Javascript Cryptography Considered Harmful

What do you mean, "Javascript cryptography"?

We mean attempts to implement security features in browsers using cryptographic algoritms implemented in whole or in part in Javascript.

You may now be asking yourself, "What about Node.js? What about non-browser Javascript?". Non-browser Javascript cryptography is perilous, but not doomed. For the rest of this document, we're referring to browser Javascript when we discuss Javascript cryptography.

Why does browser cryptography matter?

The web hosts most of the world's new crypto functionality. A significant portion of that crypto has been implemented in Javascript, and is thus doomed. This is an issue worth discussing.

What are some examples of "doomed" browser cryptography?

You have a web application. People log in to it with usernames and passwords. You'd rather they didn't send their passwords in the clear, where attackers can capture them. You could use SSL/TLS to solve this problem, but that's expensive and complicated. So instead, you create a *challenge-response protocol*, where the application sends Javascript to user browsers that gets them to send *HMAC-SHA1(password, nonce)* to prove they know a password without ever transmitting the password.

Or, you have a different application, where users edit private notes stored on a server. You'd like to offer your users the feature of knowing that their notes can't be read by the server. So you generate an AES key for each note, send it to the user's browser to store locally, forget the key, and let the user wrap and unwrap their data.

What's wrong with these examples?

They will both fail to secure users.

Really? Why?

For several reasons, including the following:

- Secure delivery of Javascript to browsers is a chicken-egg problem.
- Browser Javascript is hostile to cryptography.
- The "view-source" transparency of Javascript is illusory.
- Until those problems are fixed, Javascript isn't a serious crypto research environment, and suffers for it.

What's the "chicken-egg problem" with delivering Javascript cryptography?

If you don't trust the network to deliver a password, or, worse, don't trust the server not to keep user secrets, you can't trust them to deliver security code. The same attacker who was sniffing passwords or reading diaries before you introduce crypto is simply hijacking crypto code after you do.

That attack sounds complicated! Surely, you're better off with crypto than without it?

There are three misconceptions embedded in that common objection, all of them grave.

First, although the "hijack the crypto code to steal secrets" attack sounds complicated, it is in fact simple. Any attacker who could swipe an unencrypted secret can, with almost total certainty, intercept and alter a web request. Intercepting requests does not require advanced computer science. Once an attacker controls the web requests, the work needed to fatally wound crypto code is trivial: the attacker need only inject another <SCRIPT> tag to steal secrets before they're encrypted.

Second, the difficulty of an attack is irrelevant. What's relevant is how tractable the attack is. Cryptography deals in problems that intractable even stipulating an attacker with as many advanced computers as there are atoms composing the planet we live on. On that scale, the difficulty of defeating a cryptosystem delivered over an insecure channel is indistinguishable from "so trivial as to be automatic". Further perspective: we live and work in an uncertain world in which any piece of software we rely on could be found vulnerable to new flaws at any time. But all those flaws require new R&D effort to discover. Relative to the difficulty of those attacks, against which the industry deploys hundreds of millions of dollars every year, the difficulties of breaking Javascript crypto remain imperceptibly different than "trivial".

Finally, the security value of a crypto measure that fails can easily fall below zero. The most obvious way that can happen is for impressive-sounding crypto terminology to convey a false sense of security. But there are worse ways; for instance, flaws in login crypto can allow attackers to log in without ever knowing a user's password, or can disclose one user's documents to another user.

Why can't I use TLS/SSL to deliver the Javascript crypto code?

You can. It's harder than it sounds, but you safely transmit Javascript crypto to a browser using SSL. The problem is, having established a secure channel with SSL, you no longer need Javascript

cryptography; you have "real" cryptography. Meanwhile, the Javascript crypto code is still imperiled by other browser problems.

What's hard about deploying Javascript over SSL/TLS?

You can't simply send a single Javascript file over SSL/TLS. You have to send *all the page content* over SSL/TLS. Otherwise, attackers will hijack the crypto code using the least-secure connection that builds the page.

How are browsers hostile to cryptography?

In a dispriting variety of ways, among them:

- The prevalence of content-controlled code.
- The malleability of the Javascript runtime.
- The lack of systems programming primitives needed to implement crypto.
- The crushing weight of the installed base of users.

Each of these issues creates security gaps that are fatal to secure crypto. Attackers will exploit them to defeat systems that should otherwise be secure. There may be no way to address them without fixing browsers.

What do you mean by "content-controlled code"? Why is it a problem?

We mean that pages are built from multiple requests, some of them conveying Javascript directly, and some of them influencing Javascript using DOM tag attributes (such as "onmouseover").

Ok, then I'll just serve a cryptographic digest of my code from the same server so the code can verify itself.

This won't work.

Content-controlled code means you can't reason about the security of a piece of Javascript without considering every other piece of content that built the page that hosted it. A crypto routine that is completely sound by itself can be utterly insecure hosted on a page with a single, invisible DOM attribute that backdoors routines that the crypto depends on.

