DOI:10.1145/1506409.1506422

Article development led by acmqueue
queue.acm.org

What can be done to make Web browsers secure while preserving their usability?

BY THOMAS WADLOW AND VLAD GORELIK

Security in the Browser

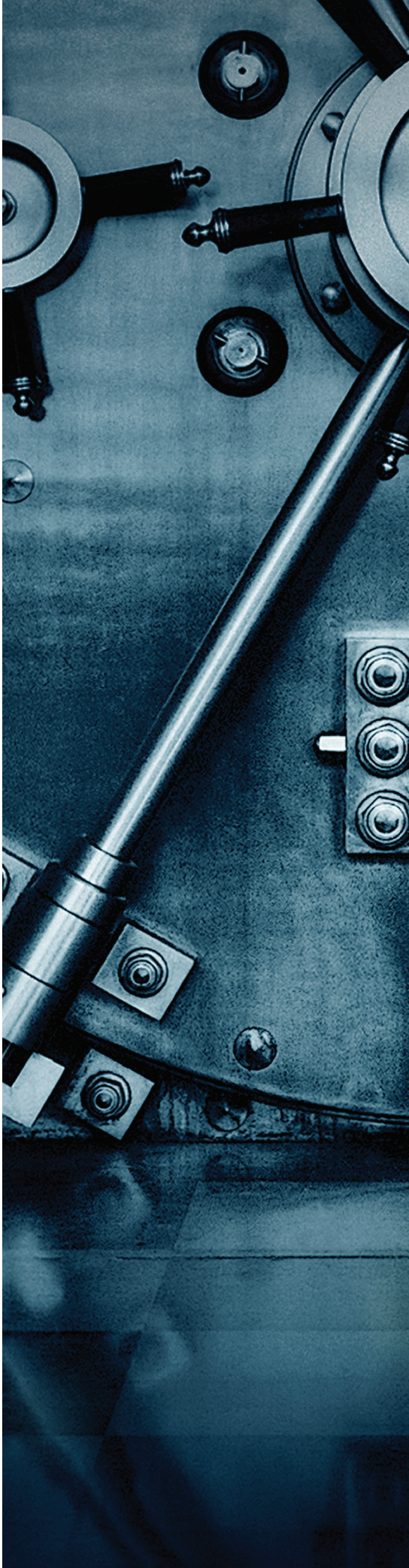


ILLUSTRATION BY JONATHAN BARKAT

"SEALED IN A depleted uranium sphere at the bottom of the ocean." That's the oft-quoted description of what it takes to make a computer reasonably secure. Obviously, in the Internet age or any other, such a machine would be fairly useless as well.

We live in interesting times. That computer on your desktop embodies the contradiction that faces a security engineer in the 21st century. It must be kept safe; and a lot of time, effort, and money is spent attempting to do exactly that. Firewalls are built to separate that machine from the Internet. Security audits tell us what programs must be deleted and what permissions changed so that the machine cannot be compromised. Virus checkers test all new software loaded on the machine for malicious content.

And yet, to make that fortress useful to us we demand that holes be chopped through the walls to permit us to run a Web browser. We complain if that browser is not given enough access to the rest of the computer. We insist on ease of use and speed, even if it makes all of our other defenses meaningless. And in many cases, we use browsers downloaded from the Internet without precaution, and configured by the owner of the desktop who has no security training or interest.

Browsers are at the heart of the Internet experience, and as such they are also at the heart of many of the security problems that plague users and developers alike.

The Use Model is Evolving...

Key features of early browsers included encryption and cookies, which were fine for the simple uses of the day. These techniques enabled the start of e-commerce, and monetizing the Web was what brought in the rest of the problems. Attackers who want money go where the money is, and there is money to be had on the Web.

Today, users expect far more from a browser. It should be able to handle sophisticated banking and shopping systems, display a wide variety of media, including video, audio, and animation, interact with the network on a micro scale (such as what happens when you move the cursor over a DVD selection in Netflix and see a summary of the movie), and update in as close to real time as possible—all without divulging sensitive information to bad guys or opening the door for attackers.

Consider AJAX, also known as Asynchronous JavaScript and XML. A Web page can contain code that establishes a network connection back to a server and conducts a conversation with that server that might bypass any number of security mechanisms integrated into the browser. The growing popularity of AJAX as a user-interface technique means an enterprise network often allows these connections, so that popular sites can function correctly.

