CSE 361: Web Security

Web tracking

Nick Nikiforakis
1993

“On the Internet, nobody knows you’re a dog.”

2015

“Remember when, on the Internet, nobody knew who you were?”
Stateful tracking

- Stateful tracking:
  - First-party / third-party cookies
  - Evercookies
  - Cache-based mechanisms
  - HSTS tracking
  - Other exotic storage mechanisms

- Find a "corner" in the browser and stuff it with data
A cookie's life
Flash Cookies

- The Flash plugin had the ability to store information in Local Shared Objects (LSOs) that are separate from HTTP cookies
  - Commonly known as Flash cookies

- These LSOs are there to remember things about the user in the context of a Flash application, like remembering user preferences and game progress
Flash Cookies

• Flash cookies were the first technology that was used to supplement the traditional cookie-based third-party tracking

• Originally, when a user deleted his cookies, that would not have an effect on Flash cookies
  • Multiple stories started appearing in 2009 and 2010 where researchers and investigative journalists were finding websites that were using Flash cookies to remember
  • Flash cookies were often used to respawn traditional HTTP cookies, a practice which was named “zombie cookies”

• Browsers later collaborated with Flash so that “Delete all cookies” eventually included Flash cookies
Evercookies

• The idea of using Flash cookies to respawn traditional HTTP cookies was taken to the extreme with Samy Kamkar’s "evercookie"
  • https://github.com/samyk/evercookie

• evercookie is a javascript API that produces extremely persistent, respawnning cookies in a browser. Its goal is to identify a client even after they've removed standard cookies, Flash cookies (LSOs), HTML5 storage, SilverLight storage, and others.
The main idea is the same
  
  - Try to hide an identifier into as many “corners” of the browser as possible and rely on one copy to respawn the rest

Methods used
  
  - Standard HTTP cookies
  - Flash LSOs
  - Silverlight Storage
  - HTML5 storage
  - HTML5 indexdb
  - HSTS Pinning
Tracking through HSTS Pinning

• HSTS is normally used to increase a user’s security by having your browser connect to your website over HTTPS even if the user asked to be connected over HTTP

• People realized that HSTS could be used for tracking
  • How?
Tracking through HSTS

1. Generate a tracking identifier of desired length, e.g. 32 bits (allowing for $2^{32}$ users)
2. Break that identifier down into binary

\[
\begin{array}{cccccc}
1 & 0 & 1 & 1 & \ldots & 0 & 1 \\
\end{array}
\]

3. Have the user’s browser conduct 32 requests to different subdomains of your domain
   3. a.tracking.com
   4. b.tracking.com
   5. c.tracking.com
Tracking through HSTS (continued)

4. For every request sent, if the corresponding bit was 1, redirect to HTTPS and then send along an HSTS header telling the browser to remember that you want to be contacted only over HTTPS.

1. a.tracking.com ➔ https://a.tracking.com
2. b.tracking.com ➔ http://b.tracking.com
3. c.tracking.com ➔ https://c.tracking.com
4. Etc.
Tracking through HSTS (continued)

5. When the user returns, try to load all 32 domains again (over HTTP), and observe which requests will come to you over HTTPS (browser honoring your previous instructions communicated via the appropriate HSTS header)

   1. https://a.tracking.com → 1
   2. http://b.tracking.com → 0
   3. https://c.tracking.com → 1

   ...

6. Based on these requests you can now reconstruct the original identifier
What is browser fingerprinting?

- **Browser fingerprinting** refers to the extraction of properties from a user's browsing environment and the use of these properties to differentiate that user from other users.
  - Distinctly different from cookies and other **stateful tracking** methods

- Usage scenarios:
  - Track users against their will, even when they delete cookies / use private mode (stateless tracking)
  - Differentiate between a legitimate owner of an account and someone impersonating that owner
  - Differentiate between bots and legitimate web users
    - Particularly when bots try to hide
Web bots

- Web bots are programs that interact with websites in automated ways
  - Benign bots
    - Page indexing, link previews, malware detection
  - Malicious bots
    - Scraping, brute-forcing credentials, stealing backup/configuration files, exploiting vulnerabilities

Source: Imperva Bot Report, 2021
Evidence of fingerprinting
Back to fingerprinting

