Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives

Hossein Saiedian, *University of Kansas*
Dan S. Broyles, *Sprint Nextel*

The same-origin policy, a fundamental security mechanism within Web browsers, overly restricts Web application development while creating an ever-growing list of security holes, reinforcing the argument that the SOP is not an appropriate security model.

One of the first security measures that Internet browsers incorporated was the same-origin policy. As early as Netscape Navigator 2.0, SOP prohibited data sharing between origins—any unique host (such as a website), port, or application protocol. So, for example, SOP prevents one site's documents from accessing the document contents or properties from other sites. Thus, the SOP makes it possible for users to visit untrusted websites without allowing them to manipulate data and sessions on trusted sites.

If you browse http://example.com/index.htm, the SOP in the browser would accept or reject script and data accesses from the following sources:

- http://example.com/about.htm (port 80): accept
- https://example.com/doc.html (port 443): reject
- http://google.com/search.php (port 80): reject
- http://dev.example.com/more.htm (port 80): reject

By default, the SOP does not allow subdomains such as dev.example.com to interact with the primary domain. However, by using the <document.domain="example.com"> script in various subdomains, the SOP permits data sharing between the pages of example.com and dev.example.com. Doing so can cause problems, however; the script would let pages from a subdomain such as userpages.example.com access and alter pages from another subdomain, such as payments.example.com.

The SOP incorrectly assumes that all directory paths within the URL belong to the same source. For example, URLs www.example.com/~john and www.example.com/~mary have the same origin, even though they belong to different users and therefore should not trust each other. Another problem with the SOP is that it prevents developers from delivering dynamic multisource data. As the Internet and Web technology have progressed, the SOP has not evolved to keep up with the security needs of a more complex system, allowing malicious users to circumvent and exploit it.

In principle, the SOP restriction is a good security measure because it aims to protect data integrity and confidentiality. However, it has not kept up with changes in Web technology. The first Web browsers were not designed with security in mind, so developers added the SOP mechanism later to meet some basic security needs.

With the advent of JavaScript, Ajax, Web services, and mashups, clever programmers and hackers have found creative ways to subvert the SOP. Any SOP exploitation can expose a Web application to attack from malicious code, even if that exploitation comes from a well-intentioned developer. In addition, those who correct security flaws must account for the Web's unique environment, such as statelessness and code mobility.[1] To further complicate matters, SOP rules and implementations differ between resources, DOM objects, XMLHttpRequests, cookies, Flash, Java, JavaScript, ActiveX, Silverlight, plug-ins, and browsers.

Inexperienced Web programmers who do not know that certain objects and actions—such as form submissions and tags like <script> and <img>—are not subject to SOP might copy JavaScript from other websites without understanding the security implications. Much like the early Web browser itself, such developers focus on functionality first and security last.

SOP weaknesses have led to attacks such as cross-site request forgery (CSRF), cross-site scripting (XSS), and Web cache poisoning. Attempts to fix these exploits have had only limited success; they tend to patch individual exploits without actually correcting the underlying security problems. In other words, the SOP is not the correct security mechanism and requires redesign to meet the access-control requirements of Web-based assets. The Web security community is still debating how best to implement such a major undertaking. However, it seems clear that the current SOP lacks two basic access-control principles: the separation of privilege and least privilege.

Inexperienced Web programmers who do not know that certain objects and actions are not subject to the SOP might copy JavaScript from other websites without understanding the security implications.

Professional Web developers know about these deficiencies and the many effective mitigation techniques available, but many websites are built by nonprofessional developers with limited experience.

NEED FOR DATA IN WEB APPLICATIONS

Internet activity is moving away from traditional searching and navigating toward an interactive and application-like activity in which browsers deliver dynamic, customized content. Users can enter their own content on Web forums, and social networking sites and mashups incorporate content from many users and third-party sites.

