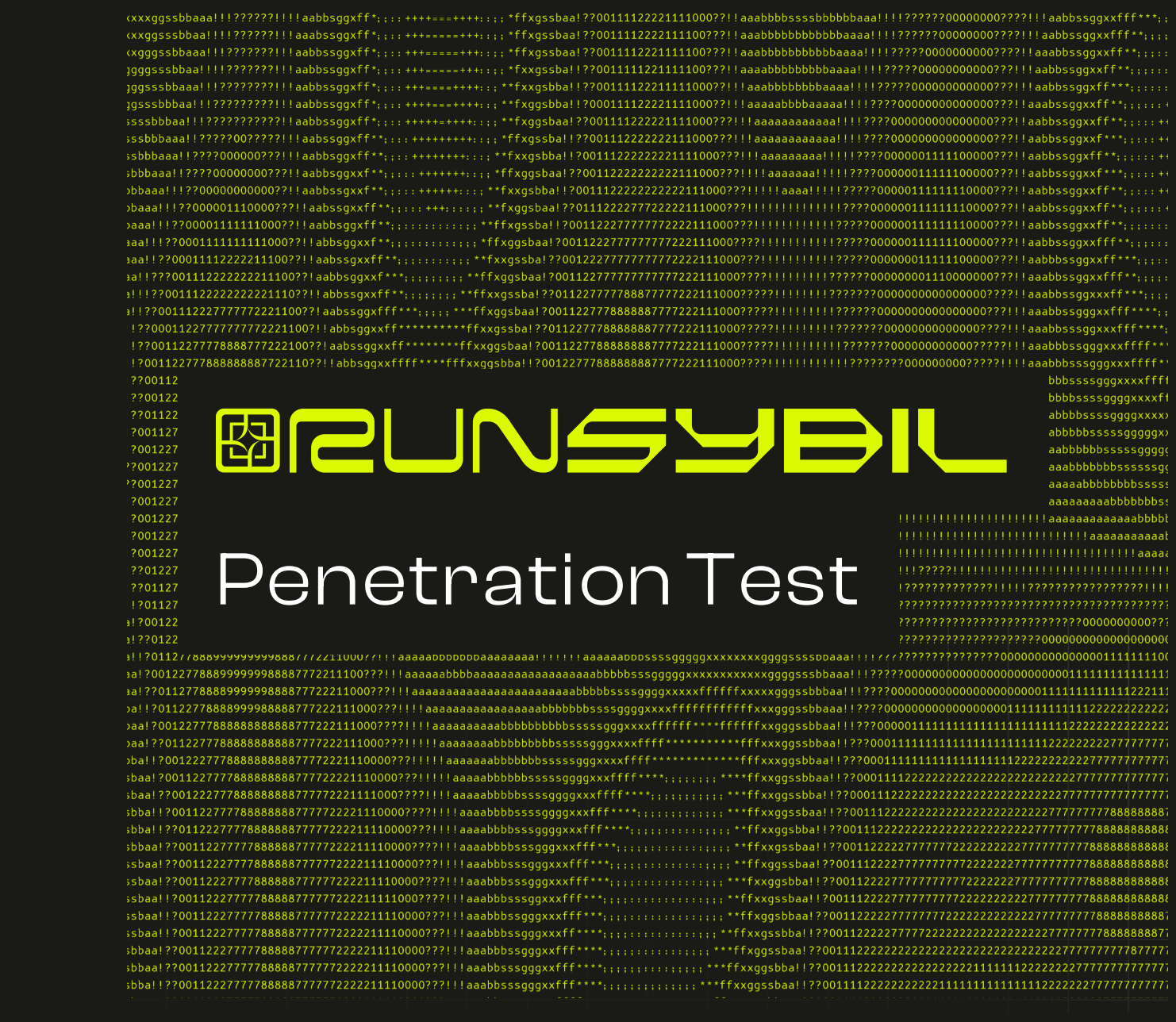

Penetration Test

July 16, 2026

2026 Portal Pentest

Acme Corp

Table of Contents

Executive Summary

- Scope
- Findings Overview

Pentest Narrative

Threat Ranking Methodology

Findings Summary

- 1. Unauthenticated Authentication Cookie Generation via /admin/Home/ExtendUserSession
- 2. Mass Assignment on Public Registration Endpoint Allows Unauthenticated Privilege Escalation via IsInternalUser Field
- 3. OS Command Injection in _testReward JSON Parameter
- 4. GraphQL Introspection Bypass via Apollo Federation Service Endpoint
- 5. Insecure Direct Object Reference in IAM Groups Endpoint Allows Cross-Tenant Data Access
- 6. Path Traversal in Checkpoint Archive Endpoint Parameter
- 7. SQL Injection in FeatureDemoLookup via platform parameter
- 8. Exposed phpinfo() Debug Page in /tmp/ Directory
- 9. Reflected Cross-Site Scripting via javascript: URI in VideoPlayer iframe src Parameter
- 10. Business Logic Flaw: Multiple Tax IDs of Same Type Allowed Per Customer

Appendix A: Assessment Scope Overview

- Targets in Scope
- Access Only Targets
- Targets Out of Scope

Appendix B: Manual Application Testing and OWASP Testing Methodology

Executive Summary

A comprehensive pentest of the main application portal.

