Cleaning Up the Mess of Web 2.0 Security (At Least Partly)

Benjamin Stritter*, Felix Freiling*, Hartmut König†, René Rietz†,
Steffen Ullrich‡, Alexander von Gernler‡, Felix Erlacher§, and Falko Dressler§

* Department of Computer Science, Friedrich-Alexander-University, Erlangen, Germany
† Computer Networks and Communication Systems, Brandenburg University of Technology Cottbus, Germany
‡ genua mbH Kirchheim near Munich, Germany
§ Computer and Communication Systems, Institute of Computer Science, University of Innsbruck, Austria

Abstract*—Everybody loves the new Web 2.0 applications. They are easy to use, fast, and can be accessed from any computer or smartphones without installation. They allow us to easily communicate and share data with each other, make shopping simple, and give us access to vast amount of information. However, Web 2.0 is also frequently mentioned in the news in connection with novel exploits, data leaks, or identity theft. Active content, tight integration, and the overall complexity of the continuously evolving Web 2.0 technology creates new risks which we can hardly grasp. Turning back is no solution, since we would lose many beloved features. But how can we get both — pleasant user experience and security — in such a messy place such as the current Web 2.0 represents? *We study the complex security situation and attack surface of Web 2.0 applications and attempt to give a brief tour through this zoo, focusing on already existing applications. We particularly outline open research challenges in this field and give recommendations how to approach these issues.

I. Introduction

With the emergence and the success of the original World Wide Web (WWW), a huge variety of different commercial and non-commerical applications have been established reaching from simple link collections via search engines and web shops to audio and video telephony applications. To facilitate the use of the Web and to make the interface to the user more interactive, various scripting languages have been proposed, among them JavaScript has established itself as the prevalent one. Later plugins have been introduced. Java and Flash most popular ones. While the capabilities of JavaScript to interact with the computing environment have been restricted from the very beginning, the execution environment for plugins is not limited by design.

The rapid development of the browsers and their constantly extended feature set was accompanied by implementation differences and incompatibilities between the features. Allthough security became a major issue for browser developers, security demands have been met insufficiently. Secure Socket Layer (SSL), for instance, is used for securing the communication between the server and the client, but it does not provide dedicated security measures for the higher layers. The most frequently observed security issue in the Pre-Web 2.0 era was the buffer overflow, which opened up the system for further intrusions. Reasons were immature implementations of JavaScript, plugins, ActiveX, and the browsers themselves.

This work was supported by the German Bundesministerium für Bildung und Forschung (BMBF) in the scope of the Padiofire project.

To protect against these attacks, personal and perimeter firewalls have been deployed to scan downloaded contents for known vulnerabilities. As the majority of scripts and plugins are not essential for accessing the respective web sites, organizations that applied stronger security policies simply removed all JavaScript, plugins, and ActiveX content, thus providing a less powerful, but more secure version of the Web page. For many modern web applications, however, this is not acceptable any longer because current web sites tend to become unusable without active content.

In contrast to the usually static old Web, the dynamic Web 2.0 contains highly interactive JavaScript-driven Web applications. This facilitates the creation of more user-generated (sometimes sensitive) content. Commercial interests increasingly connect the content with social networks, advertisements, and user tracking. Moreover, web interfaces for controlling devices like routers, phones, or industrial systems are commonplace nowadays.

To sum up Web 2.0 has become a pretty complex system which even experts do not fully understand. This complexity has produced a variety of new attack methods, which especially affect the client side (i.e., the browser). Therefore, the Web 2.0 is a pretty dangerous place from the security point of view. Nowadays, however, we depend on the Web 2.0 because we want to access to information and our data anytime and anywhere both in personal and in business life. What can be done to make the modern Web secure and keep it usable at the same time?

In this article, we illustrate the most important security issues of Web 2.0 applications that directly affect the user, e.g., by attacking the web browser. We then explain basic mitigation strategies on server and client side and in intermediate devices, such as perimeter firewalls. By doing this, we wish to bring some order into the mess of modern web applications and to show how this mess can be cleaned up — at least partly. We conclude identifying open research challenges.

