Authorization Enforcement Detection

Ehood Porat
Jerusalem College of Technology
Jerusalem, Israel
ehoodp41@gmail.com

Shmuel Tikochinski
Jerusalem College of Technology
Jerusalem, Israel
molli68@gmail.com

Ariel Stulman
Jerusalem College of Technology
Jerusalem, Israel
stulman@jct.ac.il

ABSTRACT

One of the many aspects of website security is the question of authorization breach. It is an attack in which un-authorized entities are allowed access to restricted space. As the complexity of website code increases, the human capability of handling authorization rules and semantics decreases accordingly. In this project, we demonstrate an automated authorization enforcement detection (AED) tool which allows website administrators to check if they have authorization vulnerabilities on their sites.

CCS Concepts

• **Security and privacy → security services;** *Authentication, Access control, Authorization*

KEYWORDS

Authorization; Cookies; CSRF-TOKEN;

ACM Reference format:

E. Porat, S. Tikochinski, and A. Stulman. 2017. Authorization Enforcement Detection. In Proceedings of 22nd ACM Symposium on Access Control Models and Technologies, Indianapolis, Indiana USA, July 20177 (SACMAT'17), 4 pages.
DOI: 10.1145/3078861.3084172

1 INTRODUCTION

The human mind can handle a lot of data, but is limited at some point. The increase in website code complexity causes the human tester to fail miserably. Our project's goal is to provide automated detection of authorization vulnerabilities; in effect, provide authorization enforcement in websites. We want to provide administrators or their representatives (white-hat pen-testers, etc.) with the knowledge of if and how an attacker can bypass authorization access control. According to the 2016 annual statistical report published by White Hat Security [1] the likelihood that a website will have an *insufficient authorization* based vulnerability is greater than 10%. Even in well maintained sites such as YouTube and Facebook, such authorization breaches exist [2, 3].

The motivation is clear, but is better understood with an (real) example. Creating even the simplest WordPress site using the templates provided, will contain an order of 760 calls to the function "current_user_can" in its source code. This implies that the developer conjured approximately 760 cases which require authorization checks. With such a huge number of cases, can one be sure others weren't missed? Is it even possible to hand check for this kind of vulnerability with the specific environment each of these calls was made from? Many operations are implicitly assumed authoritatively safe as the user isn't supposed to get access to what triggers them. Is this a fair assumption or the product of necessity? An automated process is clearly required.

2 BACKGROUND

2.1 Cookies

Cookies are used by web servers to differentiate between users, and allow for user tailored operations (e.g. pre-saved preferences). Typically, these cookies are provided to the client (browser) by the server as a response to some previous identification mechanism (e.g. the login process). By re-submitting these cookies a client is uniquely identified, providing the server with the mechanism by which it can also deduce whether the user has already been authenticated. Granting access to services or performing other restricted operations is based on this deduction.

2.2 CSRF

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated [4]. Thus, perceived as emanating from an authenticated user, the request will have the privileges of the victim, and is allowed to perform undesired operations (usually *state-changing* requests) on the victim's behalf. [1] places the likelihood of such a vulnerability at over 20%.

The basic mechanism coaxes the user to access some un-wanted site, and exploits the browser's mechanism of

SACMAT'17, June 21-23, 2017, Indianapolis, IN, USA

ACM 978-1-4503-4702-0/17/06…$15.00
http://dx.doi.org/10.1145/3078861.3084172

automatically including all session cookies associated with the site. As both a legitimate request and a forged one originate from the victim, the site has no way of distinguishing between them.

2.3 CSRF Tokens

Synchronizer tokens were specifically introduced to thwart cross-site request forgery (CSRF) attacks [5]. For any state changing request, the server expects to receive a one-time, *random* token, which provides a security mechanism for preventing CSRF attacks. The CSRF token was passed to the client at a prior stage (either as a hidden form field, in a response body containing JSON or others techniques). The server expects the token be returned, rejecting a request if the CSRF token fails validation.

Approximately 35% of vulnerable sites use this technique to remediate their problem [1]. Hence, any automated tool must be able to cope with these tokens in-order to accurately assess a vulnerability's presence.

3 AED

The proposed tool, Authorization Enforcement Detection (AED)[1], allows for an ADMIN to casually surf the website. Behind the scenes, every request made is intercepted by the AED Proxy and then forwarded to the web server. When a response containing a cookie is detected, the corresponding request and response are saved for future analysis. Other request/response pairs are discarded for AED efficiency. We call this the *recording phase*. When the ADMIN stops recording, the analysis phase begins.

