Mike Hearn Follow
Sep 23 · 12 min read

It's time to kill the web

Something is going on. The people are unhappy. The spectre of civil unrest stalks our programming communities.

For the first time, a meaningful number of developers are openly questioning the web platform. Here's a representative article and discussion. Here's another. Yet another. I could list more but if you're interested enough in programming to be reading this you've already read at least one hilarious rant this year about the state of modern web development. This is not one of those articles. I can't do a better job of mocking the status quo than the poor people who have to live it every day. This is a different kind of article.

That's you, frontend hacker

I want to think about how one might go about making a competitor to the web so good that it eventually replaces and subsumes it, at least for the purpose of writing apps. The web has issues as a way of distributing documents too, but not severe enough to worry about.

This is the first of two articles. In part one I'm going to review the deep, unfixable problems the web platform has: I want to convince you that nuking it from orbit is the only way to go. After all, you can't fix problems if you don't analyse them first. I'll also briefly examine why it is now politically acceptable to talk about these issues, although they are not actually new.

In part 2 I'll propose a new app platform that is buildable by a small group in a reasonable amount of time, and which (IMHO) should be much better than what we have today. Not everyone will agree with this last part, of course. Agreeing on problems is always easier than agreeing on solutions.

Part 1. Here we go.

. . .

Why the web must die

Web apps. What are they like, eh? I could list all kinds of problems with them but let's pick just two.

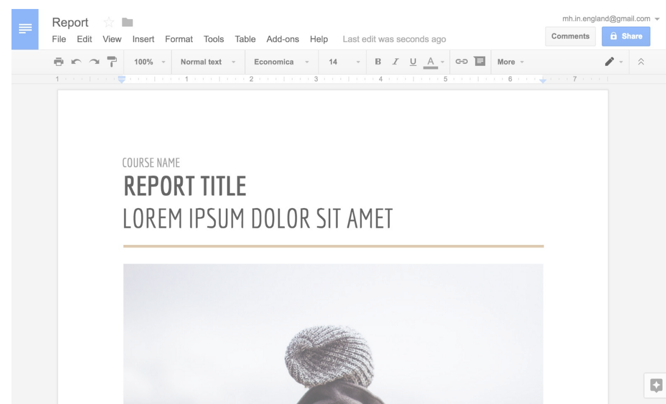
1. Web development is slowly reinventing the 1990's.
2. Web apps are impossible to secure.

Here's a good blog post on Flux, the latest hot web framework from Facebook. The author points out that Flux has recreated the programming model used by Windows 1.0, released in 1985. Microsoft used this model because it was appropriate for very slow computers, but it was awkward to develop for and so within less than a decade there was an ecosystem of products (like OWL) that abstracted away the underlying WndProc messaging system.

One of the reasons React/Flux works the way it does is that web rendering engines are very slow. This is true even though the end result the user actually sees is only a little bit fancier than what a Windows user might have seen 20 years ago:

Windows 98

Sure, the screen resolution is higher these days. The shade of grey we like has changed. But the UI you see above is rather similar in complexity to the UI you see below:



Even the icon fashions are the same! Windows 98 introduced a new trend of flat, greyscale icons with lots of padding to a platform where the previous style had been colourful, tightly packed pixel art.

But Office 2000 was happy with a 75 Mhz CPU and 32mb of RAM, whereas the Google Docs shown above is using a 2.5Ghz CPU and almost exactly 10x more RAM.

If we had gained a 10x increase in productivity or 10x in features perhaps that'd be excusable, but we haven't. Developer platforms in 1995 were expected to have all of the following, just as the price of entry:

- A visual UI designer with layout constraints and data binding.
- Sophisticated support for multi-language software components. You could mix statically typed native code and scripting languages.
- Output binaries so efficient they could run in a few megabytes of RAM.
- Support for graphing of data, theming, 3D graphics, socket programming, interactive debugging …

Many of these features have only been brought to the web platform in the past few years, and often in rather sketchy ways. Web apps can't use real sockets, so servers have to be changed to support "web sockets" instead. Things as basic as UI components are a disaster zone, there's no Web IDE worth talking about and as for mixing different programming languages … well, you can transpile to JavaScript. Sometimes.

One reason developers like writing web apps is that user expectations on the web are extremely low. Apps for Windows 95 were expected to have icons, drag and drop, undo, file associations, consistent keyboard shortcuts, do useful things in the background … and even *work offline*! But that was just basic apps. Really impressive software would be embeddable inside Office documents, or extend the Explorer, or allow itself to be extended with arbitrary plugins that were unknown to the original developer. Web apps usually do none of these things.

All this adds up. I feel a lot more productive when I'm writing desktop apps (even including the various "taxes" you are expected to pay, like making icons for your file types). I prefer using them too. And I know from discussions with others that I'm not alone in that.

