The same origin concern

(Gal Weizman@MetaMask)

Secure realms towards safer composability driven web applications

This document focuses on the "same origin concern", describing the lack of control apps have over new realms that rise under their own origin, as well as its implications on their safety, how current efforts to address it fail and what browsers can do to help ship a secure and performant solution for the problem

~ *tl;dr*

- Fortunately, web and browsers continuously **evolve towards a composability driven software development future**.
- While it is good, **such methodology bears great security risks**, where integrated software may **introduce malicious code to the composed software**, endangering it entirely.
- Therefore, ongoing efforts by security leaders are being made to introduce safety mechanisms to **allow apps to unlock composability's full potential with minimal risk**.
- One effort in particular focuses on providing better visibility and control over the flow of the app at runtime, to uncover unexpected behavior in the case of it resulting from malicious intentions - a most reasonable **natural outcome of the difficulty in securing composable web apps**.
- However, the road to secure composable software is long and yet to be over, as **critical infrastructures must continuously evolve accordingly**.
- This includes browsers, which in order to enable such initiatives might require similar adjustments, to **help composed software thrive**.
- Therefore, we wish to focus on one specifically being what we refer to as **the "same origin concern"**, where we identify a need for some level of protection/isolation of the main realm of the application from any sibling/adjacent realms of the same origin (*some subset of what the Same Origin Policy provides for cross origin realms isolation is another way of thinking about it*).
- Not only did we identify this issue, but we also bring a working shim of the solution we imagine, which effectively demonstrates the security hardening we seek for applications against this concern in **production**.
- Problem is, as part of this effort, we learned that **a shim representation of the solution is insufficient** in terms of both performance, but most importantly security.
- Which helped us arrive at some conclusions, the main one being that **adequately securing this layer of concern performantly can only be done with some help from the browser itself**.
- In this document we wish to further explain the motivation for this, the problem we encounter, the solution we attempted at and the solution we believe would be best to properly address it.

Background

- There is a **strong rising need** for browser client side runtime services, **mostly security oriented ones.**
- Previously, web applications were not constructed in a composable manner, meaning they were **not very reliant on externally developed software.**
    - The upside to this, in regards to security, was that the absolute **majority of the code** executing in the browser by the web app was **developed by the maintainer of the app** itself, giving **full clarity and control** over what the code does.
    - Therefore, security efforts were mostly focused around “external” attack vectors, meaning attack scenarios that rely on user input mostly, due to **lack of comprehension of the risk in user input** to the client side of web applications in the past.
        - A security gap which was later **filled adequately by well designed solutions** that came to be **core security mechanisms in browsers.**
- **Today’s landscape changes that**.
    - The further we go, the better we understand the **importance of software composability** and the efficiency it brings to software development and runtime, and embrace it accordingly.
        - In the build time realm, we now refer to it as **“supply chain based development”** - a justifiably popular methodology for developing applications these days.
    - However, the supply chain based development methodology and its rising adoption comes with a **natural negative side effect** of forming an **“easy” gateway for malicious code** to find its way to the app in mass;
        - Resulting in **most malicious code execution originating in supply chain breaches** as of today - a new worrying concern.
    - Attempting to address that takes us to **higher levels of complexity**, for two main reasons:
        - **Buildtime composability** - telling trusted code from untrusted code where **both are generated in the build process** is significantly harder than it was for telling “user-input” code from trusted code of the app itself.
        - **Runtime composability** - and even if build time composability became safer in the build process somehow, the web still majorly **consists of web apps that are composed of both build and runtime software**, where the latter **cannot be secured at build time** (e.g. third party scripts), and the app has no control over whatsoever.