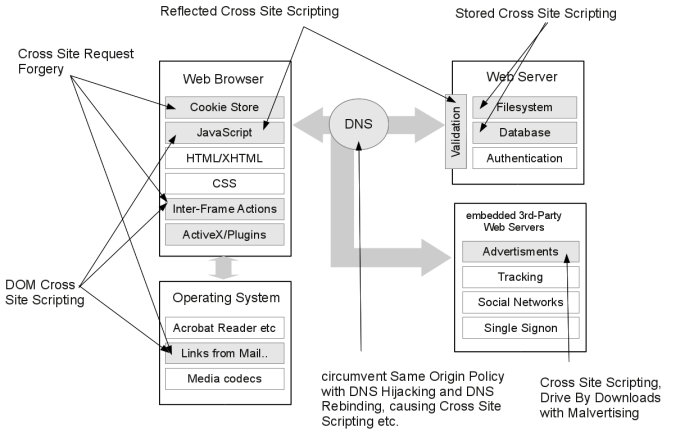

Figure 1. Complex interactions in the Web enable a variety of attacks against the client

II. The New Attacks of Web 2.0

With Web 2.0, new technologies have emerged. Asynchronous JavaScript and XML (AJAX) and Dynamic HTML (DHTML) have created the basis to share content from different sources, to mix them up, and to personalize them. Now web applications are advocated similarly to desktop applications without the overhead to install and maintain them. This motivated companies to use web applications for sharing business-critical data in the Internet or in intranets. As a side effect of the personalization capabilities of the Web, advertisement and tracking providers adapted their techniques to aggregate and combine the huge amount of freely available information about the users to create precise user profiles.

As the Web became larger and larger, and more attractive for users, new attacks have emerged that are specific to the new Web. They combine and integrate active content, content from different sources, and old risks lurking in the design decisions of browsers and web applications. On the other hand, the impact of buffer overflows decreased due to mitigation methods, such as sandboxing and Address Space Layout Randomization (ASLR). In addition, new features of HTML, such as support for Scalable Vector Graphics (SVG), canvas, and the integration of audio and video, made lots of plugins obsolete. Some plugins, like Adobe Reader, Java, and Flash, which are present in most desktop browsers, are still an attractive attack vector though.

Figure 1 attempts to give an overview over the new attacks. On the left-hand side we see the client side (including the browser and the operating system), whereas the (one or more) Web servers are depicted on the right-hand side . In between a simplified view of the network infrastructure is depicted which is fragile itself. The interaction between these components can create problems, as we explain in the following.

A. Merging of Security Domains Inside a Browser

A single browser can access web sites from different security domains, e.g., from a corporate network, an intranet, private mail, social, news, or advertisement networks. These sites can interact with each other within the browser in different ways. For an attack, the browser can even be misused as a trampoline to access internal systems from outside, evading protection techniques, such as firewalls.

a) Embedding of script.: Usually social networks and advertisement or tracking sites are directly embedded as scripts, thus being an integral part of the page and having full control over it. This often integrates multiple security domains with totally different levels of trust (see Fig. 2). Since every domain has the same access privileges, this situation easily allows code execution from unwanted sources [1]. This is also called *Cross Site Scripting (XSS)*.

```
<script src=http://malvertising.com...
```

Figure 2. Stored Cross-Site-Scripting through inclusion of untrusted 3rd-party

b) Embedding of iframes.: An iframe is a frame which embeds another page into the current page. Although a script inside an iframe is more restricted than a script directly embedded in the hosting page, it can communicate with other frames through the *postMessage* API or similar techniques. Since iframes have no visible borders with the hosting page, they can be used for *User Interface Redressing* attacks, e.g., to present a faked login dialogue instead of an advertisement (see Fig. 3). Furthermore, content can be framed within a different context by clipping or visually overlaying it, leading to *Clickjacking* [2] and similar attacks.

```
<iframe>
<form action=http://mallory>
Please login to Facebook
<input name=user ...>
<input type=password...>
```