Jim Mischel has noted that the Web browser is the platform of the future, but in its current state, the SOP makes it difficult to share remote data and exposes too many vulnerabilities.[2] JavaScript and Ajax make modern feature-rich websites possible, bringing applications directly to users and improving efficiency and performance. However, the SOP makes it difficult for Web applications from one source to obtain and display data from another. Developers use two common and powerful techniques to circumvent the SOP and obtain data from other domains; the first uses an Ajax proxy, and the second uses JavaScript object notation with padding (JSONP) script tag injection.

Ajax proxy

XMLHttpRequest objects, the cornerstone of Ajax technology, make dynamic Web applications possible. The SOP restricts XMLHttpRequest calls much like it does any other script running in a browser, allowing such requests only between applications and servers from the same source.

Imagine a Web application that displays current stock price information hosted by a remote webserver. If the user enters a URL such as www.getyourstocks.com/current.php?ticker=msft&format=json, the remote webserver will return the current stock price in the following format:

```
{
      “ticker”:”msft”,
      “current”:”24.5”,
      “lastclose”:”24.0”,
      “pctchange”:”2.1”,
      “30dayavg”:”23.45”
}
```

JavaScript makes it easy to format and display such return data on the webpage, but the SOP forbids the developer from making a request to http://getyourstocks.com from within his webpage. However, he can set up an application proxy server on his webserver, ask it to obtain the data from the other server, and deliver it through the server to the user. Since the page makes an XMLHttpRequest to the webserver proxy, which has the same origin as the Web application, the SOP allows it. A proxy server is functional, but slow and inefficient. It would be far better if the Web application could query the remote server directly.[2]

JSONP script tag injection

Another approach to getting this outside data into the Web application is to place the call to getyourstocks.com inside a JavaScript function. In this example, if getyourstocks.com supports JSONP, then the programmer could add a JavaScript function on the page called, for example, showCurrent, that displays the data once it returns from getyourstocks.com:

```
function showCurrent(data){
   // display the contents of the data
}
```

Then all the application needs is a <script> tag that makes the request to the remote server:

```
<script type="text/javascript"
src="http://www.getyourstocks.com/current.
php?ticker=msft&format=json&callback=show
current">
</script>
```

The remote server will return the requested data so that it calls the callback function, showCurrent:

```
showCurrent({
  "ticker":"msft",
  "current":"24.5",
  "lastclose":"24.0",
  "pctchange":"2.1",
  "30dayavg":"23.45"
});
```

The webpage will execute the returned JavaScript as if it were native to the page. However, if a hacker were to alter the getyourstocks.com site so that it returns malicious JavaScript instead of stock quote data, the browser running this Web app will execute the malicious code. By circumventing SOP, the developer has introduced a security hole.

So how do developers obtain data for a Web application and still maintain security? They need a better security policy. Basic security logic suggests that third-party entities should not have the same access rights as trusted entities, so a policy that lets application developers determine access rights for each object might solve many SOP issues.

HACKER EXPLOITS

Hackers use various exploits to take advantage of SOP deficiencies. There are many variations of these attacks, and hackers create new exploits all the time. Here, the purpose is not to enumerate every SOP vulnerability and attack, but to outline the fundamental flaws in the SOP so that readers can better analyze the proposed solutions.

Cross-site request forgery

Security experts identified the earliest CSRF as a confused deputy attack. In this type of attack, hackers lure a victim to a malicious website to submit a form that points to a trusted target site in which the victim might have an active session. The trusted webserver receives and processes the form submission request, which looks identical to a legitimate request from the trusted website.

CSRF includes any malicious webpage with scripts that make unauthorized requests to trusted sites, hoping to take advantage of users who have an active session with the trusted site. For example, a user visits her bank's website regularly and has an active session open with the bank when she browses to a malicious website containing code that calls the bank's webserver. The call requests a transfer to another account, robbing the unwitting user. The browser caches the user's active session information within itself, so the malicious request to the bank's server looks exactly like a legitimate user request.

