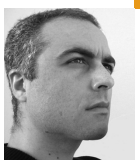
GIORGIO MAONE

hardening the Web with NoScript



Giorgio Maone is CEO and CTO of InformAction, a software development and IT consulting firm based in Italy. He's the author and main developer of NoScript, a popular open source solution enhancing browser security.

g.maone@informaction.com

NOSCRIPT IS A POPULAR SECURITY add-on for Firefox and other Web browsers based on Mozilla technology. Although it is mainly known for providing easy fine-grained script blocking at the domain level, NoScript pioneered several innovative and unique client-side countermeasures against emergent Web-based threats, such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and UI redressing (also known as "clickjacking"), which had previously believed to be addressable on the server side only.

Default Deny, Easy Allow

Since its very first release (May 2005 [1]), NoScript's core feature has been whitelist-based selective activation of "executable" Web content. JavaScript and browser plugins containing scripting interpreters and just-in-time compilers, such as Sun's Java, Adobe's Flash, or Microsoft's Silverlight, have turned the Web, which had been originally intended as an interlinked collection of static documents, into a rather anarchic executable platform with loose or nonexistent security checks. The *same-origin policy* (stating that active Web content on a certain site must not be allowed to access data or execute code from a different site) and sandboxing mechanisms (meant to prevent active Web content from reaching out of the browser and interacting with the underlying system) have long been the only security constraints enforced by browsers on Web-based "programs," but they get violated very often, because of design flaws or implementation bugs.

Design flaws, responsible for "structural" vulnerabilities such as XSS, CSRF, or UI redressing, are very unlikely to be fixed in a satisfactory way, because of compatibility concerns: their mitigation is delegated to Web development "good practices" recommendations, doomed to remain almost always unheard or misunderstood. Implementation bugs, often allowing malicious Web content to escalate privileges and compromise a user's account or system, are the reason why Web browsers and their plugins are bound to impressively tight patching and updating cycles, which are indispensable to keep an acceptable degree of security [2]. Unfortunately, the rise of a florid zero-day vulnerabilities black market, full disclosure stunts, corporate rules slowing down or banning automatic updates, leg-

acy compatibility needs, and other factors can significantly widen the exposure window of many users to unpatched browser and plugin vulnerabilities, which have quickly become a major venue of malware propagation.

Nearly every security vulnerability that has been affecting the browser or its plugins so far could be mitigated or even, more often than not, completely neutered by disabling JavaScript or, when applicable, the vulnerable plugin. In fact, almost all the security advisories about exploitable browser flaws play the "Disable JavaScript" card as the only possible workaround until a patch is available. However, in the modern Web, where many sites and applications rely on JavaScript-based techniques (e.g., DHTML and AJAX) to enhance users' "experience" or even to implement their basic functionality, this simple and effective countermeasure is often impractical.

But what if you had a quick and easy way to enable JavaScript and potentially dangerous plugins *only on those sites you trust*, either permanently or just when you need to? This is exactly what NoScript has been conceived for: enforcing a "Default Deny" policy on *active* Web content, yet providing users with the ability to whitelist trusted domains on the fly as needed, by popping up a contextual menu and selecting the proper "Allow some.trusted.domain.com" command. A subtle non-modal notification bar is displayed on the bottom of pages where active content has been disabled, to remind you that some script *might* need to be allowed if the site doesn't work properly. This feedback system has been carefully designed to be as discreet as possible and never get in your way, especially on those Web sites which do work fine even if scripting is disabled. NoScript neither begs for attention nor requires any user interaction: it tries to avoid the trap of training users to permit everything, an effect that modal security questions ("Allow this?" "Block that?") are often accused of causing.

Site-level permissions for active Web content actually had a venerable precursor in Microsoft Internet Explorer 6's "Security Zones" [3], and about nine months after NoScript's appearance, Opera 9 provided a user interface for configuring "Site specific preferences" [4], including JavaScript, Java, and Plugins. However, IE's Zones, being mainly oriented to enforcing corporate policies, are buried deep inside the Internet Options panel and quite hard to configure for end users, while Opera's implementation, albeit user-friendlier, lacks any contextual feedback system and the ability to discriminate among third-party imported scripts, both required for effective security-grade script management.

Deflecting Reflective XSS

Assuming that the whitelist policy for active content execution is effectively enforced and cannot be violated—NoScript's implementation has never been broken so far—is there still any way for malicious code to run against a user's will? Sadly, the answer is yes: quite obviously, it is sufficient for the malicious code to be injected in any of the whitelisted sites. This can be achieved by hacking the Web server that hosts the site or, much more frequently, through a Cross-Site Scripting (XSS) attack.

