RTC-Web Security Considerations

IETF 80

Eric Rescorla

ekr@rtfm.com
The Browser Threat Model

Core Web Security Guarantee: “users can safely visit arbitrary web sites and execute scripts provided by those sites.” [HCB+10]

- This includes sites which are hosting malicious scripts!
- Basic Web security technique is isolation/sandboxing
  - Protect your computer from malicious scripts
  - Protect content from site A from content hosted at site B
  - Protect site A from content hosted at site B
- In this case we’re primarily concerned with JavaScript running in the browser

The browser acts as a trusted computing base for the site
List of Issues to Consider

• Consent to communications
• Access to local devices
• Communications security
In an alternate universe: Cross-Site Requests

Victim → Gmail → Attacker

Login w/ Password

Cookie=XXX

→

GET /malicious.js

<script>XMLHttpRequest("https://gmail.com/")...

GET w/ XXX

→

Mail data

←

Mail data

Obviously this is bad... and it’s a problem even w/o cookies
The Same Origin Policy (SOP)

- A page’s security properties are determined by its origin
  - This includes: protocol (HTTP or HTTPS), host, and port
  - All these must match for two pages to be from the same origin
- Each origin is associated with its own security context
  - Scripts in origin A have only very limited access to resources in origin B
- Important: the origin is associated with the page, not where the script came from
  - Scripts loaded via `<script src="">` tags are associated with the origin of the page, not the URL for the script!
The Same Origin Policy for Page Data

- Scripts can only access page data from their own origin
  - Contents of the DOM
  - JavaScript variables
  - Cookies
  - Important exception: JavaScript pointer leakage [BWS09]

- Scripts can access any other page data from their origin
  - Includes other windows and IFRAMEs

- Frame can navigate their own children
  - This is used for cross-site communication (e.g., FaceBook Connect)
The Same Origin Policy for HTTP Requests

- JavaScript can be used to make fairly controllable HTTP requests with XMLHttpRequest() API
  - But only to the same origin
- Origin A can make partly controllable requests to origin B via HTML forms
  - But cannot read the response
  - Cross-Site Request Forgery (CSRF) defenses depend on this
- Origin A can read scripts from origin B
  - But they run in A’s context
  - This is done all the time (e.g., Google analytics)
What does all this mean for RTC?
Consent for real-time peer-to-peer communication

• Need to able to send data between two browsers
  – Unless you want to relay everything

• But this is unsafe (and violates SOP)
  – Not OK to let browsers send TCP and UDP to arbitrary locations

• General principle: verify consent
  – Before sending traffic from a script to recipient, verify recipient wants to receive it from the sender
  – Familiar paradigm from CORS [vK10] and WebSockets[Fet11]
How to verify consent for RTC-Web

- Can’t trust the server (see above)
  - Needs to be enforced by the browser

- Browser does a handshake with target peer to verify connectivity

  Alice  Server  Bob

  Connect to Bob                         Connect to Alice
  ↓                                    ↓
  Handshake                             Handshake
  ↓                                    ↓
  Media traffic                         Media traffic

- This should look familiar from ICE [Ros10]
- Restricts communication with that endpoint until handshake complete (new)
Access to Local Devices

• Making phone (and video) calls requires that your voice be transmitted to other side
  – But the other side is controlled by some site you visit
  – What if you visit http://bugmyphone.example.com?

• Somehow we need to get the user’s consent
  – But to what?
  – And when?
  – Users routinely click through warning dialogs when presenting “in-flow”

• What is the scope of consent?
  – By origin?
  – What about mash-ups?
What about communications security?

- We’ve already addressed this in the context of SIP
  - Things aren’t that different here—all the usual protocols work
- Open question: where is the keying material stored?
  - On the server?
  - In localstorage?
  - In the browser but isolated from the JavaScript? (probably best)
References


