acmqueue

Browser Security: Lessons from Google Chrome

Google Chrome developers focused on three key problems to shield the browser from attacks.

Charles Reis, Google; Adam Barth, UC Berkeley ; Carlos Pizano, Google

The Web has become one of the primary ways people interact with their computers, connecting people with a diverse landscape of content, services, and applications. Users can find new and interesting content on the Web easily, but this presents a security challenge: malicious Web-site operators can attack users through their Web browsers. Browsers face the challenge of keeping their users safe while providing a rich platform for Web applications.

Browsers are an appealing target for attackers because they have a large and complex trusted computing base with a wide network-visible interface. Historically, every browser at some point has contained a bug that let a malicious Web-site operator circumvent the browser's security policy and compromise the user's computer. Even after these vulnerabilities are patched, many users continue to run older, vulnerable versions.[5] When these users visit malicious Web sites, they run the risk of having their computers compromised.

Generally speaking, the danger posed to users comes from three factors, and browser vendors can help keep their users safe by addressing each of these factors:

- **The severity of vulnerabilities.** By sandboxing their rendering engine, browsers can reduce the severity of vulnerabilities. Sandboxes limit the damage that can be caused by an attacker who exploits a vulnerability in the rendering engine.
- **The window of vulnerability.** Browsers can reduce this window by improving the user experience for installing browser updates, thus minimizing the number of users running old versions that lack security patches.
- **The frequency of exposure.** By warning users before they visit known malicious sites, browsers can reduce the frequency with which users interact with malicious content.

Each of these mitigations, on its own, improves security. Taken together, the benefits multiply and help keep users safe on today's Web.

In this article, we discuss how our team used these techniques to improve security in Google Chrome. We hope our first-hand experience will shed light on key security issues relevant to all browser developers.

REDUCING VULNERABILITY SEVERITY

In an ideal world, all software, including browsers, would be bug-free and lack exploitable vulnerabilities. Unfortunately, every large piece of software contains bugs. Given this reality, we can hope to reduce the severity of vulnerabilities by isolating a browser's complex components and reducing their privileges.

Google Chrome incorporates several layers of defenses to protect the user from bugs, as shown in figure 1. Web content itself is run within a JavaScript virtual machine, which acts as one form of a sandbox and protects different Web sites from each other. We use exploit barriers, such as address-

space layout randomization, to make it more difficult to exploit vulnerabilities in the JavaScript sandbox. We then use a sandbox at the operating-system level to limit the process itself from causing damage, even if exploits escape the earlier security mechanisms. In this section, we discuss in more detail how these layers of defense are used.

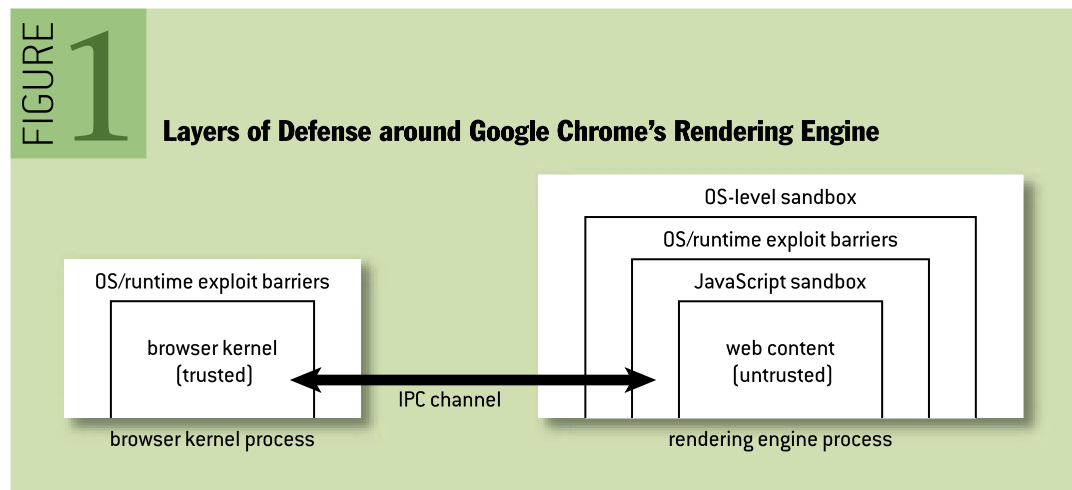


FIGURE 1 **Layers of Defense around Google Chrome's Rendering Engine**

SECURITY ARCHITECTURE

Google Chrome uses a modular architecture that places the complex rendering engine in a low-privilege sandbox, which we discuss in depth in a separate report.[1] Google Chrome has two major components that run in different operating-system processes: a high-privilege browser kernel and a low-privilege rendering engine. The browser kernel acts with the user's authority and is responsible for drawing the user interface, storing the cookie and history databases, and providing network access. The rendering engine acts on behalf of the Web principal and is not trusted to interact with the user's file system. The rendering engine parses HTML, executes JavaScript, decodes images, paints to an off-screen buffer, and performs other tasks necessary for rendering Web pages.

