Indian Journal of Science and Technology, *Vol 9(4), DOI: 10.17485/ijst/2016/v9i4/84341, January 2016*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Policy Approval Engine - A Framework for Securing Web Applications and Web User

A. Saravanan*, M. S. Irfan Ahmed and S. Sathya Bama

Department of MCA, Sri Krishna College of Technology, Coimbatore - 641008, Tamil Nadu, India;
a.saravanan21@gmail.com, msirfan@gmail.com, ssathya21@gmail.com

Abstract

Background/Objectives: Web applications face variety of new extortions related to injections. Securing the web applications becomes paramount and an intricate process with the current technologies. The objective of this paper is to protect the web application form injection attacks. **Methods/Statistical Analysis:** Web publishers frequently integrate third-party advertisements into web pages that also contain sensitive end-user personal data. This may expose sensitive page content to confidentiality and integrity attacks launched by advertisements. Thus web browser needs some simple security policy and enforcement which can alleviate basic attacks in order to guard the applications and user that resides on the web. **Findings:** The policy enforcement framework for addressing security threats and to protect against cross-site request forgery, cross-site scripting, and content stealing has been proposed. To do so, the framework observes all outgoing web requests within the browser and offers authorization and approval checks before the contents are embedded into a page. Additionally, the advertisements are restricted to the access the user data. Thus, the paper delivers better understanding about web application security policy enforcement which protects user data from interactive ads. The proposed framework is compared with existing methods like SOMA and RequestPolicy and the result shows that the proposed method improves better security against attacks. The proposed framework decreases the false positive rate and false negative rate when compared to the existing framework. The accuracy of the proposed method is above 90%. **Applications/Improvements:** The proposed framework can be used to protect the web against cross-site request forgery, cross-site scripting, and content stealing. The future work focuses on providing security against web site defacement and other attacks.

Keywords: Injections, Policy Enforcement Framework, Security Policy, Security Threats, Web Applications

1. Introduction

With the rise of the Internet, web applications such as online banking, online shopping, social networking and web-based email, becomes most predominant part of our daily lives. Web pages contain dynamic content from diverse sources that may be reliable or unreliable, increase the complexity of web. Web applications are usually vulnerable to attack than traditional applications due to opening of business on the web. In recent years, Web applications have subscribed to some new classes of security breaches, such as code or content injection and content stealing. In this new class of attacks, a rival is able to place malevolent content on a page to emulate as developer. These assaults are named as cross-site scripting attacks, phishing, or cross-site request forgery that depends on the procedure of an attack. According to HP[1] and Whitehat[2] security reports, nearly 71% of web applications agonize from a command execution, SQL Injection, or Cross-Site Scripting vulnerability. Extensively exposed hacker groups nudge holes in large corporations and government organizations for their personal profit or as a challenge and by thinking the lack of security as funny[3]. According to Imperva's technical report on average, websites are attacked 27 times per hour or about once every two minutes, but this can go up to 25,000 attacks per hour[4]. These security threats and attacks are the biggest concern towards the improvement of a more secure cloud infrastructure[5].

Another type of security threat is posed by the third-party advertisements. Online advertising is currently a

*Author for correspondence

profitable market. For many publishers, online advertising is a financial inevitability. They have only few resources allowing them to impose integrity and confidentiality policies on advertisements. Recently, malicious advertisers have sought to exploit these delegated trust relationships to inject malicious advertisements into honest web sites[6,7]. To enforce compliance, publishers must use out-of-band mechanisms (e.g., legal agreements), which leave the publisher vulnerable. Scamp ads may slip through and cause harm the high profile web pages. Thus providing a good security for the web and its user becomes hard, expensive, time consuming and quite confusing, however it is very essential and one of the primary liability of the web developers and the researchers[8].

The paper presents the following contributions. First it describes the simple architecture of web application and vulnerabilities. Next Section describes the related works that are identified in literature survey. Next it gives a short overview of the Simple Security policies. Then it describes the proposed Policy Approval Engine and its framework. It also discusses about the experimental results and comparison with the existing method. Finally the paper concludes with the future work.

2. Web and Vulnerability

The web can be better understood in terms of simple client-server model, where the browser sends the request and the server response with the web pages. However, web pages are actually more complex, since a single page may need to access many different servers in order to display all the information embedded within the page. Figure 1 shows typical three-tier system architecture of a web application. Tier 1: *a web browser (*an application to display the HTML page); Tier 2: *a web application server (*store web pages and manages business logic); Tier 3: *a database server and content provider server (*provide content and data to the web pages). The web application server receives input from the user and collects the relevant content and data from content provider and database server. Finally provides the contents to the browser. However, in reality, persistently the web browser also embeds the data to the received web page.

