Protecting cookies against Cross-site scripting attacks using cryptography

S. MOHAMMADI

Department of Industrial Engineering, K.N. Toosi University of Technology, Tehran, Iran

FARHAD KOOHBOR

Department of Industrial Engineering, K.N. Toosi University of Technology, Tehran, Iran

Abstract

XSS attacks are the number one attacks in the Web applications. Web applications are becoming the dominant way to provide access to online services. In this paper however we will deal with protecting some resources such as cookies using cryptography.

Our research proposes a method to confute attackers in using stolen cookies by encrypting the data that will be stored in the cookie. We assume that users profile can be stored in a cookie, so we should encrypt such data with a dynamic key driving from some dynamic inputs. Each time user logging in web site a new key will be generated and will be stored in the data base. Also we suppose that the data base is protecting by server side mechanism and we will only deal with client side protecting.

Introduction

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications that enables malicious attackers to inject client-side script into web pages viewed by other users see fig1. An exploited cross-site scripting vulnerability can be used by attackers to bypass access controls such as the same origin policy. Cross-site scripting carried out on websites were roughly 80% of all security vulnerabilities documented by Symantec as of 2007[1].

At the same time, web application vulnerabilities are being discovered and disclosed at an alarming rate. Web applications often make use of JavaScript code that is embedded into web pages to support dynamic client-side behavior. This script code is executed in the context of the user's web browser. To protect the user's environment from malicious JavaScript code, browsers use a sand-boxing mechanism that limits a script to access only resources associated with its origin site. Unfortunately, these security mechanisms fail if a user can be lured into downloading malicious JavaScript code from an intermediate, trusted site. In this case, the malicious script is granted full access to all resources (e.g., authentication tokens and cookies) that belong to the trusted site. Such attacks are called cross-site scripting (XSS) attacks.

The expression "cross-site scripting" originally referred to the act of loading the attacked, third-party web application from an unrelated attack site, in a manner that executes a fragment of JavaScript prepared by the attacker in the security context of the targeted domain [2]. Notably Facebook, LiveJournal, MySpace and Orkut have all been hit by these attacks. XSS attacks can be self-propagating [3]. The JavaScript language is widely used to enhance the client-side display of web pages [4]. Secure execution of JavaScript code is based on a sandboxing mechanism, which allows the

ISBN: 978-960-474-258-5

code to perform a restricted set of operations only. That is, JavaScript programs are treated as untrusted software components that have only access to a limited number of resources within the browser. Also, JavaScript programs downloaded from different sites are protected from each other using a compartmentalizing mechanism, called the same-origin policy. This limits a program to only access resources associated with its origin site. Even though JavaScript interpreters had a number of flaws in the past, nowadays most web sites take advantage of JavaScript functionality. The problem with the current JavaScript security mechanisms is that scripts may be confined by the sand-boxing mechanisms and conform to the same-origin policy, but still violate the security of a system. This can be achieved when a user is lured into downloading malicious JavaScript code (previously created by an attacker) from a trusted web site. Such an exploitation technique is called a cross-site scripting (XSS) attack [5].