This isn't an abstract problem. It's an instance of "Javascript injection", better known to web developers as "cross-site scripting". Virtually every popular web application ever deployed has fallen victim to this problem, and few researchers would take the other side of a bet that most will again in the future.

Worse still, browsers cache both content and Javascript aggressively; caching is vital to web performance. Javascript crypto can't control the caching behavior of the whole browser with specificity, and for most applications it's infeasible to entirely disable caching. This means that unless you can create a "clean-room" environment for your crypto code to run in, pulling in no resource tainted by any other site resource (from layout to UX) , you can't even know what version of the content you're looking at.

What's a "malleable runtime"? Why are they bad?

We mean you can change the way the environment works at runtime. And it's *not* bad; it's a fantastic property of a programming environment, particularly one used "in the small" like Javascript often is. But it's a real problem for crypto.

The problem with running crypto code in Javascript is that practically any function that the crypto depends on could be overridden silently by any piece of content used to build the hosting page. Crypto security could be undone early in the process (by generating bogus random numbers, or by tampering with constants and parameters used by algorithms), or later (by spiriting key material back to an attacker), or --- in the most likely scenario --- by bypassing the crypto entirely.

There is no reliable way for any piece of Javascript code to verify its execution environment. Javascript crypto code can't ask, "am I really dealing with a random number generator, or with some facsimile of one provided by an attacker?" And it certainly can't assert "nobody is allowed to do anything with this crypto secret except in ways that I, the author, approve of". These are two properties that often *are* provided in other environments that use crypto, and they're impossible in Javascript.

Well then, couldn't I write a simple browser extension that would allow Javascript to verify itself?

You could. It's harder than it sounds, because you'd have to verify the entire runtime, including anything the DOM could contribute to it, but it is theoretically possible. But why would you ever do that? If you can write a runtime verifier extension, you can also do your crypto in the extension, and it'll be far safer and better.

"But", you're about to say, "I want my crypto to be flexible! I only want the bare minimum functionality in the extension!" This is a bad thing to want, because ninety-nine and five-more-nines percent of the crypto needed by web applications would be entirely served by a simple, well-specified cryptosystem: PGP.

The PGP cryptosystem is approaching two decades of continuous study. Just as all programs evolve towards a point where they can read email, and all languages contain a poorly-specified and buggy implementation of Lisp, most crypto code is at heart an inferior version of PGP. PGP sounds complicated, but there is no reason a browser-engine implementation would need to be (for instance, the web doesn't need all the keyring management, the "web of trust", or the key servers). At the same time, much of what makes PGP seem unwieldy is actually defending against specific, dangerous attacks.

You want my *browser* to have my PGP key?

Definitely not. It'd be nice if your browser could generate, store, and use its own PGP keys though.

What systems programming functionality does Javascript lack?

Here's a starting point: a secure random number generator.

How big a deal is the random number generator?

Virtually all cryptography depends on secure random number generators (crypto people call them CSPRNGs). In most schemes, the crypto keys themselves come from a CSPRNG. If your PRNG isn't

CS, your scheme is no longer cryptographically secure; it is only as secure as the random number generator.

But how easy is it to attack an insecure random generator, really?

It's actually hard to say, because in real cryptosystems, bad RNGs are a "hair on fire" problem solved by providing a real RNG. Some RNG schemes are pencil-and-paper solveable; others are "crackable", like an old DES crypt(3) password. It depends on the degree of badness you're willing to accept. But: no SSL system would accept any degree of RNG badness.

But I can get random numbers over the Internet and use them for my crypto!

How can you do that without SSL? And if you have SSL, why do you need Javascript crypto? Just use the SSL.

I'll use RANDOM.ORG. They support SSL.

â€œJavascript Cryptography. It's so bad, youâ€™ll consider making async HTTPS requests to RANDOM.ORG simply to fetch random numbers."

Imagine a system that involved your browser encrypting something, but filing away a copy of the plaintext and the key material with an unrelated third party on the Internet just for safekeeping. That's what this solution amounts to. You can't outsource random number generation in a cryptosystem; doing so outsources the security of the system.

What else is the Javascript runtime lacking for crypto implementors?

Two big ones are secure erase (Javascript is usually garbage collected, so secrets are lurking in memory potentially long after they're needed) and functions with known timing characteristics. Real crypto libraries are carefully studied and vetted to eliminate data-dependant code paths --- ensuring that one similarly-sized bucket of bits takes as long to process as any other --- because without that vetting, attackers can extract crypto keys from timing.

But other languages have the same problem!

That's true. But what's your point? We're not saying Javascript is a bad language. We're saying it doesn't work for crypto inside a browser.

But people rely on crypto in languages like Ruby and Java today. Are they doomed, too?

Some of them are; crypto is perilous.