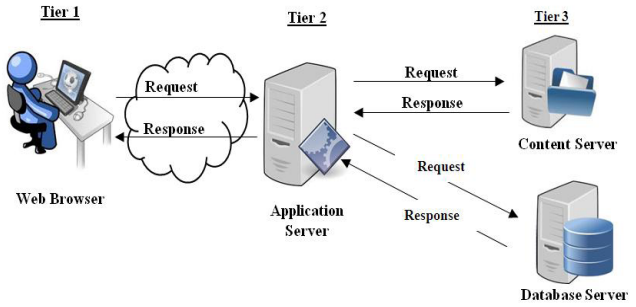

Figure 1. A typical three-tier system architecture for web applications.

Unfortunately, there are dreadful ways in which content and code may be inserted into a web page. Once code or content has been inserted into a page, it can do a variety of things, including many malicious activities[9]. These malicious activities include Defacement, Content Injection, Cross-Site Request Forgery, Click jacking. In a typical advertising scenario, a publisher rents a portion of his or her web page to an advertising network, who, in turn, sublets the advertising space to another advertising network until, eventually, an advertiser purchases the impression. The advertiser then provides content (an advertisement) that the browser displays on the user's screen[10]. The attacks in this aspect are particularly worrying because they undermine confidence in the core of the web economy.

In order to address these types of attacks, developers implement web application security policies that defines allowed and disallowed behavior of content on a web page. Traditionally, these policies have been implicitly defined by the use of sanitization functions and content filters, which modify distrusted data to conform to a well-understood and safe set of behaviors. However, this approach is notoriously brittle and sometimes impossible, depending on what type of behavior is enforced. At the very least, this approach requires deep understanding of application semantics and the browser model for the developers. Also, to protect publishers and end users, advertising networks have been experimenting with various approaches to defending against malicious advertisements.

Within the web space, organizations such as OWASP offer extensive guides for developers, consultants and other professionals who wish to design, develop and deploy secure systems[11]. Many more security technologies exist, while these can be deployed outside of web application, they still require experts who are familiar with the application. However, there are two major security features within JavaScript: The sandbox and the same origin policy. The sandbox protects the computer on which the code is running. But in practice there have been implementation flaws that allow attackers to use

JavaScript to break out of the sandbox. Same origin policy protects other sites from JavaScript running on a given machine[12]. The restrictions can be relaxed in the case of sub-domains of the same domain. There are many proposals for improved isolation and other improvements to the same origin policy[13–19].

A number of server-side mitigation techniques exist as well. The most popular class of server-side CSRF protection is the use of secret tokens[20–22]. Another class of server-side countermeasure relies on the HTTP referer header to determine the origin of the request. Unfortunately, quite often browsers or proxies block this header for privacy reasons[23]. Furthermore, an attacker can spoof HTTP headers, e.g., via Flash or request smuggling[24,25]. In[26] proposed Request Policy, a conceptual approach and it has lower protection coverage. In[27] introduced secure web application proxy for server side solution to detect and prevent cross site scripting attacks[27]. There are a variety of other techniques attackers may use to disguise injected code, such as use of HTML encodings[28]. Buffer overflow attacks were the most common type of security vulnerability attack until 2005, at which point they were supplanted by cross site scripting[29]. Rather than trying to fix all potential input and output, tainting focuses upon data which is more sensitive that can be used in traditional applications[30,31] as well as specifically for web applications[32–34].

A security policy is a formalized way of stating the intended behavior of a system. It provides constraints upon this behaviour with the goal of providing better security by restricting the actions that can be taken by a rogue or unauthorized entity. For more dynamic and context specific content, such as advertisements, verifiers and dynamic enforcement mechanisms like Caja[35] and ADsafe[36] have been proposed. For secured transaction for distributed system, a secured system web service which generate dynamic key and support parallel encryption and decryption[37]. A technique to detect Distributed Denial of Service attack by using window matrix and optimized HCF filtering technique has been proposed[38].

Also, to protect publishers and end users, advertising networks have been experimenting with various approaches to defending against malicious advertisements. However, none of these approaches addresses the need to analyze what the security policies of web applications actually are. Furthermore, many of these solutions have a variety of problems that prevent developers from fully embracing them.