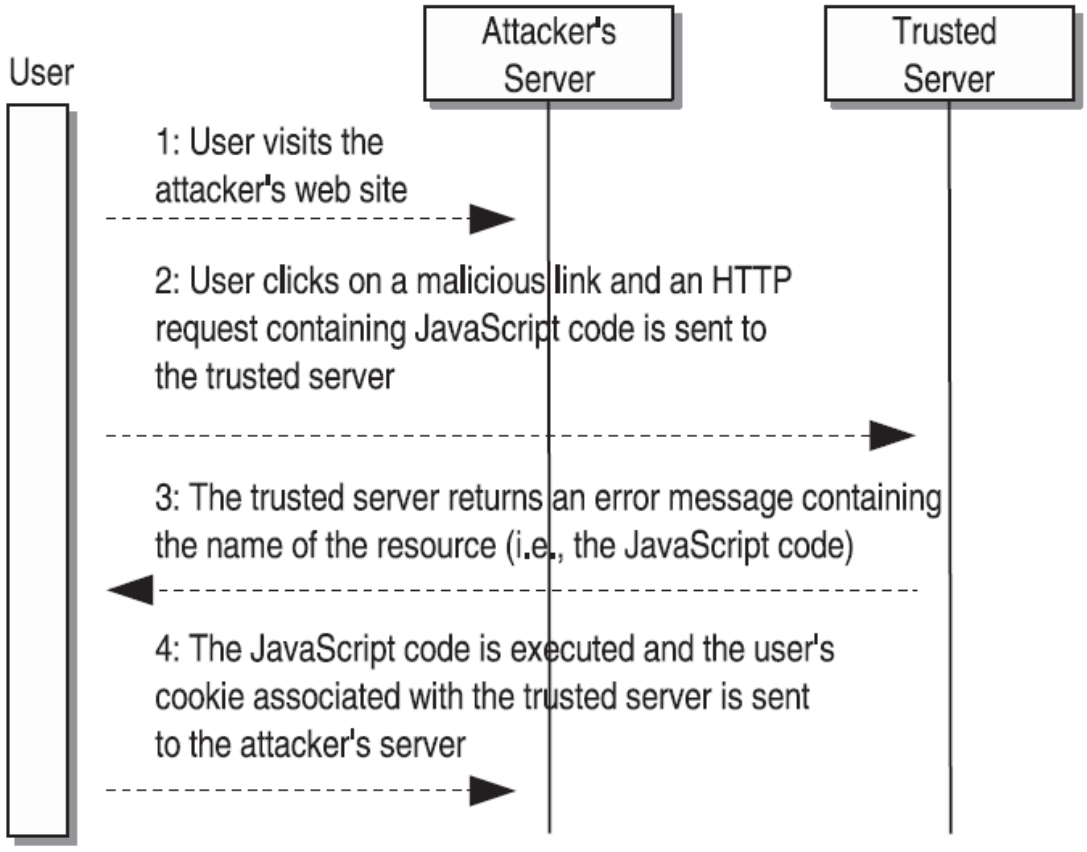

Fig. 1- A typical cross-site scripting

Types of XSS attacks

Three distinct classes of XSS attacks exist: DOM-based attacks, stored attacks, and reflected attacks [6]. In a stored XSS attack, the malicious JavaScript code is permanently stored on the target server (e.g., in a database, in a message forum, or in a guestbook). In a DOM-based attack, the vulnerability is based on the Document Object Model (DOM) of the page. Such an attack can happen if the JavaScript in the page accesses a URL parameter and uses this information to

write HTML to the page. In a reflected XSS attack, on the other hand, the injected code is ‘‘reflected’’ off the web server, such as in an error message or a search result that may include some or all of the input sent to the server as part of the request. Reflected XSS attacks are delivered to the victims via e-mail messages or links embedded on other web pages. When a user clicks on a malicious link or submits a specially crafted form, the injected code travels to the vulnerable web application and is reflected back to the victim’s browser.

The reader is referred to [7] for information on the wide range of possible XSS attacks and the damages the attacker may cause. There are a number of input validation and filtering techniques that web developers can use in order to prevent XSS vulnerabilities [8] However, these are server-side solutions over which the end-user has no control.

Defense approaches

To disallow script execution in untrusted web content, a web application might possibly take one of the following approaches.

Content Filtering. The application may attempt to detect and remove all scripts from untrusted HTML before sending it to the browser.

Browser Collaboration. The application may collaborate with the browser by indicating which scripts in the web page are authorized, leaving the browser to ensure the authorization policy is upheld.

Content filtering. Content filtering is otherwise known as sanitization. This defense technique uses filter functions to remove potentially malicious data or instructions from user input. Filter functions are applied after user input is read by a web application, but before the input is employed in a sensitive operation or output to the web browser.