Today, banks employ various measures to thwart such attacks, but other sites do not, especially those of small- and medium-size companies whose Web developers do not see the need for security measures covering SOP exploits. Using the Referrer header could be effective against CSRF, but Web applications frequently block this header because of privacy concerns, so an application that enforces it will exclude many users. Applications also strip Referrer headers from all HTTPS requests. Still worse, hackers can modify the Referrer header, making it unreliable.[3]

The SOP is not the correct security mechanism and requires redesign to meet the access-control requirements of Web-based assets.

Adam Barth and his colleagues recommended augmenting browser policy to use an Origin header as opposed to the Referrer header to provide CSRF and click-jacking protection.[4] The Mozilla security model currently proposes this fix (https://wiki.mozilla.org/Security/Origin).

Cross-site scripting

There are two basic ways attackers implement XSS. The first method, considered nonpersistent, introduces malicious scripts in GET or POST requests that show up in pages returned by the server. For example, an attacker sends an e-mail containing a specially crafted hyperlink to a trusted website; the URL string includes malicious instructions reflected in the page returned by the webserver. The attack takes place when the user clicks on the hyperlink.

The second attack, considered persistent, occurs when the attacker injects malicious scripts into GET or POST actions that the server stores and then dynamically presents to the victim. For example, a malicious user logs into a forum and adds comments containing malicious JavaScript to a discussion thread as follows.[5]

```
<html><head><title>Paul's Blog</title>
</head><body>
  <h1>Scavenger Hunt!<h1>
  <hr>
  <h2>Paul: I will award the student
  bringing me the following items:</h2>
```

```
   <ul>
   <li>Yellow #2 pencil</li>
   <li>Secretary's middle name</li>
   <li>Number of ceiling tiles in our lab
 </li>
   </ul>
   <hr>
   <h4>Comments</h4>
   Karthick: What will we get?
     <script>
     //malicious script that modifies the
     above list
     </script>
   <hr>
 </body></html>
```

The forum database stores this code as part of the thread. The code executes anytime a user browses that page.

The most vulnerable websites are those of small- to medium-size companies or institutions, which often do not understand the threat.

With the famous Samy MySpace worm, a user (Samy) exploited an XSS vulnerability in the MySpace profile form submission page and attached code to his profile. The injected code included a CSRF attack wherein anyone viewing his profile page would automatically add the same code to their profile page, make Samy their hero, and request Samy as a friend.[6] Another XSS attack technique is to embed an invalid image object and use the "onerror=" event to redirect the user to a webpage the attacker chooses. Any code injected into a webpage receives full rights as if it were part of the original page. The code can read and write other page elements, cookies, and browser history.

Proper form validation and input sanitization can prevent XSS attacks. The Samy attack infected a million user accounts in the first day, and the damage to MySpace's reputation is incalculable. The estimated cost to repair the damage from two smaller XSS attacks, Code Red and Slammer, was $2.6 billion and $1 billion, respectively.

In DOMXSS, a more recent version of an XSS attack, webpage code running on the client browser uses DOM objects related to the URL string and other environment variables that users can influence. If applications do not properly validate these objects, an attacker can use them to introduce malicious code into the page. This attack targets Flash and other embedded programmable objects that have access to user-manipulated DOM objects and environment variables. In DOMXSS attacks, the client-side code embeds the malicious code into the webpage; in traditional XSS attacks, the server embeds the malicious code.

Dynamic pharming

Dynamic pharming employs Domain Name System (DNS) hijacking to deliver a Web document with malicious JavaScript code, and then in a separate <iframe>, the attacker uses DNS rebinding techniques to load the authentic website the user expected.[7] In this way, the user is actually interacting with a trusted website while, behind the page, an attacker can monitor transactions and steal session cookies and passwords. Since the malicious page and the <iframe> appear to have the same origin, SOP allows the malicious page to interact with the legitimate website.