To mitigate vulnerabilities in the rendering engine, Google Chrome runs rendering-engine processes inside a restrictive operating-system-level sandbox (see figure 1). The sandbox aims to prevent the rendering engine from interacting with other processes and the user's operating system, except by exchanging messages with the browser kernel via an IPC channel. All HTTP traffic, rendered pages, and user input events are exchanged via such messages.

To prevent the rendering engine from interacting with the operating system directly, our Windows implementation of the sandbox runs with a restricted Windows security token, a separate and invisible Windows desktop, and a restricted Windows job object.[12] These security mechanisms block access to any files, devices, and other resources on the user's computer. Even if an attacker is able to exploit a vulnerability and run arbitrary code in the rendering engine, the sandbox will frustrate the attacker's attempts to install malware on the user's computer or to read sensitive files from the user's hard drive. The attacker's code could send messages to the browser kernel via the IPC channel, but we aim to keep this interface simple and restricted.

Getting existing code bases such as rendering engines to work fully within this type of sandbox sometimes presents engineering challenges. For example, the rendering engine typically loads

font files directly from the system's font directory, but our sandbox does not allow such file access. Fortunately, Windows maintains a system-wide memory cache of loaded fonts. We can thus load any desired fonts in the browser-kernel process, outside the sandbox, and the rendering-engine process is then able to access them from the cache.

There are a number of other techniques for sandboxing operating-system processes that we could have used in place of our current sandbox. For example, Internet Explorer 7 uses a "low rights" mode that aims to block unwanted writes to the file system.[4] Other techniques include system-call interposition (as seen recently in Xax[2]) or binary rewriting (as seen in Native Client[14]). Mac OS X has an operating system-provided sandbox, and Linux processes can be sandboxed using AppArmor and other techniques. For Windows, we chose our current sandbox because it is a mature technology that aims to provide both confidentiality and integrity for the user's resources. As we port Google Chrome to other platforms such as Mac and Linux, we expect to use a number of different sandboxing techniques but keep the same security architecture.

EXPLOIT MITIGATION

Google Chrome also makes vulnerabilities harder to exploit by using several barriers recommended for Windows programs.[8] These include DEP (data execution prevention), ASLR (address space layout randomization), SafeSEH (safe exception handlers), heap corruption detection, and stack overrun detection (GS). These are available in recent versions of Windows, and several browsers have adopted them to thwart exploits.

These barriers make it more difficult for attackers to jump to their desired malicious code when trying to exploit a vulnerability. For example, DEP uses hardware and operating-system support to mark memory pages as NX (non-executable). The CPU enforces this on each instruction that it fetches, generating a trap if the instruction belongs to an NX page. Stack pages can be marked as NX, which can prevent stack overflow attacks from running malicious instructions placed in the compromised stack region. DEP can be used for other areas such as heaps and the environment block as well.

Stack overrun detection (GS) is a compiler option that inserts a special canary value into each stack call between the current top of the stack and the last return address. Before each return instruction, the compiler inserts a check for the correct canary value. Since many stack-overflow attacks attempt to overwrite the return address, they also likely overwrite the canary value. The attacker cannot easily guess the canary value, so the inserted check will usually catch the attack and terminate the process.

Sophisticated attacks may try to bypass DEP and GS barriers using known values at predictable addresses in the memory space of all processes. ASLR, which is available in Windows Vista and Windows 7, combats this by randomizing the location of key system components that are mapped into nearly every process.

When used properly, these mechanisms can help prevent attackers from running arbitrary code, even if they can exploit vulnerabilities. We recommend that all browsers (and, in fact, all programs) adopt these mitigations because they can be applied without major architectural changes.