The assessment followed RunSybil's standard methodology for penetration tests and findings were aligned with the Common Vulnerability Scoring System (CVSS), offering a systematic and uniform approach to evaluating the severity and potential impact of the identified vulnerabilities.

Scope

This assessment covered the Acme Portal. Detailed scope information including specific URLs or components is listed in Appendix A.

Findings Overview

During the application penetration test, 10 issues were identified within the authorized scope. The breakdown by severity is as follows:

Critical vulnerabilities requiring immediate attention were identified.

Pentest Narrative

Executive Summary

A comprehensive automated security assessment was conducted against the target platform. The application is a multi-tenant SaaS platform built on a modern JavaScript framework, offering cloud-hosted data management capabilities with a Bring Your Own Cloud (BYOC) deployment model, third-party billing integration, SSO-powered authentication, and a rich API surface including REST, tRPC, and GraphQL endpoints. Testing was structured around two provisioned tenant organizations — referred to throughout as OrgA and OrgB — and spanned the full breadth of the application from unauthenticated public surfaces to authenticated admin functionality.

Over the course of the engagement, 2,420 individual test runs were executed. What follows is a narrative account of how testing progressed, what areas were examined, and the logic behind each phase.

Mapping the Attack Surface

Before any active exploitation was attempted, testers spent considerable effort understanding the application from the outside in. The first priority was extracting as much intelligence as possible from the application's publicly accessible assets.

The target application, like most modern JavaScript framework applications, ships a large volume of JavaScript to the browser — login pages, dashboard layouts, contact and join flows, pricing pages, and a long tail of webpack chunk files numbered into the thousands. Each of these was fetched and analyzed for hardcoded secrets, internal API route paths, server action IDs, and configuration values that could inform later attacks. Source maps, which translate minified bundle code back into readable source, were extracted systematically across the entire chunk range. These are a particularly high-value target because developers often forget to disable them in production, and they can expose secrets, internal service names, and authentication logic that would otherwise be opaque.

Alongside bundle analysis, testers performed forced browsing — systematically probing URL paths that are not linked from the application but may exist and respond. This was done against admin route patterns, BYOC-specific paths, unauthenticated pages, and undocumented deep routes, all using both authenticated sessions (UserA and UserB) and no session at all. The application's configuration and environment file paths were also requested directly to check for accidental exposure.

Breaking Into the Application

With the surface mapped, testing turned to the authentication layer — the front door of any multi-tenant application and a common source of critical vulnerabilities.

The platform uses a third-party identity provider for authentication, with magic links (passwordless email authentication) as the primary login mechanism and OTP as a secondary factor. Both were attacked from multiple angles. Magic link generation was tested to see if it could be triggered without authentication — a condition that would allow an attacker to initiate authentication on behalf of any email address, potentially poisoning inboxes or harvesting valid user identifiers. Rate limiting on magic link and OTP flows was probed to determine whether a brute force or enumeration attack could be sustained before throttling kicked in. The OTP endpoint was further tested as an email enumeration oracle: subtle differences in response behavior for known-good versus unknown email addresses can reveal whether an account exists.

The OAuth integration was examined for state parameter vulnerabilities — specifically, time-of-check-to-time-of-use (TOCTOU) races and CSRF via the authentication callback. JWT tokens were tested for bearer replay across sessions and for admin claim manipulation, checking whether a standard user could forge elevated permissions. Session cookies themselves were assessed for forgery potential and proper security attributes.

A full account takeover chain was constructed and executed: extracting email addresses from one part of the application, feeding them into authentication triggers, and tracing how far an attacker could progress toward full account control without having the target's credentials.

Attacking Tenant Boundaries

The most consequential class of vulnerabilities in any multi-tenant platform is the failure of tenant isolation — one customer reading or modifying another's data. This was the central focus of the engagement.

Testing operated across both OrgA and OrgB throughout. With a session belonging to OrgA, testers methodically attempted to access every significant resource belonging to OrgB: dashboard pages, resource configurations, API keys, billing records, usage statistics, invoices, usage alerts, team members, settings, integrations, and webhooks. The same was repeated in reverse. Each test represented a discrete insecure direct object reference (IDOR) scenario: can I read your data? Can I write to it?

Beyond simple object substitution, the impersonation endpoint — designed for legitimate administrative use — was attacked with creative variations: alternative body formats, HTTP method overrides, parameter pollution, and RSC path-based cross-org substitution. Identity provider user IDs were tested as alternative identifiers to reach RSC payloads directly. The tRPC API layer, which underpins much of the real-time dashboard functionality, was tested for cross-tenant IDOR against integration and resource endpoints specifically.

