SECURITY

The 10-Kilobyte Web Browser

JON HOWELL, BRYAN PARNO, AND JOHN R. DOUCEUR

Jon Howell is re-envisioning the delivery of client applications, building a large scale, consistent distributed name service, building formally verified software, and improving verifiable computation. howell@microsoft.com

Bryan Parno focuses on protocols for verifiable computation and zero-knowledge proofs, building practical, formally verified secure systems, and developing next-generation application models. parno@microsoft.com

John R. Douceur is interested in the design of distributed algorithms, data structures, and protocols, and in the measurement, evaluation, and analytical modeling of systems and networks, with particular focus on statistical analysis and simulation. douceur@microsoft.com

In theory, browsing the Web is safe: click a link, and if you don't like what you see, click "close" and it disappears forever. In practice, this guarantee doesn't hold, because the browser is complex in both implementation and specification. We designed and built an alternate Web app delivery model in which the client-side interface specification and code—the pieces that replace the browser—are extremely simple, yet can run applications even richer than today's JavaScript apps. This article describes how we achieve this goal, and suggests a path forward into a future free of today's bloated browser interface.

Today a Web browser is a 100 MB operating system. Most of those megabytes interpret JavaScript and render images, but the browser's most important job is to provide the user with the ability to visit different Web sites safely, confident that merely viewing one Web site won't have any effect on any of the other sites she uses and relies on. Reliable isolation is best achieved in a simple design. The ideal Web browser would be a VNC viewer: each site renders its own content entirely independently, and the only job of the client machine is to show the various pixels to the user.

Of course, real browsers don't have such a simple specification. They're vastly more complicated, including HTML, DOM, CSS, JavaScript, JPG, PNG, and a complex specification for how various applications might interact with one another. This complexity forms a vulnerable surface, and hence real Web browsers *don't* actually succeed in isolating different pages; users are cautioned to avoid "dangerous" links lest their browser be compromised.

This ideal VNC pixel browser may seem absurd at first, but clearly it gets isolation right. You might complain that the performance stinks because it depends on a fast, available network, but we can fix that by allowing each site vendor to borrow a little virtual machine on the client; think of it as a *pico-datacenter*. That VM is strongly isolated from the other sites' VMs, just as customers in a real datacenter, say of a cloud-hosting provider, are isolated from one another.

In this new model (Figure 1), the specification of the browser is tiny and robust. Without a simple, clear specification, isolation is unachievable. With a clear specification, like this VM+VNC analogy, seeing how isolation can be rigorously maintained is easy; we push all the challenges of deciding how sites should interact with one another to the sites themselves. Promiscuous sites can still share cookies or engage in risky, CSRF-prone behavior (e.g., hosting user-supplied content), but cautious sites (e.g., bank Web sites) now have the control to reject those complex interactions.

The proposal of a virtual machine for execution and VNC for displaying pixels gives an intuition for how simple the interface can be, but we can go even simpler. We propose a minimal client execution interface called a *picoprocess*. A picoprocess is native code running in a hardware address space. It can allocate memory and threads, use futexes to schedule threads, read a real-time clock, and set a real-time alarm. All communication—to remote servers or to neighboring processes—is via IP; thus, an attacker can't do anything more threatening on the client machine than it could do from a server. (An attacker might relay IP attacks through its presence on a client, but the client's IP packets enter the Internet outside

