Multiparty Verified Code Execution On Modern Browsers

Pablo Guerrero – siriux@gmail.com – http://greenentropy.com – October 2014

Motivation and previous work

The idea of verifying the code that is executed on the browser has been discussed for a long time. It's important, because it could open the door to implementing secure client applications that doesn't need to trust any single server, while keeping the benefits of the modern web.

The problem is that javascript (JS) and the web are considered insecure environments to implement cryptography for different reasons. A good review on the potential security risks can be found at http://matasano.com/articles/javascript-cryptography/.

The typical way this problem is solved is by installing a browser extension that is able to verify signed pages before running them using the public key of a trusted source. This has the problem of installing the browser extension, which is inconvenient in the best case, or impossible in some systems or devices that are not under the user control (enterprise computers, mobile devices and tablets, public computers, ...).

A similar scenario are the browser specific apps, that are also signed and delivered by the browser creators.

An ideal solution would involve verification of the code using multiple independent servers to avoid a single point of trust. It's also important that it's simple to use by any user, and that it can run on any modern desktop and mobile browser without installing an extension. At the same time, it has to solve the potential risks of JS described in the previous link.

Simplifying the problem

Some of the criticism for JS comes from the malleability of the language and the lack of primitives needed for cryptography, but this is no longer true. All modern browsers (including IE11) have access to window.crypto.getRandomValues that is the most basic primitive needed for cryptography.

Also, there are well written and tested JS crypto libraries for the most common algorithms. And, if you don't trust them, it's possible to use trusted implementations written in C/C++ using Emscripten.

The malleability of the language problem can be reduced to a much simple problem. Right now it's a standard practice to render web pages directly on the client using only JS. This can be easily done in such a way that the data for rendering cannot secretly execute JS from the DOM side.

At the same time, standard technologies such as AJAX, CORS, data URLs, ... allow to load extra assets as plain text before they are used in the page. This can be used to verify the assets by simple inspection or using previously computed hashes that are directly stored on the JS code. This can be automated and standardized by libraries.

These methods allow us to reduce the problem to a much simple one. If you are able to execute some verified JS code on a clean environment (known JS and DOM state), using the previous standard techniques you can be sure that the JS and DOM will remain in a known state.

Assuming you can meet the first condition, there is no reason to believe that native applications are safer for cryptography than web applications.

First approximation

In order to solve the simplified problem, we propose to approach it from the perspective of multiparty verification. The idea is to have N independent servers, that are under the control of N independent entities, that help us with the verification process.

As perfect safety is impossible, we are going to assume some reasonable things. It's unlikely that all of the N servers are controlled by an attacker. Depending of your security needs, you can choose N and the parties controlling the servers that suits you. We also assume that the client is not controlled by an attacker and that the SSL certification authorities used by the browser are not compromised either.

The idea is to have the first server providing the client the JS and HTML code in a way that it's not automatically executed. Then, the client will provide this code to the N – 1 remaining servers for verification. The client must not trust any links to the verification servers provided by the first server, and has to enter the urls by himself (but it can be reminded of the other servers using their logos, for example). If all agree the code pass the verification, then, the client can safely execute it.

We can easily see that the security is not compromised even if N-1 servers are controlled by an attacker. If the first server is compromised and provides a modified code, at least one verification server will give us a warning. And if only the verification servers are compromised, they cannot do much harm either, because they can either tell us that everything is fine, which it's true because the main server was not compromised, or give us a warning, which is preventing us from trusting the code, but doesn't compromise the security.

The straight forward way of doing this is using a local file as intermediate storage for the code. The user downloads an HTML+JS file from the first server and then provides it to the verification web pages on the other servers. After verification, the user can execute the local file on the browser.

This approach works, but it has some problems. Not all browsers allow to execute local files (at least some mobile browsers), and when they do, they don't behave in a consistent way (although it might be enough in many cases). Also, some environments can disable

the execution of JS on local files.

Another problem is that the process involves the file system, and the user has to understand it and switch between the browser and file system contexts. It can be simplified using drag and drop on many browsers, but it's still far from optimal.

Solving the problems

Intermediate storage

As we have seen, using local files has multiple problems, so the idea is to use an alternative intermediate storage for the code before it is verified. The proposed solution is to make use of bookmarklets, bookmarks that contain JS code that will be executed on the current page when clicked.

The first server will provide us with a link containing the code, that the user has to drag and drop into his bookmarks (or using any other method).

Once the code is in the bookmarklet, the user goes to the first verification page, and drags from the bookmark into a text area. This allows the page to verify the code in the bookmarklet and give us a warning if it has been modified. This process is repeated for any additional verification pages.

This can be really easy with a well designed UI. Also, this works across desktop and mobile browsers (even if in some mobile browsers is not as easy, it's still possible with a small amount of work).

Now, the user has some verified code that can be executed using the bookmarklet. When used, this code will be executed on the HTML+JS context of the current page.

We can see that this doesn't have the problems of the local file, and with the right interface and instructions, it can be very simple and fast to perform. But we have introduced another problem. Now, we control the JS code, but we don't have control over the HTML and the JS environment where the bookmarklet is executed.

Executing the bookmarklet on a clean environment

A simple and viable solution is to instruct the user to go to a random web page before executing the bookmark. Then, the bookmarklet will take over the page removing any previous content. This can be reasonably secure if we assume that an attacker cannot control this page because it's really difficult predict. But in most cases this will not be enough.

