1 Introduction

Cross-site request forgery (CSRF) is a common web application vulnerability that has plagued many popular web applications. CSRF vulnerabilities occur when a state-changing request to a web application can be guessed by an attacker except for some automatically sent authentication credential. Most frequently, the authentication credential in question is in a cookie, but it can also be a basic authentication credential kept by the browser. Given such a request, the attacker can create or modify a web page to triggers the request in whatever web browser visits it, causing an access control violation whenever a logged-in user visits the attacker’s webpage.

CSRF vulnerabilities are relatively easy to defend against: one adds and checks for an unguessable cookie attached to the request. The primary problem with this solution is that requests that require protection are not always easy to identify. Some GET requests, which are supposed to be idempotent, perform state changes (most likely because application developers find this the easiest way to make a link-like button to perform an action). Some non-GET requests do not change relevant state — they may only be a non-GET request because the size of the parameters do not fit in the URL.

This paper proposes building a tool to identify CSRF vulnerabilities by directly detecting when a request results in a state change. Instead of using only the HTTP front end of the web application, we also use access to the backend database. Given this access, it is relatively easy to determine whether a request really changes state, which should allow us to avoid false positives and false negatives that other approaches would have. An evaluation of this tool shows that this approach is a promising and practical way to find subtle CSRF vulnerabilities.

2 Related Work

2.1 CSRF Protection Tools

Tools have been developed to protected web applications from CSRF vulnerabilities. These tools assume that sensitive requests do not use the GET method (or, alternately, to identify all sensitive GET requests). These tool then adds a CSRF tokens to forms generated by the application and check those tokens on requests coming into the application. This type of protection can be implemented in web application frameworks (e.g. Ruby on Rails) or in a proxy as in RequestRodeo.
2.2 Penetration Testing Tools

Several tools exist to identify vulnerabilities in web sites. The simplest are scanning tools like Nikto[1] that use a list of canned requests and responses known to indicate vulnerabilities. More complicated tools (e.g. [9, 2, 8]) identify requests by examining requests specified by the user (manually or by using an application-supplied proxy) a proxy or by crawling the web site; given these requests, they then generate variant requests that are likely to expose vulnerabilities. These tools primarily target command injection flaws like cross-site scripting and SQL injection.

Some of these tools reportedly attempt to find CSRF vulnerabilities. To do so, however, these tools generally either assume that all POST requests and no GET requests need CSRF protection or that the user will list all sensitive requests. Then, they use some heuristic to identify whether a CSRF token is present. For example, Grendel–Scan[8] assumes only form submissions are a CSRF risk (but allows user whitelisting) and assumes that a CSRF token is present if and only if the form parameters changes on re-request, a heuristic likely to be unsuccessful on many web applications.

3 Method

3.1 Testing Scenario

Our tool is intended to be deployed on a web application which primarily manipulates data in some database. The user of our tool sets up the application they are testing in a dedicated testing environment and create two user accounts in this application. The user then must provide our tool with instructions on how to login to the two user accounts and access to the database underlying the application. Our tool accesses the web site as each of the users, examining the effects on the database underlying the database. Since we wish to assume that any changes in the database are caused by the tool’s activity, the tool must have exclusive access to the application and cannot be usefully parallelized.

Given this access to the application, our tool produces a list of requests that it believes are forgivable and change sufficiently important state on the server. We maintain sufficient information to show why an attacker would know the entire contents of the request in question or what parts of the request we believe an attacker would not be able to guess.

Detecting any state change (instead of just ‘important’ changes as we attempt to do) would be inappropriate. Though it may be disturbing in their analytics, web application developers do not consider ‘false’ log entries a security problem, but it is certainly a change in state on the server. Similarly, updates to cookie expiration times or the generation of new anti-CSRF tokens are changes in state, but this state is not persisted by the server. There are some cases where it is difficult to decide whether a particular state change is an attack; some concrete examples are presented in our results.