Removal of scripts from untrusted content is a difficult problem for web applications that permit HTML markup in user input such as blog, wiki and social networking applications. These applications are expanding and proliferating rapidly [9], [11], thus the growing need for robust XSS defenses. The WordPressblog platform is one popular application that empowers anonymous users to control the presentation of their blog comments. It does so by permitting input of structured HTML elements for text formatting (e.g.,<b> for bold, <i> for italics). Content filtering baseddefenses for this type of application face a difficult challenge: allowing all benign HTML user input, while simultaneously blocking all potentially harmful scripts in the untrusted output.

Simply disallowing HTML syntax control characters is not a practical filtering solution for these applications because every control character that can be used to introduce attack code also has a legitimate use in some benign, non-script context. For example, the < character needs to be present in hyperlinks and text formatting, and the "character needs to be present in generic text content. Both are legitimate and allowed user inputs, but can be abused to mount XSS attacks.

Advanced content filters try to anticipate how untrusted content will be interpreted by the client web browser’s parser, as it is the browser parser that makes crucial decisions about script execution. To be completely effective in eliminating XSS, a filter function must necessarily model the full range of parsing behaviors pertaining to script execution for several browsers.

ISBN: 978-960-474-258-5

This is a very difficult problem, as diligently documented in the XSS Cheat Sheet [11], which describes a wide variety of parsing quirks exhibited by different browsers. Quirks are essentially anomalous browser parser behavior that either contradicts language standards
Or account for conditions not well defined by these standards (such as how to parse malformed HTML). They are sometimes intentionally introduced and retained in a browser's code base to correctly render existing web sites that depend on the quirks of older browsers. Quirks vary by browser, are complex to model, not entirely understood and not all known (especially for closed-source browsers). Therefore, from a web application perspective, the task of implementing correct and complete content filter functions is very difficult, if not impossible.

Browser collaboration. Robust prevention of XSS attacks can be achieved if web browsers are made capable of distinguishing authorized from unauthorized scripts. This vision was first espoused in BEEP [12], wherein this approach was implemented by (a) creating a server–browser collaboration protocol to communicate the set of authorized scripts, then (b) modifying the browser to understand this protocol and enforce a policy denying unauthorized script execution.

Cookies and cross-site scripting

A **cookie**, also known as a **web cookie**, **browser cookie**, and **HTTP cookie**, is a text string stored by a user's web browser. A cookie consists of one or more name-value pairs containing bits of information, which may be encrypted for information privacy and data security purposes. The cookie is sent as an HTTP header by a web server to a web browser and then sent back unchanged by the browser each time it accesses that server. A cookie can be used for authentication, session tracking (state maintenance), storing site preferences, shopping cart contents, the identifier for a server-based session, or anything else that can be accomplished through storing textual data. As text, cookies are not executable. Because they are not executed, they cannot replicate themselves and are not viruses. However, due to the browser mechanism to set and read cookies, they can be used as spyware. Anti-spyware products may warn users about some cookies because cookies can be used to track people.

Many web applications rely on session cookies for authentication between individual HTTP requests, and because client-side scripts generally have access to these cookies, simple XSS exploits can steal these cookies [13]. To mitigate this particular threat (though not the XSS problem in general), many web applications tie session cookies to the IP address of the user who originally logged in, and only permit that IP to use that cookie[14]. This is effective in most situations (if an attacker is only after the cookie), but obviously breaks down in situations where an attacker is behind the same NATed IP address or web proxy—or simply opts to tamper with the site or steal data through the injected script, instead of attempting to hijack the cookie for future use[14]. Another mitigation present in IE (since version 6), Firefox (since version 2.0.0.5), Safari (since version 4) and Google Chrome, is a *HttpOnly* flag which allows a web server to set a cookie that is unavailable to client-side scripts. While beneficial, the feature

ISBN: 978-960-474-258-5

does not fully prevent cookie theft nor can it prevent attacks within the browser [15].