The underlying mechanism of AJAX (which, despite the name, may not necessarily use JavaScript, XML, or be Asynchronous), is a function called XMLHttpRequest,[6] originally introduced by Microsoft for Internet Explorer, but now supported by Firefox, Safari, Opera and others. XMLHttpRequest allows a part of a Web page to make what is effectively a remote procedure call to a server across the Internet and use the results of that call in the context of the Web page. It is a powerful tool, but one that is open to a number of attacks.[2]

Flash, JavaScript, and Java all allow programs written by unknown third parties to run within the browser. Yes, there are sandboxes and safeguards, but as any attacker will tell you, a big step toward penetration is getting the target machine to run your code.

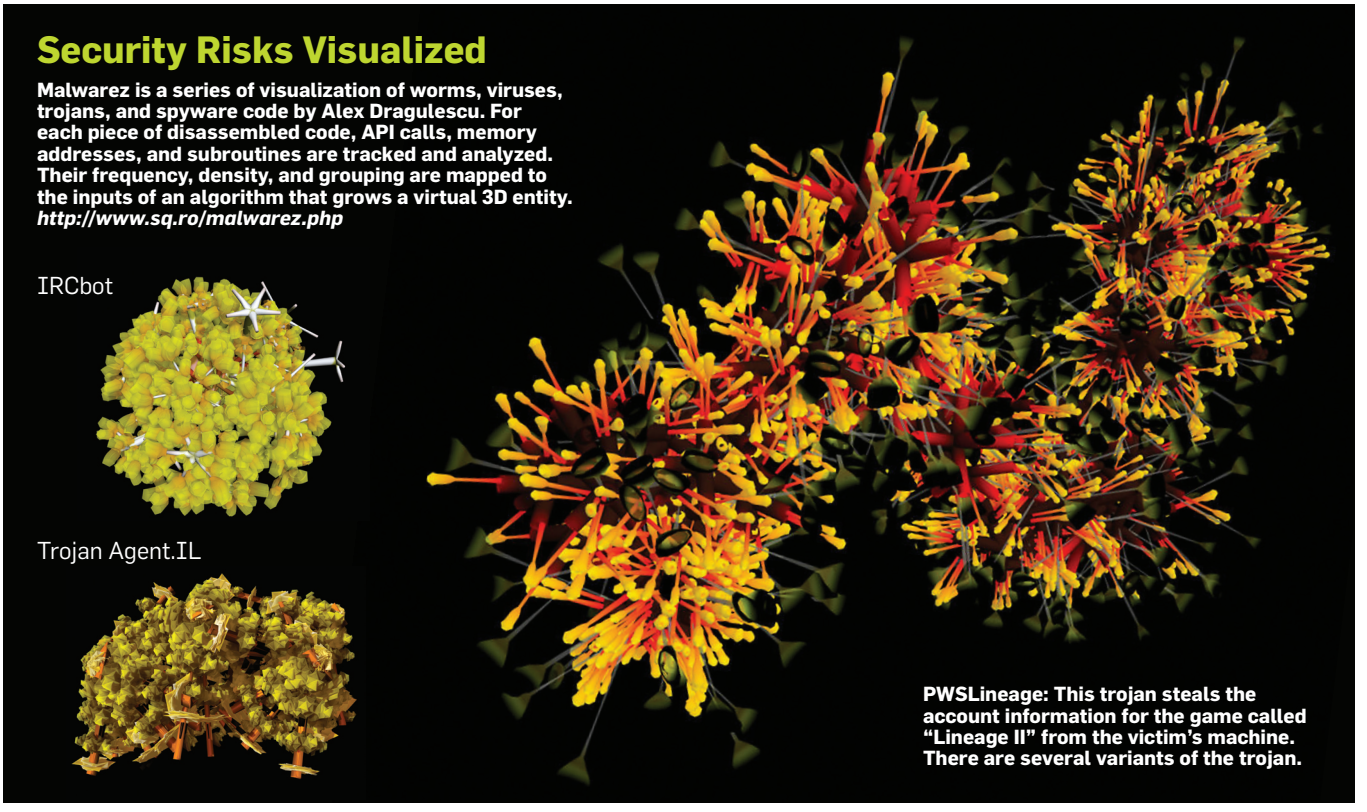
...And So Is The Threat Model

Early browsers had several major and noteworthy vulnerabilities, but they also had fewer types of attackers. The early attackers tended to be motivated by curiosity or scoring points with their peer groups. Modern browsers must defend against increasingly well-organized criminals who are looking for ways to turn browser vulnerabilities into money. They are aggressive, methodical, and willing to try a variety of attacks to see what works. And then there are those who work in gray areas, not quite violating the law, but pushing the envelope as much as possible to make a few dollars.

