DOMPurify Security Advisory

Devansh Shah

Version SA-2026-0001, 2026-05-26

Table of Contents

1. Overview

DOMPurify versions 3.4.3 through 3.4.6 fail to sanitize attached shadow roots that are nested inside a `<template>` element when the caller invokes `DOMPurify.sanitize(node, { IN_PLACE: true })`. The shadow root and every node it contains - including inline event handlers, `javascript:` URIs, and tags that DOMPurify would otherwise strip - is left untouched. When the calling application later instantiates the template into the live document, which is the entire purpose of using a `<template>`, the malicious shadow tree is inserted along with it and executes as cross-site script.

Item	Details
Vendor	Cure53
Product	DOMPurify
Affected Versions	3.4.3 through 3.4.6 (HEAD `bb7739e`, 2026-05-26)
Vulnerability Type	CWE-79: Improper Neutralization of Input During Web Page Generation
CVSS v4.0 Score	8.5 (HIGH)
CVE IDs	Pending

1.1. Affected Products

Product	Affected Versions	Fixed Version
DOMPurify	3.4.3 through 3.4.6	Pending coordinated disclosure

2. IN_PLACE Sanitization Skips Attached Shadow Roots Inside `<template>.content`

CVE ID	Pending
CWE	CWE-79: Improper Neutralization of Input During Web Page Generation
CVSS v4.0	8.5 (`AV:N/AC:L/AT:P/PR:N/UI:N/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N`)
Severity	HIGH

2.1. Description

The vulnerability is exposed by any application that hands a DOM tree to `DOMPurify.sanitize(node, { IN_PLACE: true })` when that tree contains a `<template>` element whose `.content` includes a node carrying an attached shadow root, and which subsequently instantiates the template into the live document via the standard `tpl.content.cloneNode(true)` stamping pattern. This combination appears in Web Components frameworks (Lit, Stencil, FAST, Open WC) that stamp out component instances from blueprint templates, in rich-text and WYSIWYG editors that snapshot a DOM subtree for undo or copy/paste while preserving node identity, in low-code page builders and design-system tools that hold user-authored components as templates for later instantiation, and in any custom-element loader that accepts user content through declarative shadow DOM. Attached shadow roots in the input arrive through the canonical Web Components route of `element.attachShadow(…)` on a programmatically constructed node, or through `Document.parseHTMLUnsafe` and `Element.setHTMLUnsafe`, which materialize declarative shadow DOM markers (`<template shadowrootmode="open">`) into real attached shadow roots during parsing. For the cloned subtree to carry the malicious shadow root through the stamping step, the shadow root must be marked clonable, either via the `{ clonable: true }` option to `attachShadow` or via the `shadowrootclonable` attribute when using declarative shadow DOM.

To sanitize a DOM tree in IN_PLACE mode, DOMPurify has to cover two blind spots in the browser's `NodeIterator`. The iterator does not enter `HTMLTemplateElement.content` because template content lives in a separate document. The iterator also does not enter attached shadow roots, because shadow trees are isolated from the host tree. DOMPurify works around each blind spot with its own walk.

The first workaround is the pre-pass function `_sanitizeAttachedShadowRoots`. It traverses the input subtree looking for elements with attached shadow roots and feeds every one it finds into `_sanitizeShadowDOM`. The pre-pass descends through the tree using `Element.childNodes` only.

The second workaround lives in the main iterator. When the iterator visits a `<template>`, DOMPurify explicitly recurses into `template.content` by calling `_sanitizeShadowDOM(template.content)`. Inside that recursion the function inspects each iterated node for further nested template content, but it does not inspect those nodes for attached shadow roots.

Neither workaround visits an attached shadow root that lives inside `template.content`. The pre-pass cannot see template content because template content is not part of `Element.childNodes`. The template-content recursion sees the element that hosts the shadow root, but never reads its `shadowRoot` property. As a result, the subtree under that shadow root is reached by no walk at all, and every attribute and child node inside it survives the sanitization call untouched.