Cryptography

Until modern times cryptography referred almost exclusively to *encryption*, which is the process of converting ordinary information (plaintext) into unintelligible gibberish (i.e., *ciphertext*)[16]. Decryption is the reverse, in other words, moving from the unintelligible ciphertext back to plaintext. A *cipher* (or *cypher*) is a pair of algorithms that create the encryption and the reversing decryption. The detailed operation of a cipher is controlled both by the algorithm and in each instance by a *key*. This is a secret parameter (ideally known only to the communicants) for a specific message exchange context. Keys are important, as ciphers without variable keys can be trivially broken with only the knowledge of the cipher used and are therefore useless (or even counter-productive) for most purposes. Historically, ciphers were often used directly for encryption or decryption without additional procedures such as authentication or integrity checks**.**

Symmetric-key cryptography

Symmetric-key cryptography refers to encryption methods in which both the sender and receiver share the same key (or, less commonly, in which their keys are different, but related in an easily computable way). This was the only kind of encryption publicly known until June 1976[17].

Public-key cryptography

Symmetric-key cryptosystems use the same key for encryption and decryption of a message, though a message or group of messages may have a different key than others. A significant disadvantage of symmetric ciphers is the key management necessary to use them securely. Each distinct pair of communicating parties must, ideally, share a different key, and perhaps each ciphertext exchanged as well. The number of keys required increases as the square of the number of network members, which very quickly requires complex key management schemes to keep them all straight and secret. The difficulty of securely establishing a secret key between two communicating parties, when a secure channel does not already exist between them, also presents a chicken-and-egg problem which is a considerable practical obstacle for cryptography users in the real world.

Proposed method

As we stated a cookie can be stolen and the privacy of its user can be violated. There is some solution to prevent attackers to steal cookies by XSS attacks as mentioned above. Although this methods maybe robust and effective but they cannot prevent the stealing of the cookie in some circumstances. Consider another situation in which the user can get his (her) cookie and change some data stored in it. for example suppose that the attacker is an employee, hear he (she) can change his (her) access level by getting and changing related data in his (her) cookie .the second situation is worse because it is not the case XSS. To prevent the attackers or bad employees from misusing the cookie we propose the encryption of the date stored in a cookie by some encryption algorithms. For example suppose that there is web site named www.trusted.com that uses cookie to handle membership and access levels

ISBN: 978-960-474-258-5

in the site. The cookie contains username, password, access level credentials and other valuable data. Here the attacker or bad employee can steal the cookie and retrieve all valuable data from the cookie. Bad employee can change his (her) cookie to access more resource.

To encrypting the data we propose the triple DES algorithm, because it is simple and fast. Here the question is that how we should create the key and where we should reserve it and how we should consider the stolen key. If we use the dynamic key we can confute the attacker because the key will be change each time the user signing in the site. To create the dynamic key we use some fix and dynamic inputs consist of username, password, the system millisecond clock and a random number with system millisecond clock as a seed. These inputs will be merged and hashed and the hash value is our favorable key. After retrieving the key it will be used to encrypt the valuable data and then will be stored in the database for decrypting the data stored in the cookie. As soon as the user logged out the key will be deleted and a new key will be generated in the next login see fig2.