3. Simple Web Security Policies

There are a great many issues in web security, but the biggest problem is that the attacks are often relatively easy while the solutions are unexpectedly difficult. Accordingly it is necessary to make attacks more difficult and to provide solutions that are easier to implement. Security policy is one way that we can define divisions to enhance the existing structures of the web[39]. Web security policy has challenges that differ from traditional security policy also simpler solutions are also available to enable simpler production. Solutions that are simple to implement are needed because of the defenders. Much web programmers concentrate on doing things and not on the security for the user[40]. While the web users assume that security will be the concern of programmers, unfortunately most of the web programming experts are unaware of these security breaches[41]. To make security implementation simpler, one of the classical way which isolates security from an application implementation can be employed.

Thus the main goal is to reduce the amount of time needed to create and maintain the policy. Also these policies must be simple, more usable and makes the attack harder. Thus a security policy that provides additional security while retaining their simple natures has been proposed. The next section explains the proposed framework termed Policy Approval Engine that protects web pages from unrestricted communications outside the page and protects the user data from malicious advertisement attacks by providing fine-grained control of the advertisement's privileges.

4. Policy Approval Engine – Proposed Framework

This work focuses on the framework to restrict inclusion within the browser and to restrict ad's privileges over user data exploiting the idea of isolating applications. This forbids the capricious communications from a web page to other sites. Additionally, access controls protect indiscriminate use of the content to other sites. The policy approval engine monitors all outgoing web requests within the browser and enforces a security policy and affords additional restrictions on inclusions, communications and access to user data by checking the White List before content can be embedded into a page.

The policy approval engine has a variety of options to handle captured cross-domain requests, i.e.,: *allow the*

request - requests originating from the same origin and only GET requests from cross-domain; *block the request*- POST request from cross-domain; *modify the request* - remove data from the request, such as cookies, cached credentials, and extra headers. Only safe request are sent to the Application Web Server (AWS). If the browser tries to embed the content into the web page, it checks the white list and Access Control List (ACL). The overview of the proposed framework is shown in the Figure 2. The processes depicted in the Figure 2 are as follows:

1. Web browser sends a request
2. Policy Engine may allow, block or modify the request based on the captured request. Assuming that the request is safe, it forwards the request to the Web Application Server along with the request to ACL.
3. The application server processes the request and sends the web page and ACL as response to the Policy Approval Engine.
4. The Defender Engine then gets the ACL and checks to ensure that the content provider is on the ACL.
5. Assuming that the content's server is on the ACL, the engine then checks the White List that resides in Content Server to confirm the approval of content provider.
6. If the Content provider approves, the content is then obtained from the content provider and embedded in the page.
7. Similar to ADsafe, if the web page contains any advertisements, it blocks a script from accessing any global variables and elements.

The Access Control List file is a text file which contains a list of domains which are considered as risk-free places from which code and content can be obtained. It is a static file that resides in the web application server and is created by the web application developer. The White file may contain a list of site that is allowed to load content from this content web server into their pages.

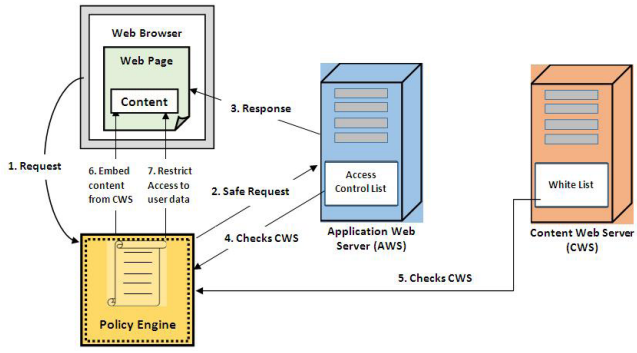

Figure 2. Overview of the proposed framework.

The sequence diagram of the proposed framework is given in the Figure 3. All requests from the web server are redirected to the policy engine that resides in the client side. The engine verifies the request and either it may accept, block or modify the request. If it accepts, it will forward the safe request to the Application Server (App_Server) along with the request to Access Control List. App_Server returns webpage and ACL. Policy Approval Engine checks to ensure that the Content Server is on the Access Control List. If App_Server does not allow inclusions from Content_Server, then the content is not loaded. If the Content_Server is included in the ACL, then the Policy Engine sends request to Content Server for White List. If Content_Server does not allow App_Server to use its content, then the engine again refuses to load the content. If Content_Server approves, the Policy Engine sends the request for the content to the Content_Server. The content provider provides the content which is then forwarded to the browser. Again if the content includes some online ads, it blocks a script from accessing any global variables and elements which is similar to ADsafe.