2.2. Affected Code

The first gap is in `_sanitizeShadowDOM` at `src/purify.ts:1733-1761`. The loop sanitizes each iterated element and recurses into `shadowNode.content` for nested templates, but never looks at `shadowNode.shadowRoot`.

```
while ((shadowNode = shadowIterator.nextNode())) {
  _sanitizeElements(shadowNode);
  _sanitizeAttributes(shadowNode);
  if (_isDocumentFragment(shadowNode.content)) {   // template-content recursion only
    _sanitizeShadowDOM(shadowNode.content);
  }
  // missing: an inspection of shadowNode.shadowRoot
}
```

The second gap is in `_sanitizeAttachedShadowRoots` at `src/purify.ts:1793-1834`. The pre-pass descends via `childNodes` only, which never enters `template.content`.

```
const childNodes = getChildNodes ? getChildNodes(root) : (root as Element).childNodes;
// missing: when root is a <template>, also recurse into root.content
```

2.3. Proof of Concept

The following self-contained page is a minimal reproducer. Chromium and Firefox display the `alert(1)` dialog when the page loads.

```
<!doctype html>
<title>DOMPurify 3.4.6 IN_PLACE bypass</title>
<body>
<script src="./purify.min.js"></script>
<script>
  // 1. Build a template whose content contains a div with an attached shadow root.
```

```
  //    The shadow root carries an <img onerror=alert(1)>.
  const tpl  = document.createElement('template');
  const host = document.createElement('div');
  host.attachShadow({ mode: 'open', clonable: true })
      .innerHTML = '<img src=x onerror=alert(1)>';
  tpl.content.appendChild(host);

  // 2. Sanitize the template in place. DOMPurify believes the tree is now safe.
  DOMPurify.sanitize(tpl, { IN_PLACE: true });

  // 3. The application instantiates the template into the live document.
  //    The shadow tree is cloned along with the host (clonable: true).
  //    The <img> fails to load, onerror fires, and alert(1) executes.
  document.body.appendChild(tpl.content.cloneNode(true));
</script>
```

After step 2, the shadow root inside `tpl.content` is unchanged. The `onerror` attribute that DOMPurify was supposed to strip is still attached to the `<img>`.

```
template.content.firstChild.shadowRoot.innerHTML:
  <img src="x" onerror="alert(1)">
```

2.4. Impact

The attacker achieves arbitrary JavaScript execution in the calling application's origin. The full impact set is the standard one for in-origin cross-site script execution: theft of session cookies and stored tokens, stored XSS that persists across reloads, registration of malicious service workers for long-lived persistence, in-origin phishing using the legitimate domain in the address bar, and lateral movement at the victim's privilege level inside the application.

3. Suggested Fix

The vulnerability is closed by extending each of the two walks so that they cover the other walk's blind spot. Either patch on its own closes the most direct attack path. Both patches together close the template-in-shadow and shadow-in-template variants symmetrically.

```
// Patch 1. In _sanitizeAttachedShadowRoots, after the existing childNodes
// walk, also recurse into <template>.content so the pre-pass enters
// template-owned subtrees.
const tagName =
  getNodeName && typeof getNodeName(root) === 'string'
    ? transformCaseFunc(getNodeName(root))
    : '';
if (tagName === 'template') {
  const content = (root as HTMLTemplateElement).content;
  if (_isDocumentFragment(content)) {
    _sanitizeAttachedShadowRoots(content);
  }
}

// Patch 2. Inside _sanitizeShadowDOM's iteration loop, inspect each iterated
// element for an attached shadow root and walk it. Guard on element nodeType,
// because the iterator also returns text, comment, CDATA, and processing-
// instruction nodes, and Element.prototype.shadowRoot is illegal on those.
const nt = getNodeType ? getNodeType(shadowNode) : (shadowNode as Node).nodeType;
if (nt === NODE_TYPE.element) {
  const sr = getShadowRoot
    ? getShadowRoot(shadowNode)
    : (shadowNode as Element).shadowRoot;
  if (_isDocumentFragment(sr)) {
    _sanitizeAttachedShadowRoots(sr);
    _sanitizeShadowDOM(sr);
  }
}
```