3.2 Website Exploration

We explore the database using standard techniques from existing web fuzzers and other penetration testing tools. Our application acts as a web crawler, but also fills in forms on the web pages with random inputs. To ensure that any potential vulnerabilities we find are not protected by strict referer checking, our crawler never sends a Referer header: we assume an attacker would prevent a referer
from being sent by, for example, submitting the form from an https or data URL. Several separate
crawls of the application are required. First, we crawl the website while not logged in to obtain a
guess as to which changes in the database are unimportant. Then, we crawl as a “victim” user to
gather a set of requests that an attacker might want to forge. Then, we crawl as an “attacker” user to
determine what information would be available to the attacker.

To allow good coverage of the application, it is necessary for both the attacker and victim user
to have maximum access to the application. Since this likely means that both users should have
administrative privileges on the application, this requirement might be a source of false positives:
it may be possible for administrators to learn information necessary to perform a CSRF attack on
other administrators and application maintainers may not consider this a security problem. These
added privileges have an unfortunate risk of causing the website or one of the user accounts to
become unusable; our tool currently requires manual intervention to handle these cases (which
occurred twice in preliminary testing — once by changing the password on the attacker account and
once by marking the website as down for maintenance).

The experiments in this paper used a web crawler we derived from Heritrix[12]. We used the
Jericho HTML parser to find HTML forms, and filled them in at random, choosing from a fixed
set of inputs for text fields. We do not make any attempts to fill in file uploads or produce AJAX
requests.

3.3 Side-Effect Detection

We fetch the entire database before and after each request. During initialization, we obtain a list
of all database tables and their schemas. For each table, we construct queries to fetch the entire
contents of the rows of the table in a consistent order. We compute a hash of the results of each
of these queries and compare the result before and after the request. Since a test deployment
should have a small database, this strategy does not propose major performance concerns. From the
comparisons, we can annotate each request with a list of which database tables it changes.

If we were concerned with all state changes, it would be sufficient for any database table to
change. Unfortunately, some of these changes may reflect unimportant data. For many web apps
such changes are frequent enough that we would be swamped by false positives. For example, some
application update a session ID cookie expiration time stored in a database on every request; if we
assumed that any database change represented a state change we would give every page as a false
positive. We use a heuristic to avoid this at the cost of some false negatives: we assume that any
changes occurring due to GET requests while not logged in represent unimportant state.

3.4 Guessability Detection

Our tool assumes that if an attacker can guess a request, then the parameters of the request most
likely appear in the attacker’s view of the same website. Given this assumption, we construct an
index of all words received when the ‘attacker’ user crawls the website. Our tool considers any words
seen in this crawl guessable and other words unguessable. We avoid falsely detecting parameters
we use to fill in forms as unguessable because we use a known set of guessed form inputs (which
appear throughout the application after the ‘victim’ user’s crawl as run). We decompose a request
into words (considering each request after URL decoding and assuming that any non-alphanumeric
characters separate words).
This method of detecting guessability is obviously imperfect. False negatives are possible: an application might, for example, use a sequence number, the Base64 encoding of some guessable string, or fixed word prepended to known word; our simple word matching would not detect the patterns in these tokens which would clearly be usable by attackers. False positives are also possible: for example, an application generating strings of mostly non-word characters as a CSRF token might erroneously be assumed to have a guessable request when it would not actually be possible for an attacker to know which sequence of non-word characters to use.

4 Results

4.1 Tested Applications

As a proof of concept, we selected a couple old web applications known to have CSRF vulnerabilities and several new applications which take countermeasures against CSRF vulnerabilities. The old application were:

- Bugzilla 3.3.1[4] — Bugzilla is a bugtracking system. This version of Bugzilla had a CSRF vulnerability[7] allowing bugs, keywords, unused flag types, and saved searches to be updated by CSRF attack.

- PHPbb 2.0.19[3] — PHPbb is a web forum. This version of PHPbb had previously published CSRF vulnerabilities allowing attackers to create and delete postings.[11]

- PHPbb 3.0.6[5] — This version of PHPbb, which apparently shares little code with PHPbb version 2, was the most recent at the time of this work and was not known to have any vulnerabilities.

- XOOPS 2.4.2[6] — XOOPS is a content-management system. At the time of work, this was the most recent version of XOOPS.