Motivation

- Because of that, there's a **need for all sorts of extra advanced security protections - for both build and runtime**. One isn't enough without the other due to how hard it is to solve this problem, but mostly because of the "runtime composability" argument above.
    - Therefore, in this piece we focus on the runtime side, without diminishing the importance of build time solutions.
- Focusing on the runtime part, this means we want to be able to "control" the behavior of the app at runtime, out of knowing that due to the rise of composability driven development adoption, **we'd never know for sure what most of the code constructing the app is going to be** (for better or worse).
    - Thanks to the infinite flexibility code provides, "control" can mean basically anything we want it to mean:
        - Observe and/or block the behavior of sensitive operations to detect runtime misconduct[1];
        - Limit the environment of the app to make it less useful for attackers to begin with[2];
        - Alter the functionality of certain APIs to harden their security;
        - *(All real examples of already existing third party tools, references below)*
- Grasping the magnitude of the problem and how **supply chain based development changes the battlefield dramatically**, **we expect to see more and more such runtime solutions** as we go forward.
    - And in that context, we want to make sure browsers are well equipped to embrace such efforts, similar to how they successfully embraced initiatives to protect web apps from "external" attack vectors like in the past.

Problem

High Level

- The architectural work on browsers in the past years was great, and it might allow us to more easily adjust them to **continuously embrace the ongoing adoption of composable development**.
- A significant deficiency we identify and would like to focus on today is the **"same origin concern"**, which **hurts maintainers of such advanced runtime solutions** the most.
- The browser is great when it comes to protecting cross origin realms from each other, but **isn't at protecting same origin realms** the same way.
- Which makes total sense - browsers were **not designed to allow composability safely** the way it turned out these days.
    - Browsers had reasons to worry about code from one domain accessing information/capabilities of other domains, but had **no reason to worry about two different code entities running on the same origin**.
- Problem is, as discussed above, we expect to see more and more sophisticated runtime tools which will aspire to apply protection to the main realm execution environment (as well as its global object) of web applications, and the lack of isolation between same origin realms (as we refer to as the "same origin concern") **completely undermines those**.
- But how so?

Technical Overview

- By *"aspire to apply protection to the main realm execution environment of web applications"*, we mean to say that given the capabilities of JS and the browser current design, anything the discussed tools attempt to achieve, **only applies to the main realm** of the web application by default.
- Let's break it down, focusing first on *"the main realm execution environment of web applications":*
    - The main realm being the default execution environment of the web app in the browser, along with its own default global object, aka the "window" object, from which all powerful capabilities can be accessed.
        - What is a realm in JavaScript?
- Understanding that part, now it's important to realize what *"aspire to apply protection"* means. Consider this overly simplified example:
- Let program X be a **standard web app**, which by standard is built on around **90% code other people on the web wrote**, which constantly updates with **no great continuant visibility** into these changes.
- This may easily allow any of those 90% to **update X with malicious code** (whether intentionally or not).
- This malicious code can not be found using a shift-left approach, like using static analysis tools run during build time, therefore the need and the focus on runtime security.

- There are dozens of areas a web app such as X would be concerned about and might wish to protect in case such malicious code execution is triggered, but for the sake of the example let's assume we wish to defend access to the **localStorage API** of app X.
- Let tool Y be a **standard runtime security solution** developed by some third party vendor, which program X decides to install in its web application in order to address the security concern described above by enhancing protection against it.
- To address this concern, web app X decides to install tool Y.
- By installing it, X grants Y the power to control the app, which it uses to apply protection rules and logics it invented.
- Focusing on our specific example, perhaps Y rewrites the behavior of the localStorage API to only allow access with some agreed upon secret:

```
JavaScript
window.localStorage.getItem = function(key, secret = '') {
	if (secret !== 'AGREED_UPON_SECRET') {
		return null // protect access to localStorage items!
	}
	return localStorage[key]
}
```

- In a broader and more realistic resolution, this technique can be highly useful for tools such as Y with helping apps such as X to better **defend their apps in runtime due to being left with currently non-sufficient solutions for the supply chain security problem** expressed above.
- However, as of today, attackers who made it into app X and wish to access the localStorage despite the protection of tool Y, can unfortunately do so **with almost zero effort**:

```
JavaScript
function getLocalStorageNaively() {
	return window.localStorage
}

function getLocalStorageBypass() {
	const ifr = document.createElement('iframe')
	return document.body.appendChild(ifr).contentWindow.localStorage
}

getLocalStorageNaively().getItem('sensitive_pii') // null
getLocalStorageBypass().getItem('sensitive_pii') // +972-5555-333
```

- What happens here, is that the attacker **leverages the concept of same origin realms to go around the protection** introduced by tool Y, by instead of accessing the

localStorage capability of the main global object of the application, which tool Y defends, it creates a new realm (using an iframe) and **steals a fresh new instance of the same localStorage capability**, which by design, tool Y cannot protect.

