctness of our approach using a software model checker. Our gateway protects a web site and all of its pages against cross-site attacks and is still able to function normally while not being attacked. We evaluate our approach by applying it to a number of important web sites and see the necessary architectural changes that would need to be made.**

I. Introduction

Many web applications are security critical, since they involve real-world monetary transactions, e.g. online auctions or online banking. Manipulating such transactions is a valuable target for attackers and protecting them is in the interest of both, the user and the web site.

Web developers may use JavaScript to enhance their applications. JavaScript is a language that can be embedded into an HTML page and is interpreted by the user's browser. The execution of the scripts in a page is sandboxed by the browser, such that all accesses and references are monitored. The policy governing the script's execution is the *same-origin policy* that loosely speaking states that a script is only allowed to access objects that originated from the same site. Objects include web pages (in the JavaScript document object model), cookies, and similar. A script that can access these objects has unlimited control over them, e.g. a script may rewrite a web page or read and send cookies to a different host. An attacker can abuse these abilities, e.g. to rewrite a money transfer to his account or forward a cookie that authenticates the user to him. A site, as used in the same-origin policy, refers to a DNS domain, e.g. $website.com$, and all objects from that domain are considered to be from the same origin.

In respect to the same-origin policy the term security context is often used. A security context is established between a user and a web site s, such that every script that executes in this context can access every object (in the user's browser) from that web site s. In order for a script to enter this security context it needs to be embedded in a web page requested from site s. If the attacker is able to embed a script in a web page a valid user is requesting from site s, he can take control of the entire context. Such a maliciously embedded script is called a cross-site scripting attack, since the script actually originated from a different site.

In order to mount this attack, the attacker exploits vulnerable web applications. Interactive web applications require user input. If this input is used by the web application to build the response web page, it might be exploited by the attacker. In particular, if the input is not checked well enough by the application. Imagine the following web application: A user requests a web page `http://www.website.com/index.php?name=Jim` and the application responds with a personalized web page:

```
<html>
 <body>
  Hello, Jim
  ...
```

If an attacker can trick a user into requesting the following page: `http://www.website.com/index.php?name=Jim <script>alert("XSS Script")</script>` then the response might look this:

```
<html>
 <body>
  Hello, Jim
  <script>alert("XSS Script")</script>
  ...
```

This script originating in the input part of the URL of the request would be executing in the attacked user's security context with web site $website.com$. The consequences can be severe, e.g.

- Cookie (and session) theft
- Browser hijacking, including but not limited to
- User monitoring and data theft
- Request forgery and fake transactions

Such an attack is called a reflected cross-site scripting attack [1], since the input is reflected by the users' browser. It is important to note, that the user does not see or interact with the input that is sent on his behalf, i.e. he has no chance

of checking it before it is sent to the web site. He simply follows a link that is presented to him.

The complexity of the script is not limited by the input length, because the attacker can load the script's code from a different page:

```
http://www.website.com/index.php?name=Jim
<script src="http://attacker.net/xss.js">
```

will result in our example application in the following response

```
<html>
 <body>
  Hello, Jim
  <script src="http://attacker.net/xss.js">
  ...
```

The web site that hosts the malicious link and the source of the script code do not need to coincide. Further details on how to mount a cross-site scripting attacks are left out, but the concept of a script embedded into the inaccessible input part of a link builds the basis of a successful reflected cross-site scripting attack.

Figure 1 shows an overview of the interactions. The attacker might present the malicious link to the user not only via a web page, but also an e-mail or other forms of electronic communication.

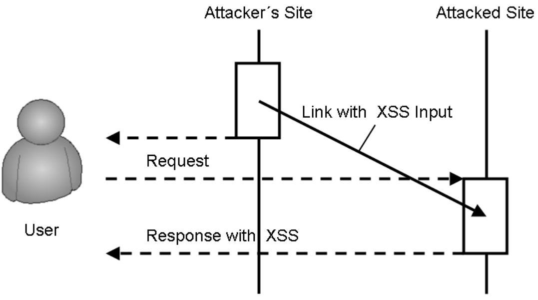

Fig. 1. Interactions during a cross-site scripting attack

The same interactions are followed by a cross-site request forgery attack. In a such an attack, the attacker exploits the existence of a session. A session is a time-limited authorization for a user to perform certain actions. The difference to cross-site scripting attacks is the goal of the request forgery attack: While a cross-site scripting attack attempts to introduce a script into the security context, the request itself suffices for a request forgery attack, since the request might initiate a forged transaction. The gateway presented in this paper is designed to prevent both reflective cross-site scripting and cross-site request forgery attacks. Throughout this paper the term cross-site attack refers to either reflective cross-site scripting or cross-site request forgery attacks.