Please login to Facebook
Username
Password

Figure 3. User Interface Redressing using an iframe

c) Shared (cookie) storage.: HTTP is by design a stateless protocol, but practically all web applications need a stateful session which consists of several HTTP requests. The states are usually encoded using cookies, i.e., session identifiers, which are issued by the server and sent back to the clients with each subsequent request. The cookies are managed in a storage inside of the browser. The major security flaw of this solution is the fact that cookies are sent back with each request to the server, even if the request originates from a different site (e.g., triggered by submitting a form or following a link, see Fig. 4). This enables so-called *Cross Site Request Forgery (CSRF)* attacks [3] from a malicious site to inject actions into existing sessions. Modern browsers generally allow web applications to persistently store even more data. This can also be used for *XSS* attacks which contain payload that is not related to the current request.

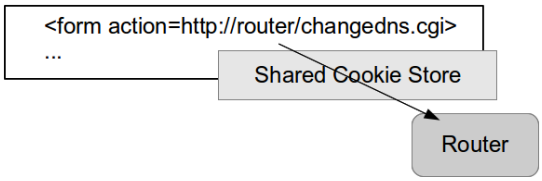

Figure 4. Cross Site Request using a form

B. Serializing Complex Data Formats As Text

HTML merges information about the structure of a page together with its content into a "flat" text representation. The web page does not only contain HTML code, but additionally also JavaScript, Cascading Style Sheet (CSS), and — with data-URLs[1] — even images and other content. To extract the structure and the content from the text representation several layers and kinds of encoding and escaping have to be unwrapped in the correct order. This makes it extremely difficult to detect and distinguish the different parts in order to ensure an unambiguous interpretation (cf. [4]). Some examples of the resulting problems are illustrated in Fig. 5.

The first example uses Unicode to encode parts of a script. It is very simple but very effective to evade signature-based detections of specific JavaScript commands. The second example uses a different encoding for CSS escaping, i.e., to call JavaScript from insight of CSS expressions. Similar escaping techniques can be used for JavaScript strings, HTML characters, and URLs. The third example is more complex. It shows how different encoding and escaping techniques can be combined.

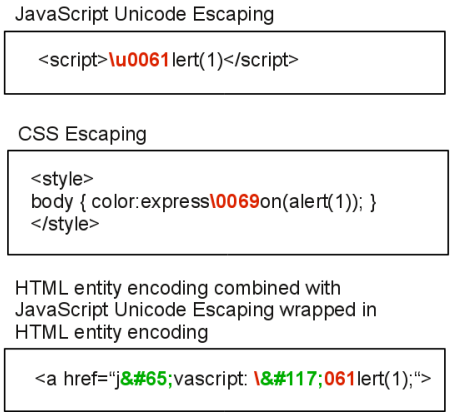

Figure 5. Flattening of structured data as text

Errors in adhering to all these encodings and escapings when creating or interpreting the text inside the browser are sources for vulnerabilities. They enable various kinds of injections, in particular XSS. These errors often occur when the processed user input is assumed to be in a specific format, without verifying it. Similar problems arise on the server side when generating database queries from the user input. Failures when checking input or not applying respective escaping rules may lead to a "break-out" of the query and the execution of the injected code, e.g., *SQL Injections*. The variety of escaping rules across different databases additionally complicates this procedure.

[1]http://www.ietf.org/rfc/rfc2397.txt

C. Incomplete or Conflicting Standards

Although the standards that are directly related to web applications, notably HTTP and HTML, are complex, they mostly fail to address the handling of ambiguous or erroneous content, thus making the interpretation implementation-dependent. A classic example is the specification of the character set of the HTML content which can be specified in various places, e.g., inside the HTTP header, as various tags inside the HTML content, or using a Byte Order Mark (BOM). So it is possible to circumvent security filters in servers or the intermediate devices using conflicting declarations. This supports various attacks, in particular XSS, as depicted in Figure 6.