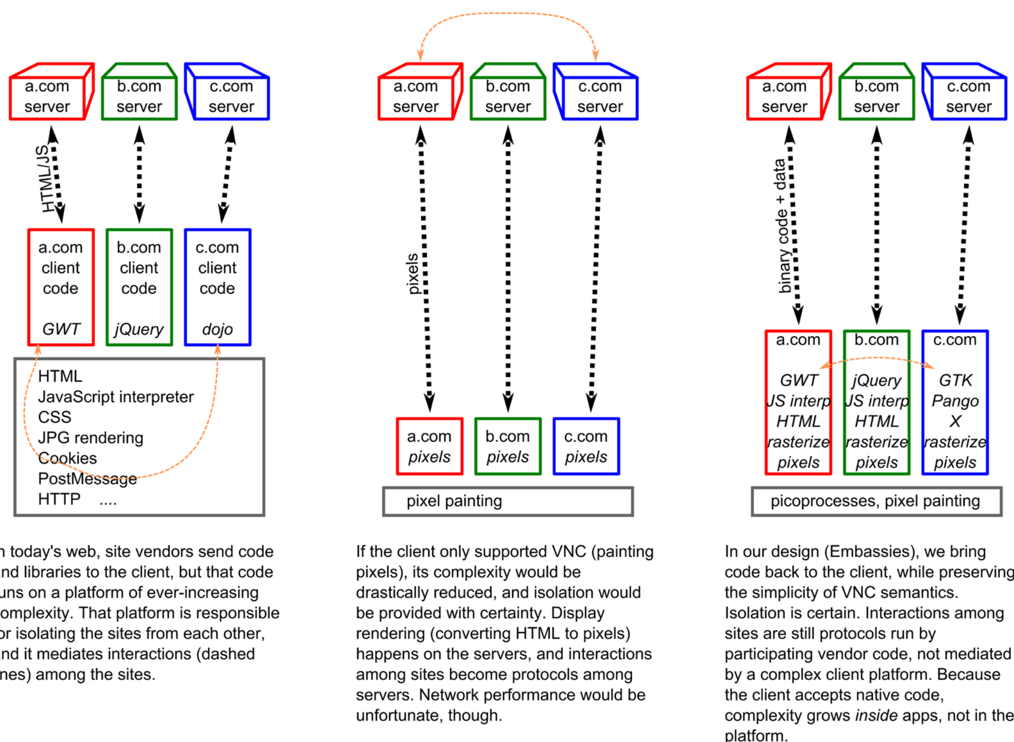

Figure 1: The current, the ideal, and a new way to browse the Web

any firewall, so the relay doesn't gain any privilege from the client's IP address or network position.) The client provides a source of randomness to enable the app to encrypt its messages over IP. Finally, the app displays its content by using its own libraries to render to an off-screen bitmap, then asking the client to paint a rectangle of pixels on the screen, semantics as simple as VNC.

This minimal interface replaces the role of the VM described above. Because it's even simpler than a conventional VM, the interface can be implemented easily on any host, from desktop OSes to native microkernels. On Linux, for example, the picoprocess is a Linux process, blocked from making Linux system calls by one of several mechanisms: kvm, ptrace, or filtering system calls down to `read` and `write` on a single open file handle to a monitor process.

Despite this tiny client picoprocess, the ability to run native code means the app itself can provide glorious complexity. The GIMP photo editor and the AbiWord word processor run in this container [2]. We also run a WebKit browser, to show how the trusted complexity of a conventional HTML browser can be repackaged as safely isolated rendering code.

This idea is ambitious: we're describing a substantial refactoring of the Web, shifting much responsibility from the browser (and the user) to the vendors that create the applications, so that visiting a site is no longer a risky proposition. But the ability to send binary code rather than JavaScript means the idea goes farther: it not only realizes the "safe click" promised by the Web, but it can bring those semantics to classic desktop applications, like the GIMP. When the plan is realized, your Webmail provider might be based on real Outlook and you might edit documents with MicrosoftWord.com or LibreOffice.org: solid desktop app code supported by its site rather than by the end user.

Our Embassies paper [1] proposes this application delivery in detail, discussing the tradeoffs consequent in shifting complexity from clients to applications. Our USENIX paper [2] shows how these complex apps can be repackaged to run inside the constrained picoprocess; source code is available [3].

How Do We Get There from Here?

The overall vision involves reconsidering several of our assumptions about the roles, responsibilities, and relationships that make up today's Web software ecosystem. Rather than end users selecting a JavaScript implementation ("download a fast new browser!"), site vendors will choose their client-side software stacks the same way they choose today among Python, Ruby, and PHP on the server side. Such an ambitious change may need to happen in small steps.

A key step on the way to Utopia is the shift from specifying client-side software in Web 2.0 (the complex amalgamation of JavaScript, DOM, CSS, and so on) to specifying it as native code that interacts through primitive low-level interfaces, such as painting raw pixels. It's an important step because it opens the door to shifting rendering components inside each application.

