

Automated detection of CSRF-worthy HTML forms through 4-pass reverse-Diff analysis

Tasos "Zapotek" Laskos <tasos.laskos@gmail.com>

10/10/2010

Abstract

In general, the majority of vulnerability detection techniques depend on fairly simple injections of strings and subsequent blind pattern matching of the body of the induced HTTP response.

These vulnerabilities include, but are not limited to, XSS, SQL Injection, File Inclusion which require no awareness of context but straightforward brute-force.

However, there are types of vulnerabilities like Blind SQL Injection and Cross-Site Request Forgery (CSRF or XSRF) that do require a certain awareness of the context under which the audit and discovery occurs. In the case of CSRF even this is not enough as CSRF, due to its abstract nature, covers a great range of scenarios, most of them completely benign.

Thus, automated CSRF detection traditionally creates a great deal of noise.

This paper is aimed towards demonstrating a fairly simple technique, dubbed "4-pass rDiff CSRF detection", in order to diminish such noise by easing the process of context establishment, i.e. allow Web Application Security Scanners to determine which HTML Form elements are worthy of being reported as vulnerable to CSRF.

Table of Contents

1 Introduction.....	3
1.1 Definitions.....	3
1.1.1 Cross-Site Request Forgery (CSRF).....	3
1.1.2 Reverse Diff (rDiff).....	4
1.1.3 4-pass rDiff analysis.....	5
2 The Challenge.....	5
3 Formal representation of the problem and solution.....	6
4 Overcoming real-world restrictions.....	6
5 A unified algorithm to identify CSRF-worthy forms (4-pass rDiff).....	7
5.1 Natural language.....	7
5.2 Pseudocode.....	8
6 Conclusion.....	8
References.....	8

1 Introduction

The “4-pass rDiff” analysis approach that will be discussed in this paper was part of the research and development efforts for the Open Source web application security scanner framework [Arachni](#)[1].

In order to move forward we first need to define the terms that will be used throughout this paper.

1.1 Definitions

1.1.1 Cross-Site Request Forgery (CSRF)

[OWASP](#)[2] defines CSRF as:

CSRF is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated. With a little help of social engineering (like sending a link via email/chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation in case of normal user. If the targeted end user is the administrator account, this can compromise the entire web application.

In order to better understand how CSRF attacks work imagine the following scenario:

You are logged-in at your bank's website (<http://bank.com>) which has an HTML form that allows you to transfer money from your account to another.

That HTML form has the following attributes:

- “action” = “<http://bank.com/transfer.php>”
- “method “ = “get”

and the following input fields:

- amount – the amount of money to be transferred
- creditAccount – the receiver's account

Executing a legitimate transfer would result in your browser requesting the following URL from the bank's HTTP server:

<http://bank.com/transfer.php?amount=<amount>&creditAccount=<account number>>

Now imagine that while your session cookies for <http://bank.com> are still valid (i.e. you are still logged-in to the bank's website), you receive an e-mail prompting you to click on a link.

Upon clicking the link you are presented with a web-page that contains the following code:

```

```

The above HTML code will cause the web browser to request the URL

<http://bank.com/transfer.php?amount=1000&creditAccount=123456>, with your valid session cookies, thinking that it was an image and causing your bank to transfer 1000€ to the “123456” account.

Something to that effect can also take place even when the transfer form uses an HTTP POST request, it is not only limited to GET requests.

The above scenario describes a critical CSRF vulnerability, one that affects business logic.

1.1.2 Reverse Diff (rDiff)

A reverse diff operation has the exact opposite effect of a regular diff operation. While a diff operation analyzes 2 strings and returns their differences, an rDiff operation will return the parts of the 2 strings that remain unchanged.

A Ruby implementation of [rDiff](#)[3] extracted from the code-base of the [Arachni](#)[1] Web Application Security Scanner Framework follows:

```
#
# Gets the reverse diff (strings that have not changed) between 2 strings
#
# @param [String] text1
# @param [String] text2
#
# @return [String]
#
def rdiff( text1, text2 )

  return text1 if text1 == text2

  # get the words of the first text in an array
  words1 = text1.split( /\b/ )

  # get the words of the second text in an array
  words2 = text2.split( /\b/ )

  # get all the words that are different between the 2 arrays
  # math style!
  changes = words1 - words2
  changes << words2 - words1
  changes.flatten!

  # get what hasn't changed (the rdiff, so to speak) as a string
  return ( words1 - changes ).join( ' ' )

end

text1 = <<END
This is the first test.
Not really sure what else to put here...
END

text2 = <<END
This is the second test.
Not really sure what else to put here...
Blah blah...
END

rdiff( text1, text2 )
# => "This is the test.\nNot really sure what else to put here...\n"
```

This simple piece of code has been proven very useful in circumstances where context irrelevant content, like advertisements and other dynamic content, needs to be discarded.

Thus giving the system an effective sense of context by enabling it to focus on elements that pertain to the functionality of a given page.

1.1.3 4-pass rDiff analysis

4-pass rDiff analysis is the name the author gave to the technique that this paper will be presenting.

4-pass refers to 4 initial HTTP requests required in order to determine which forms are worthy of CSRF testing, so as to reduce noise.

2 The Challenge

As you might have guessed, the biggest challenge one has to deal with while developing an automated system to detect CSRF vulnerabilities is context.