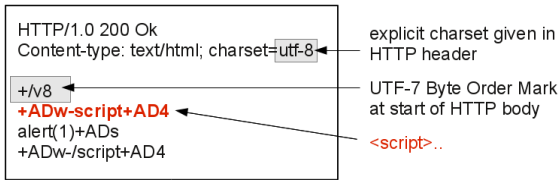

Figure 6. Ambiguous character set interpretation

Another example is the interpretation of the HTTP *Content-Type* header. According to the standard, the data type of the HTTP body is determined in the header, but this is often ignored. This is especially true when including JavaScript into HTML using the *Script-Tag*. Even with an explicit *Content-Type* of "image/gif", a content with "GIF89a=1; alert(1)" will be interpreted as JavaScript. This also helps to circumvent content filters in the server. Therefore, securing applications with user generated content is still a challenge.

D. Unjustified Trust in DNS and PKIs

Same Origin Policy (SOP) is one of the cornerstones of the browser security. It allows that contents of the same origin may freely interact among each other, but it restricts interactions with contents of different sources. It applies to the interaction in JavaScript and defines the access to cookies and the interaction of frames. The calculated origin for SOP depends on the host and domain names, and thus on the Domain Name System (DNS). But it also depends on the policies regarding effective top level domains, which can be different for each registrar. For example, although `uk` is a global top level domain, effectively all subdomains (`*.uk`) and others are used as top level with some exceptions, such as `policy.uk` etc.

Although Same Origin Policy depends on the DNS replies being authoritative and correct, this is not guaranteed. As long as DNSSec (DNS with signed replies) is not supported everywhere, queries can easily be hijacked. Even with DNSSec, a DNS server can still claim any IP address (even internal ones like 127.0.0.1) to be the address for a host name it manages. This makes *DNS Rebinding* attacks possible.

Mistrust is also justified against Public Key Infrastructures (PKIs), the basis of HTTPS. While earlier attacks often were ignored, the high profile compromise of two certification authorities (CAs) in 2011[2] and the abuse of certificates issued by mistake for intermediate CAs in 2012[3] enabled attackers to issue unauthorized certificates for major sites, such as `google.com`. This publically demonstrated the fragility of Public Key Infrastructure (PKI). While these attacks are not really specific to the Web 2.0, the strong interconnection between various websites in modern web applications make attacks on DNS and PKIs much more attractive today than in the past.

III. Practical Mitigation Methods Today

Although it seems futile to establish high security in an environment that is so complex and messy, there are some general approaches to make the Web 2.0 a better place. So what can we do to (at least partly) clean up the mess? Possible solutions can be classified according to the place where they are deployed in the network architecture: at the client, at the server, or in intermediate systems.

A. Mitigation inside the Web Browsers

Most modern browsers offer mitigation heuristics against *Reflected XSS* attacks at the client side. Reflected XSS means that a request from the client contains the actual XSS payload which is reflected by the server and then executed by the client. Usually the protection is directly implemented in the browser, but sometimes it is only available as add-on. Most browsers also apply URL blacklists, such as *Google Safebrowsing*[4], to block malicious sites. Additionally, virus scanners are used to check for local or downloaded malware. More comprehensive security suites trigger a warning when their heuristics detect "abnormal" system behavior.

Other solutions limit the impact of exploits by putting the browser into a sandbox or a newly generated virtual machine. This protects against malware spreading, but it does not help against *XSS* or *CSRF* attacks because these attacks manipulate data in the Internet or an intranet, respectively, and would not work that way. Current web browsers also do not provide protection against Stored XSS attacks when attacker data are stored on the server and later injected to arbitrary users. Browsers do not protect neither against *CSRF* nor *Clickjacking*. A noteworthy extension of Firefox is *NoScript*[5] which restricts the execution of JavaScript to only few sites and thus effectively mitigates *XSS* attacks. It also offers some *CSRF* and *Clickjacking* protection. Unfortunately, *NoScript* needs an extensive individual adaption and is only manageable by experienced users.

Even though it hampers the ability of websites to finance themselves, the use of browser extensions, such as *AdBlock-Plus*[6] to inhibit integrated advertisments, tracking, and social networks, is an effective way to remove the usuallly unjustified trust in these sites without impairing their usability.

