Prevent Cross-site Request Forgery: PCRF

Sooel Son
University of Texas, Austin
samuel@cs.utexas.edu

Abstract

CSRF attacks are one of the most prevalent and dangerous web threats at the level of XSS, SQL injection. In a CSRF attack, an adversary exploits a victim’s identity or the integrity established between a client and a server. An adversary can send any web request which would be allowed by the web server under the victim’s identity. There are many defense mechanisms against CSRF. Most existing defense plots focus on giving a random factor to the web page requests in order for page requests to be difficult to forge. However, each defense plot has its own disadvantage. In this paper, I propose one way to efficiently prevent CSRF attacks toward PHP web applications. This defense system is called PCRF: Prevent Cross-site Request Forgery attack. PCRF provides an automatic robust solution against CSRF threats by using a CSRF token. Due to the property of cryptographically secure hash function, PCRF does not require the server to store every issued token or the relation between sessions and tokens. I provide experimental results to prove that my system can defend CSRF attacks from known CSRF vulnerabilities of open-source web applications.

1. Introduction

In current web environments, many companies have tried to provide safe web services for customers and protect them from the web threats. However, there are many existing web attacks including XSS [10] and SQL injection [11]. CSRF is one of the most subtle and dangerous problems in the current web environment. CSRF threats date back to the 1990s [4], and have compromised many web servers. As a striking example, in Feb 2008, Korea’s most popular eBay’s site, Auction, was hacked by a CSRF attack; 18 million people’s personal information were revealed on the internet due to this attack [3].

CSRF uses cross-site request forgery to induce a user’s browser to send forged requests. Many existing prevention systems use the model of challenge and response to defend this threat. In Session 2, I present the cause of CSRF and the limitations of each existing model.

In this paper, I introduce PCRF system to prevent cross-site request forgery attacks. PCRF is similar to other existing systems which use a CSRF token. However, instead of storing every token to the local storage in a web server, PCRF uses a cryptographically secure hash function to verify whether the token has been previously issued from servers.

To implement PCRF, I made a PCRF filter generator which generates filters automatically based on existing PHP source codes. The PHP[12] filters which are generated from the PCRF filter generator monitor requests and modify responding HTML codes. By
modifying a setup file of the APACHE server, PCRF filters are registered in the web server and then dynamically inject tokens in HTML codes or monitor requests.

To prove the robustness of PCRF against CSRF attacks, I conducted two experiments for well-known open-source PHP web applications: phpMyAdmin [2] and WordPress[15]. In those experiments, PCRF demonstrates its ability to block known CSRF vulnerabilities.

Like other defense systems which use tokens, PCRF is weak against both passive and active attackers. However, in the application layer level, PCRF creates a robust wall against CSRF threats without any modification of existing PHP source codes or the structure of existing databases.

2. CSRF threats

Figure 1 shows the overall structure of the CSRF exploit. The purpose of this attack is to make forged requests with the victim’s identity. After victims pass the authentication procedure of the web server, victims get the proof of their identities such as a session identifier or a cookie. The adversaries’ goal is to induce victims to make forged requests with those proofs of their identities. To achieve this, adversaries force the victim’s browser to send forged requests which victim does not notice it.

The main cause of this attack is the weak binding between requests and the victim’s identity. In the real world, many web servers ensure that the requests from the user who already pass the authentication are actually originated from the user. However, adversaries have many ways for victims’ browsers, not victims, to send the forged requests. Through spam mail, third-party web sites and known vulnerabilities of open source web applications, adversaries can make victims’ browsers to execute malicious java scripts which can initiate forged requests.

Login CSRF is another variation issued by Barth [5]. Instead of making forgery requests with the identity of victim, this attack forces victims to login to a target site using the adversary’s identity. The objective of Login CSRF is to collect victim’s personal data. The victims do not know that their identity of the website has been changed to the adversaries’ on the web site. Therefore, they input personal data to the web server such as credit card numbers or search keywords. Moreover, by using a Gadget [5], adversaries can execute malicious gadgets registered to the adversaries’ accounts within the victim’s local machine.

Both CSRF and Login CSRF are subtle exploitations based on the common belief that every web request made by a browser is initiated under users’ supervision. However, in the real world, there are many ways for malicious adversaries to initiate page requests from the victim’s browser.