- Browser fingerprinting taps into a browsing environment's inherent variability
  - **Browser variability**
    - Interchangeable != Identical
    - Can we differentiate between Mozilla Firefox, Google Chrome, and wget?
  - **User variability**
    - User properties and choices that can be identified, even when using a common browser
      - System specs
      - Hardware
      - Installed fonts
      - Browser extensions
Declared identity

- As you browse the web, your browser declares its identity to the webpages you visit
  - When they ask as well as when they don't

- Main mechanisms
  - HTTP Request Headers
  - Special JavaScript objects
Typical HTTP Request headers - Firefox

Host www.securitee.org
User-Agent Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/112.0
Accept text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language en-US,en;q=0.5
Accept-Encoding gzip, deflate, br
Connection keep-alive
Upgrade-Insecure-Requests 1
Sec-Fetch-Dest document
Sec-Fetch-Mode navigate
Sec-Fetch-Site none
Sec-Fetch-User ?1
Typical HTTP Request headers - Chrome

Host        www.securitee.org
Connection   keep-alive
sec-ch-ua   "Google Chrome";v="111", "Not(A:Brand);v="8", "Chromium";v="111"
sec-ch-ua-mobile ?0
sec-ch-ua-platform  "Windows"
Upgrade-Insecure-Requests  1
User-Agent  Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.0.0 Safari/537.36
Accept      text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site none
Sec-Fetch-Mode navigate
Sec-Fetch-User ?1
Sec-Fetch-Dest document
Accept-Encoding gzip, deflate, br
Accept-Language en-US,en;q=0.9
Typical HTTP Request headers - curl

Host    www.securitee.org
User-Agent    curl/7.68.0
Accept    */*
Declared identity: Client-side

- Via JavaScript, the browser offers rendered web pages additional information about itself and your system

- Reasons that JS may want to know
  - Avoid APIs that are not implemented in your browser
  - Reflow content according to the size of your device and your screen orientation
  - Avoid CPU/battery hungry operations on mobile devices

- `navigator` object: information about the browser
- `screen` object: information about the screen/display
Browser fingerprinting: typical ingredients

- Since 2010, we have known that JavaScript reveals way too much information about our browsers

Q: Which of the above are user-specific vs. browser-specific?
How?

- navigator.userAgent
- navigator.appCodeName
- navigator.appName
- navigator.platform
- navigator.product
- navigator.productSub
- navigator.vendor

- screen.width
- screen.height
- screen.availWidth
- screen.availHeight
- screen.colorDepth
How?

navigator.plugins

(new Date()).getTimezoneOffset()
How?

- Fonts are trickier
  - No official JavaScript API to extract fonts

- Solution used by Eckersley in Panopticlick
  - Use Flash plugin where there exist font-discovering APIs

- Solution found later
  - Side-channel in JavaScript
Font Detection through JavaScript

<table>
<thead>
<tr>
<th>String</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_DO_NOT_NEED_FLASH</td>
<td>500 x 84</td>
</tr>
<tr>
<td>I_DO_NOT_NEED_FLASH</td>
<td>620 x 90</td>
</tr>
<tr>
<td>I_DO_NOT_NEED_FLASH</td>
<td>600 x 87</td>
</tr>
<tr>
<td>I_DO_NOT_NEED_FLASH</td>
<td>510 x 82</td>
</tr>
</tbody>
</table>

1. Create a large string for each font family that you want to test
2. Measure it’s width and height (offsetWidth, offsetHeight properties)
3. Compare it to the width and height of the Arial font family (typical fallback font when your browser does not have the one you asked for)
4. If dimensions are different, then the font-family is installed
Resulting fingerprints

- 94.2% of the users with Flash/Java could be uniquely identified
- Heuristic algorithms could track updates of the same browser
Canvas fingerprinting arose when the `<canvas>` element was introduced in HTML 5
- Used to draw graphics on a page via JavaScript
- 2D/3D graphics
- Games / photo editing using native JavaScript
Canvas element

- Canvas drawings can be very complicated
  - https://codepen.io/towc/pen/mJzOWJ
  - https://codepen.io/tsuhre/pen/BYbjyq

- After some point, our hardware will affect what we see
  - Type of graphics card
  - Drivers
  - Operating system

- Can we use that to fingerprint users?
  - Identical setups except graphics cards should result in different canvas fingerprints
Canvas fingerprinting recipe