Fortunately, we can use a behavior that it's available in all modern browsers, to get a clean environment to execute the code. The HTML error pages.

All modern browsers display errors using a standard HTML page, and as is generated locally on the browser it cannot be compromised by an external attacker. If we can find a way to systematically arrive to an error page, we have solved the problem.

The obvious way to generate an error page is using a wrong URL. For that, we could use two bookmarks instead of one, the first one would be a link to a wrong URL and then, from this new clean environment we execute the second bookmarklet with the code.

But we need to be very careful when generating the wrong URL, because an attacker could buy the wrong domain and inject a fake error page. To avoid this problem we can use random unused top-level domain names. This domain names are extremely expensive to buy, and it takes some time to buy one, therefore it's extremely unlikely that someone can get it in time for an attack.

Still, an attacker could compromise our DNS server and resolve those domain names (that's a targeted attack to the client, so it's less important, but still important). We can use HTTPS instead of HTTP, to require the fake server to provide a valid certificate for the wrong URL. If the domain is not valid, no certification authority should provide a valid certificate for it.

To increase security, the random string can be collectively generated by all the servers, and then, after is provided by the first server on the first bookmarklet, the other servers verify that the random string is the correct one. This way, the first server cannot generate a random looking string that he has in fact registered.

At the same time, the information about top level domains is public, and can be consulted here: http://data.iana.org/TLD/tlds-alpha-by-domain.txt. The verification servers can check it regularly to identify any suspicious new domain, and verify the first bookmark against this list.

A wrong URL would look like this: https://mysafesite.dj4j5h7sjksfjs87le3f8

It's worth noting that we cannot simply use the “new Tab” pages (that are also HTML) instead of the error pages. The problem is that many browsers generate them using things from the web (e.g. Chrome displays google and more things) that can compromise security.

Obviously, the local file can also be provided as a non-default option to the user, giving more flexibility to the advanced users.

Generalizing the method

The previously proposed methods allow to retrieve, verify and execute some previously defined code, but it can be generalized to allow choosing among different web applications. This way, it can be implemented as a generic service that can be used by many different applications.

One step further on this generalization would allow self-authenticated users to store some preferences. In this use case, a user would register a username, a public key and a set of preferences. They can modify their preferences with their private key in the future. Some kind of proof-of-work can be implemented to limit the registrations to real users.

The preferences might include the verification servers they want to use (among many

available), their favorite standard applications to be shown in the main screen, 3rd party or custom applications where they provide hashes for code they want to trust in the future, bookmarks vs local files preference, ...

It's important to note that as the registration is self-authenticated, we can have collisions on the usernames but not on the public keys. This is good, because your identity is not linked to a single server, but has the problem of uniquely identifying the user, that would find hard to remember the public key. Also, the preferences information is not strictly private, but it's not strictly public either, so it would be good if we could limit the access of this information.

A possible solution is to introduce the concept of a secret identity. This secret identity is the username plus some simple extra information that allows to disambiguate and provides some secrecy to the preferences.

Taking the idea of a secret identity literally, we propose a simple method to provide this extra information in a way that's easy to remember. The idea is that, during registration, the user selects a secret character (a pirate, a ninja, a superhero, …) Then, for this character, some random objects are selected among multiple options. A reasonable number would be 3 objects out of 16. For example, the user selects a pirate, and then, a parrot, a hook, and a sword are randomly selected among the possibilities for pirates (peg leg, eye patch, hat, …)

This is can be made very visual showing the character with the objects, so that it's easy to remember for the user. Then, when the user wants to use the service, provides the username, selects his character, and clicks on the 3 objects in a grid with all the possibilities.

It's important to note that this provides a really small level of security (low entropy, single server, ...), and it should never be used for things that require strong authentication. If we want to increase privacy a little bit, we can restrict the number of tries by ip in a given time, and also respond with random preferences when a wrong secret identity is provided.

Private data retrieval

The process described in the previous sections allows a user to use multiple web applications without trusting any single server. These applications can only access public information, the preferences, and information provided by the user. This can be enough for many applications, but others will need access to secret data that it's too long to be remembered by the user.

This can easily be solved by making use of the process described in the document “Token-Based Multiparty Password Authenticated Key Retrieval Process” (Pablo Guerrero – August 2014) that allows to retrieve a secret information (like a key) making use of multiple independent servers and a password. This process guarantees the secrecy of the information and the password unless all servers are compromised by an attacker.

When both processes are used together, they can reuse some of the elements, like the multiple independent servers. It's also possible to reuse the secret identity of the currently described process

to slowdown a denial of service attack against the other process tokens (instead of the second low entropy password).

Conclusions

The previously described process is a reliable way to execute multiparty verified code in a modern desktop or mobile browser without making use of browser extensions or external applications.

We have shown that it can be extended to support multiple applications, user preferences, and even private data, which makes it useful for most real world applications.

The proposed process is really simple, and it's easy to verify that each individual step is safe, if the assumptions are met. This can help solving the doubts about using the web for secure client-side applications.

At the same time, if simple instructions and user interfaces are provided, it could be used by non-expert users while requiring a reasonable amount of work for the benefits provided.