While most research focuses on securing the web application on the server side, there are some client-related approaches. The *CsFire* [5] browser extension blocks most cross-site requests to inhibit *CSRF*, while detecting and allowing cross-site requests used with payment or single-sign-on solutions. The browser extension *JaSPIn* [6] creates a profile of the application usage of JavaScript and enforces it later. Unfortunatly, it needs individually be tuned to websites which heavily use JavaScript functions. It has to be recreated whenever the web application changes. *Noxes* [7] instead works as a client-side proxy and tries to protect against data leakage through detecting and white-listing valid cross-site requests by analyzing the requested web page. It has a high false-positive rate and requires often user interactions.

To sum up builtin security solutions offer protection against *Reflected XSS* and widely spread malware, respectively. A better protection can be achieved by specific browser extensions and sometimes by some, still rare commercial security solutions, such as Web application firewalls. Although experts may be capable to install procedural safeguards for their individual use, there is currently no solution in sight, which provides the average user with enough protection against *Stored XSS*, *CSRF*, *User Interface Redressing*, or targeted malware attacks.

B. Server-side Approaches

The best strategy to protect web servers, of course, would be to only write secure web applications from scratch. With appropriate efforts, most of the security problems could be mitigated by the web application developers, but only if they are aware of the security problems and have enough resources to implement countermeasures.

Apart from thoroughly checking all input and output for XSS and SQL injections, there are other effective strategies at the server side. For example, a secret can be set by the server for each resource using CSRF tokens [8]. Only actions that contain the token will be executed, so that *CSRF* attempts will fail. Other examples are hardening Session-IDs to inhibit *Session-Hijacking* by guessing the Session-ID, adding frame busting code or headers to fight *Clickjacking*, and using different domain names to partition the application into security realms with interactions restricted by the *Same Origin Policy*.

Some modern browsers also support the *Content Security Policy (CSP)*[7] which restrains the execution environment in browsers based on explicit policies given in the response, such as to only include scripts, styles, or media from specific sites, to forbid inline scripts, or to prohibit the execution of dynamically created code. Applications often contain an inline script that is indistinguishable from a script injection. Even if it is only used

[2] https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html

[3] http://googleonlinesecurity.blogspot.de/2013/01/enhancing-digital-certificate-security.html

[4] http://googleonlinesecurity.blogspot.de/2012/06/safe-browsing-protecting-web-users-for.html

[5] http://noscript.net/

[6] http://adblockplus.org/

[7] https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

to describe the respective implementation of the application, it already provides a certain protection. If the application is changed to a stricter software architecture and the policy is adapted, *CSP* provides an extensive protection against *XSS*. This requires, however, that application developers are aware of security problems.

A common protection strategy for the server side is to set up a *Web Application Firewall (WAF)* to check input, to add and verify CSRF tokens, to harden session cookies, and to detect common attack patterns. If tightly integrated with the web application, a web application firewall can provide a reasonable but still insufficient protection. Some web application firewalls even have a limited ability to adapt themselves through learning the input data types of the application.

Unfortunatly, resources and knowledge are highly limited in reality, so the focus of the research is to provide better security without too much effort. Approaches, such as *S2XS2* [9] or *NonceSpaces* [10], provide a framework, in which the developer has to explicitly distinguish between trusted application data and untrusted external data. He/she has further to specify how untrusted data can propagate into a dynamically generated page. Based on these information, the server then can restrict propagation of unexpected external data. *BEEP* [11], *Blueprint* [12], and the *Content Security Policy* instead let the developer specify a security policy which has to be enforced at the client. These policies allow one to distinguish between application-specific and external data, and may even restrict executions inside the JavaScript interpreter. All of these solutions, however, require changes in the application to make it conform to the policy.

The server-side proxy *XSSDS* [13] instead tries to learn the application-specific JavaScript of the unchanged application by itself. Thereafter, it allows only known JavaScript. Additionally, it facilitates to define filters against *Reflected XSS* attacks similar to the ones used inside current web browsers. A similar approach of comparing the received page with the expected page inside a server-side proxy shows *XSS-GUARD* [14]. Instead of learning the application specific script though, a shadow page based on benign input will be generated using an adapted application and the real page will be compared with it.