3. Existing CSRF defense mechanism.

There are many existing defense plots to prevent or block CSRF attacks. Most defense schemes are classified in two categories. The systems in the first category make forgery requests harder for adversaries. Many systems adopt the scheme of CSRF tokens to achieve this objective. Every web transaction which needs to be protected starts from the users’ page requests. By adding tokens to those page requests, the defense system can check whether each page request contains a valid token. In short, the security against CSRF relies on the fact that adversaries cannot make and correctly guess a victim’s token. If the attacker has or can guess victim’s tokens, the web server will pass attacker’s forgery requests and then tokens are meaningless.
Figure 1. Basic Cross-site Request Forgery attack request flow [9]

The objective of the second category is to confirm the origin of page requests. CSRF attacks come from the cross site domain, not from inside the domain. By acquiring the integrity of requests’ origins, defense system can ignore page requests come from the cross-site domains because web transactions are usually intended for the requests initiated from the same domain, not the cross sites.

3-1. Existing token based mechanisms.

NoForge[6] uses a proxy server and tokens to protect systems against CSRF threats. NoForge bounds the defense token with the session identifier by building a relation table in the database. A proxy server is used to append tokens to every link and form action. However, due to the dependency of tokens to session identifiers, tokens cannot be used before the legitimate session is established. Thus, NoForge cannot defend against Login CSRF attacks.

CSRFx[7] and CSRF Guard[8] also use tokens. Both use the same notion as NoForge. However, instead of using the proxy server, CSRFx uses the external interface of the Apache web server program and CSRF Guard uses the external interface of the Tomcat web server. Their filter files are registered to the web server; they work as a token verifier or a generator like a proxy server in NoForge.

These systems aim to dynamically block CSRF attempts without manual modifications of web applications. Therefore, performance issues are crucial for these solutions because of the property of dynamic approach.

The common weak point is that these mechanisms use a fixed token for each session to avoid complex token managements. If an adversary has the opportunity to see a victim’s token, she can use the same token as long as a session is alive.

3-2. the origin of page requests

HTML protocol provides the referer header to show the origin of requests. Because of privacy concerns and the fact that referer headers are easily suppressed over the internet, a
referrer header is not enough to ensure the integrity of the requests’ origins. Instead of a referral header, Barth and Jackson introduced a custom header called origin header [5]. The effectiveness of this method lies in the difficulty of forging the header. Neither adversaries nor users can change header information of web pages issued from the server. To change the header information, the attacker must go below the level of application layer or the browser. However, due to the property of CSRF, it is hardly achievable to induce victims to go lower than the application layer. Most CSRF exploitation codes are made in JAVA scripts and HTML codes. However, JAVA scripts and HTML codes do not possess the ability to change headers. Therefore, as long as the origins of requests are totally believable, web servers can adopt policies that requests to specific transactions from cross-site domains are ignored.

4. Preventing Cross Site Request Forgery

PCRF is a dynamic token generating defense scheme against CSRF. PCRF’s basic goal is to prevent CSRF attacks by adding a fresh token to every web request whose target page should be protected. To achieve this goal, every page request whose destination should be protected must contain a valid token. PCRF provides filters between clients and a server to automatically handle token management. The filter determines whether an incoming page request must be checked and then decides if an incoming CSRF defense token is valid. This filter also adds CSRF defense tokens to the part of a HTML code where page requests are initiated. Due to the intervention of PCRF filters between request initiators and web servers, web servers are protected from CSRF attacks.

However, PCRF uses static information from PHP source codes to determine the range of protection. PCRF adopts the basic notion and the location of working filters in the system from CSRFx [7]. Instead of storing every token to a database, PCRF introduces a hash function and an automatic PHP source analyzing tool.

4-1. The PCRF filter generator

Figure 2 shows the overall structure of the PCRF system. To generate filters, PCRF requires PHP web application source codes, the target of protection. After PCRF gets the PHP source codes, it analyzes them and then finds out the PHP web pages which should be under protection. The list of these PHP files is called the list of protection. Generated filters are adapted to the web servers based on the list of protection. These filters prevent CSRF attacks.

The whole objective of the PCRF filter generator is to make the list of protection. For practical reasons, the system cannot block all requests because certain web pages must allow requests which do not have valid tokens for accessibility. For example, if a login page, the first page for users to use whole services, is under protection, to access that page, a user must send her request with a valid token. Otherwise, the web server would refuse the request because of the absence of the valid token. However, a user does not have token information because she did not access any web page before. In short, the web server must provide the starting pages which a user can access without a token. For this reason, web server must check the validity of tokens for specific web pages, not all pages under the domain. To achieve this objective, the PCRF filter generator automatically makes the list of protection after analyzing the source codes of PHP applications. In special cases, a server administrator should manually modify the list of protection.
4-2. PCRF pre-filter and post-filter

PCRF filters consist of two parts: the pre-filter and the post-filter. The first one validates tokens from clients; the second generates tokens and attaches them to web pages. Figure 3 demonstrates the location of each filter and its responsible parts. To validate CSRF tokens which come from clients, PCRF pre-filter intercepts the user request to certain web pages on the list of protection, and then validates tokens which come up with the user requests.