There are several ways to hijack the DNS name. For example, an attacker can set up a wireless hotspot in an airport. When unsuspecting users connect to this "free" hotspot, the attacker's router can intercept their DNS requests and redirect these requests to a site of the attacker's choosing. Victims might never suspect the attack since the browser indicates the domain that they trust and expect.

PROTECTION CONSIDERATIONS FOR DEVELOPERS

Even if a website has little traffic and contains no confidential information, the fact that someone is interested enough to visit the site makes it a potential target for attack. As Figure 1 shows, the most vulnerable websites are those of small- to medium-size companies or institutions, which often do not understand the threat. If well-mannered users can find the site, so can hackers.

It is thus important for developers of even small sites to harden their sites against attacks. No strategy can guarantee protection against SOP attacks, but many make it more difficult for such attacks to succeed. When attackers exploit SOP vulnerabilities, they can steal passwords and cookies, log keystrokes, and alter information. Having good protection against CSRF attacks buys you nothing if a hacker can hijack user sessions. Therefore, it makes sense to prioritize security measures by protecting against XSS first, CSRF second.

Scanning tools and services

Scanning services and vulnerability-checking tools find common flaws, such as not filtering user input.[8] Many popular vulnerability-checking tools are free, and they test for basic SQL injection and XSS vulnerabilities, system configuration problems, default passwords, and so on. However, they fail to detect CSRF or DOMXSS attacks, which do not exactly fit the automated scanning template; moreover, such services might not be updated regularly enough to detect the newest exploits. For a fee, professional scanning services can examine Web code and simulate traditional and nontraditional attacks in a safe environment.

Confine untrusted domain data to their own <iframe>

When dealing with untrusted content, open it up in an <iframe> with its own JavaScript execution context and its own DOM elements. Using a different domain lets you leverage the browser's SOP to isolate code and elements on the main page from malicious code or elements on the <iframe>. This disassociation is beneficial but not always possible. Mashups, for example, depend on the interaction of data and components from multiple sources.

Avoid eval() and dynamically generated code

The eval() function lets the browser execute any string as JavaScript code. Web applications that do not properly validate input data risk executing malicious code. Avoid dynamically generated code unless absolutely necessary. JSON strings are meant to be a relatively safe subset of JavaScript that lets data safely pass through a Web application's eval() function. However, attackers might attempt to pass malformed JSON strings to your application, so use regular expressions or parseJson() to check for non-JSON strings.

HTML validation and escape of untrusted data

Web servers must validate all input, including URLs, query strings, and post input. Sites that host blogs, forums, comments, reviews, and social networks let users contribute their own content, including HTML code and rich data. As previously noted, malicious users exploit SOP by uploading their own JavaScript routines. In the Samy MySpace worm example, MySpace in fact did filter the profile page submissions, but Samy circumvented the filters by breaking up the filtered words over multiple lines. Better filtering would have frustrated the attack. Server-side validation can remove potentially malicious tags and scripts from untrusted user-supplied content and reduce the threat of XSS attacks. Client-side validation is not reliable for security purposes. Use one of the freely available security-focused encoding libraries to help validate untrusted data.

Use the HttpOnly cookie attribute

The HttpOnly cookie attribute is a cookie security control option that, if set, prevents JavaScript from accessing or modifying a cookie, making it more difficult for an attacker to steal or abuse a session.[9] Other cookie security parameters are also useful. The path attribute restricts cookie access to a specific path in the URL, which makes it a little more secure than SOP. When set, the secure flag instructs the browser to only grant access to cookies from an HTTPS request. And setting the expires attribute to a date in the past makes the browser delete the cookie immediately instead of waiting until it closes.

