Device-Bound Session Credentials

W3C Web Payments WG — 2023-12-07

Arnar Birgisson & Kristian Monsen, Google
{arnarb,kristianm}@google.com

Why device-binding?

Cookie theft: Malware exfil of sessions to **offload abuse to elsewhere**.

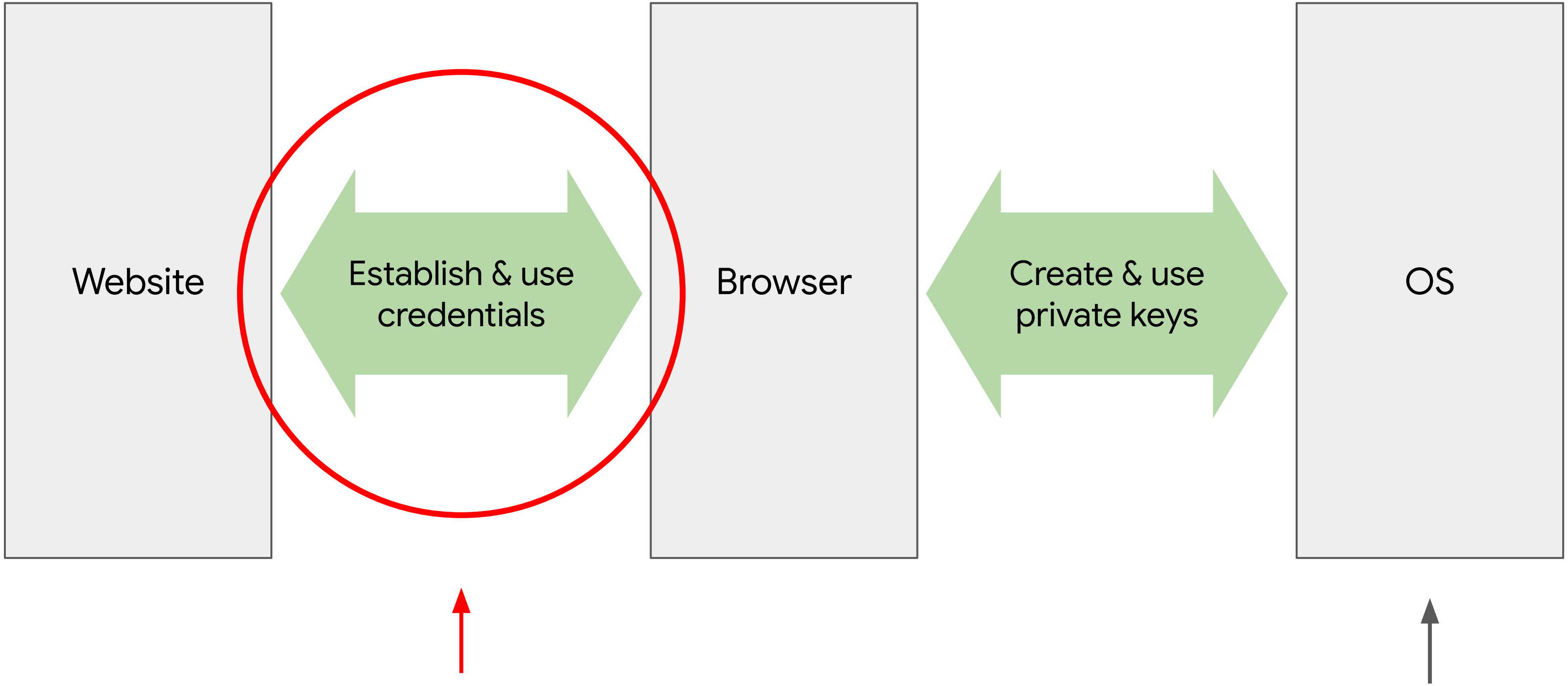
Cookies are limited to storing data (set+get) → Bearer tokens only

To use bearer tokens, browser must have access to it.

Browsers cannot protect against malware of same privilege.

Private keys instead → Offload protection to more privileged parts of the OS.

But, we need a protocol first.



Website

Establish & use credentials

Browser

Create & use private keys

OS

We will focus here

Responsible for key protection facilities.

What is difficult?

– The crypto itself. Not particularly.

– Key protection from malware → Limited resource, slow.

– Deployment and migration

– Complexity, scoping and interop with other things (like sign-in)

– API & protocol design

Key protection from malware

– OSes and devices vary in capabilities.

– Operations on keys can be slow, e.g. with TPMs.

(Cannot e.g. simply sign every request.)

Difficult but out of scope for protocol: The actual key protection

Difficult but in scope for protocol:

Flexibility for website to require signatures as needed, but not too often.

Flexibility to support improvements over time.

Deployment and migration

– Even simple websites can be complex. (s/websites/webapps/ at will)

– *When* to require signatures can be part of business logic, but lots of work.

– Web stacks are complex, and auth is cross-cutting. E.g.:

Device-binding at the TLS layer can be far away from session mgmt

Auth middleware in multiple places, using off-the-shelf libs.

Difficult and in scope for protocol:

A way to get binding without rewriting business logic or migrating stacks.

API design, managing complexity, scope

– Tempting to build the kitchen sink too.

– Browser behavior must be well-defined and predictable to aid debugging.

– Too few knobs: Rigid and inflexible, only works for some, only today.

– Too many knobs: Hard to integrate with, footguns.

Important mitigation in DBSC:

- Protocol + API is independent of sign-in (simpler), but can provide hooks to tie with pre-existing device bindings there (flexible).

Overview

1. Explicitly represent the "session" concept.
2. New functionality on website is "add-on", not a rewrite.
3. Periodic key proofs.
4. Browser manages when to send proofs.
5. The browser can *hold* other requests when proof is needed.
6. Rest of the website stays on cookies, but short-lived.

Make sessions explicit

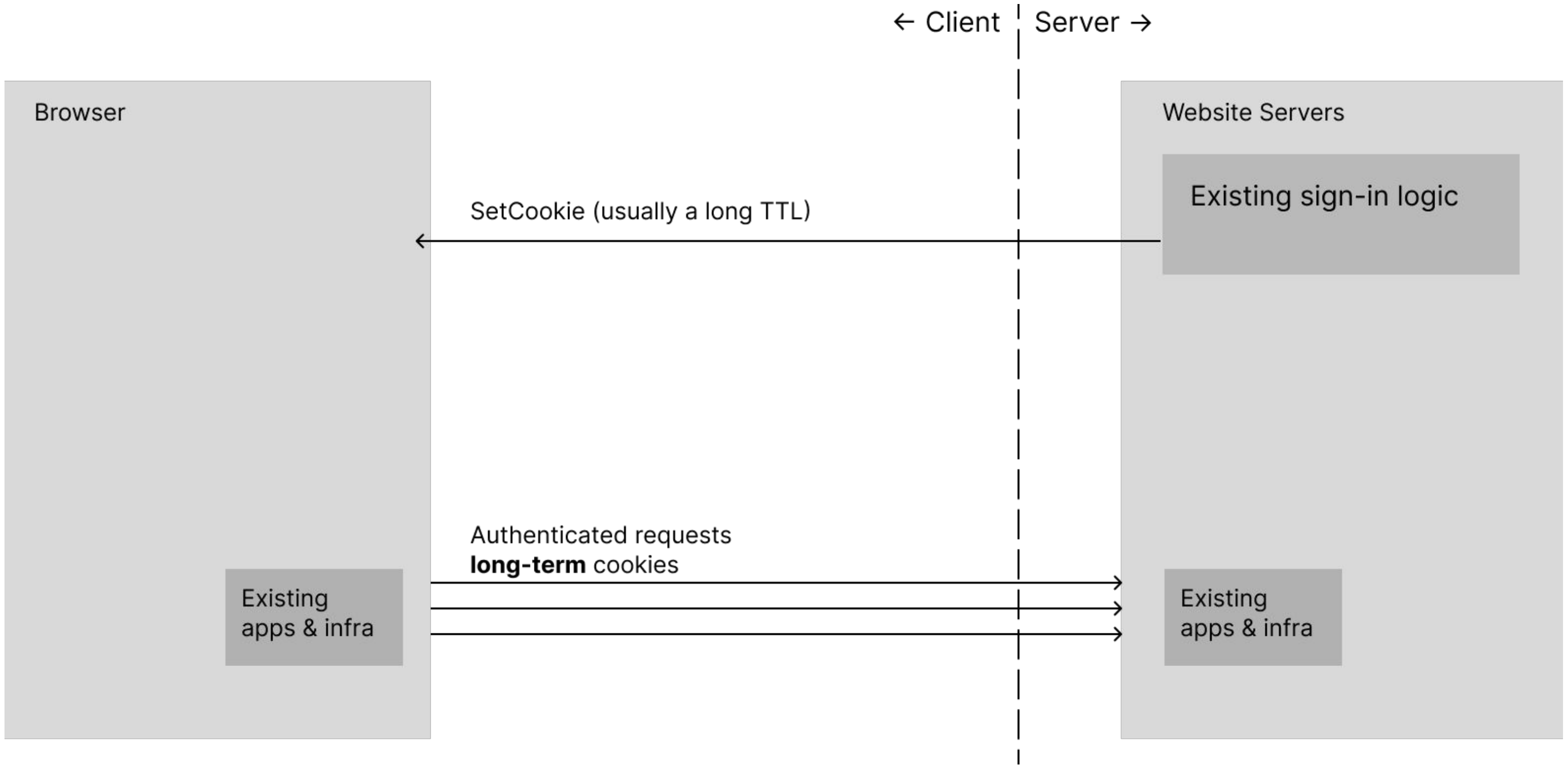
```
const session_info = await navigator.securesession.start({
  endpoint: "https://example.com/api/securesession",
  authorization: "<one-time bearer cred from sign-in>"
});

navigator.securesession.end(session_info.id);
```

– Today sessions are *implied*, hard for browser to apply semantics.

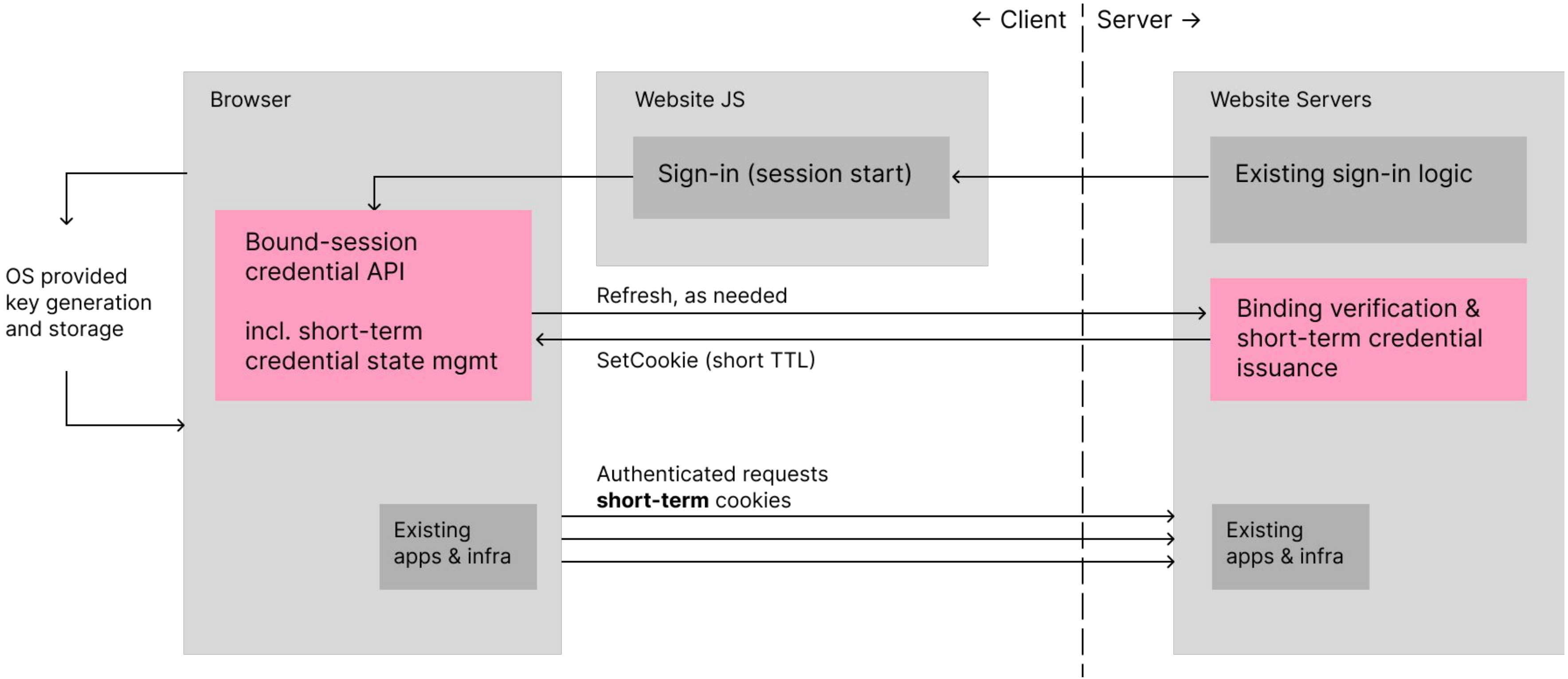
– While a session is **active**, browser does extra stuff to **maintain** it.

Note: DBSC defines minimal semantics, website decides what sessions *mean*.

New functionality is add-on



← Client
Server →
Browser
Website Servers
Existing sign-in logic
SetCookie (usually a long TTL)
Authenticated requests
long-term cookies
Existing apps & infra
Existing apps & infra



← Client

Server →

Browser

Website JS

Website Servers

Sign-in (session start)

Existing sign-in logic

Bound-session credential API

incl. short-term credential state mgmt

OS provided key generation and storage

Refresh, as needed

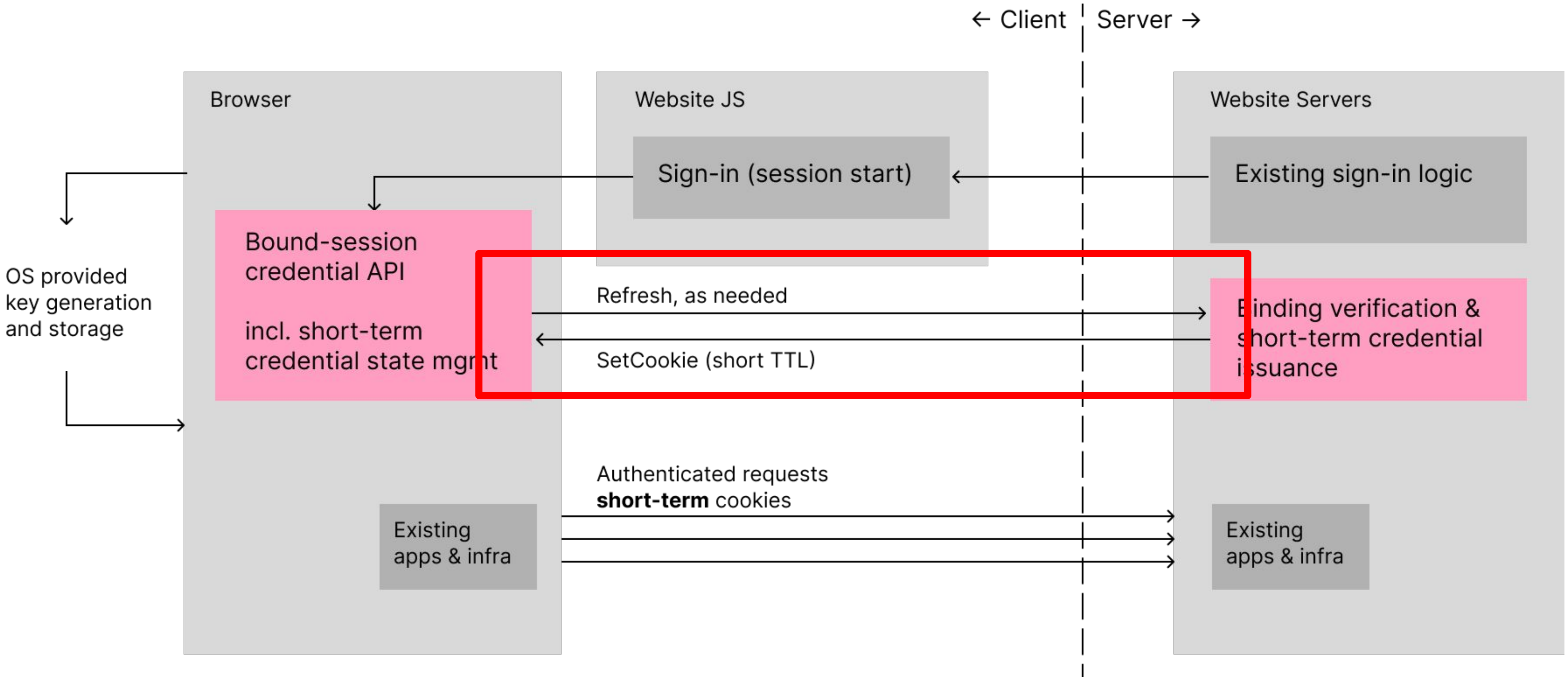
SetCookie (short TTL)

Binding verification & short-term credential issuance

Authenticated requests **short-term** cookies

Existing apps & infra

Existing apps & infra


← Client
Server →
Browser
Website JS
Website Servers
Sign-in (session start)
Existing sign-in logic
OS provided key generation and storage
Bound-session credential API
incl. short-term credential state mgmt
Refresh, as needed
SetCookie (short TTL)
Binding verification & short-term credential issuance
Authenticated requests
short-term cookies
Existing apps & infra
Existing apps & infra

Periodic key proofs

```
const session_info = await navigator.securesession.start({
  endpoint: "https://example.com/api/securesession"
});
```

```
POST /api/securesession/start
Content-type: application/json

{
  "binding_public_key":
           <new public key>
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "<server issued session id>",
  "required_cookies": [{
    "name": "auth_cookie"
  }]
}
```

Periodic key proofs

```
const session_info = await navigator.securesession.start({
  endpoint: "https://example.com/api/securesession"
});
```

```
POST /api/securesession/start
Content-type: application/json

{
  "binding_public_key":
           <new public key>
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "<server issued session id>",
  "required_cookies": [{
    "name": "auth_cookie"
  }]
}
```

Periodic key proofs

```
const session_info = await navigator.securesession.start({
  endpoint: "https://example.com/api/securesession"
});
```

```
POST /api/securesession/start
Content-type: application/json

{
  "binding_public_key":
           <new public key>
}
```

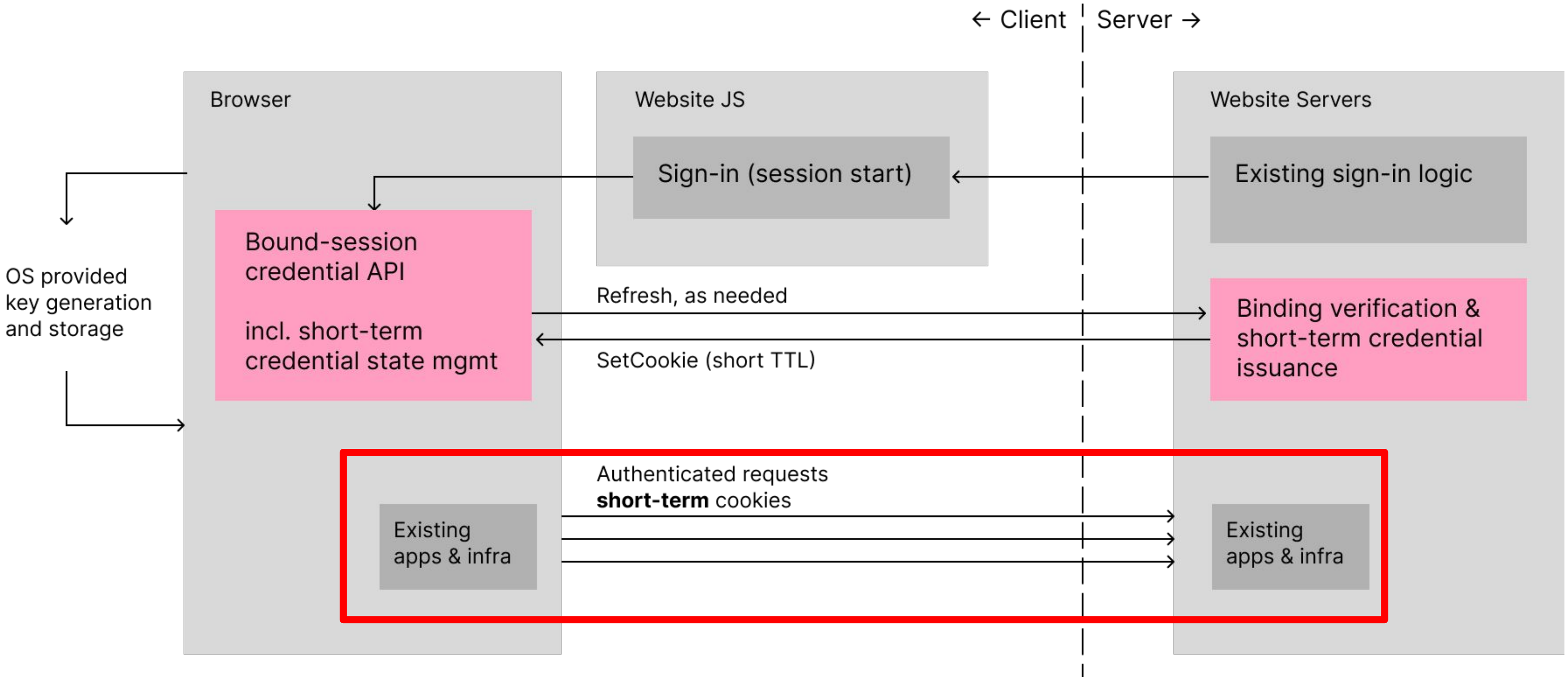
```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "<server issued session id>",
  "required_cookies": [{
    "name": "auth_cookie"
  }]
}
```

Periodic key proofs

```
const session_info = await navigator.securesession.start({
  endpoint: "https://example.com/api/securesession"
});
```

```
POST /api/securesession/start
Content-type: application/json

{
  "binding_public_key":
           <new public key>
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: auth_cookie=abcdef0123; \
            Domain=example.com; Max-Age=600;

{
  "session_identifier": "<server issued session id>",
  "required_cookies": [{
    "name": "auth_cookie"
  }]
}
```



← Client
Server →
Browser
Website JS
Website Servers
Sign-in (session start)
Existing sign-in logic
OS provided key generation and storage
Bound-session credential API
incl. short-term credential state mgmt
Refresh, as needed
SetCookie (short TTL)
Binding verification & short-term credential issuance
Authenticated requests
short-term cookies
Existing apps & infra
Existing apps & infra

Browser manages when to send proofs

```
{
  "session_identifier": "<server issued session id>",
  "required_cookies": [{
    "name": "auth_cookie"
  }]
}
```

If a required cookie is expired, the browser will contact the session endpoint again, with a fresh proof of possession.

```
GET /api/securesession/challenge HTTP/1.1
Host: example.com
Cookie: <.. as normal ..>
Sec-Session-Id: <session id>
```

```
HTTP/1.1 200 OK
Sec-Session-Challenge: \
  session_identifier=<session id>; \
  challenge=<random server issued challenge>
```

```
POST /api/securesession/refresh HTTP/1.1
Host: example.com
Content-type: application/jwt
Cookie: <.. as normal ..>

<signed JWT with body: {
  "sub": <the session identifier>,
  "jti": <the server issued challenge>,
}>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "...",
  "cookies": [{
    "name": "auth_cookie"
  }]
}
```

Identical to start response.
Each refresh can update the instructions.

```
GET /api/securesession/challenge HTTP/1.1
Host: example.com
Cookie: <.. as normal ..>
Sec-Session-Id: <session id>
```

```
HTTP/1.1 200 OK
Sec-Session-Challenge: \
  session_identifier=<session id>; \
  challenge=<random server issued challenge>
```

```
POST /api/securesession/refresh HTTP/1.1
Host: example.com
Content-type: application/jwt
Cookie: <.. as normal ..>

<signed JWT with body: {
  "sub": <the session identifier>,
  "jti": <the server issued challenge>,
}>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "...",
  "cookies": [{
    "name": "auth_cookie"
  }]
}
```

Identical to start response.
Each refresh can update the instructions.

```
GET /api/securesession/challenge HTTP/1.1
Host: example.com
Cookie: <.. as normal ..>
Sec-Session-Id: <session id>
```

```
HTTP/1.1 200 OK
Sec-Session-Challenge: \
  session_identifier=<session id>; \
  challenge=<random server issued challenge>
```

```
POST /api/securesession/refresh HTTP/1.1
Host: example.com
Content-type: application/jwt
Cookie: <.. as normal ..>

<signed JWT with body: {
  "sub": <the session identifier>,
  "jti": <the server issued challenge>,
}>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "...",
  "cookies": [{
    "name": "auth_cookie"
  }]
}
```

Identical to start response.
Each refresh can update the instructions.

```
GET /api/securesession/challenge HTTP/1.1
Host: example.com
Cookie: <.. as normal ..>
Sec-Session-Id: <session id>
```

```
HTTP/1.1 200 OK
Sec-Session-Challenge: \
  session_identifier=<session id>; \
  challenge=<random server issued challenge>
```

```
POST /api/securesession/refresh HTTP/1.1
Host: example.com
Content-type: application/jwt
Cookie: <.. as normal ..>

<signed JWT with body: {
  "sub": <the session identifier>,
  "jti": <the server issued challenge>,
}>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Set-Cookie: auth_cookie=abcdef0123; \
             Domain=example.com; Max-Age=600;

{
  "session_identifier": "...",
  "cookies": [{
    "name": "auth_cookie"
  }]
}
```

Identical to start response.
Each refresh can update the instructions.

```
GET /api/securesession/challenge HTTP/1.1
Host: example.com
Cookie: <.. as normal ..>
Sec-Session-Id: <session id>
```

```
HTTP/1.1 200 OK
Sec-Session-Challenge: \
  session_identifier=<session id>; \
  challenge=<random server issued challenge>
```

```
POST /api/securesession/refresh HTTP/1.1
Host: example.com
Content-type: application/jwt
Cookie: <.. as normal ..>

<signed JWT with body: {
  "sub": <the session identifier>,
  "jti": <the server issued challenge>,
}>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Set-Cookie: auth_cookie=abcdef0123; \
            Domain=example.com; Max-Age=600;

{
  "session_identifier": "...",
  "required_cookies": [{
    "name": "auth_cookie"
  }]
}
```

Identical to start response.
Each refresh can update the instructions.

Session refresh

– The browser only performs refreshes if there are other requests being made that are *in scope* for the session.

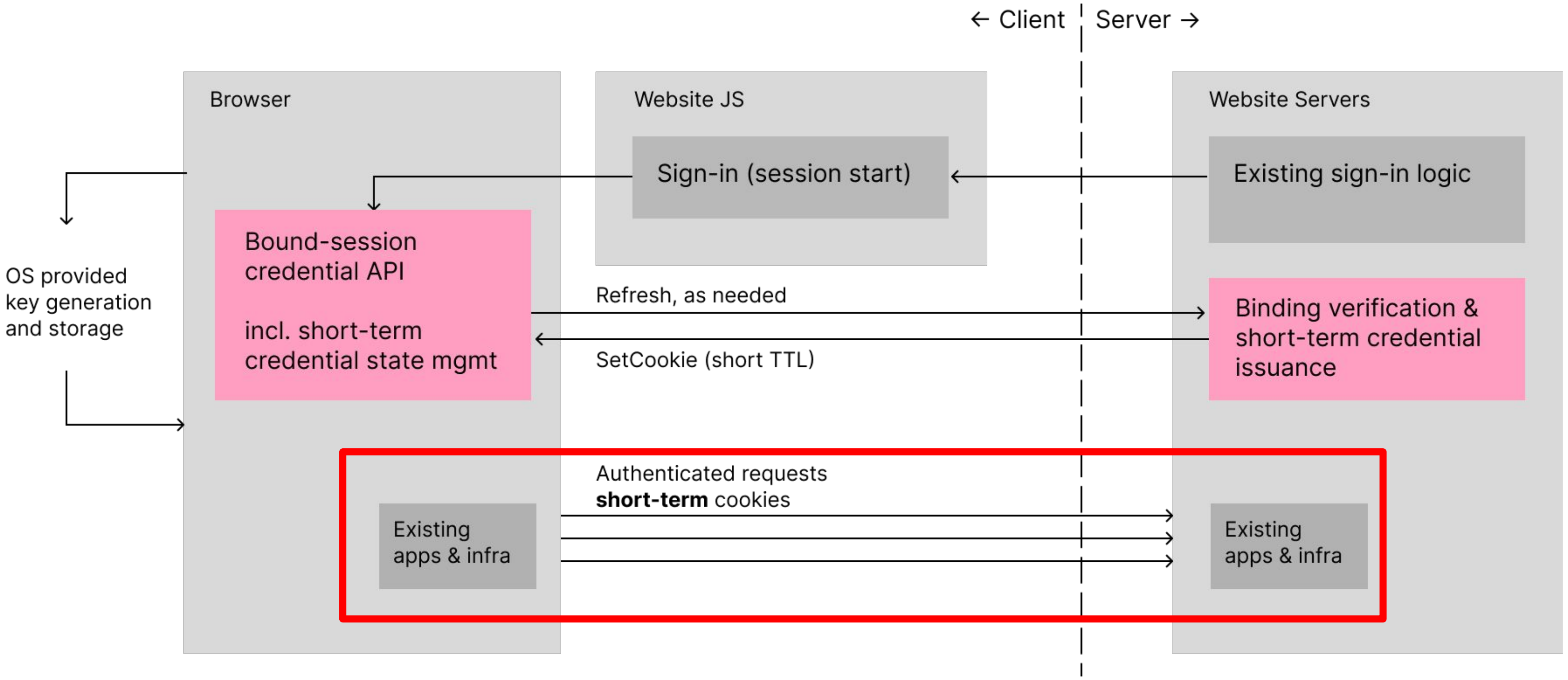
– Problem:

Refresh takes time.

What should server do with requests without those cookies? Still have to rewrite business logic?

Key choice for "migratability":

By default the **browser holds** (i.e. defers) any requests in scope ***until* the refresh endpoint replies**.



← Client | Server →

Browser

Website JS

Website Servers

Sign-in (session start)

Existing sign-in logic

OS provided key generation and storage

Bound-session credential API

incl. short-term credential state mgmt

Refresh, as needed

SetCookie (short TTL)

Binding verification & short-term credential issuance

Existing apps & infra

Authenticated requests
short-term cookies

Existing apps & infra

Browser holding requests?

– Greatly reduces the work needed on the server side.

Other requests in an active sessions still always come with a cookie.

Don't need to {upgrade auth libs, switch stacks, rewrite business logic, ...}

– Obviously introduces latency, likely user visible.

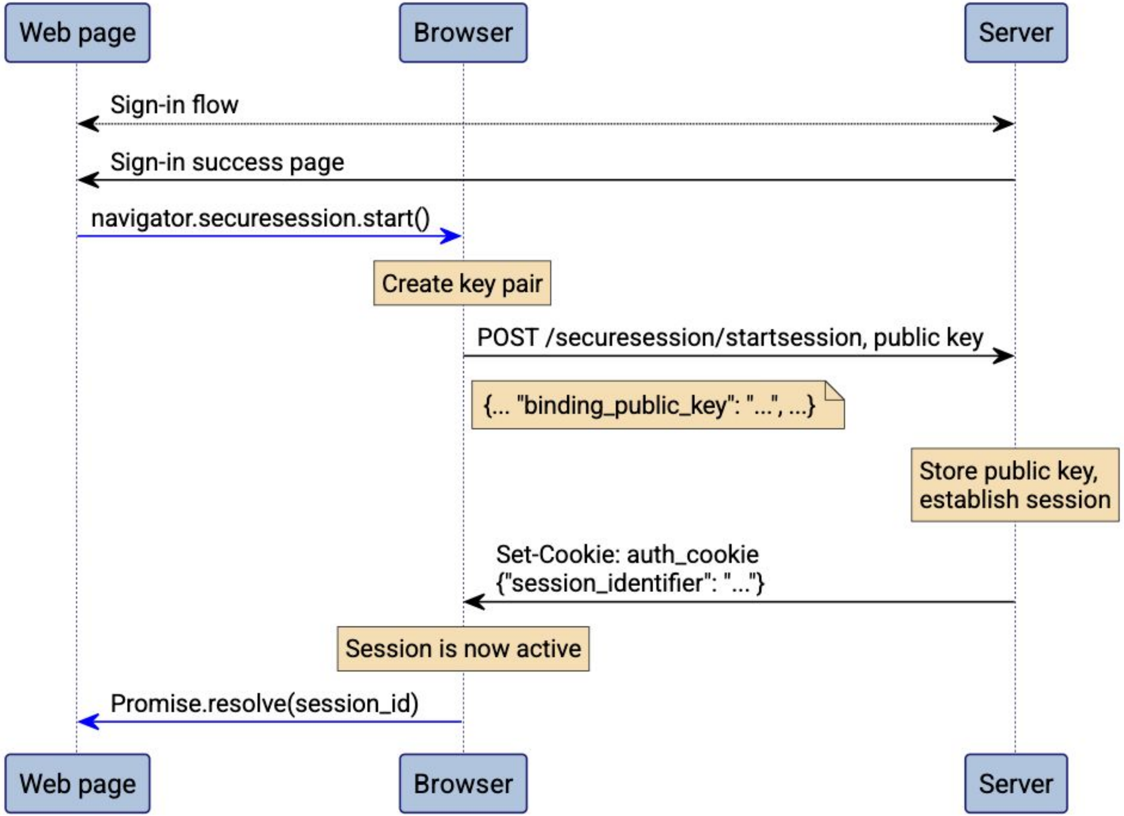
Unfortunately there is no free lunch. But:

– During active use of a website, the browser can pre-emptively refresh.

