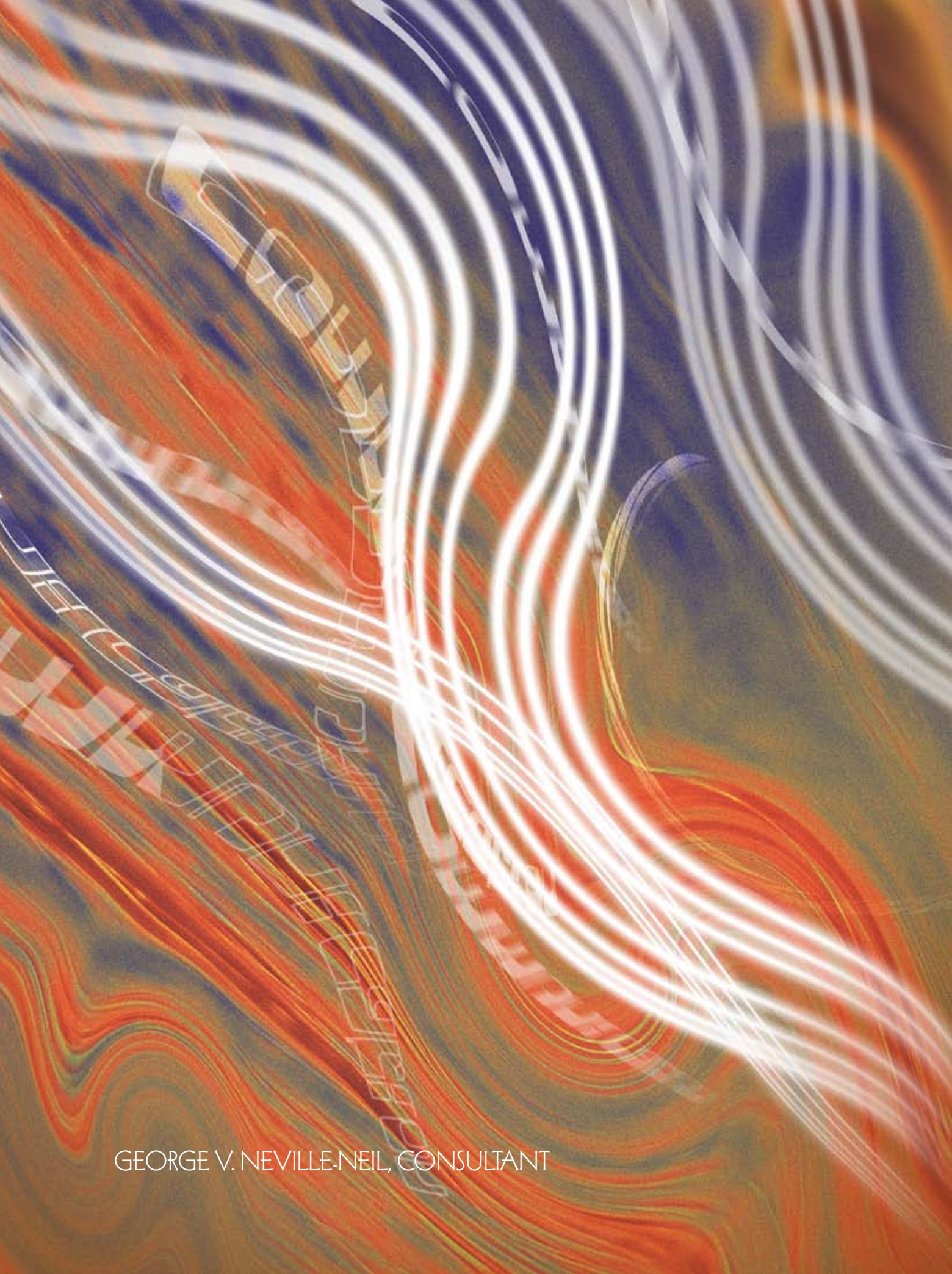
GEORGE V. NEVILLE-NEIL, CONSULTANT

Building Secure Web Applications

Believe it or not, it's not a lost cause.