XSS vulnerabilities affect those Web applications which don't properly escape their input when it is echoed back as (X)HTML output: this allows script fragments crafted by the possibly malicious user controlling the input to be executed by the browser in the context of the vulnerable site. According to studies by the Web Application Security Consortium [5] and WhiteHat Security [6], corroborated by live data from the XSS Project [7], this kind of vulnerability is the most prevalent in Web application security and

affects an overwhelming majority of Web sites, from social networks to online banking applications, no matter how popular and/or resourceful they are. Of course, XSS lowering the effectiveness of script blocking is a minor concern compared to its overall impact: XSS attacks can be used to silently steal credentials, perform stealth financial transactions impersonating a logged-in user, or set up "perfect" phishing attacks (undetectable, since the fake page comes from the real domain).

The painful awareness of these threats and the complete lack of initiative by the browser vendor against them, being considered at the time a server-side only problem which could never be mitigated on the client side, ignited the development of the first in-browser XSS filter, which was publicly released as a NoScript component called InjectionChecker in March 2007. InjectionChecker examines any cross-site HTTP request for HTML documents, looking for HTML or JavaScript fragments that could be injected in the destination page. If one is found, the request gets sanitized by stripping out the potential payload before it's sent. Initially considered with skepticism by both security researchers and browser vendors, this approach quickly demonstrated its reliability and effectiveness against Type 0 (DOM-based) and Type 1 (Reflective) XSS attacks. The main limitation of the earliest InjectionChecker versions, which were based exclusively on pattern matching, was a moderately high false-positive rate. However, after some development iterations, the analysis algorithms underwent a radical overhaul: by leveraging the browser's JavaScript interpreter itself (SpiderMonkey) in order to discriminate non-trivial and syntactically valid script injections from innocuous but suspect request data, newer versions managed to reduce the false-positive rate near to 0. Still, even though extremely rare, a cross-site request might legitimately include some valid HTML or JavaScript fragment and therefore trigger the InjectionChecker. However, this residual issue is alleviated by the non-blocking design of the filter which, rather than preventing the possibly attacked page from loading or brutally disabling its scripting capabilities, just sanitizes the request, modifying the bare minimum for the attack to fail: this approach usually keeps the landing page functional. Furthermore, the issued warning message is non-modal (like every notification from NoScript) and gives the user an option to examine the original request and replay it unfiltered, if it is deemed safe. Finally, exceptions for safe origins or targets can easily be configured to handle specific situations.

The success of NoScript's XSS filters probably encouraged browser vendors to approach this problem with fewer prejudices. In fact, even if more than one year later, Microsoft revealed that an XSS protection subsystem, impressively resembling NoScript's InjectionChecker, was being added to Internet Explorer 8 [8], and in September 2009 Adam Barth announced a similar development effort in progress for the open source Chromium browser on which Google Chrome is based [9]. Notwithstanding, both Microsoft's and Google's solutions appear quite limited compared to NoScript's: since they act on the page rather than on the request, they're unable to neutralize Type 0 (DOM-based) XSS and, at least in Microsoft's case, new XSS vulnerabilities can be introduced by the neutering routine itself, when it modifies the landing document's contents.

ClearClick vs Clickjacking

In September 2008 Jeremiah Grossman (WhiteHat Security) and Robert "RSnake" Hansen (SecTheory) generated lots of buzz when, requested by Adobe, they canceled a speech scheduled for the World OWASP AppSec conference in New York. A new exploitation technique they were going

to present, dubbed "clickjacking," implied many more critical consequences than initially thought, if combined with an otherwise minor flaw in the Flash browser plugin [10]. As was revealed after Adobe had fixed its plugin-specific issue, a remote attacker could easily modify the local Flash privacy settings and start spying on a user's activity through his microphone or Webcam [11].

While speculations about the nature of this mysterious attack flourished, some observers deduced from the available information that, even if the specific exploitation scenario was indeed new and spectacular, the underlying vulnerability was a known one, endemic in all the modern browsers but still underestimated (or, better, understated, because no obvious solution could be deployed without drastically breaking the Web as we know it): UI redressing [12]. This is the problem definition as put by Google's browser security expert Michal Zalewski:

> A malicious page in domain A may create an IFRAME pointing to an application in domain B, to which the user is currently authenticated with cookies. The top-level page may then cover portions of the IFRAME with other visual elements to seamlessly hide everything but a single UI button in domain B, such as "delete all items," "click to add Bob as a friend," etc. It may then provide [its] own, misleading UI that implies that the button serves a different purpose and is a part of site A, inviting the user to click it. Although the examples above are naive, this is clearly a problem for a good number of modern, complex Web applications. [13]

UI redress/clickjacking, in its simplicity, is actually much more faceted and difficult to approach than it seems: variants may target same-site plugin content (as in the famous Adobe case) rather than cross-site documents, the victim UI can be rendered transparent by abusing the CSS "opacity" property rather than by being covered by the parent malicious site, keyboard strokes might be solicited rather than clicks, and so on. In spite of the fact that NoScript, as noted by Jeremiah Grossman in his early interviews before full disclosure, provided protection against his Flash-based clickjacking exploit by default and against the more general scriptless UI redress attacks if users enabled the "Forbid IFrames" option, the latter configuration was much too inconvenient to be recommended to the general public.

