acmqueue

The Web Won’t Be Safe or Secure until We Break It

Unless you’ve taken very particular precautions, assume every Web site you visit knows exactly who you are.

Jeremiah Grossman, WhiteHat Security

The Internet was designed to deliver information, but few people envisioned the vast amounts of information that would be involved or the personal nature of that information. Similarly, few could have foreseen the potential flaws in the design of the Internet—more specifically, Web browsers—that would expose this personal information, compromising the data of individuals and companies.

If people knew just how much of their personal information they unwittingly make available to each and every Web site they visit—even sites they’ve never been to before—they would be disturbed. If they give that Web site just one click of the mouse, out goes even more personally identifiable data, including full name and address, hometown, school, marital status, list of friends, photos, other Web sites they are logged in to, and in some cases, their browser’s auto-complete data and history of other sites they have visited.

Obtaining all this information has been possible for years. Today’s most popular browsers, including Chrome, Firefox, Internet Explorer, and Safari, do not offer adequate protection for their users. This risk of data loss seems to run counter to all the recent marketing hype about the new security features and improvements browser vendors have added to their products over the past several years, such as sandboxing, silent and automatic updates, increased software security, and anti-phishing and anti-malware warnings, all of which are enabled by default. While all are welcome advances, the fact is that these features are designed only to prevent a very particular class of browser attacks—those generally classified as *drive-by downloads*.

Drive-by downloads seek to escape the confines of the browser walls and infect the computer’s operating system below with malware. Without question, drive-by-downloads are a serious problem—millions of PCs have been compromised this way when encountering infected Web sites—but they are certainly not the *only* threat browser users face, especially in an era of organized cybercrime and ultra-targeted online advertising.

The techniques behind attacks that obtain personal information are completely different and just as dangerous as malware, perhaps more so since the solution is far more complicated than just installing antivirus software. These attack techniques have even more esoteric labels such as XSS (cross-site scripting), CSRF (cross-site request forgery), and clickjacking. These types of attacks are (mostly) content to remain within the browser walls, and they do not exploit memory-corruption bugs as do their drive-by download cousins, yet they are still able to do their dirty work without leaving a trace.

These attacks are primarily written with HTML, CSS (Cascading Style Sheets), and JavaScript, so they are not identifiable as malware by antivirus software in the classic sense. They take advantage of the flawed way in which the Internet was designed to work. The result is that these attack techniques are immune to protections that thwart drive-by downloads. Despite the dangers they pose, they receive very little attention outside the inner circles of the Web security industry. To get a clearer picture of these lesser-known attacks, it's important to understand a common Web technology use case.

HTML allows Web developers to include remotely hosted image files on a Web page from any location across the Web. For example, a Web site located at `http://coolwebsite/` may contain code such as:

```
<img src="http://someotherwebsite/image.png">
```

This instructs a visiting browser to send a Web request to `http://someotherwebsite/` automatically, and when returned, to display the image on the screen. The developer may tack on some JavaScript to detect if the image file was loaded successfully or contained an error:

```
<img src="http://someotherwebsite/image.png" onload="successful()" onerror="error()">
```

If the image file loaded correctly, then the "successful" JavaScript function executes. If an error occurred, then the `error` function executes. This code is completely typical and innocuous, but the same functionality can also be leveraged for invasive, malicious ends.

Now, let's say `http://coolwebsite/` loaded an image file from `http://someotherwebsite/`, but that image file is accessible only if the user's browser is currently logged into `http://someotherwebsite/`. As before:

```
<img src="http://someotherwebsite/loggedin.png" onload="loggedIn()"
onerror="notLoggedIn()">
```

If the user is logged in, then the image file loads successfully, which causes the executions of `loggedIn`. If the user is not logged in, then `notLoggedIn` is executed. The result is an ability to test easily and invisibly whether a visitor is logged in to a particular Web site that a Web developer does not have a relationship with. This login-detection technique, which leverages CSRF, can be applied to online banks, social networks, Web mail, and basically anything else useful to an attacker. The attacker behind `http://coolwebsite/` just has to find the URLs that respond in a Boolean state with respect to login.