The Access Control List[42] and White Listare created and maintained by the system administrators as well

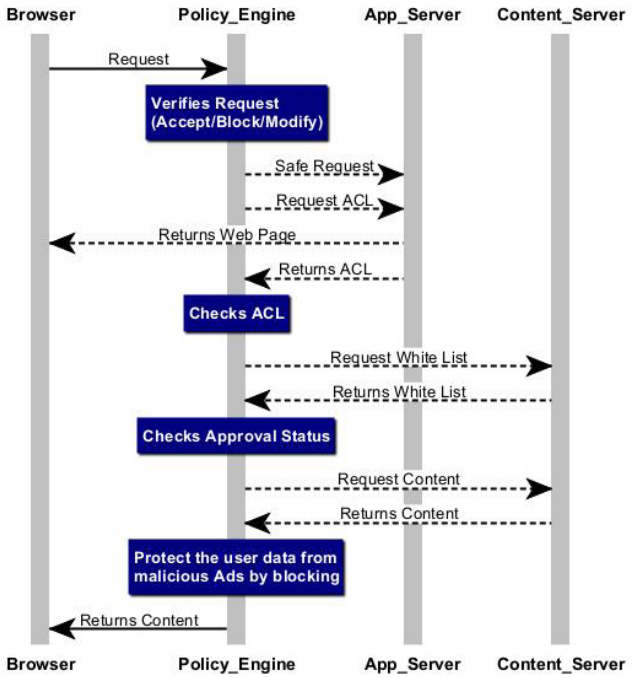

Figure 3. The sequence diagram of the proposed framework.

as web designers/developers, allowing them to specify approval for site inclusions in advance. Thus, the attackers cannot change the policy files or compromise underlying server software. In addition to these files, the policy approval engine can be extended to store the black list of the malicious websites and advertisements which endow with another layer of defense.

5. Evaluation

In order to do evaluate our framework, we developed experiments and used some performance metrics. They are

Accuracy: Accuracy measures the rate of generating correct results.

False Positive Rate: The rate at which non-vulnerable request are captured as attacks.

False Negative rate: The rate at which attacks are discarded as non-vulnerable request.

In order to verify that the proposed framework actively blocks information leakage, cross-site request forgery, cross-site scripting, and content stealing, we created examples of these attacks. In our tests, we have configured the web server and implemented all test beds with their default configurations. This insures that all implemented vulnerabilities are available for test. The sample test beds Sample1, Sample2 and Sample3 are taken for testing for which Table 1 and Figure 4 shows the accuracy rates in each sample set.

In order to compare the proposed work with the existing methods SOMA and RequestPolicy, the experiment has been conducted with known number of vulnerabilities in vulnerable web application. The maximum number of vulnerabilities was 50 which include code injection, Cross Site Scripting vulnerabilities, CSRF. In this test, we have calculated the accuracy for each method. Figure 5 describes the comparison between the proposed and the Existing frameworks. This also decreases the false positive rate and false negative rate when compared to the existing framework.

Table 1. Results for proposed framework

Samples	No. of known attacks	No. of Detected attacks	Accuracy (%)
Sample 1	30	28	93
Sample 2	30	27	90
Sample 3	30	29	97

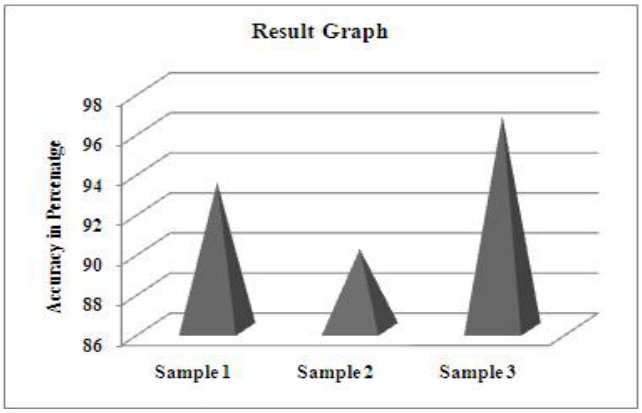

Figure 4. Result graph with accuracy rates.