With more aggressive threats come more aggressive defenders. Security experts wanting to make names for themselves can release vulnerability information about browsers faster than browser developers may be prepared to react. While the roots of this type of disclosure

Security Risks Visualized

Malwarez is a series of visualization of worms, viruses, trojans, and spyware code by Alex Dragulescu. For each piece of disassembled code, API calls, memory addresses, and subroutines are tracked and analyzed. Their frequency, density, and grouping are mapped to the inputs of an algorithm that grows a virtual 3D entity. ***http://www.sq.ro/malwarez.php***



PWSLineage: This trojan steals the account information for the game called "Lineage II" from the victim's machine. There are several variants of the trojan.

are often driven by noble motives, the results can be devastating if they are not handled properly by all parties.

The flip side of early disclosure is the *zero-day exploit*. In this type of attack, an attacker learns of a flaw in a browser and moves to exploit it and profit from it before the security community has an opportunity to mount a defense.

Injection attacks (sometimes known as *cross-site scripting*, XSS) are when an attacker embeds commands or code in an otherwise legitimate Web request. This might include embedded SQL commands, stack-smashing attempts, in which data is crafted to exploit a programming vulnerability in the command interpreter, HTML injection, in which a post by a user (such as a comment in a blog) contains code intended to be executed by a viewer of that post.

Cross-site reference forgery (XSRF) is similar to XSS but it basically steals your cookie from another tab within your browser. This is relatively new, since tabbed browsing has only become popular in the last few years. It's an interesting demonstration of how a browser feature sometimes amplifies old problems. One of the reasons Google engineers implemented each tab in a separate process in Chrome was to avoid XSRF attacks.

A similarly named but different attack is the *cross-site request forgery*, in which, for example, the victim loads an HTML page that references an image whose `src` has been replaced by a call to another Web site, perhaps one that the victim has an account on. Variations of this attack include such things as mapping networks within the victim's enterprise for later use by another attack.

Add to this threats that are more social and less technical in nature—phishing,[5] for example, where a victim might receive a perfectly reasonable email message from a company that he does business with containing a link to a Web site that appears to be legitimate as well. He logs in, and the fake Web site snatches his username and password, which is then used for much less legitimate purposes than he would care for. A phishing scam depends much more on the gullibility of the user than the technology of the browser, but browsers often take much of the blame.

There are attacks of this nature based on the mistyping or misidentification of characters in a host name. A simple example of this would be that it is tricky to spot the difference between "google.com" and "googIe.com" (where the lowercase "L" has been replaced by an uppercase "I") in the sans-serif font so frequently used by browser URL entry fields. Expand that attack to Unicode and internationalization and you have something very painful and difficult to defend against.

The browser designer faces the Goldilocks problem. Either the porridge is too cold (not usable due to the demands of the security lockdown), or too hot (easy to abuse because not enough security measures are in place, or are too weak). Designing a configuration that is "just right" is nearly impossible because of evolving threats, uncovered bugs, and differing user tolerances for frustration.

Cookies are a long-used mechanism for storing information about a user or a session. They can be stolen, forged, poisoned, hijacked, or abused for denial-of-service attacks.[4] Yet, they remain an essential mechanism for many Web sites. Looking through the list of stored cookies on your browser can be very educational.

Similar to browser cookies are Flash Cookies. A regular HTTP cookie has a maximum size of 4KB and can usually be erased from a dialog box within the browser control panel. Flash Cookies, or Local Shared Objects (LSO)s are related to Adobe's Flash Player. They can store up to 100KB of data, have no expiration date, and normally cannot be deleted by the browser user, though some browser extensions are becoming available to assist with deleting them. Although Flash is run with a sandbox model, LSOs are stored on the user's disk and may be used in conjunction with other attacks.

In addition to Flash Cookies, the ActionScript language (how one writes a Flash application) supports XMLSockets that give Flash the ability to open network communication sessions. XMLSockets have some limitations—they aren't permitted to access ports lower than 1024 (where most system services reside), and they are allowed to connect only to the same subdomain where the originating Flash application resides. However, consider the case of a Flash game covertly run by an attacker. The attacker runs a high-numbered proxy on the same site, which can be accessed by XMLSockets from the victim's machine and redirected anywhere, for any purpose, bypassing XMLSocket limitations. This trick has already been used to unmask users who attempt to use anonymizing proxies to hide their identities.