C. Solutions for Intermediate Devices

A common practice to securely connecting a company to the Internet is the use of firewalls. While simple packet filters up to Layer 4 do not offer any protection against Web-specific attacks, application layer proxies and Intrusion Detection System (IDS)/Intrusion Prevention System (IPS) with deep packet inspection capabilities can explore the application protocols and block or manipulate connections. These systems are often called *Secure Web Gateway (SWG)*, or increasingly marketed as *Next Generation Firewall (NGFW)* or *Unified Threat Management (UTM)*. They all inspect the traffic at least up to the HTTP layer and provide thus the possibility to classify and filter the traffic based on URLs. They are also capable of scanning traffic for viruses or other malware. Advanced solutions can additionally bridge and inspect Transport Layer Security (TLS) traffic. Few of them are capable of inspecting the traffic for *Reflected XSS* attacks, of normalizing HTML, and of removing scripts, plugins, or ActiveX embeddings.

Another solution is to crawl the Web in advance to detect malware. Analysis methods applied are virus scanning, the detection of common exploits, obfuscation patterns, or abnormal behavior within a sandboxed execution. Results of this analysis are then provided in the form of URL blacklists, e.g., *Google Safebrowsing*[8]. Since these methods do not integrate real-life interactions with web sites, they cannot detect attacks on the application logic, such as *CSRF*, *Stored XSS*, or *Clickjacking*.

Although apparently no current intermediate devices offer reasonable protection against the new Web 2.0 attacks, there seems to be no major research in this area.

Table I
ATTACK COVERAGE OF DIFFERENT PLACEMENT STRATEGIES

attack	clients	intermediate systems	server
major attacks			
attacks against servers			**x**
cross-site scripting	**p**		**p**
credential/session prediction			**x**
session fixation			**x**
cross-site request forgery	**p**		**x**
buffer overflow	**x**		**x**
malware	**p**	**p**	**p**
URL redirector abuse			**x**
minor attacks			
integer overflows			
content spoofing			
remote file inclusion			x
HTTP response splitting		x	x
HTTP request splitting		x	x
null byte injection			
routing detour			
XML external entities			x

x - good attack coverage
p - partial attack coverage

D. Attack Coverage

Existing protection solutions against the malicious use of web applications differ in the attack coverage and in placement strategies. Table I lists the currently known attack methods based on the classification system of the web application security consortium[9]. Attacks against the integrity of servers or the intermediate devices network infrastructure do not directly affect the client side, but may result in various attacks against the client. Therefore, we subsume attacks, such as brute force, denial of service, HTTP request or response smuggling, path traversal, predictable resource location, SOAP array abuse, mail or OS command injections, LDAP, SSI, SQL, XPATH, XML and XQUERY injections, XML attribute blowup, and XML entity expansion, as attacks against servers.

[8]http://googleonlinesecurity.blogspot.de/2012/06/safe-browsing-protecting-web-users-for.html

[9]http://projects.webappsec.org/w/tags/show?tag=Threat%20Classification

The upper part of Table I, which is loosely based on the OWASP Top Ten Project [15], lists the major attacks. With appropriate effort, applications can be protected fairly well against the major attacks on the server side. However in real-life, most servers are not protected in this way. Therefore, attack mitigation strategies are required at the client side and in intermediate devices.

IV. Open Research Challenges

To secure web traffic in the Web 2.0 era novel protection methods are required. Currently deployed solutions do not provide adequate security.

Of server-side approaches there are only several ones which usually try to cover application-specific vulnerabilities, such as insecure cookie flags or missing *CSRF* protection. Although it would have been better to fix the Web application itself in most cases, these approaches nevertheless are reasonable to secure specific flaws in existing applications.

More effective approaches are based on the distinction between trusted and untrusted content, but they require modifications in web applications and browsers. Some of the most promising ideas led to the implementation of the *Content Security Policy (CSP)* in recent browsers which provides a significantly better protection against *XSS* attacks, but it lacks adoption in Web applications. Even if new applications may use *CSP*, most existing Web applications will never be adapted.