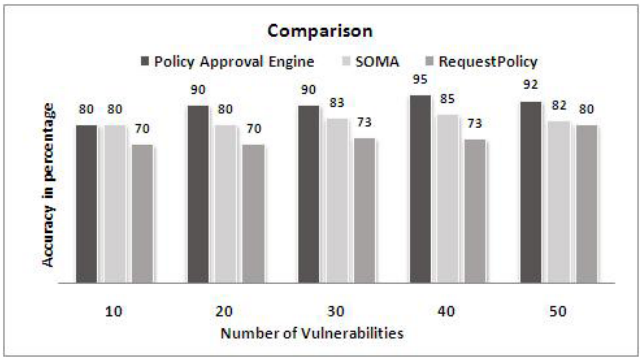

Figure 5. Comparison of accuracy.

6. Conclusion and Future Work

The policy enforcement framework to protect the web against cross-site request forgery, cross-site scripting, and content stealing is presented. The results and comparison of accuracy shows that this framework provides more security against various attacks. The framework observes all outgoing web requests within the browser and offers authorization, approval checks before the contents are embedded into a page and restricts the access to the advertisements by malicious codes. However, the data can be modified in transit and it does not stop attacks from trusted content providers/trusted origins. Also, the attack may compromise the pre-approved sites. Thus the future work focuses on providing security against web site defacement and other attacks.

7. References

1. Dausin M, Eisenbarth M, Gragido W, Hils A, Holden D, Jagdale P, Lake J, Painter M, Puzic A. Full year top cyber security risks report. Hewlett Packard; 2010. Available from:

http://dvlabs.tippingpoint.com/img/FullYear2010%20Risk%20Report.pdf
2. Grossman J. Whitehat website security statistics report. Whitehat Security; 2010. Availablr from: https://www.whitehatsec.com/resource/stats.html
3. Gray P. Why we secretly love lulzsec, Risky.biz; 2011 Jun 8. http://risky.biz/lulzsec
4. Web application attack report. 5th ed. Imperva's Application Defense Centre; 2014. Available from: http://www.imperva.com/docs/hii_web_application_attack_report_ed5.pdf
5. Uddin M, Memon J, Alsaqour R, Shah A, Rozan MZA. Mobile agent based multi-layer security framework for cloud data centers. Indian Journal of Science and Technology. 2015 Jun; 8(12):1–10.
6. Kaplan D. Malicious banner ads hit major websites. SC Magazine; 2007. Available from: http://www.scmagazine.com/malicious-banner-ads-hit-major-websites/article/35605/
7. Mills E. Malicious flash ads attack spread via clipboard. CNET; 2008. Available from: http://www.cnet.com/news/malicious-flash-ads-attack-spread-via-clipboard
8. Abhinivesh M, Garg M, Acharjya DP. Secured transaction for distributed service system. Indian Journal of Science and Technology. 2015 Jan 1; 8(S2):160–4.
9. Oda T. Simple security policy for the web; 2011. p. 1–236.
10. Finifter M, Weinberger J, Barth A. Preventing capability leaks in secure JavaScript subsets. NDSS. 2010; 99:1–14.
11. Kang A, Wiesmann A, et al. A guide to building secure web applications and web services. The Open Web Application Security Project (OWASP); 2005.
12. Ruderman J. Same origin policy for javascript. 2008; Available from: https://developer.mozilla.org/En/Same origin policy for JavaScript
13. Schuh J. Same-origin policy part 2: Server-provided policies? 2008. Available from: http://taossa.com/index.php/2007/02/17/same-origin-proposal
14. Howell J, Jackson C, Wang H J, Fan X. MashupOS: Operating system abstractions for client mashups. Workshop on Hot Topics in Operating Systems; USA. 2007. p. 1–7.
15. Wang HJ, Fan X, Howell J, Jackson C. Protection and communication abstractions for web browsers in MashupOS. 21st ACM Symposium on Operating Systems Principles (SOSP); 2007. p. 1–16.
16. De Keukelaere F, Bhola S, Steiner M, Chari S, Yoshihama S. Smash: Secure cross-domain mashups on unmodified browsers. Proceedings of the 17th International Conference on World Wide Web; China. 2007. p. 535–44.
17. Oda T, Wurster G, van Oorschot P, Somayaji A. SOMA: Mutual approval for included content in web pages. Proceedings of the 15th ACM Conference on Computer and Communications Security, (CCS'08); 2008. p. 89–98.
18. Microsoft Live Labs. Web sandbox; Available from: http://websandbox.livelabs.com/
19. A vocabulary and associated APIs for HTML and XHTML, W3C Working Draft, World Wide Web Consortium; 2009. Available from: https://www.w3.org/TR/2009/WD-html5-20090423/
20. Jovanovic N, Kirda E, Kruegel C. Preventing cross site request forgery attacks. IEEE International Conference on Security and Privacy in Communication Networks (Secure Comm); Baltimore, USA. 2006. p. 1–10.
21. Sheridan E. OWASP CSRFGuard Project; 2008.
22. Esposito D. Take advantage of ASP.NET built-in features to fend off web attacks. MSDN Library; 2005. Available from: https://msdn.microsoft.com/en-us/library/ms972969.aspx
23. Barth A, Jackson C, Mitchell C. Robust defenses for cross-site request forgery. Proceedings of the 15th ACM Conference on Computer and Communications Security; USA. 2008. p. 75–88.
24. Klein A. Forging HTTP request headers with Flash; 2006. Available from: http://www.securityfocus.com/archive/1/441014.
25. Linhart C, Klein A, Heled R, Orrin S. HTTP request smuggling. Computer Security Journal. 2006; 22(1):13.
26. Samuel J, Zhang B. RequestPolicy: Increasing web browsing privacy through control of cross-site requests. Privacy Enhancing Technologies, Springer Berlin Heidelberg; 2009. p. 128–42.
27. Wurzinger P, Platzer C, Ludl C, Kirda E, Kruegel C. SWAP: Mitigating XSS attacks using a reverse proxy. IEEE Proceeding of the 2009 ICSE Workshop on Software Engineering for Secure Systems; 2009. p. 33–9.
28. R Snake. XSS (cross site scripting) cheat sheet esp: For filter evasion; 2008.
29. Christey S, Martin RA. Vulnerability type distributions in CVE. MITRE Corporation; 2007. Available from: cve.mitre.org/docs/vuln-trends/
30. Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 12th Annual Network and Distributed System Security Symposium (NDSS). Internet Society; 2005. Available from: valgrind.org/docs/newsome2005.pdf
31. Xu W, Bhatkar S, Sekar R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. 15th USENIX Security Symposium; 2006. p. 15. Available from: http://www.cse.iitd.ac.in/~siy107537/sil765/readings/usenix_sec06.pdf
32. Nguyen-Tuong A, Guarnieri S, Greene D, Shirley J, Evans D. Automatically hardening web applications using precise tainting. Proceeding of 20th IFIP International Information Security Conference; 2005. p. 295–307.
33. Vogt P, Nentwich F, Jovanovic N, Kruegel C, Kirda E, Vigna G. Cross site scripting prevention with dynamic data tainting and static analysis. 14th Annual Network and Distributed System Security Symposium, Internet Society; 2007. p. 1–12.