The main idea is to enforce that the input to the web application originates from the user's browser. This prevents both attacks just described by preventing the first step of the attack. The attacker will not be able to send input on behalf of the user. It is more important to capture the method of attack and not its consequences, such that our gateway does not prevent scripts from different sources, but prevents the attacker from using a malicious link to embed such a script. In the end, preventing either the effect or the method prevents cross-site attacks, as they are currently executed, but the two different views on the problem lead to two different solutions. The main contribution of the paper is to present and evaluate this strikingly simple idea that could become part of best practices for web sites, yet eliminates the problem as other much more complex research solutions, e.g. [6], [7] (combined).

The remainder of the paper is organized as follows: Section II formalizes and defines the problem; section III outlines the concepts of our solution with details of the algorithm and proves it correct under given security assumptions. Section IV describes our formal model and verification of the system in Alloy [4], [5]. We evaluate our solution by applying it to a number of important web applications (section VI) and compare our solution to existing solutions in section V. Section VII presents our conclusions.

II. Definitions

A. Problem Description

A reflected cross-site scripting attack or cross-site request forgery attack occurs when a user follows a malicious link. This link has been tainted with some input that exploits a vulnerability on the target web site, such that a script can execute in the security context of the user's browser and the target web site.

Definition 1. *Reflected Cross-Site Scripting or Cross-Site Scripting Request Forgery attacks can occur when a user requests a page from a target web site s by following a link from another web site s' ($s \neq s'$) that contains some input.*

It is important to highlight that this definition captures the method of cross-site attacks and not their impact. We believe that this is the right way of looking at the problem, since it led us to the simple solution presented in this paper. This solution compares favorably in our opinion to existing solutions and therefore we argue that this definition correctly captures the problem.

The impact of cross-site attacks can be achieved in other ways, e.g. stored cross-site scripting attacks [1] or a copy-paste operation of attacker provided input. A stored cross-site scripting attack occurs when a user requests a page that has been tainted with the input of some other user, i.e. the other user's input has been stored on the server. Such attacks can only occur in web applications that allow this kind of information flow. We do not claim to protect against stored cross-site scripting attacks and copy-paste of attacker provided input is currently not being exploited. Our solution does not force an architecture of copy-paste operation for legal input as our examples in the evaluation sectionVI show. Furthermore, forcing the user to copy and paste cross-site scripting input clearly raises the bar for an attack. First, the attacker cannot

hide the input anymore, as he can do with malicious links. Second, the user should be cautious what he copies and pastes input into sensitive fields, e.g. an auction or banking amount. In combination, the user should at least carefully consider what he copies into a field.

B. Building Blocks

We use two techniques of web browsers: cookies and referer strings. A cookie is a small piece of data the web browser sends to the web site with every request of the user that can be modified by the web site. Cookies are stored on the user's computer and are associated with the web site (same origin policy), i.e. a web site only receives the cookies associated with it. A web site may be associated with more than one cookie.

We use cookies to authenticate users, i.e. the user operating the browser.

Definition 2. *An authentication cookie is a cookie that web site s accepts as a proof of the user's identity.*

Such a cookie can be issued by the web site after the user completed an authentication protocol, e.g. password check. One way to construct an authentication cookie would be to use a message authentication code with the user name and a secret key known only to the web site. Then the cookie cannot be forged without knowledge of the secret key and when presented it is tied to the user's identity.

Authentication cookies are only one way of authenticating users to web applications. Others include session URLs, HTTP authentication and client-side SSL. Our solution extends to all of them, but we focused on authentication cookies, since they are used in many real-world applications and portals and to simplify analysis.

Another tool used, is the referer string. The referer string is an optional HTTP header field in the request and it contains the URL of the web page referring to the requested page. The authors of [7] discourage the use of the referer due to a recommendation in [2] and its optional character. If the necessary precautions are taken by web browser developers we believe that the referer string provides a great tool and it can be assumed that a web site developer can request the user to enable it in its web browser. That is, we assume that a referer string is present in the request or the request will be denied or redirected to a page advising the user to enable it. It could be interesting for web browser developers to allow limiting the disclosure of the referer string by the same origin policy.