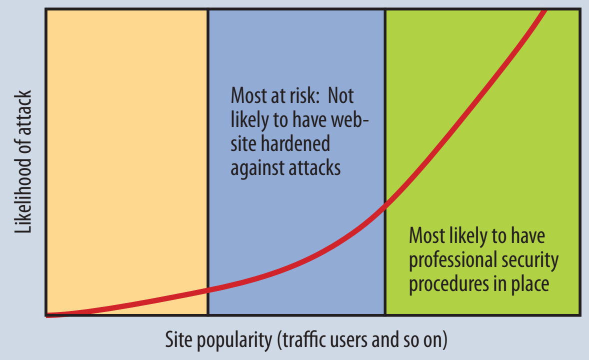

Figure 1. Website popularity versus likelihood of attack. The most vulnerable websites are those of small- to medium-size companies or institutions, which often do not understand the threat.

Use cryptographic tokens or captchas for high-risk GET/POST requests

Jeremiah Grossman[6] and Thomas Schreiber[10] both advocated including a cryptographic token to all links and forms that modify server-side data to provide strong protection against CSRF attacks. Presenting the user with a link or form that includes an unpredictable element specific to that action, user, and session disrupts these attacks. To implement this token, you can use a hidden input form element with a value from a keyed cryptographic hash like HMAC_sha1(Action_Name + Secret, SessionID). Before the server executes the request, it generates the same code or hash and compares it against the user-submitted one; if the two do not match, the server aborts the action.

Captcha and other challenge-response mechanisms are also effective, but they affect the user experience, so use them with care. Cryptographic synchronizer tokens are not visible to users and require no additional user steps.

Avoid third-party code

Instead of pointing to a JavaScript or image file on a remote server, copy the file to your webserver and reference it from there. Do not trust third-party ads. If a webpage must contain ads, make sure they come from a reputable company with an excellent security record.

Any scripts injected into the website, including those for ads, have complete access to all webpage content.

Third-party scripts lower the website's security boundary. If a hacker alters that script, that and every other site using the same script will put confidential information at risk. Multiple websites using scripts from a handful of entities creates the potential for a single point of compromise. If it is absolutely necessary to use third-party scripts, check them for vulnerabilities at secunia.com.

User precautionary reminders

If a Web application involves sensitive user information, then its users will probably appreciate security reminders. Use e-mail or website notifications encouraging users to log out immediately after using the Web application; turn off JavaScript or use white-list plug-ins like No-Script; use a different browser to access secure or sensitive websites than the one being used to browse the Internet freely, especially with tabbed browsing; and do not allow browsers to remember usernames and passwords.

Basic security logic suggests that third-party entities should not have the same access privileges as trusted entities.

Mozilla Content Security Policy

One important security measure recently implemented in Mozilla Firefox 4 is the Content Security Policy (CSP). Aimed at mitigating XSS and click-jacking attacks, the CSP employs a set of directives that define the security policy for all types of webpage content on the webpage (https://wiki.mozilla.org/Security/CSP/Specification). The Web developer or administrator specifies a list of hosts or URIs that can supply each content type. Additionally, the CSP restricts common attack vectors in the client browser, denying inline <script> tags, calls to eval(), and other methods of creating code from strings.

NEW BROWSER SECURITY MODELS

The preceding techniques mitigate but do not solve the root problem—the lack of an appropriate security model. Many proposed new models aim to alleviate browser security, but two that stand out as innovative and noteworthy also complement each other.

Cryptographic server identity (locked SOP)

Chris Karlof and colleagues introduced a method that enforces access control not only via a website's host, port, and application protocol, but also through the webserver's cryptographic identity: a browser only grants access if the server's public Secure Sockets Layer (SSL) key matches the key from the locked Web objects.[7] This is crucial to protect against pharming attacks, which manipulate DNS records and return the attacker's IP address with the target site's name. Victims are unaware that they are under attack since the URL in their browser shows the expected host name. Using the webserver's cryptographic identity, the browser would detect and deny any server whose cryptographic identity does not match the website's SSL public keys.

This proposal includes two policies: weak and strong locked SOPs. In traditional SSL server connections, the browser warns the user if the SSL certificate is unsigned or if it has any errors, but users tend to ignore the warning. SSL warnings can indicate a DNS spoofing or man-in-the-middle attack. Using the weak locked SOP, the browser would only allow a locked Web object to access another locked Web object if the standard SOP would have allowed it and if the object's certificate had no errors or warnings.