In these days of phishing and near-daily announcements of identity theft via large-scale data losses, it seems almost ridiculous to talk about securing the Web. At this point most people seem ready to throw up their hands at the idea or to lock down one small component that they can control in order to keep the perceived chaos at bay.

Before going any further, let me first define the three main problems that people are trying to solve by building secure Web applications:

- The first problem most people encounter is *authentication*. How does the application know who is accessing it and what they are allowed to access?
- Problem two is the ability of an attacker to trick users, once they have authenticated, into doing work on the attacker's behalf. I call this problem *request forgery*.
- The last problem is the risk involved in hosting UGC (user-generated content) on a Web site.

In this article I discuss each of these problems, the current state of existing solutions, and the unsolved problems that continue to plague large Web applications.

THE BROWSER SECURITY MODEL

The Web is, by its nature, a difficult system to secure. To provide a unified user experience with adequate performance, most large Web applications are distributed, and the requests and replies used to get work done in any system attempt, as much as possible, to be stateless. Because each request attempts to be stateless, the server itself must maintain all the state governing a user's actions. Unlike an application, but like remote login, no single, long-lived session is maintained between the client—aka the user's browser—and the server. To make a flow of Web pages seem more like a Web session, there are several tricks to maintaining and communicating state on each request between the browser and the server. Authentication is a good example.

To authenticate themselves to a Web application, users must prove they are who they claim to be. Most applications have username and password schemes where each user is assigned a unique username and then selects a password that is known only to that user. Once the user has successfully given the username and password to the application, the user moves to a logged-in state. These concepts are familiar to anyone who has used a computer in the past 50 years. They come directly from the multi-user time-sharing systems of the 1960s.

The problem is, whereas with a time-sharing system users had some sort of unique mapping between them

Building Secure Web Applications

and a communications device such as a terminal or long-lived network connection, with the Web that is not the case. The user could, for example, be using a service on mail.foo.com and then decide to click on the calendar and access calendar.foo.com, which is likely to be a completely different set of servers and may even be in a different physical location from the mail servers. They are still related services provided by foo.com, however, and the switch between them should appear seamless to the user.

The way in which large Web applications maintain authentication state is to depend on *cookies*, which are set by servers and stored in the browser, and to depend on the browser to implement what is now called the *browser security model*, though it ought to be called the *cookie security model*. This model is a contract between the servers and the browser, stating that the browser will send cookies only to a server in a domain that set them in the first place. The domain is defined by the DNS name assigned to the servers. In the previous example, foo.com is the domain name under which the cookies are set.

As in any protocol implemented independently by several parties, different implementations have different bugs. This means that the security model as implemented by Mozilla in Firefox is likely slightly different from the one implemented by Microsoft in Internet Explorer, and those are different from the one implemented in Opera, and so on. The security model is actually the union of all the available implementations. Protecting your system from only the browser with the biggest market share always leads to trouble.

A concrete example will aid in understanding the security model. On every request the user's Web browser sends the set of cookies that it believes belongs to the domain that the user is working with. If the user is working with the domain mybank.com, then the browser will send the cookies that have been set in the mybank.com domain. If the user moves to another domain—for example, freemail.com—then the browser sends only the cookies that are set by freemail.com servers. Changing any part of the domain changes the set of cookies that the browser is willing to send. Thus, freemail.com and freemail.net are not equivalent; they are distinct domains.

COOKIES

In a Web application the user's cookies act as an authentication token to use the system. I do not wish to address the problem of correctly implementing cookies as login tokens here. Suffice it to say that cookies need to be cryptographically signed to make sure that they aren't tampered with and have to be issued only after a valid password or other piece of information is verified by the servers. The two problems with cookies that I address here are: How much scope should they have, and for how long should they be valid?

The *scope* of a cookie refers to how many services it can access. From the user's perspective, a company's whole Web site represents a single application. If, for example, a company provides e-mail, calendars, blogs, and product reviews, the users would like to log in once and then be able to access all those systems equally. Having to type a password for each subsystem, such as mail, calendar, blogs, and reviews, is considered a poor user experience and would not be tolerated.