Definition 3. *The referer string is the URL of the web page linking to the web page requested which is sent along with the request.*

III. Solution

The web site will be divided into two types of pages: entry pages and regular pages. Entry pages do not accept any input, i.e. all provided input is either filtered by a gateway or discarded. The idea is to allow access to regular pages (which can be applications, servlets or scripts) only via an access path over an entry page, i.e. each visit must originate at an entry page (therefore the name) and can then proceed to regular pages. This way we enforce that all input of regular pages originates from the user's browser and not some malicious link trying a cross-site attack. Navigating in regular pages, e.g. using the browser's back button is still feasible.

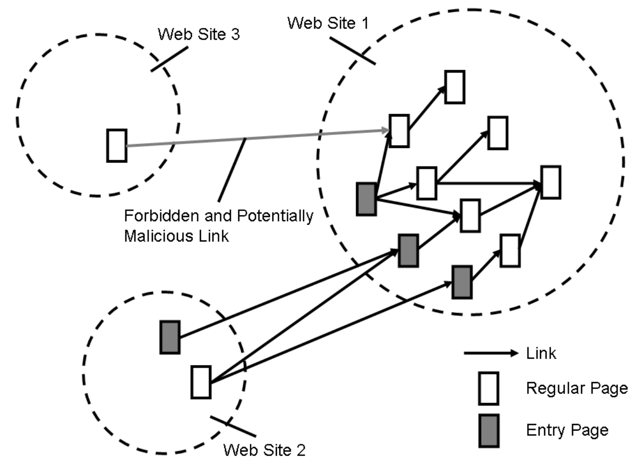

Fig. 2. Web site with entry and regular pages

Natural candidates for entry pages are login pages, home pages or portal fronts. Bookmarks can be set only to entry pages. So a careful selection of these and adaptation to the application is necessary. We discuss details for certain applications in section VI. Figure 2 depicts the entry and regular pages concept.

A. Algorithm

This section describes the algorithm of a gateway protecting a site s. Furthermore, we prove that this algorithm prevents cross-site attacks with target s under explicitly stated security assumptions.

Assume each protected site s has a gateway G that

1) for each access to an entry page, strips all input in the form of query URLs or POST request data, and
2) for each access to a regular page, verifies that
    a) a cookie has been set that authenticates a user
    b) the referer string of the request originates from site s.

    Otherwise, the request will be redirected to a default entry page (and processed as before).

An example set of HTTP headers for a reflected cross-site scripting attack and its response are shown below:

```
GET /regular_page.php?name=<script>
     alert(%22xss%22)<%2Fscript>  HTTP/1.1
Host: www.website.com
Referer: http://www.attacker.net/
                        malicous_link.html
```

```
Cookie: auth=username:
          9100ada74e758c76d0ccb2595747c910

HTTP/1.1 302 Found
Location: http://www.website.com/
                        default_entry.html
```

In this example a user followed a page with a malicious link requesting a sensitive regular page on the web site with input supplied by the attacker. The gateway would detect that, as would be necessary for a regular page, the referer string does not originate at the web site and redirect the request to the default entry page, although the cookie which has been automatically sent by the web browser correctly authenticates the user.

For our algorithm to be secure we place the following assumptions on our mechanisms:

Assumption 1. *A user's authentication cookie cannot be forged or obtained outside of the security context between the user's browser and the web site.*

Security against forgery can be achieved with a message authentication code as outlined above as long as the server's key is kept secret. Note that such cookies are transferable and therefore need to be protected at their storage place (usually along with the user's web browser). Therefore the same-origin policy and its security context are of utmost importance, since they protect the cookie against scripts executing outside that security context. Nevertheless scripts are not the only means of acquiring such a cookie, as other malware, such as browser helper objects or Trojan horses, are not bound by the same origin policy and can access the files directly at the operating system layer. These forms of attack are excluded by the above assumption, but they also require a higher degree of compromise to the user's computer than cross-site attacks which can occur during regular browsing.

Assumption 2. *In a safe browser (which we assume is used by all our users) one cannot forge the referer string of a request.*

This assumption is one of secure implementation of the browser and most browser developers subscribe to the goal of secure and safe implementation. Obviously mistakes occur and may violate this assumptions, but a regularly patched, up-to-date version of a common web browser should fulfill this assumption.