There was clearly a need for a specific countermeasure, which had not been conceived yet. So on October 7, 2008, after a week-long design and coding marathon, a prototype of the ClearClick NoScript module could be finally released [14]. ClearClick's concept is almost as simple as UI redressing itself: whenever a mouse or keyboard interaction is engaged with a cross-site framed document or an embedded plugin object, event processing gets temporarily suspended while two screenshots of the involved item are compared: one taken from the top-level window (reproducing the user's point of view), the other taken after isolating and opacizing the event target. If the two images match, the user can see "the naked truth" and the original mouse or keyboard event processing is immediately resumed. Otherwise, the situation is considered suspect because the event target is concealed, transparent, or otherwise not clearly visible: a warning is issued, showing both the screenshots for easy visual verification and allowing the user to judge if the interaction needs to be aborted or not.

Some months later Microsoft announced with a fanfare [15] that "clickjacking protection" was being added to IE8, but it was quickly exposed [16] as an "X-Frame-Options" HTTP header which should be sent by Web sites when they do not want to be framed: an opt-in proposal requiring Web developers' cooperation, then, not comparable to a client-side automatic solu-

tion like ClearClick. Nevertheless, NoScript implemented this feature as well (just a few hours after it had been revealed) for compatibility's sake, while Apple's Safari 4 and Google's Chrome 2 followed the lead later. However, as Google's "Browser Security Handbook" itself explains,

> So far, the only freely available product that offers a reasonable degree of protection against the possibility is NoScript (with the recently introduced ClearClick extension). To a much lesser extent, an opt-in defense is available [for] Microsoft Internet Explorer 8, Safari 4, and Chrome 2, through a X-Frame-Options header, enabling pages to refuse being rendered in any frames at all (DENY), or in non-same-origin ones only (SAMEORIGIN) [18].

ABE Patrolling the Web's Borders

Cross-Site Request Forgery (CSRF) had been called "the sleeping giant" [19] back in 2006, because it was as ubiquitous as it was misunderstood. If a Web application is vulnerable, a malicious site can perform unintended actions (e.g., to transfer funds or change router settings) on behalf of the users who are browsing it, by silently sending a known HTTP "command" request through one of the many automatic navigation vehicles provided by HTML and JavaScript. The browser will automatically add authorization information, either as a session cookie or an Authorization header. Three years later the giant has awakened, even though some progress has been done in prevention: awareness grew among developers, and support for countermeasures, such as explicit security tokens, has been introduced in popular Web application frameworks.

However, as usual, mitigation is left to Web authors' skill and good will, with no help from the client side and no control in user's hands. ABE (Application Boundaries Enforcer), a project sponsored by the NLnet Foundation [20], tries to improve this situation.

Released as a NoScript component in June 2009 [21], but planned to be also decoupled from the Firefox add-on and ported to different browsers, ABE is a firewall-like system which allows users, Web developers or trusted third parties (subscription providers) to configure "Rulesets" declaring the boundaries of one or more Web applications. Rules are expressed using a syntax [22] which should look natural to any system administrator. This rule, for instance, can be used to protect Gmail against CSRF attacks:

```
Site mail.google.com
Accept from SELF, www.google.com
Deny
```

It causes Gmail (mail.google.com) to reject (Deny) all the potentially forged requests, identified as those coming from any site except mail.google.com itself (SELF) and www.google.com, the domain from which the login form for the Google application is served. Selectors can be much more fine-grained than these, allowing glob patterns and regular expressions to be combined in site specifications and HTTP methods to be used as criteria to match requests. Documentation and examples are available on the project Web site, http://noscript.net/abe/.

Rules are enforced at the beginning of the HTTP load cycle, preventing malicious requests from doing any harm. Furthermore, since it lives inside the browser, ABE knows the real origin of each request, allowing decisions to reliably depend on this information but not requiring it to be leaked through the wire, unlike the Referer HTTP header which, indeed, often gets sup-

pressed or forged because of privacy concerns and, for this reason, must not be trusted.

Any Web site can protect its boundaries by providing an ABE rule set in its root directory, but they can't override the user's own rule sets or those on other sites. ABE refreshes site-provided rule sets when a session starts, then hourly, but honors HTTP caching hints if provided.