Figure 2: Three applications that currently run in an Embassies picoprocess

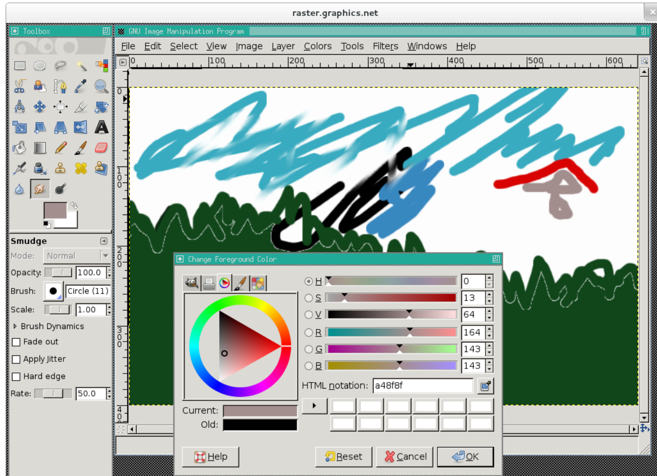

Figure 2a: GIMP in an Embassies picoprocess

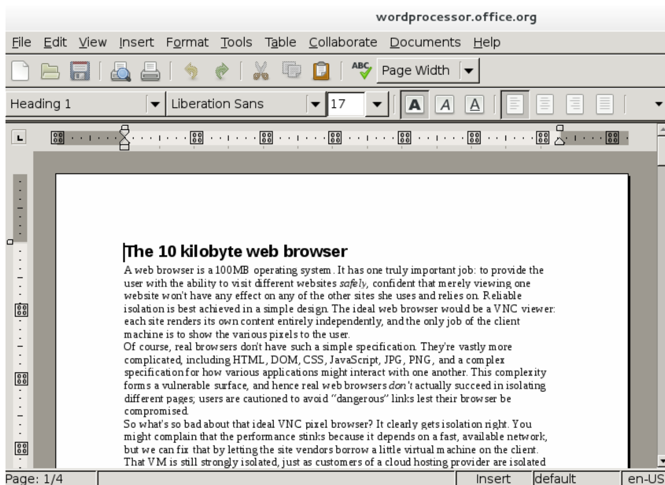

Figure 2b: AbiWord in an Embassies picoprocess

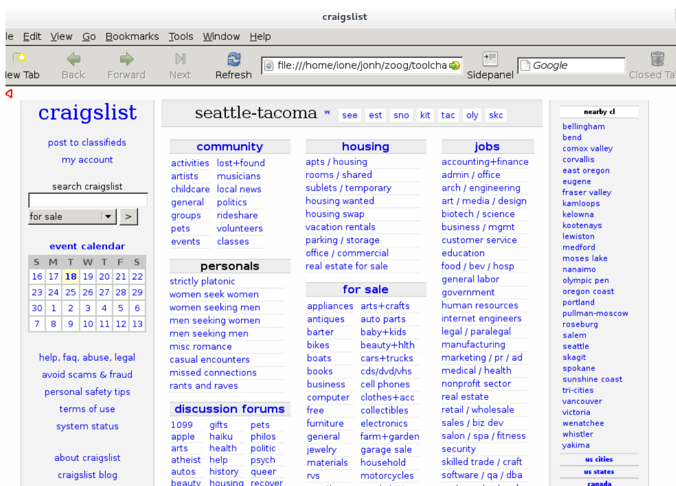

Figure 2c: WebKit in an Embassies picoprocess

But it's also a step that's compelling all by itself. The Xax [4] and Native Client [5] projects, both introduced in 2008, showed that delivering binary code to the client and executing it safely is feasible. Those systems were interesting enough to let us send down interesting components: Doom on NaCl, or PDF and OpenGL renderers on Xax.

Going beyond components to full applications exposes big opportunities. We can already package up GIMP and make it a Web app. We can do the same for the Gnumeric spreadsheet; add a bit of "cloud" and you have made an open-source alternative to Google Docs' spreadsheet. We can fit KDE Marble (a spinning globe) into a picoprocess; that is the foundation of a Google Earth alternative that doesn't require a trusted plugin. The opportunity to deliver rich apps is exciting in itself, even before we reach the ambitious goal of gutting the browser.

Challenges to Delivering Rich Apps

This goal is within reach. We have the technology; however, three tasks remain. First, we need to settle on a suitably shaped native code container. Second, we need to publish a picoprocess browser plugin. Third, we need to wrap up cool apps and publish them as Web apps.

How the Native Code Container Affects Deliverability

We said above that Xax delivered fairly modest stacks of libraries. Xax suffered from a practical burden: a high cost of modifying libraries and applications to run in the new environment. The Xax system replaced the ubiquitous glibc with a patched-together uclibc. In practice, that broke some libraries, and required linking others statically rather than dynamically. This approach worked only for short stacks of libraries. As we tried to enlarge the library stack, each new package required a new effort to disassemble its build system, and some software couldn't even conceive of being built as a static library. These are mundane concerns, but they proved a practical barrier to our ambitions of porting rich desktop apps.

NaCl has encountered similar challenges, for similar reasons. NaCl's isolation mechanism requires modifying the compiler's code generation step to produce code that NaCl can verify is safe. This requirement implies perturbing the build process (and often the link steps) of each package. We suspect that the NaCl team encountered a mundane but tedious and expensive burden much like the one that affected our Xax development.