COMPATIBILITY CHALLENGES

One of the major challenges for implementing a security architecture with defense in-depth is

maintaining compatibility with existing Web content. People are unlikely to use a browser that is incompatible with their favorite Web sites, negating whatever security benefit might have been obtained by breaking compatibility. For example, Google Chrome must support plug-ins such as Flash Player and Silverlight so users can visit popular Web sites such as YouTube. These plug-ins are not designed to run in a sandbox, however, and they expect direct access to the underlying operating system. This allows them to implement features such as full-screen video chat with access to the entire screen, the user's webcam, and microphone. Google Chrome does not currently run these plug-ins in a sandbox, instead relying on their respective vendors to maintain their own security.

Compatibility challenges also exist for using the browser's architecture to enforce the *same-origin policy*, which isolates Web sites from each other. Google Chrome generally places pages from different Web sites into different rendering-engine processes,[11] but it can be difficult to do this in all cases, as is necessary for security. For example, some frames may need to be rendered in different processes from their parent page, and some JavaScript calls need to be made between pages from different origins. For now, Google Chrome sometimes places pages from different origins in the same process. Also, each rendering-engine process has access to all of the user's cookies, because a page from one origin can request images, scripts, and other objects from different origins, each of which may have associated cookies. As a result, we do not yet rely on Google Chrome's architecture to enforce the same-origin policy.

Recently, some researchers have experimented with browsers (such as OP[7] and Gazelle[13]) that do attempt to enforce the same-origin policy by separating different origins into different processes and mediating their interaction. This is an exciting area of research, but challenges remain that need to be overcome before these designs are sufficiently compatible with the Web. For example, supporting existing plug-ins and communication between pages is not always straightforward in these proposals. As these isolation techniques improve, all browsers will benefit.

REDUCING THE WINDOW OF VULNERABILITY

Even after we have reduced the severity of vulnerabilities, an exploit can still cause users harm. For example, a bug might let a malicious Web-site operator circumvent the same-origin policy and read information from other Web sites (such as e-mail). To reduce the danger to users, Google Chrome aims to minimize the length of time that users run unpatched versions of the browser. We pursue this goal by automating our quality-assurance process and updating users with minimal disruption to their experience.

AUTOMATED TESTING

After a vulnerability is discovered, the Google Chrome team goes through a three-step process before shipping a security patch to users:

1. The on-duty security sheriff triages the severity of the vulnerability and assigns an engineer to resolve the issue.
2. The engineer diagnoses the root cause of the vulnerability and writes a patch to fix the bug. Often security patches are as simple as adding a missing bounds check, but other patches can require more extensive surgery.

3. The patched binary goes through a quality assurance process to ensure that (a) the issue is actually fixed; and (b) the patch has not broken other functionality.

For a software system as complex as a Web browser, step 3 is often a bottleneck in responding to security issues, because testing for regressions requires ensuring that every browser feature is functioning properly.

The Google Chrome team has put significant effort into automating step 3 as much as possible. The team has inherited more than 10,000 tests from the WebKit project that ensure the Web platform features are working properly. These tests, along with thousands of other tests for browser-level features, are run after every change to the browser's source code.

In addition to these regression tests, browser builds are tested on 1 million Web sites in a virtual-machine farm called ChromeBot. ChromeBot monitors the rendering of these sites for memory errors, crashes, and hangs. Running a browser build through ChromeBot often exposes subtle race conditions and other low-probability events before shipping the build to users.

SECURITY UPDATES

Once a build has been qualified for shipping to users, the team is still faced with the challenge of updating users of older versions. In addition to the technical challenge of shipping updated bits to every user, the major challenge in an effective update process is the end-user experience. If the update process is too disruptive, users will defer installing updates and continue to use insecure versions.[5]

Google Chrome uses a recently open-sourced system called Omaha to distribute updates.[6] Omaha automatically checks for software updates every five hours. When a new update is available, a fraction of clients are told about it, based on a probability set by the team. This probability lets the team verify the quality of the release before informing all clients. When a client is informed of an update, it downloads and installs the updated binary in a parallel directory to the current binary. The next time the user runs the browser, the older version defers to the newer version.