There is only little research to secure the client side in insecure web applications. Although valuable context, such as the application state or other runtime information, is available at the client side, currently no approach takes the full advantage of these benefits. Further research is required here. Solutions for intermediate devices are very rare and currently limited to specific attack vectors. Their main focus is malware detection, often based on an offline analysis in combination with URL blacklists. Single attack vectors, such as reflective XSS, are sometimes covered as well, but none of the existing approaches provides a comprehensive protection against existing threats.

The presented approaches and solutions offer only partial protection using heuristics, require adaptations on client and server side, or an extensive manual configuration is needed that only few users are capable to perform. In order to provide a reasonable protection against web attacks research is required in the following areas:

- *Protection of the web browsers* against typical Web 2.0 attacks. There is an urgent need for solutions which provide solid protection without requiring extensive configurations through the user. Since the usage of web applications is highly individual and use patterns change fast, it is not sufficient to provide this protection only for the most important web applications.
- *Transparent protection against attacks on both sides in intermediate devices* like firewalls. The protection must adapt itself to individual and fast moving usage patterns on the client side and continuously changing Web applications on the server side without losing reliability. Manual configurations are also not feasible here.
- *Creation of secure and easy to use application frameworks for the server-side* to make future Web applications more secure with less effort.
- *Rethinking the interaction between browser, server, and all the other components of the Web* to provide security by default.

While the chance of a restart are promising it is certainly not acceptable in practice. Therefore, security solutions have to start with the mess we currently have.

V. Conclusion

In conclusion, it can be said that Web 2.0 security is to be considered one of the most challenging issues in the field. Cleaning up the mess (at least partly) is not only a matter of providing solutions for singular problems and weaknesses. We identified several open research challenges, most prominently focusing on real-time identification of active content both at the browser and at the serve side. One possible way to approach the mentioned challenges is to investigate more intelligent firewall systems doing application layer inspection of traffic beyond application layer, i.e., inspecting contents transported in Web 2.0 application layer protocols.

References

[1] CERT, "Malicious HTML Tags Embedded in Client Web Requests," https://www.cert.org/historical/advisories/CA-2000-02.cfm, February 2000.

[2] R. Hansen and J. Grossman, "Clickjacking," http://www.sectheory.com/clickjacking.htm, 2008.

[3] P. Watkins, "Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images)," http://www.tux.org/~peterw/csrf.txt, June 2001.

[4] R. Hansen, "XSS Filter Evasion Cheat Sheet," https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, April 2015.

[5] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens, "Automatic and Precise Client-Side Protection against CSRF Attacks," in *ESORICS*, ser. Lecture Notes in Computer Science, V. Atluri and C. Díaz, Eds., vol. 6879. Springer, 2011, pp. 100–116.

[6] P. Raman and C. U. (Canada)., *JaSPIn: JavaScript Based Anomaly Detection of Cross-site Scripting Attacks*, ser. Canadian theses. Carleton University (Canada), 2008.

[7] E. Kirda, C. Krügel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *SAC*, H. Haddad, Ed. ACM, 2006, pp. 330–337.

[8] OWASP, "Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet," https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[9] H. Shahriar and M. Zulkernine, "S2XS2: A Server Side Approach to Automatically Detect XSS Attacks," in *DASC*. IEEE, 2011, pp. 7–14.

[10] M. V. Gundy and H. Chen, "Noncespaces: Using randomization to defeat cross-site scripting attacks," *Computers & Security*, vol. 31, no. 4, pp. 612–628, 2012.

[11] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *WWW*, C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, Eds. ACM, 2007, pp. 601–610.

[12] M. T. Louw and V. N. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009, pp. 331–346.

[13] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," in *ACSAC*. IEEE Computer Society, 2008, pp. 335–344.

[14] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," in *DIMVA*, ser. Lecture Notes in Computer Science, D. Zamboni, Ed., vol. 5137. Springer, 2008, pp. 23–43.

[15] OWASP, "OWASP Top Ten Project," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2012.