But many of them aren't, because they can deploy countermeasures that Javascript can't. For instance, a

web app developer can hook up a real CSPRNG from the operating system with an extension library, or call out to constant-time compare functions.

If Python was the standard browser content programming language, browser Python crypto would also be doomed.

What else is Javascript missing?

A secure keystore.

What's that?

A way to generate and store private keys that doesn't depend on an external trust anchor.

External what now?

It means, there's no way to store a key securely in Javascript that couldn't be expressed with the same fundamental degree of security by storing the key on someone else's server.

Wait, can't I generate a key and use it to secure things in HTML5 local storage? What's wrong with that?

That scheme is, at best, only as secure as the server that fed you the code you used to secure the key. You might as well just store the key on that server and ask for it later. For that matter, store your documents there, and keep the moving parts out of the browser.

These don't seem like earth-shattering problems. We're so close to having what we need in browsers, why not get to work on it?

Check back in 10 years when the majority of people aren't running browsers from 2008.

That's the same thing people say about web standards.

Compare downsides: using Arial as your typeface when you really wanted FF Meta, or coughing up a private key for a crypto operation.

We're not being entirely glib. Web standards advocates care about *graceful degradation*, the idea that a page should at least be legible even if the browser doesn't understand some advanced tag or CSS declaration.

"Graceful degradation" in cryptography would imply that the server could reliably identify which clients it could safely communicate with, and fall back to some acceptable substitute in cases where it couldn't. The former problem is unsolved even in the academic literature. The latter recalls the chicken-egg problem of web crypto: if you have an acceptable lowest-common-denominator solution, use that instead.

This is what you meant when you referred to the "crushing

burden of the installed base"?

Yes.

And when you said "view-source transparency was illusory"?

We meant that you can't just look at a Javascript file and know that it's secure, even in the vanishingly unlikely event that you were a skilled cryptographer, because of all the reasons we just cited.

Nobody verifies the software they download before they run it. How could this be worse?

Nobody installs hundreds of applications every day. Nobody re-installs each application every time they run it. But that's what people are doing, without even realizing it, with web apps.

This is a big deal: it means attackers have many hundreds of opportunities to break web app crypto, where they might only have one or two opportunities to break a native application.

But people give their credit cards to hundreds of random people insecurely.

An attacker can exploit a flaw in a web app across tens or hundreds of thousands of users at one stroke. They can't get a hundred thousand credit card numbers on the street.

You're just not going to give an inch on this, are you?

Nobody would accept any of the problems we're dredging up here in a real cryptosystem. If SSL/TLS or PGP had just a few of these problems, it would be front-page news in the trade press.

You said Javascript crypto isn't a serious research area.

It isn't.

How much research do we really need? We'll just use AES and SHA256. Nobody's talking about inventing new cryptosystems.

AES is to "secure cryptosystems" what uranium oxide pellets are to "a working nuclear reactor". Ever read the story of the radioactive boy scout? He bought an old clock with painted with radium and found a vial of radium paint inside. Using that and a strip of beryllium swiped from his high school chemistry lab, he built a radium gun that irradiated pitchblende. He was on his way to building a "working breeder reactor" before moon-suited EPA officials shut him down and turned his neighborhood into a Superfund site.

The risks in building cryptography directly out of AES and SHA routines are comparable. It is capital-H Hard to construct safe cryptosystems out of raw algorithms, which is why you generally want to use high-level constructs like PGP instead of low-level ones.

What about things like SJCL, the Stanford crypto library?

SJCL is great work, but you can't use it securely in a browser for all the reasons we've given in this document.

SJCL is also practically the only example of a trustworthy crypto library written in Javascript, and it's extremely young.

The authors of SJCL themselves say, "Unfortunately, this is not as great as in desktop applications because it is not feasible to completely protect against code injection, malicious servers and side-channel attacks." That last example is a killer: what they're really saying is, "we don't know enough about Javascript runtimes to know whether we can securely host cryptography on them". Again, that's painful-but-tolerable in a server-side application, where you can always call out to native code as a workaround. It's death to a browser.

Aren't you creating a self-fulfilling prophecy about Javascript crypto research?

People don't take Javascript crypto seriously because they can't get past things like "there's no secure way to key a cryptosystem" and "there's no reliably safe way to deliver the crypto code itself" and "there's practically no value to doing crypto in Javascript once you add SSL to the mix, which you have to do to deliver the code".

These may be real problems, but we're talking about making crypto available to everyone on the Internet. The rewards outweigh the risks!

DETROIT --- A man who became the subject of a book called "The Radioactive Boy Scout" after trying to build a nuclear reactor in a shed as a teenager has been charged with stealing 16 smoke detectors. Police say it was a possible effort to experiment with radioactive materials.

The world works the way it works, not the way we want it to work. It's one thing to point at the flaws that make it hard to do cryptography in Javascript and propose ways to solve them; it's quite a different thing to simply wish them away, which is exactly what you do when you deploy cryptography to end-users using their browser's Javascript runtime.