Next, consider that a malicious Web-site owner might want to go one step further and "deanonymize" a Web visitor, which is to say, learn the visitor's real name. Assume from the previous example that the attacker can determine if the visitor is logged into Twitter, Facebook, Google+, etc. Hundreds of millions of people are persistently logged in to these online services every day. These Web sites, and many like them, are designed that way for convenience.

The next thing an attacker could take advantage of is those familiar third-party Web widgets, such as Twitter's "Follow," Facebook's "Like," and Google's "+1" buttons.

While these buttons may seem innocent and safe enough, nothing really technically prevents Web sites from placing those buttons within an HTML container, such as a `div` tag, making those buttons transparent, and hovering them just under a Web visitor's mouse pointer. This is done so that when visitors click on something they see, they instead automatically Follow, Like, or +1 whatever else the bad guy wants them to. This is a classic case of clickjacking—an attack seen in the wild every day.

Here's why this flaw in the Internet matters: since the attacker controls the objects behind those buttons, after the user clicks, the attacker can tell exactly "who" just Followed, Liked, or +1'ed on those online services (e.g., Twitter: "*User X Followed you*." Facebook: "*User X Liked Page Y*."). To deanonymize the Web visitor, all the attacker needs to do is look at the public profile of the user who most recently clicked. That's when the fun begins for the attacker and trouble begins for the unsuspecting Internet user.

One more longstanding issue, "browser intranet hacking," deserves attention. This serious risk, first discussed in 2006, remains largely unaddressed to this day. Browser intranet hacking allows Web-site owners to access the private networks of their visitors, which are probably behind network firewalls, by using their browsers as a launch point. This attack technique is painfully simple and works equally well on enterprises and home users, exposing a whole new realm of data.

The attack flow is as follows: a Web user visits a malicious Web site such as `http://coolwebsite/`. That site instructs the visitor's browser to make a Web request to an IP address or host name that the visitor can get to but the attacker cannot, such as 192.168.x.x or any non-routable IP as defined by RFC-1918. Such requests can be forced through the use of IMG tags, as in the earlier example, or also through the use of `iframe`, `script`, and `link` tags:

```
<iframe src="http://192.168.1.1/" onload="detection()">.</iframe>
```

Depending on the detectable response given from the IP address, the attacker can use the Web visitor's browser to sweep internal private networks for listening IP Web servers. Locating printers, IP phones, broadband routers, firewalls, configuration dashboards, and more.

The technique behind browser intranet hacking is similar to the Boolean-state detection in the login-detection example. Also, depending on whether the user is logged in to the IP/Hostname, this type of attack can force the visitor's browser to make configuration changes to the broadband router's Web-based interface through well-known IPs (192.168.1.1, 10.10.0.1, etc.) that can be quickly enumerated. The consequences of this type of exploitation can be devastating as it can lead to all traffic being routed though the attacker's network first.

Beyond login detection, deanonymization, and browser intranet hacking are dozens of other attack techniques possible in today's modern browsers. For example, IP address geo-location tells, roughly speaking, what city/town a Web visitor is from. The user-agent header reveals which browser distribution and version the visitor is using. Various JavaScript DOM (Document Object Model) objects make it trivial to list what extensions and plugins are available—to hack or fingerprint. DOM objects also reveal screen dimensions, which provides demographic context and whether the user is using virtualization.

The list of all the ways browser security can be bent to a Web-site owner's will goes on, but the point is this: Web browsers are not "safe"; Web browsers are not "secure"; and the Internet has fundamental flaws impacting user (personal or corporate) security.

Now here's the punch line: the only known way of addressing this class of problem adequately is to "break the Web" (i.e., negatively impact the usability of a significant percentage of Web sites). These issues remain because Web developers, and to a large extent Web users, demand that certain functionality remain available, and that functionality is what makes these attacks possible.