Theorem 1. *Gateway G prevents cross-site attacks on a protected web site s.*

Proof: If the cross-site link is to an entry page, the gateway will allow no input and the attack fails. If the cross-site link is to a regular page, the request must contain

- an authentication cookie
- a referer string from site s

Otherwise, the input is discarded and the attack fails.

If the request has the authentication cookie, it was made from the user's browser (from assumption 1). If the request was made by the user's browser, then it contains the correct referer string (from assumption 2). If the referer string is from site s, then it cannot be a cross-site attack, since the input originated from the same site. ∎

IV. Formal Verification

We verified the security of our algorithm and liveness of the system using a software model checker: Alloy [4], [5]. Alloy is an object-oriented modeling language and finite-state analyzer based on first-order logic. Given a system description (a logical formula) the Alloy analyzer tries to find a model (an instance of the system) that satisfies or disproves the formula. Although Alloy is limited to finite states, we believe that due to the bounded nature of our system, the results generalize.

We aimed at ensuring that all aspects of the problem could be captured. Hence, we not only proved the algorithm under the assumptions, but also proved that the model is powerful enough to capture cross-site attacks. Our model ensures three logic formulas or statements:

1) Cross-site attacks are possible, i.e. the model is powerful enough.
2) Cross-site attacks are not possible on protected web sites, i.e. our algorithm prevents cross-site attacks.
3) Protected web sites can take input, i.e. our algorithm allows for regular web applications.

We model our system with 3 base objects: browsers, (web) sites, and (web) page requests. A web site is an abstract object with no association. Safe web sites are web sites protected by a gateway as described in section III-A. Although a gateway is deployed at the web site in a network we model it as a restriction on the requests allowed.

A web browser object is an abstraction of a real browser. We assume it is tied to one user, i.e. there is no explicit model of a user. A web browser is composed of a set of web pages and cookies where the web pages are the display to the user. Whether they are displayed concurrently or sequentially is not important and not modeled. In general, we do not model time. Instead, the final state is modeled as a history (similar to the history list in a web browser) and if an attack was successful it is contained in the history.

Nevertheless one transition needs to be modeled: From safe state to the state where a successful attack has occurred. If only the final state is modeled, then the initial state could have already contained a successful attack. We therefore divided the web pages of the browser (its history) into two sets: an initial and a subsequent set. Later the assumption is made that the initial state is free from successful attacks. The analyzer shows that starting from this safe state, a state with a successful attack in the history cannot be reached.

In a web browser cookies are abstracted as a reference to a web site. The contents of the cookie are not modeled, but, in the safe state, we assume that the attacker does not have any cookies. This assumes as before that cookies are not forgeable and have not been compromised already.

We combine web pages and requests into one logical object, since web pages are the result of a request. These objects

have a source site, some optional input and an optional link to another page. For simplicity we assume that each page contains at most one link, since the user can follow at most one link on a page. A real web page can be seen as a set of model objects. The link consists of a destination site and a reference to another page. The file and directory names of a web page are not modeled, since we place no restrictions on it. For input we only model its source (and not its content). Input can either originate from the link's source or the page itself, i.e. the user enters it into the browser.

The restrictions placed on the initial state are that no page initially open in any browser has any input and that as stated before an attacker has no cookies.

The transition between web page requests is defined by function `click` which models a user's click on a link in the web browser. A transition occurs in a web browser from a source page to a destination page. The restrictions on the transition are

- the source page is in the browser's history.
- the destination page (and web site) are the target of source page's link.
- the destination may take input from the source page or the browser.
- the destination page is in the browser's history (without the start pages).
- if the destination page is from a protected web site, the referer and cookies are checked.

From regular (unsafe) web browsers referer strings are not checked, since they are forgeable and therefore not reliable. The transition restrictions are enforced on the history of the web browser, such that any non-initial page must be generated by the function `click`.

This completes the model and we can check its properties. Cross-site attacks provide input to a web page that originated at another site. The analyzer shows successful attacks on unprotected sites and does not find a counterexample for protected sites. We check the functionality of protected web site by verifying that it can receive some input.

Our model is checked with two browsers which allows for the mix of safe and regular browsers which allow for referer forgery, but have no cookies, two web sites, which are necessary for a successful attack and eight web pages. The choice of eight web pages is somewhat arbitrary, but from the restricted number of relations it is not clear how a larger number may result in a possible attack. The complete Alloy model is listed in appendix A.