Conclusion

- The explanation above clearly demonstrates the problem, where under the belief we should see more and more runtime security tools, those **won't be able to accomplish their mission as long as the "same origin concern" exists**.
- It is important to clarify the definition of the problem in addition to the demonstration of it.
- On the one hand, the fact that the browser successfully protects cross origin realms from each other is **fantastic**.
    - It does so by making sure neither has sync access to the other's global object, DOM, storage or any other strong capabilities, as well as not being able to perform any actions on behalf of it.
- On the other hand, the fact that the browser offers no optional subset of the protection it offers for cross origin realms to same origin realms, is the **"same origin concern".**
- Without yet being certain on the best way to achieve it, the browser should **offer web apps a way to opt in to apply some protection against sync JS access between same origin realms**.

Solution

Present

- As of today, **this problem is not solved**. Both the theoretical example above and real world tools (see references [1] & [2]) **can be neutralized** to a large (if not full) extent by a malicious actor.
- To practically address this problem, there is **the Snow project initiative** by LavaMoat@MetaMask which implements the desired solution to this problem as we see it, by:
    - Providing a simple API that allows anyone in the page to chronology register a callback to the creation of all same origin realms in the app;
    - This callback will be invoked with the WindowProxy object of every new same origin realm that comes to life in the page - before being passed on to its original creator.
        - Which means Snow gives the registrar **first access to any same origin realm** that is being created in the page, **even if by a malicious entity**.
- **Snow is implemented as a JavaScript shim**, and is effectively being used to protect the MetaMask app in production.
- If to lean on the former example, Snow allows you to **easily ship any security mechanism to all same origin realms, which was not possible before**:

```javascript
// Use Snow
SNOW((win) => {
              win.localStorage.getItem = function(key, secret = '') {
              if (secret !== 'AGREED_UPON_SECRET') {
                     return null // protect access to localStorage items!
              }
              return localStorage[key]
       }
}

// Snow protects same origin realms
getLocalStorageNaively().getItem('sensitive_pii') // null
getLocalStorageBypass().getItem('sensitive_pii') // null
```

- However, being a JS shim, Snow carries a number of **core disadvantages** in its approach, mainly:
    - **Security** - same origin realms can be created in so many ways and introduce so many edge cases, which makes it **close to impossible** to successfully synchronously hook into all possible ways to create a new realm when implemented over JS.

    - **Performance** - The only way to achieve such a state with JS is by hooking into many core functionalities (aka monkey patching) and extending their behavior. This technique usually introduces a **performance overhead.**
        - That is because virtualizing so many parts of runtime JS **inevitably introduces significantly more processing** to calculate information the **browser can easily grant us itself**.
        - The performance argument is **not to be overlooked**, because even if in a world where achieving this goal was possible security-wise (which it **most likely isn't**) - that would still introduce too big of a **performance overhead for apps to actually adopt it**, leaving those continuously **vulnerable to the security hazards composability brings** with it.
- The security aspect however, is the one we're mostly concerned with.
    - After maintaining Snow for 2 years now, we are still **overwhelmed by the amount of ways** the community helped discover **to bypass the virtualization layer** we were trying to achieve, which with time brought us to understand the **complexity of solving this issue** from the JS-shim approach and the chances of it **keep being bypassed** again and again, probably **endlessly**.
        - Therefore, this helped us reach both probably the most secured open sourced shim, but also the realization that **a fully secured one isn't possible to create without the help of a native browser solution.**