So the choice of isolation container can have a profound effect on the ease of migrating apps to the new environment. NaCl's choice of verification based on software-fault-isolation (SFI) is driven by a desire to attach untrusted libraries onto the side of an existing browser, right inside the same process. For our ambitions, this objective is a red herring: even today's NaCl libraries don't need tight coupling with the browser; rich apps will stand

further alone; and, ultimately, we'd like to see the browser disappear entirely. Because it doesn't offer intra-process isolation, the picoprocess can exploit MMU protection, and hence provides a familiar execution environment for existing code.

Still, that decision wasn't enough to make porting easy in Xax. We made two fine-grained changes from Xax to Embassies that worked out well. The first was that, where Xax allowed the application to control its address space layout, Embassies only allows the app to ask for *how much* memory it needs, not where it goes. This actually increases the burden on the app—the executable must be position-independent—but it makes implementing the host much easier. In Xax, each new host added weird new address-space restrictions; in Embassies, this problem disappeared entirely.

More importantly, the main reason we couldn't use glibc or dynamic libraries in Xax was that we had no support for the x86 segment registers, used for thread-local storage (TLS). That meant we had to compile all components with --no-tls, and we couldn't find a way to use dynamic linking without TLS. The x86 segment-as-TLS is a goofy hack in any case; it uses deprecated hardware to compensate for the architecture's tiny register set. Because contemporary operating systems rely on paging for memory protection, this (ab)use of segmentation hardware has no security risk. By adding it to the Embassies picoprocess x86 specification, we're able to use standard glibc, conventional shared library linkage, and, hence, just about every package as is, with binary compatibility. (This whole discussion is moot on any other, sane architecture, where TLS just uses a conventional program register.)

The result—the Embassies specification for a native code container—is a spec to which a wide variety of rich apps can be ported with little effort. We've ported AbiWord (word processor), Gnumeric (spreadsheet), Gnucash (accounting), Midori (WebKit-based HTML renderer), GIMP (raster graphics design), Inkscape (vector graphics design), and Marble (3D globe). At the same time, the container is small, well-specified and secure, and practical to implement on any host platform.

A Browser Plugin

Now that we know what shape the container should be, achieving the initial step of delivering rich apps as Web apps is within reach: we need to implement the container as a plugin for the popular browsers, and test it for security.

Performance

We've described this new model using a strong analogy to the Web, to appeal to its "safe click" semantics. That doesn't mean we have to keep the Web's online requirement, or that we have to fetch our (now 100 MB) apps every time we open a site.

Whereas conventional browsers include caching behavior, Embassies apps control their own bootstrap and caching. An app can fetch its 100 MB of program image from any cache on the Internet and then check its hash to ensure they are the right bits. That cache can be an untrusted app on the same machine, obviating the need for network connectivity. The local cache can transmit the image in a single IPv6 jumbo frame, making app start fast; we see start time overheads of ~100 ms. Thus "sending big apps" is only an intuitive abstraction borrowed from today's Web; in deployed Embassies, it's fast and works when disconnected.

Once the app is running, native code enables performance better than JavaScript. The picoprocess's isolation comes from paging hardware, and hence introduces no overhead; CPU-intensive GIMP rotations are just as fast inside Embassies as on desktop GIMP.

Delivering Cool Apps

With an appropriate container available as a ubiquitous plugin, it's time to start packaging desktop apps as Web pages. Our ATC '13 paper [2] (and published code [3]) lays out how to achieve this packaging, showing it working for lots of apps, from a spreadsheet to an interactive 3D globe map. These apps need a little modification to make useful Web sites: for example, they need plumbing so that saving a document routes the content to client- or server-side Web storage.

The long-term vision is an exciting one: it promises finally to make browsing "safe," and broadens browsing to include both Web apps and desktop apps. Even if you don't yet buy that vision, the first step down the road is exciting all by itself: delivering all our favorite desktop apps as easily as clicking a link.

References

[1] Jon Howell, Bryan Parno, and John R. Douceur, "Embassies: Radically Refactoring the Web," USENIX Symposium on Networked Systems Design and Implementation (NSDI), awarded "Best Paper," April 2013.

[2] Jon Howell, Bryan Parno, and John R. Douceur, "How to Run POSIX Apps in a Minimal Picoprocess," USENIX Annual Technical Conference (ATC), June 2013.

[3] http://embassies.codeplex.com/.

[4] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch, "Leveraging Legacy Code to Deploy Desktop Applications on the Web," USENIX Symposium on Operating Systems Design and Implementation (OSDI), December 2008.

[5] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," IEEE Symposium on Security and Privacy, May 2009.