I think the web is like this because whilst HTML as a document platform started out with some kind of coherent design philosophy and toolset, HTML as an app platform was bolted on later and never did. So basic things like file associations don't exist, yet HTML5 has peer to peer video streaming, because Google wanted to make Hangouts and Google's priorities dictate what gets added. To avoid this problem you need a platform that was designed with apps in mind from the start, and then maybe add documents on top, rather than the other way around.

. . .

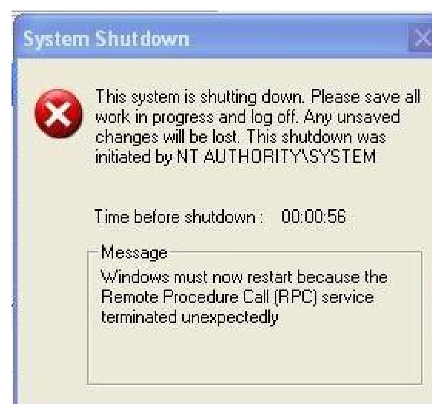
Web apps are impossible to secure

At the end of the 1990's a horrible realisation was dawning on the software industry: security bugs in C/C++ programs weren't rare one-off mistakes that could be addressed with ad-hoc processes. They were everywhere. People began to realise that if a piece of C/C++ was exposed to the internet, exploits *would* follow.

We can see how innocent the world was back then by reading the SANS report on Code Red from 2001:

> *"Representatives from Microsoft and United States security agencies held a press conference instructing users to download the patch available from Microsoft and indicated it as "a civic duty" to download this patch. CNN and other news outlets following the spread of Code Red urged users to patch their systems."*

Windows did have automatic updates, but if I recall correctly they were not switched on by default. The idea that software might change without the user's permission was something of a taboo.



First signs of a Blaster infection

The industry began to change, but only with lots of screaming and denial. Back then it was conventional wisdom amongst Linux and Mac users that this was somehow a problem specific to Microsoft … that *their* systems were built by a superior breed of programmer. So whilst Microsoft accepted that it faced an existential crisis and introduced the "secure development lifecycle" (a huge retraining and process program) its competitors did very little. Redmond added a firewall to Windows XP and introduced code signing certificates. Mobile code became restricted. As it became apparent that security bugs were bottomless "Patch Tuesday" was introduced. Clever hackers kept discovering that bug types once considered benign were nonetheless exploitable, and exploit mitigations once considered strong could be worked around. The Mac and Linux communities slowly woke up to the fact that they were not magically immune to viruses and exploits.

The final turning point came in 2008 when Google launched Chrome, a project notable for the fact that it had put huge effort into a complex but completely invisible renderer sandbox. In other words, the industry's best engineers were openly admitting *they could never write secure C++ no matter how hard they tried*. This belief and design has become a de-facto standard.

Now it's the web's turn

Unfortunately, the web has not led us to the promised land of trustworthy apps. Whilst web apps are kind of sandboxed

from the host OS, and that's good, the apps themselves are hardly more robust than Windows code was circa 2001. Instead of fixing our legacy problems for good the web just replaced one kind of buffer overflow with another. Where desktop apps have exploit categories like "double free", "stack smash", "use after free" etc, web apps fix those but then re-introduce their own very similar mistakes: SQL injection, XSS, XSRF, header injection, MIME confusion, and so on.

This leads to a simple thesis:

I put it to you that it's impossible to write secure web apps.

Let's get the pedantry out of the way. I'm not talking about literally all web apps. Yes you can make a secure HTML Hello World, good for you.

I'm talking about actual web apps of decent size, written under realistic conditions, and it's not a claim I make lightly. It's a belief I developed during my eight years at Google, where I watched the best and brightest web developers ship exploitable software again and again.

The Google security team is one of the world's best, perhaps *the* best, and they put together this helpful guide to some of the top mistakes people make as part of their internal training program. Here's their advice on securely sending data to the browser for display:

> *To fix, there are several changes you can make. Any one of these changes will prevent currently possible attacks, but if you add several layers of protection ("defense in depth") you protect against the possibility that you get one of the protections wrong and also against future browser vulnerabilities. First, use an XSRF token as discussed earlier to make sure that JSON results containing confidential data are only returned to your own pages. Second, your JSON response pages should only support* `POST` *requests, which prevents the script from being loaded via a script tag. Third, you should make sure that the script is not executable. The standard way of doing this is to append some non-executable prefix to it, like* `])}while(1);</x>` *. A script running in the same domain can read the contents of the response and strip out the prefix, but scripts running in other domains can't.*

> *NOTE: Making the script not executable is more subtle than it seems. It's possible that what makes a script executable may change in the future if new scripting features or languages are introduced. Some people suggest that you can protect the script by making it a comment by surrounding it with* `/*` *and* `*/` *, but that's not as simple as it might seem. (Hint: what if someone included* `*/` *in one of their snippets?)*

Reading this ridiculous pile of witchcraft and folklore always makes me laugh. It *should* be a joke, but it's actually basic stuff that every web developer at Google is expected to know, just to put some data on the screen.