Users can add their own rules, which take precedence over the ones pushed by trusted third parties and Web applications, by editing the initially empty USER rule set accessible from the ABE panel, among NoScript's Advanced options. A visual UI to build rules contextually, during navigation, is under development.

The only ruleset provided at installation time, labeled SYSTEM, includes just one rule:

```
Site LOCAL
Accept from LOCAL
Deny
```

This quite obviously means that requests toward local sites (i.e., private IPv4 and IPv6 networks according to RFC 3330 and RFC 4193) are blocked unless they come from origins which are local as well. Such a rule automatically protects intranets against scanning and CSRF attacks toward internal Web applications and devices (e.g., router hacking) initiated from malicious Internet Web sites [23].

Did You Know?

Although often described as a "simple" script blocker, NoScript features multiple additional security enhancements, completely independent of its script-blocking core. Some users may believe that maintaining a whitelist of trusted sites allowed to run scripts is too tedious in this AJAXified world. Nevertheless, they should give NoScript a try: no matter if they give up and resort to "Allow Scripts Globally (dangerous!)," the InjectionChecker, ClearClick, and ABE components, unattended and silent in the background, will keep delivering a degree of protection against XSS, clickjacking, and CSRF that is currently unmatched by any other available Web browser technology.

REFERENCES

[1] NoScript's public release versions history: https://addons.mozilla.org/en-US/firefox/addons/versions/722.

[2] T. Duebendorfer and S. Frei, "Why Silent Updates Boost Security," *ETH Tech Report 302,* May 5, 2009: http://www.techzoom.net/publications/silent-updates/.

[3] http://www.microsoft.com/windows/ie/ie6/using/howto/security/setup.mspx.

[4] "Opera 9 introduces 'Site specific preferences' User Interface," February 7, 2006: http://snapshot.opera.com/windows/w90p2.html

[5] Web Application Security Consortium, "Web Application Security Statistics 2007": http://www.Webappsec.org/projects/statistics/.

[6] WhiteHat Website Security Statistics Report: http://www.whitehatsec.com/home/resource/stats.html.

[7] The XSS Project: http://www.xssed.com.

[8] Giorgio Maone, "NoScript's Anti-XSS Filters Partially Ported to IE8," July 3, 2008: http://hackademix.net/2008/07/03/noscripts-anti-xss-filters-partially-ported-to-ie8/.

[9] Adam Barth, "Reflective XSS protection (for reals this time)," Chromium-dev Group, September 4, 2009: http://groups.google.com/group/chromium-dev/browse_thread/thread/d2931d7b670a1722/d56bdfccfcef677f.

[10] Robert Hansen, "Clickjacking," September 15, 2008: http://ha.ckers.org/blog/20080915/clickjacking/.

[11] Robert Hansen and Jeremiah Grossman, "Clickjacking," September 12, 2008: http://www.sectheory.com/clickjacking.htm.

[12] Mark Pilgrim, "This Week in HTML 5—Episode 7," September 29, 2009: http://blog.whatwg.org/this-week-in-html-5-episode-7.

[13] Michal Zalewski, "Dealing with UI Redress Vulnerabilities Inherent to the Current Web," WHATWG Mailing List, September 25, 2009: http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-September/016284.html.

[14] Giorgio Maone, "Hello ClearClick, Goodbye Clickjacking": http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/.

[15] Giorgio Maone, "Ehy IE8, I Can Has Some Clickjacking Protection?" January 27, 2009: http://hackademix.net/2009/01/27/ehy-ie8-i-can-has-some-clickjacking-protection/.

[16] Giorgio Maone, "IE8's 'Clickjacking Protection' Exposed," January 28, 2009: http://hackademix.net/2009/01/28/ie8s-clickjacking-protection-exposed/.

[17] Giorgio Maone, "X-FRAME-OPTIONS in Firefox," January 29, 2009: http://hackademix.net/2009/01/29/x-frame-options-in-firefox/.

[18] Michal Zalewski (Google Inc.), "Arbitrary Page Mashups (UI Redressing)," Browser Security Handbook: http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_%28UI_redressing%29.

[19] Jeremiah Grossman, "CSRF, the Sleeping Giant," September 26, 2006: http://jeremiahgrossman.blogspot.com/2006/09/csrf-sleeping-giant.html.

[20] NLnet Foundation's NoScript/ABE page: http://www.nlnet.nl/project/noscriptabe/.

[21] Giorgio Maone, "Meet ABE," June 30, 2009: http://hackademix.net/2009/06/30/meet-abe/.

[22] Giorgio Maone "ABE—Rules Syntax and Capabilities": http://noscript.net/abe/abe_rules.pdf.

[23] Jeremiah Grossman, "Hacking Intranet Websites from the Outside," Black Hat (USA)—Las Vegas, August 3, 2006: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf.