Fingerprinting

- Why fingerprinting hinders user privacy
  - Do
- How fingerprinting works
  - Aside: passive and active fingerprinting
  - Aside: measuring fingerprintability
- What do browsers do against fingerprinting?
  - Not supporting certain powerful APIs
  - The user permissions gateway
  - Adding unpredictability
  - Enforcing a privacy budget
- Use a broad testing environment
  - Do
  - Cross-browser testing tools
- Only rely on the user-agent string for coarse information
  - Fuzzing
    - Do
- Client Hints

Fingerprinting means trying to identify a user when they return to your website, or identifying the same user across different websites. Many characteristics may differ between your setup and someone else's. For example, you may be using a different type of device and a different browser, have a different screen size, and have different fonts installed. If I have the font "Dejavu Sans" installed and you do not, then any website can tell the difference between you and me by checking for that font. This is how fingerprinting works; you build up a collection of these data points, and each provides more ways to distinguish between users.

A more formal definition might look like this: Fingerprinting is the action of using obvious and non-obvious long-lived characteristics of a user's setup to attempt to distinguish them from as many other users as possible.

Why fingerprinting hinders user privacy

There are some edge cases where fingerprinting a user is important: fraud detection, for example. But fingerprinting can also be used to track users across sites, and that tracking is often done without users consenting to it, or, in some cases, on the basis of an invalid consent that does not adequately inform the user. When that's done, those users will often find this somewhat disquieting and feel rather betrayed.

Fingerprinting means finding ways to covertly distinguish one user from another. Fingerprinting can be used to recognize that it's still the same user on the same website, or to recognize the same user in two different browser profiles at the same time. This means that fingerprinting can be used for tracking users across sites. The deterministic and overt methods of tracking, such as storing a cookie with a unique user-specific ID, can be to some extent observed by users and controlled (and the previous module explained some of these approaches). But fingerprinting is more difficult to avoid exactly because it is covert; it relies on unchanging characteristics and most likely happens invisibly. This is why it's called "fingerprinting". It is difficult at best to change your fingerprint, whether your digital one, or the ones on the ends of your fingers.

Browser vendors *know* that users do not like being tracked, and are continually implementing features to limit fingerprinting (some of which we saw in the previous module). Here, we're looking at how these features may affect your business requirements and how to still do what you want to do in a privacy-protecting way. This is more about how browser protection *against* fingerprinting will affect what you do and how, rather than about how it will stop you *doing* fingerprinting.

In practice, most developers and most businesses have no need to fingerprint users. If your app requires users to sign in, then your users are identifying themselves to you, with their consent, and in a way that they can unilaterally opt out of at any time they choose. This is a privacy-protecting method of understanding which users are logged in. Your app may not require users to sign in at all, which is even more protective of your users' privacy (and you are then collecting just the data you need).

Do

Assess your third parties for fingerprinting. As part of the third parties module, you may already have a list of any third-party services you're including and the web requests that they make. It may be possible to inspect those requests to see which data is being passed back to the originator, if any. However, this is often difficult; fingerprinting is by nature a covert process which involves requesting data that isn't subject to user approval.

It's worth also reading the privacy policies of your third-party services and dependencies to look for indications of fingerprinting in use. It's sometimes referred to as "probabilistic matching", or as part of a suite of probabilistic matching techniques, as opposed to "deterministic matching".

How fingerprinting works

It can be helpful to look at https://amiunique.org/ and the EFF's Cover Your Tracks for an example of fingerprinting. Each will attempt to show you the characteristics of your personal browser setup which may differ from other people's setups. Frequently your personal combination of all these attributes is unique to you, or at least to a small group of similar people; this can be used to covertly track you. These websites aren't the whole story, and any uniqueness estimates are often how distinguishable you are among their visitors (who skew more technical and more privacy-focused) rather than among the whole internet, but the principles are sound.

Aside: passive and active fingerprinting

There is a useful distinction here to be drawn between passive and active fingerprinting techniques. A passive fingerprinting technique is one which uses information that is given to the website by default; an active fingerprinting technique is one which explicitly interrogates the browser for extra information. The reason this distinction is important is that browsers can attempt to detect and intercept, or mitigate, active techniques. APIs can be restricted, or gatewayed behind a dialog asking the user's permission (and therefore alerting the user that they're being used, or allowing the user to deny them by default). A passive technique is one which uses data that's already been given to the website, often because historically, in the nascent days of the information superhighway, that information was given to all sites. The user-agent string is an example of this, and we'll look at this in more detail further on. It was considered useful in providing quite a lot of information about the user's browser, version, and operating system, in order that a website could choose to present different things based on that. However, this also increases the amount of distinguishing information made available, information which helps identify one user from another; and so such information is increasingly no longer available, or at least frozen so that it is no longer distinguishing. If what you do relies on this information—for example, if you're taking different code branches depending on the user-agent—then this code may break as browsers increasingly freeze or stop that information. Testing is the best defence here (see later).

Aside: measuring fingerprintability

The technical measure for how much information each of these data points provides is called *entropy* and is measured in bits. A feature where there are many different possible values (such as the list of fonts installed) can contribute a lot of bits to the total, whereby something without much

distinguishing power (such as which operating system you're using) may only add a few. The HTTP Almanac describes how existing fingerprinting libraries automate this process of combining the responses from many different APIs into a "hash", which may identify only a small group of users, perhaps even only one. Maud Nalpas covers this in some detail in this YouTube video, but, in short, imagine that you were to see a list of your friends with their favourite music, favourite food, and the languages they speak … but with their names removed. It is quite likely that any one person's list will uniquely identify them among your friends, or at least narrow down the list to only a few people. This is how fingerprinting works; the list of things you like becomes the "hash". With that hash, identifying a user as the same person on two different unconnected sites becomes easier, which is the goal of tracking: to circumvent the user's desire for privacy.

What do browsers do against fingerprinting?

Importantly, browser vendors are very aware of many different ways for a website (or a third party included on a website) to calculate a distinguishing fingerprint for a user, or for differing bits of information to contribute to the uniqueness of that fingerprint. Some of these ways are explicit and deliberate, such as the browser's user-agent string, which often identifies the browser, operating system, and version in use (and so contributes to distinguishing you from me, if you and I are using different browsers). Some of the ways are not deliberately created to be fingerprintable but end up being so anyway, such as the list of fonts, or video and audio devices available to the browser. (The browser doesn't have to *use* these devices, just get a list of them by name.) And some have been established to be contributors to a fingerprint well after release, such as the exact pixel rendering of antialiasing of fonts on a canvas element. There are many more, and each way in which your browser differs from mine adds entropy and so potentially contributes to a way to tell the difference between you and me, and identify an individual person as uniquely as possible across websites. https://amiunique.org has a long (but definitely not comprehensive) list of potential fingerprint-contributing features, and the list grows all the time (as there is heavy monetary interest in being able to fingerprint users, even if the users don't want that or perhaps don't expect it).

Not supporting certain powerful APIs

The response from browser vendors to all of these approaches to calculating a user's fingerprint is to find ways to reduce the amount of entropy available from these APIs. The most restrictive option is to not implement them in the first place. This has been done by some major browsers for a variety of hardware and device APIs (such as NFC and Bluetooth access from client-side web apps), with fingerprinting and privacy concerns cited as reasons why they have not been implemented.

This, obviously, can affect your apps and services: you can't use an API at all in a browser that doesn't implement it, and this can restrict or entirely cut off some hardware approaches from consideration.

The user permissions gateway

A second approach taken by browser vendors is to prevent API or data accesses without some sort of explicit user permission. This approach is often taken for security reasons as well—a website should not be able to take pictures with your webcam without your permission! But here, privacy and security can have similar interests. Identifying someone's location is a privacy violation in itself, of course, but it's also a contributor to the uniqueness of a fingerprint. Requiring permission to see geolocation does not decrease the extra entropy that a location adds to the uniqueness of that fingerprint, but it basically eliminates *using* geolocation for fingerprinting because it's no longer being done invisibly. The whole point of fingerprinting is to *surreptitiously* distinguish users from one another. If you are prepared for the user to know that you're trying to identify them, then you don't need fingerprinting techniques: have the user create an account and log in with it.

Adding unpredictability

A third approach taken in some cases is for browser vendors to "fuzz" the responses from APIs to make them less granular and thus less identifying. This was described as part of the randomized response mechanism in the data module as something you can do when collecting data from users, to avoid inadvertently collecting data which is identifying. Browser vendors can take this approach to API data made available to web apps and third parties as well. An example of this is the very accurate timing APIs used to measure page performance from `window.performance.now()`. The browser knows these values to microsecond accuracy, but the values returned are deliberately reduced in precision by rounding to the nearest 20 microsecond boundary to avoid their use in fingerprinting (and also for security to avoid timing attacks, admittedly). The goal here is to ensure the APIs remain useful, but to make the responses less identifying: in essence, to provide "herd immunity" by making your device look more like everyone else's device rather than being peculiar to you. Safari presents a simplified version of the system configuration for this very reason.

Enforcing a privacy budget

The Privacy Budget is a proposal that suggests that browsers estimate the information revealed by each fingerprinting surface. It hasn't been implemented yet in browsers. The goal is to allow powerful APIs while preserving user privacy. Learn more about the privacy budget proposal.

Use a broad testing environment

All of these will affect how you build apps and services. In particular, there is a wide diversity of responses and approaches across browsers and platforms in this area. This means that testing your work in multiple different environments is *critical*. This is, of course, always important, but it can be reasonable to assume that HTML rendering or CSS will be constant for a given rendering engine, no matter which browser or platform that engine is in (and so it can be tempting to test in only one Blink-based browser, for example). This is emphatically not the case for API usage precisely because browsers that share a rendering engine may differ dramatically on how they harden their API surface against fingerprinting.

Do

- Check your own analytics and audience to guide the set of browsers you should prioritize when testing.
- A good set of browsers to test in are Firefox, Chrome, Edge, Safari on desktop, Chrome and Samsung Internet on Android, and Safari on iOS. This ensures you test across the three major rendering engines (Gecko in Firefox, various forks of Blink in Chrome, Edge, and Samsung Internet, and Webkit in Safari), and on both mobile and desktop platforms.
- If your site may also be used on less common devices, such as tablets, smartwatches, or gaming consoles, test there as well. Some hardware platforms can lag behind mobile and desktop for browser updates, meaning that some APIs may be unimplemented or unavailable in the browsers on those platforms.
- Test with one or more browsers which claim user privacy as a motivator. Include upcoming pre-release and test versions of your most common browsers where you can and if they're available to you: Safari's technology preview, Chrome's Canary, Firefox's Beta channel. These give you the best chance of identifying API breakage and changes that affect your sites before those changes affect your users. Similarly, bear your users' environments in mind with reference to any analytics you have present. If your user base has high numbers of older Android phones, be sure to include those in your tests. Most people do not have the fast hardware and latest releases that a development team does.
- Test using both a clean profile and in incognito/private browsing mode; the likelihood is that you have already granted the permissions required in your personal profile. Test what happens if you deny permission to the site for any question.
- Explicitly test your pages in Firefox's fingerprinting protection mode. Doing so will show permission dialogues if your page is attempting fingerprinting, or will return fuzzed data for some APIs. This helps you confirm if third parties included in your service are using fingerprintable data, or if your own service depends on that. You can then consider whether the deliberate fuzzing makes it more difficult to do what you need. Consider making corrections accordingly to obtain that data from another source, do without it, or use less granular data.

- As previously discussed in the third-parties module, it's also important to audit your third-party dependencies to see if they are using fingerprinting techniques. Passive fingerprinting is difficult to detect (and impossible if a third party does it on their server) but fingerprinting mode may flag some fingerprinting techniques, and looking for uses of navigator.userAgent or unexpected creation of `<canvas>` objects may also reveal some approaches that deserve further scrutiny. It is also worth looking for uses of the term "probabilistic matching" in marketing or technical material describing a third party; this can sometimes indicate the use of fingerprinting techniques.

Cross-browser testing tools

Testing your code for privacy purposes is difficult to automate, and the things to look for when testing manually are described earlier. What happens when you deny permission to the site for any APIs it tries to access, for example, and how is that presented to the user? An automated test cannot judge whether the site acts so as to help the user trust it, or conversely to encourage the user to distrust it, or think that something is being hidden.

However, once the site has been audited, the testing of APIs to confirm that nothing has broken in newer browser versions (or in upcoming "beta" and "preview" versions) can be automated, and largely should be as part of your existing test suite. Something to consider with your automated test tools, when working with API surface coverage, is that most browsers allow some level of control over which APIs and features are available. Chrome does so through command line switches, as does Firefox, and having access to these in the testing tool setup will allow you to run certain tests with APIs turned off or on. (See, for example, Cypress's browser-launch plugin a nd puppeteer's launch.args parameter for ways to add browser flags when launching.)

Only rely on the user-agent string for coarse information

Taking another example, browsers have since the beginning of the web sent a description of themselves with every request in the HTTP User-Agent header. For nearly as long, people have been exhorting web developers to not use the contents of the user-agent header to serve different content to different browsers, and for all that time web developers have done it anyway, with a certain amount of justification in some (but not all) cases. Since browsers don't want to be singled out for a suboptimal experience by websites, this results in every browser pretending to be every other browser, and the user-agent string looking something like:

`Mozilla/5.0 (Linux; Android 6.0.1; SGP771 Build/32.2.A.0.253; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/52.0.2743.98 Safari/537.36`.

This claims, among other things, to be Mozilla/5.0, a browser which was released at the same time that the first astronauts boarded the International Space Station over two decades ago. The user-agent string is a rich source of entropy for fingerprinting, of course, and to mitigate that fingerprintability, browser manufacturers have either already frozen the user-agent header or are working towards doing so. This is another example of changing the data that an API provides without necessarily removing the API entirely. Sending a blank user-agent header would break countless websites which assume that it is present. So in general, what browsers are doing is removing some of the detail from it, and then keeping it mostly unchanged from then on. (You can see this happening in Safari, Chrome, and Firefox.) This protection against detailed fingerprinting essentially means that you cannot rely on the user-agent header being accurate anymore, and if you are doing so then it's important to find alternative data sources.

To be clear, the data in the user-agent isn't going entirely away, but it is available at a lower granularity, or is sometimes inaccurate because an older but unchanging number may be reported. For example, Firefox, Safari, and Chrome all cap the reported macOS version number to ten (see Update on user-agent string reduction for more discussion here). The exact details for how Chrome plans to reduce data in the user-agent string are available at User-Agent Reduction but, in short, you can expect that the reported browser version number will only contain a major version (so the version number will look like 123.0.0.0, even if the browser is version 123.10.45.108), and the OS version will be without detail and will freeze to one of a small number of unchanging choices. So an imaginary Chrome version 123.45.67.89 running on an imaginary "Windows 20" would report its version number as:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36
```

The core information you need (the browser version) is still available: it is Chrome 123, on Windows. But the subsidiary information (the chip architecture, which Windows version, which version of Safari it's imitating, the browser minor version) will no longer be available after the freeze.

Compare this with a "current" Chrome user-agent on a different platform:

`Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36`,

and it can be seen that the only thing that differs is the Chrome version number (104) and the platform identifier.

Similarly, Safari's user-agent string does show a platform and a Safari version number, and also gives an OS version on iOS, but all else is frozen. So an imaginary Safari version 1234.5.67 running on an imaginary macOS 20 might give its user-agent as:

`Mozilla/5.0 (Macintosh; **Intel Mac OS X 10_20_0**) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/15.1 Safari/605.1.15`,

and on an imaginary iOS 20 it might be:

`Mozilla/5.0 (iPhone; CPU **iPhone OS 20_0** like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/**20.0 Mobile/15E148 Safari/605.1.15**`.

Again, the core information (this is Safari, it's on iOS or macOS) is available, and iOS Safari still provides an iOS version number; but much of the ancillary information that was available in the past has since been frozen. Importantly, this includes the Safari version number, which is not necessarily available.

The changes to the reported user-agent were hotly debated. https://github.com/WICG/ua-client-hints#use-cases summarises some of the arguments and reasons for the change, and Rowan Merewood has a slide deck with some strategies for migrating away from using the user-agent for differentiation, in the context of the UA Client Hints proposal explained further on.

Fuzzing

Fuzzing is a term from security practice, where APIs are called with unexpected values in the hope that they handle those unexpected values badly and expose a security issue. Web developers should be familiar with cross-site scripting (XSS), which involves adding malicious script to a page, often because the page doesn't correctly escape injected HTML (so you do a search query with the text `<script>` in it). Back-end developers will be aware of SQL injection, where database queries that don't correctly validate user input expose security issues (as notably illustrated by xkcd with Little Bobby Tables). Fuzzing, or fuzz testing, is more properly used for automated attempts to provide many different invalid or unexpected inputs to an API and to check the results for security leaks, crashes, or other bad handling. These are all examples of deliberately providing inaccurate information. Here, though, it's being done preemptively by browsers (by making the user-agent deliberately incorrect), to encourage developers to stop relying on that data.

Do

- Check your codebase for any reliance on the user-agent string (a search for `navigator.userAgent` is likely to find most occurrences in your client-side code, and your backend code will likely be looking for `User-Agent` as a header), including your dependencies.
- If you find uses in your own code, work out what the code is checking for, and find another way to do that differentiation (or find a replacement dependency, or work with the dependency upstream by filing issues or checking with them for updates). Sometimes browser differentiation is necessary to work around bugs, but the user-agent will increasingly become not the way to do this once it is frozen.
- You may be safe. If you’re only using the core values of brand, major version, and platform, then these will almost certainly still be available and be correct in the user-

agent string.

- MDN describes good ways to avoid reliance on the user-agent string ("browser sniffing"), the main one of which is feature detection.
- If you are reliant on the user-agent string in some way (even when using the few core values that remain useful), it's a good idea to test with upcoming user-agents that will be in new browser releases. It is possible to test with those upcoming browser versions themselves by means of beta or technology preview builds, but it's also possible to set a custom user-agent string for testing. You can override the user-agent string in Chrome, Edge, Firefox, and Safari, when doing local development, in order to check how your code deals with different user-agent values that you might receive from users.

Client Hints

One major proposal to provide this information is User-Agent Client Hints, although this is not supported across all browsers. Supporting browsers will pass three headers: `Sec-CH-UA`, which gives a browser brand and version number; `Sec-CH-UA-Mobile`, which indicates if the request comes from a mobile device; and `Sec-CH-UA-Platform`, which names the operating system. (Parsing these headers is less easy than it seems because they are Structured Headers rather than simple strings, and this is enforced by browsers sending "tricky" values, which will be handled incorrectly if not properly parsed. This is, as earlier, an example of "fuzz testing" being done preemptively by the browser. A developer using this data is required to handle it properly because the data is designed so that improper or lazy parsing will likely give bad results, such as showing brands that don't exist, or strings that don't close properly.) Fortunately, this data is also made available by the browser to JavaScript directly as `navigator.userAgentData`, which in a supporting browser might look something like this object:


```
{
  "brands": [
□{
  □"brand": " Not A;Brand",
  □"version": "99"
□},
□{
  □"brand": "Chromium",
  □"version": "96"
□},
□{
  □"brand": "Google Chrome",
  □"version": "96"
□}
```

```
  ],
  "mobile": false
}
```

□□□ #2
□ □□□ □□□ 2026-10-31 22:54:36 UTC
□ □□□ □□□ 2026-10-31 22:54:39 UTC