The PCRF post-filter generates tokens and injects tokens into the original HTML at the time when the page is delivered to the clients from the server. After a web server generates a web page for a page request, the PCRF post-filter in the web server checks an HTML page, generated from the web server, and then injects the tokens. Finally, the modified web page is transferred to a request initiator. The filter decides whether tokens must be added to the page based on HTML codes. If the generated HTML contains a submission form through POST or GETS methods, the post-filter automatically attaches a generated token to the submission routine. The PCRF post-filter also checks every link in the HTML code and the source of the frameset and then adds tokens only if the destinations of links and the sources of the frameset are under the web server’s domain. Therefore, every outgoing link does not contain tokens; tokens are attached to only the links to the web server itself. Tokens are added to incoming links because tokens are only needed for the verifying server. It also decreases the possibility of revealing the tokens to adversaries outside the web server. Figure 4 demonstrates the part of a HTML code after passing the post-filter. The PCRF post-filter adds tokens to HTML codes with the name of ‘PCRFt.’

4-3. Token generation and protocol

The whole security of PCRF against CSRF depends on the token generation and validation. The basic scheme of token validation is to ensure that incoming tokens with the
requests are issued from the web server. Actual token generation rule is as follows:

\[
\text{PCRF token} = \{\text{Timestamp, Nonce}, \\
\quad \text{Hash (Timestamp, [SID], Nonce, Client IP, Client Info, SK)}\} \\
\text{(SK: web server’s secret key, SID: session identifier, optional)}
\]

Every PCRF token has a nonce for freshness and a timestamp to determine the token’s living time. To bind the token to the user’s browser, the PCRF token should have the user’s local knowledge which is difficult for adversaries to guess. If the session is already established between users and servers, PCRF uses the session identifier. Otherwise, PCRF uses the name of the client’s browser, version and the type of operating system as client information. PHP provides this information through the reference of a server table: \$_SERVER[‘HTTP_USER_AGENT’]. Figure 5 shows the protocol of the PCRF token.

Because of the properties of a cryptographically secure hash function, CSRF tokens cannot be made by adversaries unless they know SK. The only way to get a valid token is to obtain from the web server. Therefore, the attacks to anonymous people are infeasible without previous knowledge about a victim’s IP and SID or local information.

An adversary might use her tokens to induce forgery requests from the victim. However, these attempts will be hard to achieve because attackers must guess a client’s IP and local information. Moreover, every PCRF token has a living time. To succeed at a replay attack, an adversary must get a victim’s IP and local information and then receive tokens from server with the spoofed IP address. Moreover, by using that token, an adversary must induce victims to send the request with that token during the living time of the token.

![PCRF token protocol](image)

**Figure 5. PCRF token protocol**

### 4.4. Contribution

Many existing CSRF defense schemes use token matching algorithms. However, these systems store issued tokens on a database in web servers to map tokens to user’s identity or session. Moreover, due to the close relation between tokens and session identifiers, these systems cannot defend against the Login CSRF attack.

PCRF overcomes those two shortages by importing the hash function. The chain between the user and the token’s ownership is not on the session identifier but on the client’s IP address and local information. Therefore, the web server can check the legitimacy of the tokens before the sessions are established. PCRF checks whether the token received from a requestor were previously issued to that requestor. Therefore, to circumvent the checking routine, an adversary must know the victim’s IP address and local information which are used to get the token. Moreover, an adversary must request the token to the web server by spoofing her IP to get the forged token before the victim’s token’s living time is expired. In short, PCRF can be weak against both passive and active attackers like other existing defense
systems. However, this limitation is the fundamental issue which every IP based solution has because IP address is not a unique feature of each computer and IP can be spoofed for malicious purposes. For example, PHP session management is also weak against IP spoofing attacks.

PCRF requires a small amount of system resources to check tokens. The only resource required is the power to compute a hash function for every request with a token. Therefore, PCRF is much lighter than other systems which require big database tables to manage tokens and sessions.

5. Implementation

The implementation of PCRF consists of two filters and a program which generates filters. The PCRF routine, which generates two filters, is made in JAVA. This program requires the user’s input which indicates the location of PHP source files. The JAVA program checks all PHP files under the user’s specified domain. It then finds all PHP files which contain contexts that web servers execute transactions based on the information received from a page request through POST or GET methods. The algorithm finding PHP files relies on the string matching not PHP parsing. The two filters generated from this program are actually two PHP files: ‘prehead.php’ and ‘posttail.php.’ The JAVA program automatically sets a web server to include those two filter files. The APACHE web server suggests outer interfaces to locate the filters between the user and the HTML generator engine. Thus, by registering these filters to the APACHE web server, every incoming request that occurs before the HTML is generated, and outgoing responses after generating the HTML, is intercepted or hijacked by two filters. These two PHP source codes are executed inside the server whenever web servers accept page requests. As mentioned in Section 4, these two filters monitor every page requests to web servers and check the validity of tokens if the page request’s target page is on the list that should be protected.