3.1 The Analysis

The analysis is composed of two parts. The first is gathering information for sending a proper request, containing valid tokens, with modified cookies. We will refer to this part as *round one*. The actual sending of modified requests and analysis of the response will be referred to as *round two*. Both rounds use a dictionary (hash-table), the format of which is detailed in Tables 1 and 2 for rounds one and two, respectively.

TABLE 1. **RESPONSE Dictionary**

key	***value***	***purpose***

res-id[a]	Another dictionary	{
		Token-1 → [],
		Token-2 → []
		}

Token-1	Array of tokens	Tokens found in the original response.
Token-2	Array of tokens	Tokens found in the response to copy request made in round one.

[a.] *The identifier of the RESPONSE. All RESPONSEs in the dictionary contain a token*

[1] Tool can be found at https://github.com/ehood/aed

TABLE 2. **REQUEST Dictionary**

Key	***Value***	***Purpose***
Req-id[b]	Array of dictionaries	{[{Token-Index → value, Res-id→ value}]}
Res-id	Identifier of the Response	To identify the response from which the token came from.
Token-index	Number	The index of the token in the above response.

[b.] *The identifier of the REQUEST*

3.1.1 Round one. Every recorded request (those that contained cookies in the corresponding response) is replicated. This is done for two reasons: The first is to allow for flagging false positives which can occur due to one-time operations (e.g. file delete); operations which will invariably return different results the second time they are executed. As CSRF-token protection is prevalent, AED must be able to cope with supporting sites as well. Hence replication allows for capturing CSRF-tokens needed for the proper execution of future requests.

The check is achieved by iterating over the response tokens (Table 1), looking for a match in subsequent requests. This proves that the request uses a token. This information is stored in the request dictionary (Table 2), accordingly. As clients must be provided with valid CSRF-tokens before they can re-submit them, AED checks for requests with such embedded tokens. Our core assumption is that a token invariably would have been contained in some preceding response (usually the one directly preceding its use). Before sending the *copy* request, we will replace the expected token (stored in Table 1 token-1) with the corresponding token (stored in the token-2 array).

For extracting tokens from the response, however, a simple difference comparison does not suffice. Although one can safely assume that different responses to the same request will be identical except for the random unique tokens, common ending characters shared between the sequences will crop the tokens early, thwarting the difference comparison technique. Examination of poplar CSRF-token implementations, however, allows for flagging certain characters as the start and end of the token values. These are used in Algorithm 1, *StartChars* and *EndChars* for common starting and ending characters, to extract the tokens from the responses.

Algorithm 1 Find Tokens in responses

```
1:  procedure FindTokens(response1, response2)
2:      i ← 0
3:      while i < length[response1] ⋀ i < length[response2] do
4:          if response1[i] ≠ response2[i] then
5:              j ← k ← i
6:              while i < length[response1] ⋀ response1[i] ∉ EndChars do
7:                  i ← i + 1
8:              end while
9:              while k < length[response2] ⋀ response2[k] ∉ EndChars do
10:                 k ← k + 1
11:             end while
12:             while j ≥ 0 ⋀ response1[j] ∉ StartChars do
13:                 j ← j − 1
14:             end while
15:             if i ≠ k then
16:                 token1 ← token1 + response1[j : i]
17:                 token2 ← token2 + response2[j : k]
18:             end if
19:             response1 ← response1[: i]
20:             response2 ← response2[: k]
21:             i ← 0
22:         else
23:             i ← i + 1
24:         end if
25:     end while
26:     return token1, token2                                    ▷
27: end procedure
```

Algorithm 1 explained: The algorithm loops through the two responses until one terminates. As long as the charcters match, the search index is incremented (lines 22-23). Upon encountring a character discrepancy (line 4), the start of a CSRF-token is suspected. Hence, the end of the token is searched for with the help of *EndChars* (lines 6-8 for the first response, and 9-11 for the second). As tokens might begin with the same sequence (e.g. *AODF11dk* for the first and *AODF05po* for the second), we must reverse search (lines 12-14) to find where the real token begins (with the help of *StartChars*). The tokens are then extracted for storage.

3.1.2 Round two. For every request/response pair recorded in *round one*, we check if the request requires a token for it to be accepted. As the request dictionary was updated in *round one* to contain all requests that have tokens, all pairs not in the dictionary do not require a token. The algorithm used for acquiring an additional token in shown in Algorithm 2.