Clickjacking is a relatively new attack, in which attackers present an apparently reasonable page, such as a Web game, but overlay on top of it a transparent page linked to another ser-

vice (such as the e-commerce interface for a store at which the victim has an account). By carefully positioning the buttons of the game, the attacker can cause the victim to perform actions from their store account without knowing that they've done so.

Security vs. Usability

Usability and security have long been at odds with each other in software design. The browser is no exception to that rule.

When browsing the Web or downloading files the user constantly needs to make choices about whether to trust a site or the content accessed from that site. Browser approaches to this have evolved over time—for example, browsers used to give a slight warning if you accessed a site with an invalid HTTPS certificate; now most browsers block sites with invalid certificates and make the user figure out how to unblock them. Similar approaches are taken with file downloads. Internet Explorer tends to ask the user several times before opening a downloaded file, especially if the file is not signed. Prompting the user for actions that are legitimate most of the time often creates user fatigue, which makes the user careless in walking the tightrope between software with a "reasonable but not excessive" security posture and a package that is either too open for safety or too closed to be useful. Most browsers today have evolved from the "make the user make the choice" model to the "block and require explicit override action" model.

In some cases the security of the browser has had a major impact on Web site design and usability. Browsers present a clear target for identity theft malware, since a lot of personal information flows through the browser at one time or another. This type of malware uses various techniques to steal users' credentials. One of these techniques is form grabbing—basically hooking the browser's internal code for sending form data to capture login information before it is encrypted by the SSL layer. Another technique is to log keyboard strokes to steal credentials when the user is typing information into a browser. These techniques have spawned various attempts by Web site designers to provide more advanced authentication methods, such as multifactor authentication with a hardware token and use of various click-based keyboards to avoid key loggers. In some cases some banks ask the user to authenticate each transaction with a hardware token. Although some of these techniques definitely improve security, they can place a pretty heavy burden on the end user.

Modern browsers must defend against increasingly well-organized criminals who are looking for ways to turn browser vulnerabilities into money. They are aggressive, methodical, and willing to try a variety of attacks to see what works.

Another usability feature of the Web browser that has been attacked by malware is the auto-complete functionality. Auto-complete saves the form information in a safe location and presents the user with options for what he typed before into a similar form. Several families of malware, such as the Goldun/Trojan Hearse, used this technique very effectively. The malware cracked the encrypted autocomplete data from the browser and send it back to the central server location without even having to wait for the user to log in to the site.

Given all the vulnerabilities out there and the willingness of attackers to exploit them, you might think that users would be clamoring for more security from their browsers. And some of them do...as long as it doesn't prevent any of their desired features from working.

Let's start with the browser software itself. From a security engineering perspective, the obvious choice for browser software (or any software) is to ship it in a "locked down" state, with all security features turned on, and let the user or enterprise weaken the security by enabling functions that they want. Consumer software that has done this has generally failed in the marketplace. Consumers want security, but they don't want to think about it or configure it. If the shipped configuration does what they want, they probably will not alter the configuration much, if at all.

So the browser designer faces the Goldilocks problem. Either the porridge is too cold (not usable because of the demands of the security lockdown) or too hot (too easy to abuse because not enough security measures are in place, or are too weak). Designing a configuration that is "just right" is nearly impossible because of evolving threats, uncovered bugs, and differing user tolerances for frustration.

There are a number of documents available that list steps one can take to lock down a Web browser. For example, one of those steps often is something like "Disable JavaScript." But few people actually ever do that—at least not

permanently, because using a browser with JavaScript turned off is annoying, and in many cases prevents you from visiting sites you have legitimate reasons to visit.

Cookies, while sometimes flushed to solve a problem, are essential to many Web sites, and having them disabled will prevent a wide range of services from working.

What is a Browser Designer To Do?

Browser developers have been working overtime to try and address some of these issues— and with some success— but it is definitely an uphill battle.

Proactive and reactive developers can generate an endless series of software updates. As a responsible defender, your dilemma is that allowing user these untested updates may break applications or even introduce security holes, but not allowing them may leave your enterprise open to even more serious attacks.

Distributed management provides some help in this area, but all major browsers are weaker than many defenders would like them to be. Microsoft provides the free Internet Explorer Administration Kit, which sets the bar for enterprise browser deployment and management tools, but that bar is lower than many would desire. FirefoxADM, an open source project for managing collections of Firefox browsers, is far more limited but a step in the right direction. FrontMotion provides a Web-based tool that allows a defender to create packages with approved software, configuration, and plugins for Firefox. All are available for the Windows platforms only.