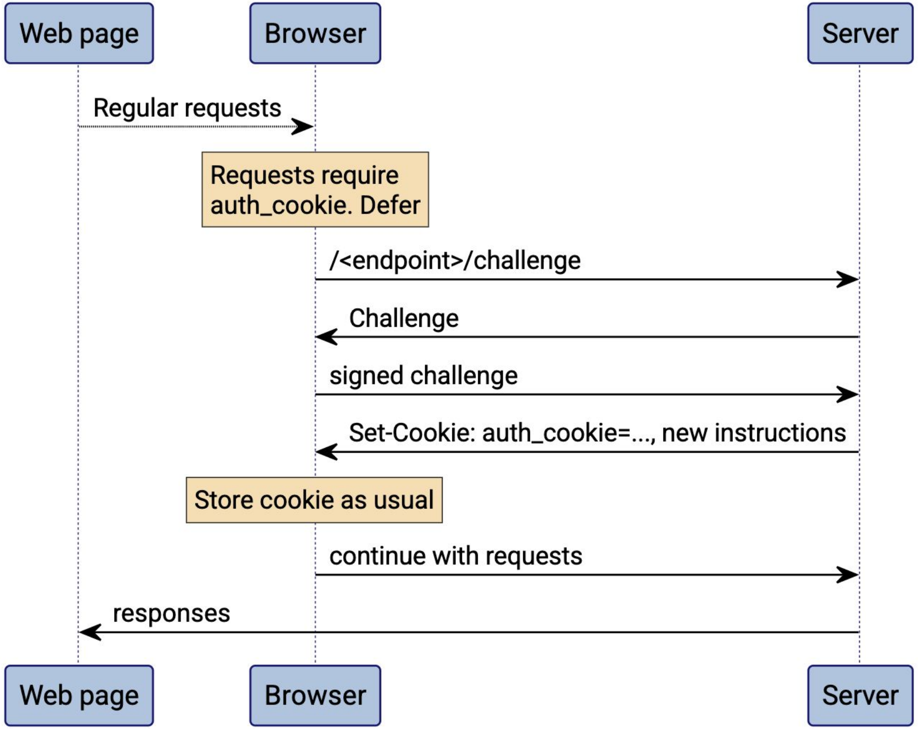
– Note that browser is also doing *one* refresh on behalf of many requests.

Q&A

Start Session


Web page
Browser
Server
Sign-in flow
Sign-in success page
navigator.securesession.start()
Create key pair
POST /securesession/startsession, public key
{... "binding_public_key": "...", ...}
Store public key, establish session
Set-Cookie: auth_cookie
{"session_identifier": "..."}
Session is now active
Promise.resolve(session_id)
Web page
Browser
Server

Session Refresh



Google

When is refresh performed?

```
{ ...
  "cookies": [{
    "name": "auth_cookie"
  }]
}
```

When the browser is about to make a request to this origin, but there is no current cookie named "auth_cookie".

I.e.: **The cookie expiration time (set by server) controls the cadence.**

We expect browsers to optimize:

Preemptively refresh if the user is actively using a website.

```
{ ...
  "cookies": [
    {
      "name": "main_auth_cookie",
      "exclude_paths": "/static"
    },
    {
      "name": "sensitive_action_cookie",
      "include_paths": "/changepassword"
  ]
}
```

Room for defining richer instructions, e.g.

- Paths of requests that are subject to blocking
- Multiple cookies (e.g. different TTLs, legacy systems)
- Challenge optionality
- (maybe) An option to do non-blocking refreshes for certain cookies/paths.

Extra round trips? Ugh

```
GET /api/securesession/challenge HTTP/1.1
Host: example.com
Cookie: <.. as normal ..>
Sec-Session-Id: <session id>
```

```
HTTP/1.1 200 OK
Sec-Session-Challenge: \
  session_identifier=<session id>; \
  challenge=<random server issued challenge>
```

– The server can, on any regular response, issue challenges pre-emptively.

– A challenge header does not trigger refresh, it's just stored for later.

Note: This *does* mean changing existing endpoints, but minimally.

What if I want a signature *now*?

- On any regular response (not refresh), just expire required cookies

```
HTTP/1.1 200 OK
Set-Cookie: auth_cookie=poof; expires=Thu, 01 Jan 1970 00:00:00 GMT
```

- The next request to the server will trigger refresh (and be held).

I thought this was about cookie alternatives?

– It *is*. With DBSC the real session credential is the private key.

The server issued session id can be used to "remember" things you currently remember in your long-term auth cookies.

– The required_cookies instruction is there for backwards compatibility and easier integration with existing web stacks.

– We have lots of ideas for alternative auth "in-between-key-proofs" that are not cookies, come talk to us.

Many details...

- Multiple concurrent sessions? – yes
- Sharing keys between multiple sessions? – no
- Does session mean a user is signed in? – Up to the website, browser is agnostic to the meaning
- Can the server still manipulate short-term cookies outside refreshes? – yes
- Does force expiring a cookie trigger refresh? – yes

...and open questions

- Cross-origin, cross-site? Maybe coupled with First-Party Sets
- HTTP-header driven, or Javascript API driven?
- Should the website also be able to trigger one-off signatures?
- Should we also support non-blocking instructions?
- Should browser signal rate limits on signatures?
- Are challenges optional?
- Other types of instructions?

Client certificates and TokenBinding

– Binding at the TLS level requires no changes to apps/business logic. Good.

– However, auth & session management are app-level constructs.

– Impedance mismatches

e.g. TokenBinding keys implicitly created so auth stacks have to take-them-or-leave-them.

– TLS endpoints are different from app endpoints

On clients: E.g. mandated TLS terminating proxies, roaming clients.

On servers: Frontend infra at large orgs and hosting providers

The simplest API possible?

- Could offer just two methods:

createDeviceBoundPrivateKey() : PublicKey, key-id

proveKeyPosession(key-id, challenge) : Signature

- Great if business logic can integrate checks. E.g. high-risk actions.
- Harder to be secure by default, e.g. with periodic signatures. E.g.

What triggers signatures when needed?

What to do with requests after last signature is too old?

When signature is old, probably many requests in flight already.