For the strong locked SOP, the browser tags locked Web objects with the public key of the webserver at the other end of the SSL connection. A browser implementing the strong policy would only allow access between locked Web objects if the standard SOP would have allowed it and if their tags match.

In a dynamic pharming attack, the attacker controls the main page, and the <iframe> contains the genuine trusted website, but only the true website server could produce the cryptographic credentials necessary to verify its identity. Therefore, the strong locked SOP would prevent the attacker's page from accessing anything on the genuine page. By authenticating the server in this way, the strong locked SOP prevents dynamic pharming attacks. The locked SOPs, however, do nothing to thwart XSS or CSRF attacks.

Escudo Web protection

Basic security logic suggests that third-party entities should not have the same access privileges as trusted entities. If various sections within webpages included access-control mechanisms, a programmer could wall off untrusted scripts and content from accessing or changing trusted code or sensitive information.

Escudo is a new Web browser protection model that uses mandatory access control to wall off content from various sources and levels of trustworthiness.[7] By enforcing access rules similar to those found in some file systems, it seeks to enforce the separation of privilege and the principle of least privilege, the lack of which contributes heavily to XSS and CSRF attacks. With Escudo, Web developers identify the principles and objects in the code along with their levels of trustworthiness, and the Web browser implements those access decisions.

Unlike the CSP, which lists allowable sources of each content type for the whole page, Escudo is more granular; it identifies access rights to specific sections and elements of the page, regardless of the content source. Developers assign all the elements of each webpage to a protection ring based on the trustworthiness of those elements and their protection requirements. The developer is free to apply as many rings as is necessary to protect the application's security.

Escudo lets developers define the meaning of a particular ring number, but ring level 0 is the most privileged. Principals can access elements with equal or lesser privilege, so a principal in ring level 2 can only perform operations on elements in ring levels 2 and higher. To assign ring levels, the developer encapsulates various page elements inside a div tag with a new attribute called ring. In addition to the ring boundaries, Escudo also incorporates access-control lists (ACLs), which let developers specify the minimum privilege level to read, write, or use a particular element.

The following code example defines a set of rings and access-control assignments:

```
<div ring=2 r=1 w=0>
  ...
  <div ring=3 r=2 w=0>
    ...
  </div>
</div>
```

In this code segment, the outer ring is level 2. The ACL assignments require that a principal must have a ring level of 1 to read the element (r = 1), and ring level 0 to modify it (w = 0). The combination of ring levels and ACLs gives Escudo a high degree of access granularity and lets developers employ the principle of least privilege in various parts of their applications.

For nested rings, inner rings must have a lower privilege level than outer rings, or else the Escudo security policy in the browser ignores them. This prevents untrusted sources from injecting code with a higher privilege level. Furthermore, div tags can include markup randomization attributes such as nonces to prevent injected code from splitting the div tag and creating a new div region with elevated privileges. Properly configured, the Escudo-enabled browser assigns untrusted principals to the least-privileged ring, where they cannot access or alter the rest of the page.

As the following example shows, Escudo rings separate untrusted page elements from trusted elements, thereby preventing a malicious user from altering the content:

```
<html><head><title>Paul's Blog</title>
</head><body>
  <div ring=2 r=0 w=0 x=0
  nonce=23409750497590487>
    <h1>Scavenger Hunt!<h1>
    <hr>
    <h2>Paul: I will award the student
    bringing me the following items:</h2>
    <ul>
    <li>Yellow #2 pencil</li>
    <li>Secretary's middle name</li>
    <li>Number of ceiling tiles in our
    lab</li>
    </ul>
   </div nonce=23409750497590487>
   <hr>
   <h4>Comments</h4>
   <div ring=3 r=1 w=1 x=1
   nonce=23409750497590487>
    Karthick: What will we get?
    <div ring=0 r=0 w=0 x=0>
      <script>
      //malicious script to modify the
      above list
      </script>
    </div>
   </div nonce=23409750497590487>
   <hr>
</body></html>
```

The attempt to embed a ring level of 0 will also fail, because it resides within ring level 3.