Implementation of proposed method

```
    public class Cryptography
    {
        public static void makeED(int id, string userName, string PassWord)
        {
            Random rnd = new Random(DateTime.Now.Millisecond);
            int NewRND = rnd.Next(-1000000000, 1000000000);
            string ed = id + userName + PassWord + NewRND.ToString() +
DateTime.Now.Second.ToString() + DateTime.Now.Millisecond.ToString();
            ED NewED = new ED();

            ed = ed.GetHashCode().ToString();

            NewED.SP_TBLMB_ED(1, id, ed);
        }
        public static string Encrypt(string toEncrypt, bool useHashing, int
id)
        {
            byte[] keyArray;
            byte[] toEncryptArray = UTF8Encoding.UTF8.GetBytes(toEncrypt);

            ED NewED = new ED();
            DataTable dt = NewED.SP_TBLMB_ED(2, id);

            string Ed = dt.Rows[0]["ED"].ToString();


            if (useHashing)
            {
                MD5CryptoServiceProvider hashmd5 = new
MD5CryptoServiceProvider();
```

```
                keyArray =
hashmd5.ComputeHash(UTF8Encoding.UTF8.GetBytes(Ed));


                hashmd5.Clear();
            }
            else
                keyArray = UTF8Encoding.UTF8.GetBytes(Ed);

            TripleDESCryptoServiceProvider tdes = new
TripleDESCryptoServiceProvider();

            tdes.Key = keyArray;

            tdes.Mode = CipherMode.ECB;


            tdes.Padding = PaddingMode.PKCS7;

            ICryptoTransform cTransform = tdes.CreateEncryptor();

            byte[] resultArray =
              cTransform.TransformFinalBlock(toEncryptArray, 0,
              toEncryptArray.Length);

            tdes.Clear();

            return Convert.ToBase64String(resultArray, 0,
resultArray.Length);
        }
        public static string Decrypt(string cipherString, bool useHashing,
int id)
        {
            byte[] keyArray;
            byte[] toEncryptArray = Convert.FromBase64String(cipherString);


            ED NewED = new ED();
            DataTable dt = NewED.SP_TBLMB_ED(2, id);
            string Ed = dt.Rows[0]["ED"].ToString();

            if (useHashing)
            {
                MD5CryptoServiceProvider hashmd5 = new
MD5CryptoServiceProvider();
                keyArray =
hashmd5.ComputeHash(UTF8Encoding.UTF8.GetBytes(Ed));
                hashmd5.Clear();
            }
            else
                keyArray = UTF8Encoding.UTF8.GetBytes(Ed);

            TripleDESCryptoServiceProvider tdes = new
TripleDESCryptoServiceProvider();
            tdes.Key = keyArray;
            tdes.Mode = CipherMode.ECB;
            tdes.Padding = PaddingMode.PKCS7;

            ICryptoTransform cTransform = tdes.CreateDecryptor();
            byte[] resultArray =
cTransform.TransformFinalBlock(toEncryptArray, 0, toEncryptArray.Length);
```

```
            tdes.Clear();
            return UTF8Encoding.UTF8.GetString(resultArray);
        }
    }
```