Actually you can do all of that and it still doesn't work. The HEIST attack allows data to be stolen from a web app that implements even all the above mitigations and it doesn't require any mistakes. It exploits unfixable design flaws in the web platform itself. Game over.

Not really! It gets worse! Protecting REST/JSON endpoints is only one of many different security problems a modern web developer must understand. There are dozens more (here's an interesting example and another fun one).

My experience has been that attempting to hire a web developer that has even heard of all these landmines always ends in failure, let alone hiring one who can reliably avoid them. Hence my conclusion: if you can't hire web devs that understand how to write secure web apps then writing secure web apps is impossible.

The core problem

Virtually all security problems on the web come from just a few core design issues:

- Buffers that don't specify their length
- Protocols designed for documents not apps
- The same origin policy

Losing track of the size of your buffers is a classic source of vulnerabilities in C programs and the web has exactly the same problem: XSS and SQL injection exploits are all based on creating confusion about where a code buffer starts and a data buffer ends. The web is utterly dependent on textual protocols and formats, so buffers invariably must be parsed to discover their length. This opens up a universe of escaping, substitution and other issues that didn't need to exist.

The fix: All buffers should be length prefixed from database, to frontend server, to user interface. There should never be a need to scan something for magic characters to determine where it ends. Note that this requires binary protocols, formats and UI logic throughout the entire stack.

HTTP and HTML were designed for documents. When Egor Homakov was able to break Authy's 2-factor authentication product by simply typing "../sms" inside the SMS code input field, he succeeded because like all web services Authy is built on a stack designed for hypertext, not software. Path traversal is helpful if what you're accessing is an actual set of directories with HTML files in them, as Sir Tim intended. If you're presenting a programming API as "documents" then path traversal can be fatal.

REST was bad enough when it returned XML, but nowadays XML is unfashionable and instead the web uses JSON, a format so badly designed it actually has an entire section in its wiki page just about security issues.

The fix: Let's stop pretending REST is a good idea. REST is a bad idea that twists HTTP into something it's not, only to work around the limits of the browser, another tool twisted into being something it was never meant to be. This can only end in tears. Taking into account the previous point, client/server communication should be using binary protocols that are designed specifically for the RPC use case.

The same origin policy is another developer experience straight out of a Stephen King novel. Quoth the wiki:

> *The behavior of same-origin checks and related mechanisms is not well-defined in a number of corner cases … this*

> *historically caused a fair number of security problems.*

> *In addition, many legacy cross-domain operations predating JavaScript are not subjected to same-origin checks.*

> *Lastly, certain types of attacks, such as DNS rebinding or server-side proxies, permit the host name check to be partly subverted.*

The SOP is a result of Netscape bolting code onto a document format. It doesn't actually make any sense and you wouldn't design an app platform that way if you had more than 10 days to do it in. Still, we can't really blame them as Netscape was a startup working under intense time pressure, and as we already covered above, back then nobody was thinking much about security anyway. For a 10 day coding marathon it could have been worse.

Regardless of our sympathy it's the SOP that lies at the heart of the HEIST attack, and HEIST appears to break almost all real web apps in ways that probably can't be fixed, at least not without breaking backwards compatibility. That's one more reason writing secure web apps is impossible.

The fix: apps need a clear identity and shouldn't be sharing security tokens with each other by default. If you don't have permission to access a server you shouldn't be able to send it messages. Every platform except the web gets this right.

There are a bunch of other design problems in the web that make it hard to secure, but the above examples are hopefully enough to convince.

. . .

Conclusion

HTML 5 is a plague on our industry. Whilst it does a few things well those advantages can be easily matched by other app platforms, yet virtually none of the web's core design flaws can be fixed. This is why the web lost on mobile: when presented with competing platforms that were actually *designed* instead of organically grown, developers almost universally chose to go native. But we lack anything good outside of mobile. We desperately need a way of conveniently distributing sandboxed, secure, auto-updating apps to desktops and laptops.

Ten years ago I'd have been crucified for writing this article. I expect some grumbling now too, but in recent times it's become socially acceptable to criticise the web. Way back then, the web was locked in a competition with other proprietary platforms like Flash, Shockwave and Java. The web was open but it's survival as a competitive platform wasn't clear. Its eventual resurgence and victory is a story calculated to push all our emotional buttons: open is better than closed, collective ownership is better than proprietary, David can beat Goliath etc. Many programmers developed a tribal loyalty to it. Prefixing anything with "Web" made it instantly hip. Suggesting that Macromedia Flash might actually be good would get your geek card revoked.

But times change. The web has grown so fat that calling it open is now pretty meaningless: you have no chance of implementing HTML5 unless you have a few billion dollars you'd like to burn. The W3C didn't meet its users needs and is now irrelevant, so unless you work at Google or Microsoft you can't meaningfully impact the technical direction of the web. Some of the competing platforms that were once closed opened up. And the JavaScript ecosystem is imploding under the weight of its own pointless churn.

It's time to go back to the drawing board. Next time: how we can do that.