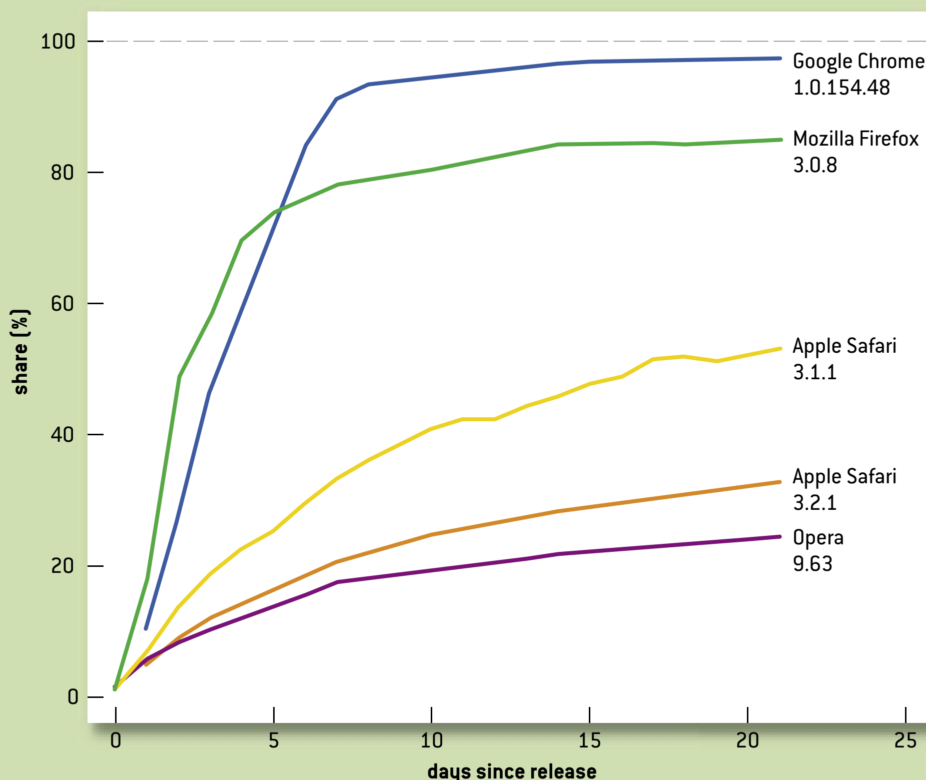
This update process is similar to that for Web applications. The user's experience is never disrupted, and the user never has to wait for a progress bar before using the browser. In practice, this approach has proven effective for keeping users up to date. A recent study of HTTP User-Agent headers in Google's anonymized logs reveals how quickly users adopt patched versions of various browsers.[3] We reproduce their results in figure 2. In these measurements, Google Chrome's auto-update mechanism updates the vast majority of its users in the shortest amount of time, as compared with other browsers. (Internet Explorer is not included in these results because its minor version numbers are not reported in the User-Agent header.)

REDUCING FREQUENCY OF EXPOSURE

Even with a hardened security architecture and a small window of vulnerability, users face risks from malicious Web-site operators. In some cases, the browser discourages users from visiting known malicious Web sites by warning users before rendering malicious content. Google Chrome and other browsers have taken this approach, displaying warning pages if a user tries to visit content that has been reported to contain malware or phishing attempts. Google works with StopBadware.org to maintain an up-to-date database of such sites, which can be used by all browsers.

One challenge with using such a database is protecting privacy. Users do not want every URL they

FIGURE 2 **Auto-Update Mechanisms in Google Chrome**



Credit: Thomas Duebendorfer and Stefan Frei.[3]

visit reported to a centralized service. Instead, the browser periodically downloads an efficient list of URL hashes without querying the service directly. To reduce the space required, only 32-bit prefixes of the 256-bit URL hashes are downloaded. This list is compared against a list of malicious sites. If a match is found for a prefix, the browser queries the service for the full 256-bit hashes for that prefix to perform a full comparison.

Another challenge is minimizing false positives. Google and StopBadware.org have tools to help publishers remove their pages from the database if they have been cleaned after hosting malware. It is also possible for human errors to flag sites incorrectly, as in an incident in January 2009 that flagged all URLs as dangerous.[9] Such errors are typically fixed quickly, though, and safeguards can be added to prevent them from recurring.

These services also have false negatives, because not every malicious page on the Web can be cataloged at every point in time. Although Google and StopBadware.org attempt to identify as many malicious pages as possible,[10] it is unlikely to be a complete list. Still, these blacklists help protect users from attack.

CONCLUSION

There is no silver bullet for providing a perfectly secure browser, but there are several techniques

that browser developers can use to help protect users. Each of these techniques has its own set of challenges.