V. Related Work

The widely accepted countermeasure to cross-site scripting attacks, reflected and stored, is input checking. If the input is appropriately filtered no scripting code should be able to enter the output (web page). The fact that the practice of filtering is not trivial is supported by the numerous flaws in web applications that have been reported. Filtering also prevents other web application or general application vulnerabilities, e.g. SQL injection attacks or buffer overflows.

Some languages, e.g. PERL [11], offer language support for identifying necessary filtering. An input variable is "tainted" until it has been checked and the compiler can identify if tainted variables are being processed. A filtering layer can also be built beneath the language that provides protection without web application developer interaction [9]. Our gateway is not supposed to replace input filtering, but rather augment it, i.e. input filtering is an important software engineering practice, but due to its difficult nature our solution can provide an additional layer of protection in case of flaws in the filtering algorithm. This is particularly important, since exploits in filtering algorithms may spread very fast, e.g. using worms (malicious programs self-replicating in the network), and filtering algorithms are often shared via libraries. Libraries expose a vulnerability to many web sites and all applications are dependent on the library developers to update their code.

Several solutions for cross-site attacks have been proposed in the scientific literature [3], [6], [7], [8], [10]. Noxes [8] proposes a client-side proxy that filters cross-site scripting attacks by disallowing requests that do not belong to the web site. This allows it to prevent most stored cross-site scripting attacks. It does not prevent cross-site request forgery attacks or cross-site scripts that do not make requests outside the web site, e.g. ones that create a forged request to trigger a malicious transaction. Nevertheless it provides a good complement to our server-side solution and can protect the user in case the web site has not been protected.

Ismail et. al. [3] present a client-side proxy that compares request and response and if it detects reflection of malicious characters, then these are disabled. This client-side proxy can protect only against reflected cross-site scripting attacks and therefore does not complement our gateway well. It does not prevent cross-site request forgery attacks and rewrites the server's response.

Vogt et. al. [10] use dynamic data tainting to prevent injected scripts to leak data to the attacker. They implemented an extension to popular Firefox web browser that tracks data this way. It does not prevent cross-site scripting attacks from forging transactions or cross-site request forgery attacks, but nicely complements our gateway, since it provides protection against some effects of stored cross-site scripting attacks.

Jovanovic et. al. [7] describe a server-side solution for preventing cross-site request forgery attacks. It does not prevent cross-site scripting attacks, but it does not rely on the referer string and instead rewrites requests and responses by adding a token to the URL.

SessionSafe [6] presents a server-side solution to session theft via (reflected and stored) cross-site scripting attacks, i.e. this solution does not prevent cross-site scripting attacks per se, but rather prevents successful attacks from stealing the session. Other attacks via cross-site scripts, e.g. requests to create malicious transactions are still possible, e.g. by modifying the input. Also cross-site request forgery attacks are still possible.

VI. EVALUATION

A. Performance Discussion

No server-side solution [6], [7] presents absolute performance figures, such that we could compare them to an implementation of ours. We will discuss the performance advantages in qualitative terms and argue why we believe our solution is more efficient.

Two types of costs can be distinguished in deploying a server-side protection mechanism. First, the cost of setting up the system and, second, the cost for every request.

Jovanovic et. al. [7] implemented their solution as a PHP wrapper and proxy and it requires no further changes to the environment. Although their method can be extended to other languages, such a language constraint can be quite hampering for real-world web applications or portals. Every request is intercepted by the proxy. During the request only the header needs to be processed and a hash table look up is performed, but during the response the entire body needs to be parsed and URLs rewritten. Although the authors report no noticeable difference in the web application, no measurements have been taken.

SessionSafe [6] poses high costs on the environment: Several (web) servers need to be deployed and DNS entries maintained. There is a mapping between web requests and the DNS entries making replication and load balancing more difficult. Furthermore, response rewriting needs to process not only URLs, but also JavaScript and the HTML structure of the web page. Additional JavaScript code executes in the user's web browser potentially delaying display.

On the other hand, our gateway can be deployed in front of any web application written in any language and even load balanced ones. The gateway only needs to be able to verify the cookie which could e.g. be done by making the cookie of the following format: $username|MAC(username, key)$. If the gateway has the key, it can verify the cookie (and the user's identity) with one message authentication code computation. Furthermore, no output rewriting is necessary. The output of the web application is just forwarded. It is anticipated that particularly the lack of response rewriting significantly reduces the delay for every request.