To test the capability of PCRF defending against CSRF attack, I used some sample source codes and two PHP open-source web applications: phpMyAdmin[2] and

![Image of code and output](image.png)

Figure 6. The exploiting code of known CSRF vulnerability of phpMyadmin and the result of victim’s server

6. Experiments

To test the capability of PCRF defending against CSRF attack, I used some sample source codes and two PHP open-source web applications: phpMyAdmin[2] and
WordPress[15]. ‘phpMyAdmin’ is the web application program for web server managers to change and maintain the database easily without any manual SQL query inputs. phpMyAdmin also uses the token to defend the CSRF attack. However, it is not enough because phpMyAdmin version 2.11 has a hole to pass the simple exploit which can cause for attackers to add millions of meaningless databases through the CSRF attack [1]. Figure 6 shows a simple exploit code and the consequence of this exploit.

However after introducing PCRF filters to the web servers, PCRF blocked this exploit completely. It is too obvious why the exploit cannot succeed. Every request to web pages on the list of protection should come with the legitimate token. However, in this case, forgery requests do not have any token.

I also had an experiment for WordPress, an open-source PHP web application for the management of personal blogs. WordPress version 2.1.1 has a CSRF/XSS vulnerability [14]. By succeeding at a forgery request, attackers can execute malicious java script codes in the victim’s browser. The starting point of this attack is the CSRF vulnerability of WordPress. PCRF also blocked this threat based on the verification of tokens.

7. Limitation.

PCRF has certain limitation because of the implementation design. The pre-filter and the post-filter use an internal buffer in the PHP engine to implement filters. Every web page generated by the web server is not directly sent out but is stored in the internal buffer. The post-filter flushes a web page stored in the internal buffer after finishing all transactions in the web server. Thus, if a web application uses the function calls that manage internal buffers, such as ob_start() and ob_flush(), PCRF filters cannot work well.

To prepare the case that the web application is suddenly terminated during the execution by using die() and exit() functions, PCRF registers a shutdown handler to the PHP engine. Therefore, when a web application registers its own shutdown handler to the PHP, PCRF might lose the chance to prevent CSRF attacks.

PCRF has one assumption that the web page initiating a page request and the web page accepting that request is different. Sometimes, a web page’s request target is the web page itself. If this page should be protected, PCRF would report a false alarm. The requests to access the web page on the list of protection must contain the valid tokens. Therefore, users cannot access that web page by manually typing a domain address without the tokens. To avoid this case, a server manager who would use PCRF should manually modify the list of protection.

In the PCRF filter generator, to make the list of protection, the filter generator analyzes PHP source codes. Filter generator finds every web page which contains couple of keywords which require client’s data such as $_POST[] and $_GET[]. Because the generator’s finding algorithm is the string matching, not parsing PHP source codes, the list of protection might not contain some files which should be on the list. Therefore, manager should manually add the file to the list for the soundness. To avoid this, I tried to make a PHP parser with JFlex and CUP for a month. I succeeded to make the hierarchy of pages. However I didn’t have much time to make the control or data flow of PHP source codes. Thus, PCRF remains some portions to improve for the soundness of system.

8. Conclusion

Due to the weak chain between request initiators and established sessions’ identities, naïve web servers are vulnerable to CSRF threats. To prevent this attack, many defense systems such as NoForge and CSRFx suggest verifying tokens. These systems make the
connection between a token and a session id to check the token validity. However, because of the dependency on the session identifiers, these systems cannot defend against Login CSRF attacks.

PCRF approaches this attack in a different way. PCRF uses tokens to ensure that request were issued from the web server previously. Instead of using session identifiers, PCRF uses clients’ IP address and local information. These are the ingredients of a cryptographically secure hash function which binds tokens to users. In short, the fact that PCRF tokens do not rely on sessions provides the safety against not only Login CSRF attacks but also normal CSRF attacks.

Although PCRF is weak against an eavesdropper in the middle, like other existing token defense systems, PCRF suggests an efficient way to prevent CSRF attacks with a little change to the web server. PCRF filters dynamically add tokens to a HTML code which goes to clients and monitors the request without changing the legacy source codes. Moreover, PCRF doesn’t need an additional database table to store the tokens.

In conclusion, I suggest that PCRF can create a robust wall against CSRF attacks for many vulnerable web servers on the Internet with the light computation of hash functions.

References