1. Draw something (ideally complicated) in a canvas, read it out as an image

2. Different browsers/hardware combinations will create slightly different images

3. Use these differences to differentiate between users

Demo: https://jsbin.com/mopoxafodi/edit?html,js,console,output
Extension popularity and fingerprinting

- ABP: > 10M users
- Grammarly: > 10M users
- LastPass: > 5.7M users
- Honey: > 4.3M users
- Pocket: > 2.8M users
- Avira: > 10M users
- Save: > 10M users
- Hover Zoom: > 1.3M users
- HubSpot: > 0.6M users
Extensions are “more private”…

- Previous research showed that plugins were one of the most powerful features for browser fingerprinting:
  - https://panopticlick.eff.org
  - https://amiunique.org
- Plugins are fading away…
- In comparison to plugins, there is no API for a web page to enumerate available browser extensions!

Are extensions really undetectable?
Doesn’t seem to be the case…
Privacy and Security implications

• Discovering targets for known exploits in browser extensions
  • E.g. popular password managers (*LastPass*, *Blur*, etc.)
• Exposing sensitive extensions installed by browser users
  • E.g., *Mailvelope*, VPN extensions, discount alerts, political add-ons, etc.
Seasonal/joke extensions

I created a browser extension that puts masks on faces on the internet for a safer browsing experience. #coronavirus
Extensions as a fingerprinting feature?!

Ghostery

AdBlock Plus

UAControl

... 

SmartVideo

User 1

User 2

... 

User N
How can websites discover browser extensions?

- **Web Accessible Resources**
  - Resources within a browser extension explicitly marked as “web accessible” by developers

- **Presence of extension artifacts in a web page**
  - Evidence that an extension is present by extra content/missing content
Web Accessible Resources

- Extension resources (i.e. files) are not accessible to the general web
- Occasionally, browser extensions need to inject content into the pages where they are active
  - JavaScript libraries
  - Images
- These resources are therefore marked by developers as Web Accessible
Web Accessible Resources

• Intention
  • Wherever a file is necessary, drop a reference to that web-accessible resource, pulling it directly from the extension’s directory
    • `<img src="chrome-extension://[PACKAGE ID]/icons/icon24.png"/>`
    • `<script src="chrome-extension://[PACKAGE ID]/localLib/jQuery/jQuery-1.10.2.js"/>`

• Problem
  • Browsers do not differentiate between links added by the extension vs. those added by the page
    1. Page tries to load a given WAR of a given extension
    2. If the resource loads, that extension is present
Web Accessible Resources

• Attack
  1. Collect as many extensions as possible from an extension store
  2. Keep those that use at least one WAR
  3. Write a script that tries to load one WAR for each extension in the user’s browser
  4. Record success or failure

• How many extensions are fingerprintable via WARs?
  • 50% reported by Sjösten et al, CODASPY 2017
  • 27% reported by Gulyás et al, WPES 2018
  • 28.7% reported by Karami et al., NDSS 2020
Detecting extensions through their side effects

• Users install extensions to get additional functionality
  • Add features
    • Tab managers, password managers, reminders, proofreaders
  • Remove “features”
    • Trackers, ads, page bloat, offensive content
  • Change content
    • Simplify pages, change background color
YouTube – No Extensions

Hackers Official Trailer #1 - Matthew Lillard Movie (1995) HD

431,097 views • Oct 5, 2012
YouTube – Magic Actions extension (1M+ users)
LastPass installed

<input type="password" style="cursor: pointer; background-image: url(\"data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAABAAAAAA
...yeBAAAAAE1FTkSuQmCC\"); background-attachment: scroll; background-size: 16px 18px; background-position: 98% 50%; background-repeat: no-repeat;">
Extensions have visible side-effects!