Future

- What would the "perfect" solution look like in the future? The short answer is - **we're not sure**, but we clearly imagine some **native browser support** to eliminate the downsides that come with a JS shim solution such as Snow.
- We can however emphasize the main needs from whatever that solution might be:
    - Allow web apps to opt in to a feature that allows the main realm of the app to **remain secure against other same origin realms.**
- The solution can end up being anything on the spectrum in between two ends:
    - **Zero access approach** - two same origin realms can become protected from one another by simply **disallowing synchronous access between them completely.**
        - Imagine a way for apps to opt-in to a feature that enforces zero synchronous access between same origin realms, exactly like we already have by default for cross origin realms (aka SOP).
    - and,
    - **Full access approach** - Continue to allow full synchronous JS access between same origin realms as-is, but provide the app the **ability to register JS code to "shape" every new same origin realm** that is brought to life in the app (or to delegate that ability to third party vendors).
        - In other words - Similar to the end result Snow brings (but implemented differently being a native browser solution rather than a JS shim).

- While the former would make life easier for runtime security tools for not having to deal with same origin realms anymore to begin with, it might **not be sustainable** when taking into account backwards compatibility with how the web is currently being used.
    - Which is to say, it is highly likely that many web apps make use of sync JS access across same origin realms, while wanting to remain protected against the “same origin concern”.
- Which leaves us with the latter solution, which would **allow runtime security tools to ship their protection mechanism to all same origin realms safely, automatically and seamlessly,** while continuing to allow apps to leverage same origin realms as they wish.
    - Which, again, is pretty much what Snow does.
- If to consider this path, this is what we generally imagine:

Proposal

- In order to achieve the desired state, browsers should **leverage already existing APIs**.
- We want to enable web apps to instruct the browser to **run a piece of code when a new same origin realm is initialized**, before any other JS code gets to run.
- We see the **CSP mechanism** as a **great potential candidate** for that for a few main reasons:
    - CSP implementation in the browser is rather **good at enforcing rules on realms and delegating those recursively** to child realms - which is exactly what the “same origin concern” is about!
    - CSP is a **strong and most etiquette mechanism for enforcing security policies** dictated on behalf of the web app.
- That is why, **CSP could make a good candidate** for delivering such an instruction.
- In other words, it would make sense to ask **CSP to load a list of JS files by referencing remote paths** to them, in the order they were listed by.
    - Since this is a sensitive capability, it should be treated like one, meaning such a file must be **enforced to be fetched from the same origin as the app**, leaning on the precedent established by Service Workers.
    - Which also means, the proposed CSP directive must be one that cannot be dictated using the `<meta/>` HTML tag.
- So to take that into consideration on top of the localStorage example presented above, we imagine something similar to this:

JavaScript

```
// /scripts/realm.js
window.localStorage.getItem = function(key, secret = '') {
	if (secret !== 'AGREED_UPON_SECRET') {
		return null // protect access to localStorage items!
	}
	return localStorage[key]
}
```

Unset

```
CSP: "run-on-same-origin-realm /scripts/realm.js /scripts/<OPTIONALLY_2ND_SCRIPT>.js;"
<html>
	<script>
		localStorage.getItem('sensitive_pii') // null
	</script>
	<iframe id="xyz" src="about:blank"></iframe>
	<script>
		const ifr = document.getElementById('xyz');
		ifr.contentWindow.getItem('sensitive_pii') // null
	</script>
</html>
```

- This example demonstrates a standard web application that leverages the desired CSP directive to remotely fetch a same origin script resource to be executed in the context of every new realm environment that the app loads in the same origin as the app itself, including the default main realm of course.
- The app may choose to pass more than one single remote path in case it wants to delegate that ability to more than just one entity, in which case the ability would be delegated to the listed entities chronologically.
- Thus, **allowing the app to regain true control over its same origin realms, with which it can protect itself against by shaping it to its liking.**
    - A power which is only fair to grant the app.

References

- [1] e.g. PerimeterX CodeDefender, Jscrambler Webpage Integrity and more
- [2] e.g. MetaMask LavaMoat and more