Firefox and Google's Chrome browser have implemented "sandboxes," in which code run by the browser (such as JavaScript or Flash) is run in a compartmentalized area of the program that provides only limited resources for the program to run and whose design is heavily scrutinized for security flaws. Internet Explorer uses a zone-based security model, in which security features are enabled or disabled depending on what site is being accessed. Under Vista, it runs in what is known as Protected Mode, which limits the operating system privileges that the browser program can exercise.

However, open source developers must be especially careful about designing and implementing sandbox systems because their sandbox source code is available to the attacker for study and testing. This is, of course, no surprise to the sandbox developers and one reason why open source sandboxes tend to improve quickly.

Browser developers have come up with several ways to combat phishing attacks as well, primarily heuristics to detect an attempted visit to a fraudulent site, techniques to aggregate lists of and warn about known phishing sites, and augmentation of login security.

Injection attacks are most properly defended against at the server, but the victim will often be the browser user, not the server owner. Therefore, browsers may implement policies that hamper the injection attack by limiting where resources may be accessed from within a particular page.

Firefox has aggressively pursued a strategy of patching known vulnerabilities and generates updates regularly. Internet Explorer 7 is a significant improvement over Internet Explorer 6 in this regard, though many more known-but-unpatched vulnerabilities exist in IE 7 than in Firefox. Chrome seems to be emulating Firefox, though it lacks the mindshare of the other two at the moment so fewer eyeballs are looking critically at it for flaws.[3]

Some browser developers are employing and refining their system for detecting, reporting, and responding to security flaws. Mozilla.org, the support and development organization for Firefox, enlists open source developers to assist with code reviews and offers open bug tracking systems so that bugs can be reported and the follow-up tracked.

From a defender point-of-view, these efforts are a mixed blessing. Because browser software may be freely downloaded from the Internet by any user, all browsers are suspect. A prudent defender might hope that the browser is sufficiently rugged, but he cannot count on that fact. Desktop *nix systems and Mac OS X allow browser software to be run at a lower permission level than Windows often does, but that safeguard may be circumvented by other user-driven configuration changes.

Conclusion

From a network security perspective, a browser is essentially a somewhat controlled hole in your organization's firewall that leads to the heart of what it is you are trying to protect. While browser designers do try to limit what attackers can do from within a browser, much of the security relies far too heavily on the browser user, who often has other interests besides security. There are limits to what a browser developer can compensate for, and browser users will not always accept the constraints of security that a browser establishes.

As this issue gets more exposure, browser developers are cooperating to some degree to share strategies for defense. Google has published an excellent *Browser Security Handbook*[1] that compares various browser features and defenses.

Attack and defense strategies are coevolving, as are the use and threat models. As always, anybody can break into anything if they have sufficient skill, motivation and opportunity. The job of browser developers, network administrators, and browser users is to modulate those three quantities to minimize the number of successful attacks.

And that is a very big job indeed. C

Related articles on queue.acm.org

Criminal Code
Tom Wadlow and Vlad Gorelik
http://queue.acm.org/detail.cfm?id=1180192

Cybercrime: An Epidemic
Team Cymru
http://queue.acm.org/detail.cfm?id=1180190

Building Secure Web Applications
George Neville-Neil
http://queue.acm.org/detail.cfm?id=1281889

References

1. Google. *Browser Security Handbook*; http://code.google.com/p/browsersec/wiki/Main.
2. ISecPartners. *Attacking AJAX Applications*; http://www.isecpartners.com/files/iSEC-Attacking_AJAX_Applications.BH2006.pdf.
3. Wikipedia. Comparison of Web Browsers; http://en.wikipedia.org/wiki/Comparison_of_web_browsers.
4. Wikipedia. HTTP cookie; http://en.wikipedia.org/wiki/HTTP_cookie.
5. Wikipedia. Phishing; http://en.wikipedia.org/wiki/Phishing.
6. Wikipedia. XMLHttpRequest; http://en.wikipedia.org/wiki/XMLHttpRequest.

Thomas A. Wadlow is a network and computer security consultant, and the author of *The Process of Network Security*, Addison-Wesley Professional, 2000.

Vlad Gorelik is vice president of engineering at AVG Technologies where he heads up the development of behavioral malware detection and removal technologies. Previously he spent several years as CTO of Sana Security, leading the company's efforts in creating products to fight malware. He has multiple patents and filed patent applications in software technology and computer security.