There are other risks with having too tightly defined cookies and requiring users to type their passwords more often. One of the reasons that phishing is so successful is that users have been conditioned to type their passwords in response to a particular page. As long as users are presented with a page that looks similar to the login page, they will type their passwords. Studies have shown that even savvy computer users can be fooled by a well-constructed, but fake, login page.[1] The more often users type their passwords, the higher the risk that they will be phished. Most large services allow their users to sign in once and then to roam about the entire site for some period of time. Only when the users do something that is deemed to be more serious, such as purchasing a product, are they asked again to prove who they are by typing a password.

Deciding how long cookies should remain *valid* is also a security issue. Clearly, making cookies valid for only one transaction would be ridiculous and annoying, requiring a password for each page transition. At the opposite end of the spectrum, making a cookie valid for all time is equally ridiculous because of the potential for theft and abuse.

The problems of cookie scope and lifetime have not been solved. Most sites, however, have their own rules of thumb, which they attempt to keep internally coherent.

REQUEST FORGERY

Once users have logged in and their browsers are dutifully sending their cookies to the server on each request,

another problem arises. What if users are able to make modifications to their data on the site they're using? For example, they could be writing a restaurant review or a blog entry or even changing their password. If the URL of the service that is being used to perform the update is easily guessable, then the user is open to request forgery.

In a request-forgery attack, the attacker creates an URL that will cause something to happen to the user's account when the user clicks on it. Imagine a blogging system where, to delete an entry, the user posts an URL such as

```
http://myblog.com/username/delete=blogentry
```

where username is the user's username and the blogentry is a number.

An attacker wanting to delete a victim's blog entry needs only to craft an URL using the victim's real username (usually visible on the blog) and the appropriate blogentry number inserted, and then trick the victim into submitting that URL. The attacker can send the victim an URL, such as the one shown below, or place it in a Web page that the victim is likely to read:

```
<a img=http://myblog.com/victim/delete=blogentry>
```

The user doesn't see anything, but if this image tag is successfully deployed in a page on the myblog.com site and the user views it, then the blog entry will be deleted.

The server needs a way of figuring out that the user actually meant to take an action. One way is to request the password again on such an action, or the server can make each URL unique in some way, usually by requiring that a signature be generated and put into any URL that takes an action on behalf of the user.

JAVASCRIPT

The challenges we've been talking about so far were present in the static Web content of the 1990s, but an even thornier set of problems is associated with the dynamic Web of the past decade. Whereas it used to be simply that static HTML pages were served to the browser, now code is actually executed in the browser.

This code is often written in JavaScript, an interpreted language that the browser executes while it is displaying a page to the user. Almost everything interesting that can be seen on the Web or any advanced UI tricks are performed with bits of JavaScript downloaded from the Web server and operating within the browser. JavaScript can do just about anything, including looking at all the data in the page the user is looking at—after all, it was designed to manipulate pages and make them look more interesting. JavaScript can also grab and manipulate the user's cookies and other metadata.

Browsers do not follow the same restrictions on executing JavaScript that they do on setting cookies. A browser will accept JavaScript from any location that a page tells it to.

Unlike cookies, JavaScript does not adhere to a security model based on the domain name of the site. Once a piece of JavaScript has control of your browser, it can do anything, in any domain it likes, no matter from where the code was served or from where it makes additional requests.

One of the risks with JavaScript is that a company has to trust what it's serving up to its users. A company that serves third-party JavaScript is taking a serious risk with its site because if the third-party JavaScript is malignant, or just contains errors, those problems will directly affect everyone who is served that page. The problem, as the next section explains, is not just a third party serving JavaScript. Many sites now allow users to customize their pages by adding JavaScript to them.

USER-GENERATED CONTENT

Most people who write JavaScript for a living are not malware authors, and though they make mistakes, they are genuinely interested in doing things right. The biggest challenge facing Web sites now is making UGC safe.