In particular, browsers should minimize the danger that users face using three techniques:

- Reduce attack severity by applying the principle of least privilege in the browser architecture. This technique limits the damage caused when an attacker exploits a vulnerability.
- Reduce the window of vulnerability by ensuring updates are developed and deployed as quickly as possible. This technique minimizes the number of vulnerable browsers an attacker can target.
- Reduce how often users are exposed to attacks by filtering out known malicious content. This technique protects users during vulnerable time windows.

The Google Chrome team has focused on each of these factors to help provide a secure browser while preserving compatibility with existing Web content. To make Google Chrome even more secure, we are investigating further improvements to the browser's security architecture, such as mitigating the damage that plug-in exploits can cause and more thoroughly isolating different Web sites using separate sandboxed processes. Ultimately, our goal is to raise the bar high enough to deter attackers from targeting the browser. Q

REFERENCES

1. Barth, A., Jackson, C., Reis, C., and Google Chrome Team. 2008. The Security Architecture of the Chromium Browser; http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.
2. Douceur, J.R., Elson, J., Howell, J., Lorch, J.R. 2008. Leveraging legacy code to deploy desktop applications on the Web. In *Proceedings of Operating Systems Design and Implementation (OSDI)*.
3. Duebendorfer, T., Frei, S. 2009. Why silent updates boost security. ETH Tech Report TIK 302; http://www.techzoom.net/silent-updates.
4. Franco, R. 2005. Clarifying low-rights IE. *IEBlog* (June); http://blogs.msdn.com/ie/archive/2005/06/09/427410.aspx.
5. Frei, S., Duebendorfer, T., Plattner, B. 2009. Firefox (in)security update dynamics exposed. *ACM SIGCOMM Computer Communication Review* 39(1).
6. Google. Omaha: Software installer and auto-updater for Windows. Google Code; http://code.google.com/p/omaha/.
7. Grier, C., Tang, S., King, S.T. 2008. Secure Web browsing with the OP Web browser. In *Proceedings of IEEE Symposium on Security and Privacy*.
8. Howard, M., Thomlinson, M. 2007. Windows Vista ISV Security; http://msdn.microsoft.com/en-us/library/bb430720.aspx.
9. Mayer, M. 2009."This site may harm your computer" on every search result. The Official Google Blog (January); http://googleblog.blogspot.com/2009/01/this-site-may-harm-your-computer-on.html.
10. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N. 2007. The ghost in the browser: analysis of Web-based malware. In *Proceedings of the First Usenix Workshop on Hot Topics in Botnets* (April).
11. Reis, C. Gribble, S.D. 2009. Isolating Web programs in modern browser architectures. In *Proceedings of European Conference on Computer Systems (Eurosys)* (April).

12. Sandbox. Chromium Developer Documentation. 2008; http://dev.chromium.org/developers/design-documents/sandbox.
13. Wang, H. J., Grier, C., Moshchuk, A.,King, S. T., Choudhury, P.,Venter, H. 2009. The Multi-Principal OS Construction of the Gazelle Web Browser. Microsoft Research Technical Report (MSR-TR-2009-16); http://research.microsoft.com/pubs/79655/gazelle.pdf.
14. Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narul, N., Fullagar, N. 2009. Native Client: a sandbox for portable, untrusted x86 native code. In *Proceedings of IEEE Symposium on Security and Privacy.*

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

CHARLES REIS is a software engineer at Google working on the Google Chrome Web browser. He recently completed his Ph.D. in the Department of Computer Science and Engineering at the University of Washington. His research focuses on improving the reliability and security of Web browsers and Web content. He received B.A. and M.S. degrees in computer science from Rice University. At Rice, he was the second lead developer for DrJava, a widely used educational programming environment.

ADAM BARTH is a postdoctoral fellow at the University of California, Berkeley. His research focuses on the security of modern Web browsers, including their security policies, enforcement mechanisms, and security user interfaces. He is a contributor to the Chromium, WebKit, and Firefox open source projects and is an invited expert to the W3C HTML and Web Applications working groups. He holds a Ph.D. and M.S. in computer science from Stanford University and a B.A. in computer science and mathematics from Cornell University.

CARLOS PIZANO is a senior software engineer at Google working on the Google Chrome Web browser. He has an M.S. degree in computer engineering from the University of New Mexico and a B.S. in electrical engineering from Universidad Javeriana. His work focuses on security and sandboxing for Internet-facing applications.

© 2009 ACM 1542-7730/09/0600 $10.00