Today's major browser vendors, whose guiding light is market share, are only too happy to comply. Their choice is simple: be less secure and more user-adopted, or be secure and obscure. This is the Web security trade-off—a choice made by those who do not fully understand or appreciate, or are not liable for, the risks they are imposing on everyone using the Web.

NONSTARTER SOLUTIONS

To fix login detection, a browser might decide not to send the Web visitor's cookie data to off-domain destinations (those different from the hostname in the URL bar) along with the Web requests. Cookies are essential to tracking login state. The off-domain destination could still get the request, but would not know to whom it belonged. This is a good thing for stopping the attack.

Not sending cookies off-domain, however, would break functionality for any Web site that uses multiple hostnames to deliver authenticated content. The approach would break single-click Web widgets such as Twitter's "Follow," Facebook's "Like," and Google's "+1" buttons. The user would be required to perform a second step. It would also break visitor tracking via Google Analytics, Coremetrics, and so on. This is a clear nonstarter from the perspective of many.

To fix clickjacking, Web browsers could ban `iframes` entirely, or at least ban transparent `iframes`. Ideally, browser users should be able to "see" what they are really clicking on. Suggesting such a change to `iframes`, however, is a losing battle; millions of Web sites rely upon them, including transparent `iframes`, for essential functionality. Notable examples are Facebook, Gmail, and Yahoo! Mail. You don't normally see `iframes` when they are used, but they are indeed everywhere. That level of breakage is never going to be tolerated.

For browser intranet hacking, Web browsers could prohibit the inclusion of RFC-1918 resources from non-RFC-1918 Web sites. This would essentially create a break point in the browser between public and private networks. One reason that browser vendors say this is not doable is that some organizations actually do legitimately include intranet content on public Web sites. Therefore, because some organizations (which you have never heard of and whose Web sites you'll never visit) have an odd use-case, your browser leaves the private networks you are on, and that of hundreds of millions of others, wide open.

As shocking as this sounds, try looking at the decision not to fix the problem from the browser vendors' perspective. If they break the uncommon use-case of these unnamed organizations, the people within those organizations are forced to switch to a competing "less-secure" browser that allows them to continue business as usual. While the security of all other users increases for the browser that makes the change, that browser vendor loses some fraction of market share.

SECURITY CHASM

The browser vendors' unwillingness to risk market share has led to the current security chasm. Dramatic improvements in browser security and online privacy are held hostage by backward compatibility requirements related to how the Internet was designed. Web-browser vendors compete with each other in trench-style warfare, gaining ground by scratching for a tiny percentage of new users, everyday—users who don't pay them a dime, while simultaneously trying to keep every last user they already have.

It's important to remember that mainstream browsers are essentially advertising platforms. The more eyeballs browsers have, the more ads are delivered. Ads, and ad clicks, are what pay for the whole party. Anything getting in the way of that is never a priority.

To be fair, there was one important win recently when, after years of discussion, a fix was applied to CSS history sniffing. This is the ability of a Web site to uncover the history of other Web sites a user had visited by creating hyperlinks on a Web page and using either JavaScript or CSS to check the color of the link displayed on the screen. A blue link meant the visitor had not been there; purple indicated the user had visited the site. This was a serious privacy flaw that was simple, effective, and 10,000-URLs-per-second fast to execute. Any Web site could quickly know where you banked, shopped, what news you read, adult Web sites frequented, etc.

The problem of CSS history sniffing finally got so bad and became so high profile that roughly 10 years after it first came up, all the major browser vendors finally broke the functionality required for the attack. Many Web developers who relied on the underlying functionality were vocally upset, but apparently this was an acceptable level of breakage from the browser vendors' perspective.

When the breakage is not acceptable, but the issue is still bad, new opt-in browser security features are put forth. They generally have low adoption rates. Prime examples are Content Security Policy, X-Frame-Options, Origin, Strict Transport Security, SSL (Secure Sockets Layer), Secure and HttpOnly cookie flags, etc. Web-site owners can implement these solutions only when or if they want to, thereby managing their own breakage. What none of these features do is to allow Web users to protect themselves, something every browser should enable its users to do. Right now, Web security is in a holding pattern—waiting for the bad guys to cause enough damage—which then should give enough juice to those with the power to take action.