All of the tested applications make some attempt at CSRF protection: PHPbb 3, XOOPS, and Bugzilla generate secret tokens, and PHPbb 2 passes the session ID through form parameters to some sensitive actions.

We setup these applications on a local server against a local MySQL database. We created two accounts in each application and gave each account administrator privileges. We constructed login instructions for our tool by hand, and pointed the tool at the front end URL and the MySQL database each application was configured to use. Our tool was configured to crawl a maximum of 5000 pages and to go to a maximum distance from the entry page of 6 hops. The tool would first perform a crawl while not logged in, then as the “victim” user, then as the “attacker” user. Detected CSRF vulnerabilities and known false results are shown in Figure 1.

4.2 Performance and Coverage

With our conservative settings intended to ensure reasonable coverage, making a total of 15000 requests per application, analyzing each of the tested applications takes about 1 to 3 hours. Based on timings from our crawler, it appears that checking the database was not the bottleneck in these
<table>
<thead>
<tr>
<th>Application</th>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>True positives</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHPbb 2.0.19</td>
<td>Send private messages</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Update user information</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Reply to a forum post</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Create a new forum post</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>*Mark private messages as read</td>
<td>—</td>
</tr>
<tr>
<td>Bugzilla 3.3.1</td>
<td>Update existing bug</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Update user preferences</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Add “quips” (quotes displayed a top of all pages)</td>
<td>—</td>
</tr>
<tr>
<td>XOOPS 2.4.2</td>
<td>*Mark private messages as read</td>
<td>—</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Logout user</td>
<td>—</td>
</tr>
<tr>
<td><strong>False positives</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHPbb 2.0.19</td>
<td>*Fail to login as user for administration panel</td>
<td>Increments last_login_tries column of users table; user apparently locked out with more than three failed attempts</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>*Search forum posts</td>
<td>Search results are cached in database table (but attacker can edit default search results seen by victim)</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Read profile</td>
<td>User last active time in users table</td>
</tr>
<tr>
<td>PHPbb 3</td>
<td>View index of forums</td>
<td>Updating last login time</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>View forum post</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>XOOPS 2.4.2</td>
<td>Read entry page</td>
<td>Table of ‘online’ users</td>
</tr>
<tr>
<td><strong>Known false negatives</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHPbb 2.0.19</td>
<td>Delete private messages</td>
<td>Not covered in crawl as victim</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Edit configuration settings</td>
<td>Failed to reauthenticate as administrator</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Login into (attacker) account</td>
<td>Not generated in crawl as victim</td>
</tr>
<tr>
<td>Bugzilla 3.3.1</td>
<td>Login into (attacker) account</td>
<td>Not generated in crawl; would be whitelisted by baseline crawl</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Logout user</td>
<td>DB change whitelisted by baseline crawl</td>
</tr>
</tbody>
</table>

* = questionable classification

Figure 1: Detection results and failures
crawls and that the vast majority of the quoted runtimes were spent waiting for replies from the web applications.

Even with these conservative crawling settings, our coverage was not perfect. For example, in PHPbb 2, though we found the form to delete private messages, our crawl, apparently through bad luck, never sent a request that actually caused private messages to be deleted, causing a false negative.

To evaluate coverage, we can examine which database tables saw changes in our crawls. Based on this analysis, we most notably discovered that we failed to cover, some administrative functionality in both versions of PHPbb 2 and 3. Manual inspection showed that this problem was most likely due to our crawler not handling a secondary authentication screen required to reach that functionality. We also found some additional gaps in coverage based on tables that were not hit, for example PHPbb 3’s private message filter rules and PHPbb 2’s manipulation of polls. We believe that most of these problems could mitigated with a better crawler or, at worst, with a crawler guided by a few human-generated requests.

4.3 Borderline True Positives

Our tool detected some CSRF attacks which we anticipate that only some web application developers would consider legitimate attacks but that are not obviously false positives. These are cases where an attacker would be able to mark messages as read for the victim.