34. Perlsec. Perl 5.10.0 documentation; 2006. Available from: dev.perl.org/perl5/news/2007/perl-5.10.0.html
35. Google Caja. A source-to-source translator for securing javascript-based web content Google Developers. Available from: https://code.google.com/p/google-caja/
36. Douglas Crockford. Adsafe: Making javascript safe for advertising; 2011. p. 297–306.
37. Abhinivesh M, Garg M, Acharjya DP. Secured transaction for distributed service system. Indian Journal of Science and Technology. 2015; 8(S2):160–4.
38. Devi GU, Priyan MK, Balan EV, Nath CG, Chandrasekhar M. Detection of DDoS attack using optimized hop count filtering technique. Indian Journal of Science and Technology. 2015; 8(1):1–6.
39. Chandra JV, Challa N, Pasupuleti SK. Intelligence based defense system to protect from advanced persistent threat by means of social engineering on social cloud platform. Indian Journal of Science and Technology. 2015; 8(28):1–9.
40. Meyer EA, Murtaugh T, Maria JS, Stevens K, Zeldman J. Findings from the list apart survey. A list apart; 2010. Available from: http://aneventapart.com/alasurvey2010
41. Wurster G, Van Oorschot PC. The developer is the enemy. Proceedings of the 2008 Workshop on New Security Paradigms Workshop (NSPW'08); 2008. p. 89–97.
42. Cox RS, Hansen JG, Gribble SD, Levy HM. A safety-oriented platform for web applications. Proceeding of the 2006 IEEE Symposium on Security and Privacy; Brekeley/Oakland, CA. 2006. p. 350–64.