- **ABP**: > 10M users
- **Grammarly**: > 10M users
- **LastPass**: > 5.7M users
- **Honey**: > 4.3M users
- **Pocket**: > 2.8M users
- **Avira**: > 10M users
- **Save**: > 10M users
- **Hover Zoom**: > 1.3M users
- **HubSpot Sales**: > 0.6M users
Results for top 10K Chrome extensions

• **9.2%** introduce detectable changes on any arbitrary URL  
  *(any webpage can fingerprint)*

• **16.6%** introduce detectable changes on popular domains  
  *(popular websites can fingerprint)*

![Graph showing the percentage of detectable extensions by popularity and fingerprintability.](image-url)
Types of DOM modifications

• Out of 1,656 detectable extensions, more than 86% have at least one completely distinct on-page side-effect!
• These DOM side-effects are a stable property of extensions as the fraction of detectable extensions remains the same even after 4 months

<table>
<thead>
<tr>
<th>Type</th>
<th>Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>New DOM node</td>
<td>78.7%</td>
</tr>
<tr>
<td>Changed attribute</td>
<td>41.6%</td>
</tr>
<tr>
<td>Removed DOM node</td>
<td>15.8%</td>
</tr>
<tr>
<td>Changed text</td>
<td>4.7%</td>
</tr>
</tbody>
</table>
WAR Fingerprinting  

- Easy to extract signatures  
- Easy to perform in a user’s browser  
- No user interaction necessary  
- Developers can straightforwardly get rid of WARs  

Behavioral Fingerprinting  

- Easy to perform in a user’s browser  
- Difficult to get rid of fingerprintable behavior  
- Difficult to extract signatures  
- User interaction may be necessary
The middle way

• Combine the best of both worlds
  • “Easy” extraction of fingerprints
  • No user interaction necessary
  • Difficult for developers to make their extensions non-fingerprintable

• CSS-based extension fingerprinting
  • Like web developers, extension developers want their extensions to look good
  • Use Cascading Style Sheets to do just that
    • When modifying a page, add some CSS to style the modifications
Styles and extensions (Dr. Web Link Checker)

```css
.r.drwebThreatLink {
  background-repeat: no-repeat;
  width: 86px;
  height: 84px;
  background-position: 0 0;
  background-image: url(data:image/png;base64,....);
}
```

```html
<div class="drwebThreatLink">(trigger)</div>
```
Style-based fingerprinting

- **Problem**: No differentiation between content injected by the extension vs. content that’s already there by the page
  - Would break some extensions if we did differentiate between the two
Style-based fingerprinting

• Attack:
  1. Focus on the extensions that inject CSS to pages
    • 98.5% of CSS are injected statically
  2. Identify CSS rules that translate to visible changes
  3. Inject decoy elements, at least one for each extension
  4. Measure whether these elements were “styled”
    • No user interaction necessary
Style-based fingerprinting results

• Out of 116K evaluated extensions
  • 6,645 inject style sheets on all websites
  • 3,475 have at least one CSS “trigger” that is unique to them
    • The remainder can still be differentiated from each other through the type of styling
  • 1,074 could not be fingerprinted via existing methods
• Fingerprinting done in ms
NON-DECLARED IDENTITY
Modern browsers are all-inclusive software platforms

• Modern browsers are constantly evolving
  • Streamlined extension frameworks
  • Push notifications
  • Custom web components
  • WebRTC
  • Payment Request API

• Rough size statistics (LOC = Lines of Code)
  • Google Chrome is 16 M. LOC
  • Mozilla Firefox is 18 M. LOC
  • Linux Kernel is 16.8 M. LOC
More features, greater fingerprintable surface

• As we keep on adding more and more features, we are expanding the fingerprintable surface of the browser
  • Same concept with traditional software vulnerabilities
  • More code leads to more bugs

• Despite following standards, browsers will behave in different and undocumented ways from on-another
  • Identify the behavior, reveal the browser
Namespace fingerprinting

• Browsers make various objects globally available to running scripts
  • Global variables
  • Global methods

• These are normally invisible to running scripts

• Is it possible that different browsers using different JS engines make different symbols globally accessible in their namespace?
What Is SSL / TLS?

- Secure Sockets Layer and Transport Layer Security protocols
  - Same protocol design, different crypto algorithms
- De facto standard for Internet security
  - “The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications”
- Deployed in every Web browser; also VoIP, payment systems, distributed systems, etc.
Setting up a TLS channel

Image credit: Cloudflare
Handshake Protocol Structure

ClientHello

ServerHello,
[Certificate],
[ServerKeyExchange],
[CertificateRequest],
ServerHelloDone

(Certificate],
ClientKeyExchange,
[CertificateVerify]

switch to negotiated cipher

Finished

Record of all sent and received handshake messages

switch to negotiated cipher

Finished
ClientHello

Client announces (in plaintext):
- Protocol version he is running
- Cryptographic algorithms he supports
- Fresh, random number
ClientHello (RFC)