4.4 False Positives

Our tool detected some false positives. In our experiments, these were all due to requests which did change state but did not (as far as we could tell on manual inspection) change relevant state. These were generally requests which did, in fact, cause some data in the database to change but for which this data was not relevant to a CSRF attack. For example, this data included updates to user last-attempted-logged in times which were not visible through any interface on the website.

4.5 Known False Negatives

As discussed previously, the most serious cause of false negatives in preliminary runs of our study was insufficient coverage in some runs. We need to cause the vulnerable request to have an effect as the victim and then most likely find the form that submitted as the attacker (to have all the values that part of that request). We did not obtain full database table coverage of any of the tested applications. For most of the missing tables, this was not because we were incapable of causing those changes, but because the crawl with which we produced the results for this paper was “unlucky.” (Some other runs done during development did cover many of these missed tables.) Since this category of false negatives is not a fundamental limitation of our technique, we believe that a more refined tool would avoid most of them.

A secondary in this study, but more fundamental source of false negatives is the granularity of our change detection. For example, this prevented us from detecting that one could logout users with CSRF in Bugzilla 3.3.1 even though this caused a change in the database because the relevant table (of session cookies) was changed on every almost every request for a non-logged-in user. We
suspect that similar problems may have resulted in further (but unidentified) false negatives in these experiments.

5 Future Work

5.1 Improving Crawling

Integrating this approach into an existing penetration/fuzz testing tool would likely improve its performance considerably. In this evaluation, we mitigated the relatively poor discovery by running a very long crawl. Better heuristics to avoid wasted duplicate requests, presumably already implemented in existing tools, would substantially help coverage. We might also gain by using a strategy, like some existing tools, that is seeded by templates from user generated requests. Additionally, because our tool accesses the backend storage, it could determine when crawling is likely complete based on which changes to the database have been observed.

5.2 Change Detection

One source of errors was the poor granularity of our change detection. Real applications mix data that should be protecting against CSRF with data which is not: the most egregious example we observed in this proof of concept was forum topic view counts mixed with the forum contents themselves, but less strange examples exist. Fortunately, we detected related CSRF vulnerabilities because other tables were used for some of the forum post related data. In general, however, the poor granularity of our detection seems like a likely source of unknown false negatives.

A similar relatively easy extension would be handling other types of logical state changes. For example, we could detect changes in cookies sent to the browser (which, for example, Bugzilla uses to store state besides a session ID), changes to the file system, etc.

5.3 Guessability Detection Improvements

The current indexing strategy does not handle some cases where strings are guessable by an attacker but do not appear. Two cases of this are likely most important. The first is where an attacker can cause a string to leak through the Referer header from the attacked website. A modified version of our tool could detect some instances of this automatically: first, we would crawl the website as an appropriately privileged user randomly filling in forms to contain a sentinel URL; then, while crawling as the victim user, we would mark the URL as known whenever a link or other resource at the sentinel URL was discovered. Besides Referers, it would also be helpful to search for sequence numbers and other near-matches where an attacker could easily enumerate all possibilities to find the target token.

Another concern is that some unguessable parameters may be optional. For example, an application might pass a session ID through every form, which we might interpret as a CSRF token, but it might not check that session ID if a session ID is present in a cookie. An improved tool could test, for each request determined to be CSRF-protected, whether the request omitting unguessable parameters still results in relevant side-effects.
5.4 Assessing More Applications

To fully evaluate this tool, we would need to apply to a larger range of web applications, which we could not do due to time limitations.

6 Conclusion

We demonstrated that our method for detecting CSRF vulnerabilities, by directly detecting the two preconditions, privilege state changes and guessability by an attacker, is practical. In these tests, our tool detected several CSRF vulnerabilities with a relatively low rate of false positives. The primary obstacle our tool faced, poor coverage of the application under test, can likely be mitigated by integrating with prior work on black box testing of web applications. We believe that with such changes, this approach would allow for the creation of a practical tool for legacy web application maintainers that would find CSRF vulnerabilities that would previously only be identified through manual inspection.

References

[7] Bugzilla team. 3.2, 3.0.6, 2.22.6, and 3.3.1 security advisory. [http://www.bugzilla.org/security/2.22.6/](http://www.bugzilla.org/security/2.22.6/) February 2009.