The past few years have seen an explosion of sites that allow users to upload and share content. Whether it's blog entries, reviews, audio, JavaScript, photos, or video, all of these data types are generated by users and then shared with thousands, perhaps millions, of other users worldwide. It's simply not possible to have human editors review all uploads by all users before they are pushed out onto the network. Virus-scanning software and other blacklist-based solutions go only so far, as the scanning software needs to be constantly updated with new signatures to prevent new attacks from spreading.

You might think that as long as code is not shared among users, there is no way for a virus to spread, but with the advent of the JPEG virus, which inhabits the header of JPEG-encoded pictures, even allowing users to upload photos provides a vector for viruses. Sites that allow users to customize the look of their pages with JavaScript are easy targets for viruses and worms written in that language. The challenge for any site that wishes to allow users to upload and share data is how to do it safely and how to protect the users from each other.

Building Secure Web Applications

The UGC challenge has a spectrum of solutions. The first is simple: do not allow any UGC and forgo making that a part of the service. Many large sites actually do exclude some or all UGC, but the market pressure is to use more. Those sites that do allow UGC see their popularity, and hence market value, increase.

Most sites now allow some form of UGC, and the smart ones do this by having a very restrictive whitelist of allowable things the user can upload. These whitelists can take on many forms. If the UGC is in the form of HTML data—for example, an online review—then users are restricted in which tags they can use in their text. They are usually allowed bold, underline, italic, and other simple formatting attributes, but they are not allowed to upload full HTML or to use script tags. In the case of photo sharing, the photo can be programmatically resized when it is uploaded, which destroys any data the user had in the JPEG header. To prevent people from instigating a denial-of-service attack on a video-sharing site, the uploaded videos are limited in size. Each type of UGC must be taken on a case-by-case basis. What works for photos may not work for video and vice versa.

Deciding how to handle UGC is similar to deciding on any of the challenges mentioned in this article. What you do is highly dependent on what your system wants to do and what data you need to protect. If users wind up with cookies as their authentication mechanism, then you cannot allow them to upload JavaScript because they will be able to programmatically steal cookies and thereby gain control over other users' data.

WHAT THE FUTURE HOLDS

At this point you may be tempted to unplug your computer and throw it away in frustration, but that isn't a very productive solution. The fact is, the Web, and the Internet that underlies it, was designed as a collaborative system for people who were, for the most part, going to behave well and not act maliciously. We are now long past the era where those design principles made sense. What is needed to make the Web a more secure system that people can use and want to use on a daily basis are a few features that were left out of the original design.

A better model of who can do what and to whom when a user is viewing a Web page is one step in the right direction. The browser security model, which depends on domain names to decide who receives the user's metadata, is probably too coarse of a model to make applications any more complicated than they already are. Every mashup is a security smash-up waiting to happen. People are still designing new systems for the Web that actually attempt to get around the limited browser security model, allowing unfettered access to data in one domain from another. This is clearly moving in the wrong direction.

Depending solely on cookies—in particular, those that have a scope global to the entire domain of a Web site—does not work well for systems with multiple services. It's time to find a new way of implementing sessions in the sessionless protocol that is HTTP requests. Finding a good way of preventing request forgery must be a component of any solution proposed for the session problem.

The RIA (rich Internet application), where more work is done using JavaScript or Flash in the browser, is the next frontier for security issues on the Web. Issues such as input validation and preventing malicious data from getting into the user's workspace become more difficult when the source code is downloaded to, and visible in, the browser. It will be very important for programmers to make sure that any sensitive data is verified by the server before the application acts on it.

The problems with UGC will continue because that is where the Web is going, and sites that can't handle serving UGC will be surpassed by sites that can. The solutions that are most needed now and in the future are those that allow users to generate and share content safely. Q

REFERENCES

1. Dhamija, R., Tygar, J.D., Hearst, M. 2006. Why phishing works. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*: 581-590.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

GEORGE V. NEVILLE-NEIL is a consultant on networking and operating system code. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking and operating systems. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who has made San Francisco his home since 1990.