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites;
    CompressionMethod compression_methods;
} ClientHello
Everyone's different

- Different TLS Clients implement things slightly differently
  - Chrome/Chromium support GREASE, a mechanism for catching interoperability issues between clients and servers
  - Firefox and Safari do not support GREASE
  - Command-line tools built using Python, curl, Perl, will have different TLS libraries than both Chrome and Firefox

```go
import "net/http"

resp, err := http.Get("https://example.com/")
```
Bots and TLS fingerprinting

- TLS fingerprinting works well against bots
  - 558 unique fingerprints shared over 10M requests
    - Small number of tools and libraries
- 86.2% of bots claiming Firefox/Chrome were fake
  - Matching signatures of curl, libwww-perl, Go, and Python
- Exploitation attempts do not match real browser fingerprints

<table>
<thead>
<tr>
<th>Tools</th>
<th>Unique FPs</th>
<th>IP Count</th>
<th>Total Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go-http-client</td>
<td>28</td>
<td>15,862</td>
<td>8,708,876</td>
</tr>
<tr>
<td>Libwww-perl or wget</td>
<td>17</td>
<td>6,102</td>
<td>120,423</td>
</tr>
<tr>
<td>Pycurl</td>
<td>26</td>
<td>3,942</td>
<td>80,374</td>
</tr>
<tr>
<td>Python-urlib3</td>
<td>8</td>
<td>2,858</td>
<td>22,885</td>
</tr>
<tr>
<td>NetcraftSurveyAgent</td>
<td>2</td>
<td>2,381</td>
<td>14,464</td>
</tr>
<tr>
<td>msnbot/bingbot</td>
<td>4</td>
<td>1,995</td>
<td>44,437</td>
</tr>
<tr>
<td>Chrome-1 (Googlebot)</td>
<td>1</td>
<td>1,836</td>
<td>28,082</td>
</tr>
<tr>
<td>Python-requests 2.x</td>
<td>11</td>
<td>1,063</td>
<td>754,711</td>
</tr>
<tr>
<td>commix/v2.9-stable</td>
<td>3</td>
<td>1,029</td>
<td>5,738</td>
</tr>
<tr>
<td>Java/1.8.0</td>
<td>8</td>
<td>308</td>
<td>1,710</td>
</tr>
<tr>
<td>MJ12Bot</td>
<td>2</td>
<td>289</td>
<td>28,065</td>
</tr>
<tr>
<td>Chrome-2 (Chrome, Opera)</td>
<td>1</td>
<td>490</td>
<td>66,631</td>
</tr>
<tr>
<td>Chrome-3 (Headless Chrome)</td>
<td>1</td>
<td>80</td>
<td>2,829</td>
</tr>
<tr>
<td>Chrome-4 (coc_coc_browser)</td>
<td>1</td>
<td>4</td>
<td>111</td>
</tr>
<tr>
<td>Total</td>
<td>113</td>
<td>38,239</td>
<td>9,879,326</td>
</tr>
</tbody>
</table>

**TABLE V: TLS fingerprint of malicious requests**

<table>
<thead>
<tr>
<th>Type</th>
<th>Python</th>
<th>Golang</th>
<th>libwww</th>
<th>Chrome</th>
<th>Firefox</th>
<th>Unknown</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backdoor</td>
<td>231</td>
<td>1,718</td>
<td>349</td>
<td>3</td>
<td>482</td>
<td>2,783</td>
<td></td>
</tr>
<tr>
<td>Backup File</td>
<td>411</td>
<td>171</td>
<td>84</td>
<td>0</td>
<td>1,803</td>
<td>2,469</td>
<td></td>
</tr>
<tr>
<td>Exploits</td>
<td>275</td>
<td>18,283</td>
<td>607</td>
<td>0</td>
<td>390</td>
<td>19,555</td>
<td></td>
</tr>
<tr>
<td>Fingerprinting</td>
<td>1,524</td>
<td>3,670</td>
<td>630</td>
<td>139</td>
<td>7,226</td>
<td>13,189</td>
<td></td>
</tr>
</tbody>
</table>

**Source:** Good Bot, Bad Bot: Characterizing Automated Browsing Activity