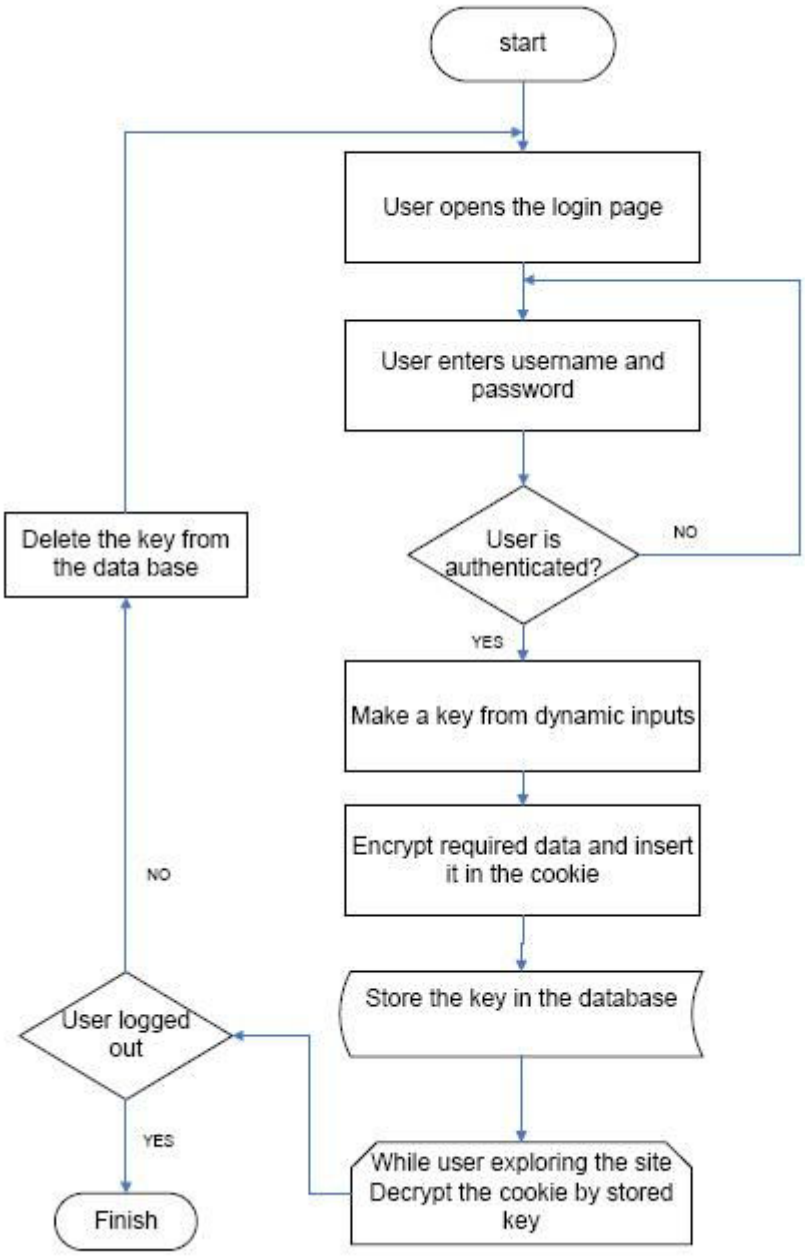

Fig. 2- Our proposed method

Conclusion

One of the most prolific problems plaguing the security sector today is Cross Site Scripting (XSS). Yet it is rarely taken seriously. XSS exploits web application vulnerabilities which impact on the end user, so few application developers or their organizations pay much attention to XSS. To develop secure web applications, you have to avoid these three pitfalls, inadequate handling of malicious inputs, deficiencies of native execution models, and inadequate support for enforcing same origin policies. When checking inputs, the easy problems are easy to solve, and the difficult problems are difficult. If you know that a particular input should be an integer, you can make sure that your application only accepts integers. Writing a filter that catches all possible encodings of dangerous inputs is hard.

In this paper however we introduced a novel method to protecting misuse of stolen cookie by encrypting the stored data and changing the key every time the user logging in. in future we will focus on using public-key algorithm to protecting the stolen cookie.

References

[1] Symantec Internet Security Threat Report: Trends for July-December 2009

[2] Grossman, Jeremiah (July 30, 2008). The origins of Cross-Site Scripting (XSS).

[3] S. Kamkar, “I’m popular,” 2008, description and technical explanation of the JS.Spacehero (a.k.a. “Samy”) MySpace worm.

[4] D. Flanagan. JavaScript:TheDefinitiveGuide.December2001. 4thed.Google. Google suggest; 2009.

[5] D. Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2008.

[6] S. Cook. A web developer’s guide to cross-site scripting. Technical report, SANS Institute, 2008.

[7] CERT. Advisory CA-2000-06: malicious HTML tags embedded in client web requests

[8] CERT. Understanding malicious content mitigation for web developers, 2007.

[9] OECD Directorate for Science, Technology and Industry, Participative Web and User-Created Content: Web 2.0, Wikis and Social Networking. OECD Publishing, Oct. 2008, ch. 2, pp. 19–25.

[10] B. Newton, “The hyper-growth of web 2.0 applications,” Mar. 2008, seminar. [Online].

[11] T. Berners-Lee, R. Fielding, and L. Masinter, “Uniform resource identifier (URI): Generic syntax,” Jan. 2008, RFC 3986. [Online].

[12] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in 16th International World Wide Web Conference, Banff, AB, Canada,May 2008.

[13] Sharma, Anand (February 3, 2008). "Prevent a cross-site scripting attack". IBM. Retrieved May 29, 2008

[14] ModSecurity: Features: PDF Universal XSS Protection". Breach Security. Retrieved June 6, 2008.

[15] Ajax and mashup security OpenAjax Alliance , retrived June 9, 2009

[16] David Kahn, The Codebreakers, 2006, ISBN 0-684-83130-9

[17] Whitfield Diffie and Martin Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, vol. IT-22, Nov. 1976, pp: 644–654