B. Applications

1) Online Banking: We modeled the example online banking application after the author's online banking application, although without access to the source code. The homepage offers a form for logging into the online banking application, i.e. the homepage is the only entry page and the client does not need to present a cookie to access it. Then after successfully verifying the supplied username and password credentials the application redirects the user to the main menu which offers a selection of options. Some of these, e.g. money transfers or stock market orders, take input and then display a confirmation page. Some include more than one step, but the main structure remains. One can always return to the main menu. The application structure is displayed in figure 3.

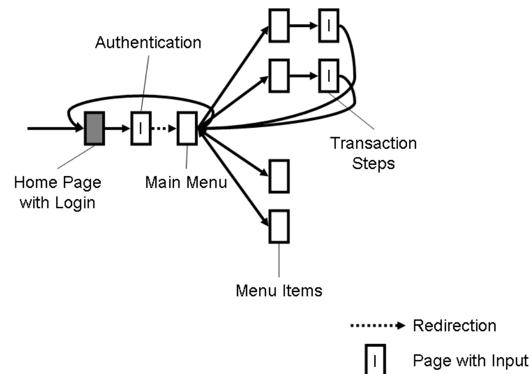

Fig. 3. Online Banking

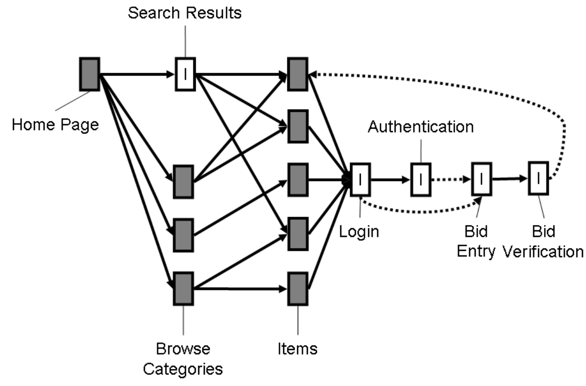

Fig. 4. Online Auctions

2) Auctions: The auction web application is modeled after `www.ebay.com`, but focuses only on some main aspects and does not cover the entire application. Our auction application provides browsing, searching and bidding. There are some changes that would need to be made to the application compared to the existing one from what we can infer by analysis. First, item pages need to be pages without input, such that they can be bookmarked, searched by external search engines and forwarded to others. We cannot asses the impact of this change, but the web pages do not need to be static, but can include variable parts, such as the highest bid. The search page is a regular page and cannot be accessed without visiting an entry page, e.g. the home page, first. Note, that in figure 4 we simplified the link structure. The search page also can be accessed without authenticating first and therefore would not need to check the cookie or a special "non-authenticated cookie" can be set by all entry pages, if the user is not authenticated yet. Bids are entered into a special page accessible only after authentication. The authentication page is preceded by a login page, that is only displayed if the user is not yet authenticated. Otherwise, she is redirected to the bid entry page automatically. After bid verification

one is redirected to the item page again. To be able to identify the item all pages involved in the bidding process need to take the item identification as input. Evaluating the auction application the item pages present a major architectural change. The complicated authentication for different pages present a challenge that can be met by the gateway, if regular pages can be configured to not need authentication or better the "non-authenticated cookie". A particular problem remaining which renders our solution insufficient for auction pages is that of stored cross-site scripting attacks. One can enter malicious input as a bid that may be displayed at another user's page. In summary, our gateway can provide only limited protection at a potentially high cost for auction applications.

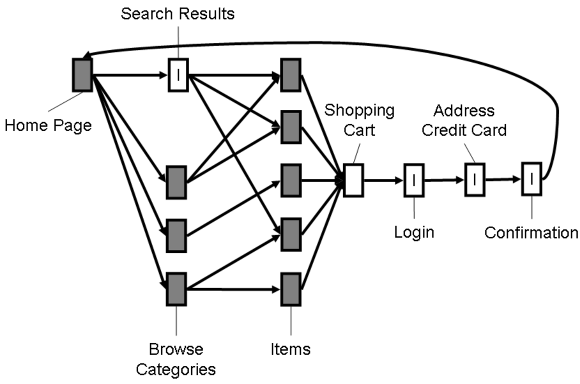

Fig. 5. Online Retailer

3) Online Retail: The retail web application is modeled after `www.amazon.com`, and its figure (fig. 5) very much resembles online auctions. However, our gateway is much better suited for online retailing than online auctions. The search page (and shopping cart) present the same authentication challenge, since they do not need to be authenticated, but again the "non-authenticated cookie" is the best option. We assumed that the contents of the shopping cart are kept in a different page. Fortunately the items' pages are already pages without input in the case of online retailing and there is no associated cost of converting them. The checkout procedure proceeds similarly to bidding, it only includes more pages and steps in the procedure. The one-click shopping option was not investigated due to its rather complicated implementation. Summarizing, our gateway is well suited for online retailing.

