UC Berkeley - CS 261 - 9/14/2012

Privilege Separation

David Wagner

Scribe: David Schinazi

Same Origin Policy

SOP is an attempt to put a clean security policy / architecture on the web, it's hard since the web evolves fast.

We want it to be safe to visit any website (trusted or untrusted), safe as in the website not interacting with browser or computer more than it should.

In SOP the origin refers to the domain name serving the web site, the protocol used (http(s)), and also the ports used.
- any downloaded content is tagged with its origin
- executing javascript runs with permissions related to that origin, e.g. it is allowed to send requests to the same origin, see cookies, HTML-5 client-side storage, etc. Therefore each origin is responsible for protecting their server. Javascript can attach hooks (keyboard events, page load events), modify the HTML DOM tree, but only on pages from the same origin.

However this has exceptions:
- if HTML from a.com asks for a javascript file from b.com, that code will be run with a.com privileges
- javascript from a.com can send a request to b.com but is not allowed to read response from b.com
- a.com can embed an image from b.com (the image will be displayed but javascript on a.com cannot access the contents of the image, the image is tagged with a b.com origin), the same applies to <iframe>

One of the advantages of this is that the request to b.com will be sent with the user's cookies for b.com, but javascript from a.com is not allowed to access those cookies.
If the a.com server itself sends the request to b.com, then it can send the results down to the user. However in that case the user's cookies won't be enclosed.

Example of attack: (CSRF Cross Site Request Forgery) user has visited bank.com and is now on evil.com, then evil.com triggers a request to bank.com asking the bank to send money. Since the request is sent with the user's cookies, it is authenticated.

Solution: on bank.com main webpage, the link to send money includes a random token that evil.com cannot guess. That token is downloaded with the HTML from the bank main page, therefore preventing evil.com from seeing request responses is necessary to make this safe.

Chromium

Why does it separate components?
For example, a vulnerability in renderer does not allow access to the complete file system. If you are visiting evil.com, the vulnerability could allow the attacker to access all passwords for evil.com but for that you don't even need a vulnerability, evil.com can change its site to send your evil.com passwords directly with javascript. We're not sure if Chromium efficiently protects your other passwords in the case of a renderer vulnerability. A renderer vulnerability could also possibly leak all cookies.

Chromium decided to not implement SAO because they decided to be compatible with all websites, and implementing it without braking the web would cause each javascript from a different origin to be run in a new renderer, and each of its accesses to the HTML DOM through an API, which would be very slow and complicated to achieve. All projects bringing this level of security were incompatible with many websites (e.g. Facebook Connect).

The reason Chromium made the choice to support all websites versus improved security is that a user that finds a website working on Firefox and not on Chromium will switch to Firefox since security is not visible to users (and most users don't care).

Privilege Separation

Why use privilege separation?
- If application needs root access (e.g. SMTP server must listen on port 25), we can reduce the root part of the program to a small program (smaller means safer), so a vulnerability in the rest of the code allows the attacker to control it but does not allow him to control the part that has root access. We can also separate another process that can append files to all user's inbox.
- Other example: Wireshark (packet parser and displayer). The parser is one of the best examples of why privilege separation is useful: parsers are prone to vulnerabilities and are often closed boxes (one input, one output) therefore can easily be sandboxed.
- SSH server: an idea would be to separate a *Master* - running as root that handles the TCP port and the initial connections - from several *Slaves* that transfer the bytes from the wire to the user's shell. The slaves run as the user therefore a vulnerability does not give access to a root shell, or to the connection private keys.

How can we evaluate the improved security from changing the architecture?
- How big (size of code) is the trusted code (the one that runs at root)
- Knowing common vulnerabilities, what percentage of them would be in the trusted code?