This is due to the fact that the majority of CSRF occurrences are benign, they don't affect business logic and have no consequences.

Technically, the following url:

<http://www.google.co.uk/search?q=csrf>

verifies that Google's search form is indeed vulnerable to Cross-Site Request Forgery.

However, there are virtually no consequences to this, rendering it unworthy of reporting or even taken under consideration.

Thus, the ability to determine which forms affect business logic is crucial in order for the scan to produce meaningful results.

A trivial process for the human cognitive abilities but one that has troubled web application security scanner developers for years.

3 Formal representation of the problem and solution

In order to effectively solve the problem of context algorithmically, one has to break down said context into the set of circumstances that comprise it.

A top-down approach will be taken, in order to adapt one's cognitive process into a suitable algorithm, starting from the highest level.

At the highest level, the following 2 criteria must be met in order for a form to be at risk of CSRF:

1. Valid session cookies must exist in the browser, reflecting a logged-in user.
2. A form must affect business-logic, i.e. affect the user's data on a given site.

Extrapolating from the 1st criterion we can safely assume that a CSRF-worthy form has a significantly bigger chance of occurring when a user is logged-in.

Following this assumption, the process of determining which forms are at risk becomes trivial since one simply needs to request the same web page twice:

- Once with valid session cookies of a logged-in user
- Once without cookies, simulating a guest user.

And then focus the audit towards forms that only appear during a logged-in session i.e. disregard forms that are identical to both guests and logged-in users.

The aforementioned approach should be taken for each web-page individually.

Using this technique one can significantly diminish noise created by blindly auditing all form elements, which would render the results of the audit relatively unusable, and provide a meaningful report.

4 Overcoming real-world restrictions

The previous chapter presents a valid approach in order to determine which form elements are most likely to affect business logic in an ideal and homogeneous environment.

However, real-world web-sites are wildly heterogeneous and differ at how they generate dynamic context-irrelevant content, like advertisements, tag-clouds or a list of recent comments.

Moreover, there are cases where a single website will generate such content in a unique way on a per page basis.

In order for an automated system to be usable without the need to be manually adjusted for each web-site, or each page in a worst case scenario, it needs to be able to dynamically adjust itself.

This would create the need to implement a sophisticated Artificial Intelligence system, leading to over-complication which would in turn lead to more unforeseen problems that would require a solution.

However, should one examine the situation in a more abstract manner, it will be evident that A.I. would be overkill since there is no need to identify content that has no relevance to the context but only the need to disregard it.

Reverse-diff (rDiff) analysis will be used in order to facilitate that effect.

By requesting each page twice and passing the two HTML responses to the rDiff algorithm one can identify content that remains unchanged between page reloads (depending on the situation multiple iterations may be used in order to increase precision).

Thus, enabling the system to only deal with content that is pertinent to the current context.

5 A unified algorithm to identify CSRF-worthy forms (4-pass rDiff)

In order for the approaches discussed in this paper to be applicable as a part of an automated system, a unified algorithm must be constructed.

One that will allow for identification of forms that affect business logic and handling of heterogeneous environments.

The author has dubbed this algorithm “4-pass rDiff” and it will be presented both in natural language and pseudocode.

For both representations of the algorithm let:

- URL/url be the URL of the page under audit.
- rDiff/rdiff be an implementation of the rDiff algorithm presented in chapter “Definitions”.

5.1 Natural language

1. Two HTTP GET requests need to be made for the URL with no cookies, so as to simulate a guest (logged-out) user.
2. The 2 response bodies (HTML code) must be passed to the rDiff algorithm in order to remove irrelevant content. Let “HTML1” be the HTML code returned by the rDiff algorithm.
3. Two HTTP GET requests need to be made for the URL with valid session cookies of a logged-in user.
4. The 2 response bodies (HTML code) must be passed to the rDiff algorithm in order to remove irrelevant content. Let “HTML2” be the HTML code returned by the rDiff algorithm.
5. HTML1 must be analyzed in order to extract forms. Let these forms be “Forms1”.
6. HTML2 must be analyzed in order to extract forms. Let these forms be “Forms2”.
7. Let “CSRF-Forms” be the (Forms2 minus Forms1). Thus extracting the forms that only appear upon submission of the cookies of a logged-in user.

5.2 Pseudocode

```
html_guest1 = http_get( url, empty_cookies )
html_guest2 = http_get( url, empty_cookies )
html_guest  = rdiff( html_guest1, html_guest2 )

html_user1  = http_get( url, user_cookies )
html_user2  = http_get( url, user_cookies )
html_user   = rdiff( html_user1, html_user2 )

forms_guest = extract_forms( html_guest )
forms_user  = extract_forms( html_user )

csrf_forms = forms_user - forms_guest
```

6 Conclusion

During testing, 4-pass rDiff analysis has greatly lessened CSRF noise and in many cases removed it completely.

Although this paper presents this technique as a way to identify HTML forms that are more likely to affect business logic, it can also be utilized as a general way to establish context in automated systems and solve a lot of intricate problems like detection of Blind SQL Injections, custom 404 pages – especially ones that include dynamically generated content – and many more.

And it has indeed successfully tackled those problems, in real-world scenarios, as well.

References

1. <http://github.com/Zapotek/arachni>
2. http://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29
3. <http://github.com/Zapotek/arachni/blob/b38bffc85639a3aaa1023aff233059ed779e6fb/lib/module/utilities.rb#L47>