4) Webmail: Our simplified web mail application is based on `www.gmail.com`. It offers mail sending, reading and searching. The web application structure strongly resembles online banking with an initial login page. This method of authentication is well suited for our gateway. A particular challenge for web mail is to display e-mails with HTML content. This content may not, at least immediately, execute scripts in the user's web browser, since they may be malicious. E-mails may originate outside of the web application and even outside of web browser, such that stored cross-site scripting attacks are possible. Furthermore, as stated before HTML is

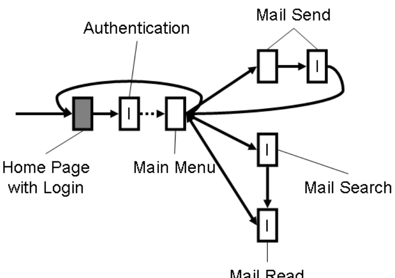

Fig. 6. Web Mail

legal input for e-mails, such that sophisticated filtering and parsing of the HTML is necessary. Our gateway presents a good complement to this advanced filtering that can protect mail sending and searching.

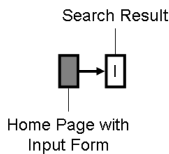

Fig. 7. Search Engine

5) Search Engines: Search application do not present a threat by themselves, but are nevertheless important targets, since they are very often start pages for users to browse the Internet. So being able to place a malicious script via a search engine and then control the users' web browser is a viable attack strategy. The example search web application is modeled after `www.google.com`. It does not allow to bookmark searches or directly link to queries any longer, so it is more restrictive than the model search engine. This can be a problem considering applications, such as queries from within the users' browser by toolbars or similar. A direct link to the search result page is necessary. Furthermore search engine usually do not authenticate users, but rather operate without user authentication. Therefore the cookie setting and verification mechanism in our gateway should be disabled completely. Nevertheless protection would hold, since there is no threat to the user's identity. Since search engines usually only have one input form, it might be worth protecting it without our gateway just by strong filtering.

6) SAP Portal: Finally, the application that triggered the development of the gateway. A simplified version of the SAP portal is depicted in figure 8. No access is possible without passing through the portal front and authenticating, although authentication can be done and is often done using client-side SSL certificates. Once the portal front has been passed all other (regular) pages can be accessed and browsed. The user always remains authenticated. Input is only required by some regular pages. Our gateway is a perfect fit for such portal applications and prevents all cross-site attacks at almost no cost.

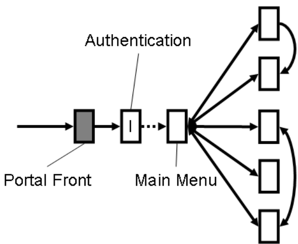

Fig. 8. SAP Portal

C. Limitations

This section summarizes the limitations of our proposed technique, some of which have already been mentioned throughout the paper.

First, the browser must send a trustworthy referer string. Unfortunately not all browser are configured to send the string by default. In case of a missing referer string the user should be directed to a separate page and be instructed to configure his browser correctly. We strongly encourage the browser developers to consider the option of extending the same-origin policy to referer strings. Although attacks to forge referer strings have been described, browser developers are working on fixing them. No standard method, e.g. via JavaScript, exists to send arbitrary referer strings from a browser.

Second, bookmarks can only refer to entry pages. This limitation strongly relates to the application section and the general architecture of the web site. Dynamically built pages that take input and are on-the-fly generated from the database cannot be entry pages. Static pages which take no input can be entry pages and therefore be bookmarked. Therefore this limitation is only a major one in case of dynamic web page architectures.

Third, multi-site scenarios where multiple web sites (domains) collaborate, e.g. e-commerce and credit card processing, lead to an increase in the number of allowed referers. The multi-site web application should be configured like a single domain web application, but obviously this requires a minor configuration overhead.

VII. Conclusions

We presented a gateway that prevents cross-site attacks, i.e. both reflected cross-site scripting and cross-site request forgery attacks. It is efficient and effective. It can be deployed in front of every web application and only needs to process the request entirely protecting the web site against the method of cross-site attacks. No response rewriting is done and therefore we believe that our solution is more scalable than other proposed solutions. We believe that our solution is the first that prevents cross-site attacks and has the potential for real-world deployment given its performance characteristics.