Using attributes in the HTTP header, Escudo rings can also protect cookies, browser API code, and browser history. Escudo is backward compatible; Web browsers that do not support the mechanism will simply ignore the Escudo attributes in the div tag and implement SOP as always.

Web developers need better control and security mechanisms. Data and code from untrusted sources should not have the same privileges as the trusted programmer's code. Looking forward, the CSP should continue to improve; future development may include protection attributes similar to Escudo to allow fine-grained access control. Escudo itself could benefit by incorporating a valid-only flag, similar to the locked SOP, instructing the browser to ignore items within the div tag unless the SSL certificate(s) for the locked items in the tag are valid.

As the Internet evolves, the Web browser's fundamental security mechanism also must evolve. If the future Web browser is to be an effective user interface for experiencing the Internet, privacy, trust, and security will be among its most important qualities.

References

1. A. Rubin and D. Geer, "A Survey of Web Security," *Computer*, Sept. 1998, pp. 34-41.
2. J. Mischel, "Browser Applications and the Same Origin Policy," *informIT*, 6 Aug. 2010; www.informit.com/guides/content.aspx?g=dotnet&seqNum=809.
3. J. Grossman, "Cross-Site Request Forgery: The Sleeping Giant," *WhiteHat Security*, July 2007; www.whitehatsec.com/home/assets/WPCSRF072307.pdf.
4. A. Barth, C. Jackson, and J.C. Mitchell, "Robust Defenses for Cross-Site Request Forgery," *Proc. 15th ACM Conf. Computer and Communications Security* (CCS 08), ACM Press, 2008, pp. 75-87.
5. K. Jayaraman et al., "ESCUDO: A Fine-grained Protection Model for Web Browsers," *Proc. IEEE 30th Int'l Conf. Distributed Computing Systems* (ICDCS 10), IEEE CS Press, 2010, pp. 231-240.
6. J. Grossman, "Cross-Site Scripting Worms and Viruses: The Impending Threat and the Best Defense," *WhiteHat Security*, Apr. 2006; http://net-security.org/dl/articles/WHXSSThreats.pdf.
7. C. Karlof et al., "Dynamic Pharming Attacks and Locked Same-Origin Policies for Web Browsers," *Proc. 14th ACM Conf. Computer and Communications Security* (CCS 07), ACM Press, 2007, pp. 58-71.
8. M. Curphey and R. Arawo, "Web Application Security Assessment Tools," *IEEE Security & Privacy*, July/Aug. 2006, pp. 32-41.
9. S. Crites, F. Hsu, and H. Chen, "OMash: Enabling Secure Web Mashups via Object Abstractions," *Proc. 15th ACM Conf. Computer and Communications Security* (CCS 08), ACM Press, 2008, pp. 99-107.
10. T. Schreiber, "Session Riding: A Widespread Vulnerability in Today's Web Applications," *SecureNet GmbH*, Dec. 2004; www.securenet.de/papers/Session_Riding.pdf.

Hossein Saiedian *is a professor of software engineering in the Department of Electrical Engineering and Computer Science at the University of Kansas, where he also is a member of the Information and Telecommunication Technology Center. His research focuses on software engineering, particularly technical and managerial models for quality software development. Saiedian received a PhD in computer science from Kansas State University. He is a senior member of IEEE. Contact him at saiedian@ku.edu.*

Dan S. Broyles *is an engineer in the technology development organization at Sprint Nextel, where he provides spectrum analysis applications and automation tools for internal engineering teams. He received an MS in information technology from the University of Kansas. Contact him at daniel.s.broyles@sprint.com.*