BEYOND THE STATUS QUO

The path toward a more secure Web has a few options. We could establish a brand-new World Wide Web, or an area within it. A Web platform designed to be resilient to the current laundry list of problems, however, will forever plague its predecessor. For the moment, let's assume we technically know how to make a secure platform, which is a big *if*.

The next step would be to convince the developers behind the millions, potentially hundreds of millions, of important Web sites to move over and/or build atop version two. Of course, the promise of a "more secure" platform would not be sufficient incentive by itself. They would have to be offered something more attractive in addition. Even if there were something more attractive, this path would only exchange our backward-compatibility problem for a legacy problem, which is likely to take years, perhaps a decade or more, to get beyond.

There is another path—one that already has a demonstrated model of success in mobile applications. What you find there basically amount to many tiny Web browsers connected to the mobile version of the main Web site. The security benefit provided by mobile platforms such as Apple's iOS and Google's Android is that the applications are isolated from one another in both memory and session state.

For example, if you launched Bank of America's mobile application, logged in, did your banking, and then subsequently launched Facebook's mobile application and logged in, neither app has access to the other app's session, as would be the case in a normal desktop Web browser. Mobile applications have little to no issues regarding login detection, deanonymization, and intranet hacking. If mobile platforms can get away with this level of application and login-state isolation, certainly the desktop world could as well.

By adopting a similar application model on the desktop using custom-configured Web browsers (let's call them DesktopApps), we could address the Internet's inherent security flaws. These DesktopApps could be branded appropriately and designed to launch automatically to Bank of America's or Facebook's Web site, for example, and go no further. Like their mobile application cousins, these DesktopApps would not present an URL bar or anything else making them look like the Web browsers they are on the surface, and of course they would be isolated from one another. Within these DesktopApps, attacks such as XSS, CSRF, and clickjacking would become largely extinct because no cross-domain connections would be allowed—an essential precondition.

DesktopApps would also provide an important security benefit to Chrome, Firefox, Internet Explorer, and Safari. Attacks such as login detection and deanonymization would be severely hampered. Let's say Web visitor X uses only a special DesktopApp when accessing the Web sites of Bank of America, Facebook, or whatever else and never uses the default Web browser for any of these activities. When X is using Chrome, Firefox, or Internet Explorer and comes across a Web site trying to perform login detection and deanomymization, well, X has never logged in to anything important in that browser, so the attacks would fail.

What about intranet hacking? The answer is to break the functionality, as described earlier. Web browsers should not allow non-RFC-1918 Web sites to include RFC-1918 content—at least not without an SSL-style security exception. One or all of the incumbent browser vendors need to be convinced of this. If that mystery company with an odd use-case wants to continue, it should have a special corporate DesktopApp created that allows for it. It would be far more secure as a result, as would we all.

This article has outlined a broad path to fix Web security, but much is left unaddressed about how to roll out a DesktopApp and get the market to adopt such practices. Beyond just the security benefits, other features are needed to make DesktopApps attractive to Web visitors; otherwise there is no incentive for browser vendors to innovate. There's also lobbying to be done with Web-site owners and developers. All of this makes fixing the Internet a daunting task. To get past security and reach our final destination—a world where our information remains safe—we must develop creative solutions and make hard choices.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

JEREMIAH GROSSMAN
is the founder and CTO of WhiteHat Security, where he is responsible for Web security R&D and industry outreach. As a well-known security expert and industry veteran, he has written dozens of articles featured in prominent media outlets around the world and has been a guest speaker at many industry events and universities. He is a cofounder of WASC (Web Application Security Consortium) and was previously named one of *InfoWorld's* Top 25 CTOs. He serves on the advisory boards of two start-ups, Risk I/O and SD Elements, and is a Brazilian Jiu-Jitsu Black Belt. Before founding WhiteHat, he was an information security officer at Yahoo!

© 2012 ACM 1542-7730/11/1100 $10.00