Our gateway uses three techniques: web page classification, referer string and cookies. Given a referer string and a cookie we can assure that the input of a web page originated in the user's browser. By allowing input only to a limited set of pages, which cannot be the first pages of a visit (thereby preventing malicious links with input to the web site), cross-site attacks are prevented.

Besides describing the details of the algorithm, another contribution of the paper is a thorough analysis of the solution and the system as a whole. We have given a security proof to highlight the assumptions made and modeled the system using a software model checker. We have shown that our solution prevents cross-site attacks to a protected web site, by showing that

- cross-site attacks are possible in our model.
- cross-site attacks are not possible on protected sites in our model.

Furthermore, we verified that our system is live, i.e. web applications on protected web sites can function normally.

We have shown how to deploy our solution in front of several web applications. It is well suited for a large number of them and we believe it should be part of best practices for such web applications. Most importantly, it does not impact on the usability of the web sites.

VIII. Acknowledgements

We would like to thank Martin Johns for valuable feed-back on an early version of the paper and the anonymous reviewers for their insightful suggestions.

References

[1] S. Cook. A Web Developers Guide to Cross-Site Scripting. *Technical Report, SANS Institute*, 2003.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, *IETF RFC 2616*, 1999.

[3] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. *Proceedings of the International Conference on Advanced Information Networking and Application*, 2004.

[4] D. Jackson. Automating First-Order Relational Logic. *Proceedings of ACM Conference on Foundations of Software Engineering*, 2000.

[5] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *Technical Report, MIT Laboratory for Computer Science*, 2000.

[6] M. Johns. SessionSafe: Implementing XSS Immune Session Handling. *Proceedings of European Symposium on Research in Computer Security*, 2006.

[7] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. *Proceedings of IEEE International Conference on Security and Privacy in Communication Networks*, 2006.

[8] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. *Proceedings of the 21st ACM Symposium on Applied Computing*, 2006.

[9] T. Pietraszek, and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. *Proceedings of Recent Advances in Intrusion Detection*, 2005.

[10] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[11] L. Wall, T. Christiansen, R. Schwartz, and S. Potter. Programming Perl. *OReilly*, 1996.

Appendix

```
module xss

// --- Basic Objects

// web sites
sig site {
}

// web sites protected by our gateway
sig safe_site extends site {
}

// web page request
sig page {
  source : site,
  input  : lone page,
  dest   : lone site,
  link   : lone page
}

// browser
sig browser {
  initial : set page,
  pages   : set page,
  cookies : set site
}

// honest user's browser
sig safe_browser extends browser {
}

// --- Safe Start State Assumption

// no input to start with
fact {
  all p : page, b : browser | {
    p in b.initial => no p.input
  }
}

// no cookies already lost
fact {
  all b : browser {
    b in browser - safe_browser
      => no b.cookies
  }
}

// --- Transitions

// user click
pred click (b  : browser,
            p1 : page,
            p2 : page) {
  p1 in (b.pages + b.initial)
  p2.source = p1.dest
  p2 = p1.link
  p2.input = p1 ||
              p2.input = p2 ||
              no p2.input
  p2 in b.pages
  p2.source in safe_site => {
    // check referer
    b in safe_browser
      => { some p2.input
              => p1.source = p2.source }
    // check cookie
    some c : site | c in b.cookies &&
                    c = p2.source
  }
}

// ensure state transitions
fact {
  all p1 : page | {
      some b : browser | p1 in b.initial
    ||
      some b : browser, p2 : page |
        click(b, p2, p1)
  }
}

// --- Assertations

// cross-site attack
assert xss {
  no p : page, b : browser
    |  p in b.pages
    && some p.input.source
    && p.source != p.input.source
}

// no cross-site attack to
// a protected site
assert noxss {
  no p : page, b : browser
    |  p in b.pages
    && some p.input.source
    && p.source != p.input.source
    && p.source in safe_site
}

// regular use
assert script {
  no p : page, b : browser
    |  p in b.pages
    && p.source in safe_site
    && some p.input
}

// check 'em
check xss for 2 site, 8 page, 2 browser
  expect 1
check noxss for 2 site, 8 page, 2 browser
  expect 0
check script for 2 site, 8 page, 2 browser
  expect 1
```