The apparently intractable problem of JavaScript cryptography

My own privacy app, PassLok, is written in JavaScript. As the researchers of Matasano security so aptly put it in this article, entitled: JavaScript Security Considered Harmful, there are serious problems with using JavaScript for crypto applications. To summarize:

1. JavaScript code is easy to modify before it reaches the computer, and even after. Since any modification to the code could introduce an undetectable backdoor, any kind cryptography would be insecure.
2. If securing JavaScript delivery requires the use of SSL, then why not just use SSL to communicate with a truly secure server?
3. JavaScript does not include any cryptography-grade functions, such as a random number generator. Lacking this, any operation using JavaScript would be less secure than anticipated.
4. The best crypto library for JavaScript is SJCL, and the SJCL authors admit that "it is not feasible to completely protect against code injection, malicious servers and side-channel attacks."

I am not going to try and refute all of these statements, which I hope I have summarized accurately. In this article, however, I will attempt to show that in some cases these problems are not as big as to stop the show and, in fact, may be smaller than in non-JavaScript crypto applications. The application I am going to refer to, of course, is going to be PassLok.

One thing should be understood before we even start. PassLok is a calculator. It takes input from the user, performs complex math on it, and puts out an answer. PassLok, by design, does not communicate outside the browser. It runs on a browser because I saw it convenient to use a browser, which every modern computer and smartphone has, in order to run the math code.

So let's look at the problems one by one:

Securing the code:

All code designed to run in a computer is liable to be tampered with before it reaches the computer, or even after. This is not a property of JavaScript code only. So the criticism hurled against JavaScript crypto code could be used just as well against any other code. Coders try to combat the tampering of newly delivered code several ways:

1. They hand-carry the code, to make sure no one messes with it. This does no longer seem feasible in the 21st century, especially if the code is large.
2. They generate a hash of the code (MD5, SHA1 or SHA256 are popular), which users can check before they actually use the code. The hash is published on a location different from the code source, so a hacker would have to hack two different servers in order to get away with tampering. This is very common because the cost is minimal. Unfortunately, since the step is optional most users tend to ignore it.

3. They can go a step further and generate a digital signature, certified by a trusted Certificate Authority (CA), which has to match the code. Verifying the signature is done automatically if the code is a browser extension or is installed from an app store such as Apple or Google Play.

PassLok can use methods 1 and 2, and hopefully 3 as well once native app versions are developed. The PassLok code at the time of this writing (version 1.7.01) is a little over 300kB in size, quite small by modern standards. It can be sent as an email attachment, carried in portable flashdrives, and even embedded inside pictures and media files. As genuine copies multiply, the ability of a hacker to get away with messing with the code decreases proportionally. Typically, a modern compiled app runs to several MB in size, and its ability to be replicated and multisourced decreases proportionally.

It is even easier to get a hash of a web app such as PassLok than it is to obtain it for a compiled program. Independent web-based hash engines are numerous, which can easily take the hash of a piece of text, which is what PassLok is. The problem is obtaining a representation of the source code that would be consistent enough to replicate the hash. This is obtained with the "view source" function included in most web browsers, which downloads a fresh copy of the code, rather than with the "save as" function, which saves the code as modified by the browser after loading. The tricky part is to ensure that the encoding is correct, for it affects the value of the hash.

Then user can then compare the computed hash with the published hash, but how does he/she know that the published hash has not been tampered with as well so it matches a tampered copy of the code? Well, this problem is not unique to JavaScript code, either, but PassLok has a unique solution to it. Along with the hash, a video of the author reading that hash is also published. Those who doubt the authenticity of the hash (and know the author's face and voice) only need to watch a one-minute video to be assured that the hash is authentic. Forging a video is admittedly much harder than changing a string of numbers.

Ah, but the code can be modified after it reaches the computer. What of it?

This would be true of most JavaScript code, but PassLok has no further communications with the outside world after it has loaded. Remember that it is a mere calculator at its core. An attacker would have to inject malicious code before PassLok loads, and have it waiting for the moment. This is a zero-day attack, which is another way to call an act of God. If the computer has been created evil, no precaution taken by PassLok or any other program, JavaScript or native for that matter, would be sufficient. Solution: if you suspect the computer might have been compromised, don't use it. Because PassLok is so portable, it makes it easier to search for greener pastures than most programs.

To SSL or not to SSL

That is the question, but let me backtrack a bit. Why trust SSL, either? Is SSL as trustworthy as we are led to believe? There have been reports of CA's issuing certificates to phony servers, and is widely believed that powerful actors like the NSA could manipulate SSL at will if they saw it necessary. What of the server that we are connected to by SSL? Can it be trusted?

Perhaps the weakest link in today's SSL chain is the remote server. Sure, we can connect to it without fear of intromission, but can we trust that server? How do we know it's not logging all our transactions in order to deliver them to an enemy? It's not like it wants to, it could be forced to it by a secret court order, and we wouldn't be the wiser.

I submit that the healthier thing to do in the current climate, is not to trust a remote server. Whatever you can do locally, do locally. Transmit no secrets, no matter how secure the transmission channel seems to be. If this is followed, then there is plenty of room for JavaScript programs that run locally, although programs that communicate with a server would remain suspect.

PassLok does make use of SSL for one thing, and that is delivering the code. Once this is done, the code is cached locally and only gets re-downloaded if it is updated on the server. SSL is necessary in order to improve the security of code delivery. The article I quoted above says: "You can't simply send a single Javascript file over SSL/TLS. You have to send all the page content over SSL/TLS". We'll, this is precisely how it's done with PassLok, so I guess from this perspective they'd have to agree that PassLok isn't delivered any less securely than a native program.

But you are ultimately trusting the server itself. There is no way around it but, you know what?, it is exactly the same problem with any other program, browser-based or native. With PassLok you choose who you want to trust among several sources, and you can still compare what you obtain from different sources. When PassLok becomes really popular there will be many mirrors, greatly improving the likelihood that you are downloading genuine code.

Cryptographic primitives

I am addressing issues 3 and 4 together, since PassLok is based on SJCL. Can SJCL be trusted, cryptographically speaking?

SJCL has been around for a couple years now and nobody has raised any alarm about anything being seriously wrong. So, short of waiting a few years for bad things to happen, this is as good as we can get. If a user is wary of using relatively new code (after it has been tested against test vectors, of course), he/she can always stay away and wait. But many people might want to jump in, and thus help to uncover unknown weaknesses. We'll all be better off in the long run because of it.