Algorithm 2 explained: For each item of *requestDictionary*, we check if the response that contained the token is dependent on requests that required tokens themselves in order to operate correctly. Since the ID of a request and its corresponding response is the same, checking the *requestDictionary* at the index of the response containing a token is analogous to checking if the request uses a token. When a token is not required, a simple request is made and the token in the response is extracted and stored in the *tokens* array (lines 11-15). When a token is required, we recursively run the current function on the new request, and assign the extracted token to a temporary variable, *t*. A request is generated containing the new acquired token (stored in *t*), and the returned tokens are extracted from the response (lined 3-9).

After acquiring new tokens, we re-send the requests with modified cookies; expecting response "strings" to differ as un-

Algorithm 2 Get new Token for request

```
1:  procedure GetNewToken(requestId)
2:      for item ∈ requestDictionary[requestId]
3:      if item ∈ requestDictionary[res_id] then
4:          t ← GetNewToken(item[res_id])
5:          response ← sendRequestWithNewTokens(item[res_id], t)
6:          responseOriginal ← GetOriginalResponse(item[res_id])
7:          t1, t2 ← FindTokens(response, responseOriginal)
8:          token ← t2[item[token_index]]
9:          tokens ← tokens + token                  ▷ Tokens is an array
10:     else
11:         response ← sendRequest(item[res_id])
12:         responseOriginal ← GetOriginalResponse(item[res_id])
13:         t1, t2 ← FindTokens(response, responseOriginal)
14:         token ← t2[item[token_index]]
15:         tokens ← tokens + token                  ▷ Tokens is an array
16:     end if
17:     end for
18:     return tokens
19: end procedure
```

authenticated (hence, un-authorized) cookies have been submitted. Similar results are treated as authorization breaches.

The actual analysis is done with ssdeep [6], a program for computing fuzzy hashing. This enables us to compute the *similarity* of "strings". If we find a matches with similarities being greater than 95%, it is tagged as a *breach*. If the similarity score is only greater than 80% but less than 95%, it is flagged *suspicious*. Below these thresholds the request is marked as safe.

4 SIMILAR TOOL

The only tool that remotely resembles our work is *Autorize*, a Burp add-on [7]. The solutions, however, differ. Besides dealing with CSRF-token enabled sites which *Autorize* cannot cope with, AED preforms a deeper analysis than simply comparing response codes and length. This is a very important distinction, as similar pages with minute differences (e.g. different embedded name tags) will not be detected by *Autorize* as the page length varies. The analysis performed by AED, will detect these variations.

5 FUTURE IMPROVEMENTS

AED's performance can be further improved by enabling the user to interact with the tool (e.g. enter a valid CSRF-Token), allowing for specific tweaks that might not be caught with the automated deduction mechanism. This will reduce false negatives. In addition, filters can be added to remove specific requests the user doesn't want inspected. Currently, longer session degrades the tool's performance, as many inconsequential paths are automatically inspected; hence, manual or automatic filtering would lighten the tool's load, further improving its performance.

6 DEMONSTRATION SCENARIO

We will present a template bases WordPress site, which, as mentioned in the section 1, contains multiple calls to the

"current_user_can" function for authorization checking of the current user. An authorization vulnerability will be deliberately inserted to a page the user was not supposed to have accessed. We then show that by omitting even one call to the aforementioned function, the entire site can be manipulated at will.

The demonstration will include not only simple pages, but also pages that are "protected" with CSRF-tokens. We will detect, duplicate and extract the tokens needed for AED to properly find the vulnerability.

REFERENCES

[1] "Web applications security statistics report", White Hat Security, 2016. Online: https://info.whitehatsec.com/rs/675-YBI-674/images/WH-2016-Stats-Report-FINAL.pdf

[2] Swati Khandelwal, "How hackers could delete any YouTube video with just one click", The Hacker News. April 1, 2015. Online: http://thehackernews.com/2015/04/hack-delete-youtube-video.html

[3] Swati Khandelwal, " Facebook Vulnerability Allows Hacker to Delete Any Photo Album", The Hacker News. Feb. 12, 2015. Online: http://thehackernews.com/2015/02/hacking-facebook-photo-album.html

[4] Cross-Site Request Forgery (CSRF). OWASP. Online: https://www.owasp.org/index.php?title=Cross-Site_Request_Forgery_(CSRF)&oldid=227768

[5] Wichers, D., Petefish, P., Sheridan, E., Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. OWASP. Online: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[6] SSDEEP. Online: http://dfrws.org/sites/default/files/session-files/paper-identifying_almost_identical_files_using_context_triggered_piecewise_hashing.pdf

[7] Tawily, B., and Dotta, F., Autorize. Online: https://github.com/Quitten/Autorize/blob/master/Autorize.py

[8] Video of the tool: https://www.dropbox.com/s/7up6h46g2zew4pu/AED.mp4?dl=0