Admin routes — audit logs, billing admin, organizations, users, and the admin root — were probed via React Server Component (RSC) payload requests. RSC is a framework feature that delivers server-rendered data payloads to the client; if the authorization checks on these payloads are misconfigured, a standard user can silently escalate to admin-level data access simply by constructing the right URL.

Escalating Privileges

Parallel to the cross-tenant work, testers pursued privilege escalation within a single tenant — attacking the role and permission model to elevate a standard user to an admin, or to claim entitlements the account was not provisioned for.

Role self-assignment was attempted via the admin user management endpoint and through billing parameter manipulation in org settings. RBAC gaps were hunted across team management, the org join flow, server action authorization checks, and invite action permissions. During invite creation, role injection was tested — supplying a higher-privileged role value in a parameter that should be server-controlled. Plan header manipulation was tested as a way to signal a higher subscription tier to the server without actually paying for it. Parameter pollution was used against audit log and admin endpoints to see if duplicate or conflicting parameters could confuse authorization logic into granting access.

Abusing the Invite System

The invite flow, which allows organization members to bring in new users, represents a complex state machine that is frequently prone to logic flaws. It received dedicated, thorough attention.

Invite tokens were tested for replay — using a token intended for one organization to gain membership in another. Token freshness validation was probed to determine whether expired or consumed tokens could be reused. Email validation bypass within invite chains was tested to see whether the invitation system's email ownership checks could be circumvented. Email enumeration via API invite responses was assessed. The full invite flow for OrgA was mapped in detail to identify all server actions in the chain and their parameter surfaces, and race conditions were tested on both the join-new-org flow and the registration endpoint to look for double-execution vulnerabilities.

Finding Paths to Internal Infrastructure (SSRF)

Server-Side Request Forgery was one of the broadest and most technically varied areas of the engagement. The attack premise is straightforward: trick the server into making HTTP requests on the attacker's behalf, potentially reaching cloud metadata services, internal databases, or Kubernetes control planes.

Two primary entry points stood out. The first was an image generation endpoint that accepts a URL parameter to fetch remote content. This was targeted with a comprehensive SSRF payload library targeting AWS IMDSv1 (`169.254.169.254`), GCP metadata (`metadata.google.internal`), Azure IMDS, and Alibaba Cloud metadata endpoints. A full exploitation chain was executed against this endpoint. The second major entry point was the framework's built-in image proxy, which fetches and optimizes remote images. This proxy was tested for cloud metadata access, internal hostname allowlist bypass, localhost variants, IPv6 representation, and alternative protocol handlers.

BYOC configuration flows — where users supply custom cloud endpoint URLs — were a natural SSRF surface and were tested against all the same metadata targets. Webhook URL fields and

integration settings URL fields for both organizations were tested similarly, as user-supplied URLs that the server may fetch.

Across all entry points, a comprehensive set of SSRF bypass techniques was applied: decimal, octal, and hex IP representations; double URL encoding; IPv6-mapped IPv4 addresses; DNS rebinding using public resolvers; file URI and alternative scheme handlers; Gopher and Dict protocol handlers; URL authority confusion; redirect chains; and path traversal through any proxy behavior. Port scanning was conducted to enumerate what internal services might be reachable — targeting database ports, Kubernetes API server ports, and a broad range of other high-priority internal addresses.

Injection Testing

Every significant user-controlled input in the application was assessed as a potential injection point.

XSS testing covered stored and reflected variants. High-value injection targets included dashboard input fields, settings pages, API key name fields, team management fields, integration configuration values, data returned from the backend, resource configuration, and contact form fields. BYOC configure and setup flows were specifically targeted for stored XSS — a particularly dangerous variant because the malicious payload persists in the database and executes for every user who views the affected page.

The email subscribe endpoint — likely a marketing or compliance form — was tested for SQL injection, NoSQL injection, Server-Side Template Injection (SSTI), CRLF injection, email header injection, and XSS. The analytics ingest proxy was tested for forged event injection, HMAC signature bypass, array-based event injection, and CORS policy abuse. Namespace fields were tested for injection, and integer overflow was tested against two specific query parameters (`q` and `w`) that suggest numeric processing.

Business Logic & Billing Attacks

The billing subsystem was treated as a primary attack target, consistent with how consequential billing vulnerabilities tend to be for SaaS platforms.

The payment provider's setup intent server action was tested three ways: called with no authentication at all to check for zero-auth execution, called with OrgB credentials against OrgA's setup intent to check for cross-org access, and exercised with live authenticated sessions for full exploitation confirmation. Billing plan IDs were substituted in subscription upgrade API calls to attempt plan escalation — gaining a higher-tier plan without paying for it. Cross-org billing IDOR was tested across usage alerts, usage-exceeded notifications, and invoices. The billing callback URL was tested for open redirect, which could be used to silently redirect users away from the platform after completing a billing action. Payment session state injection was tested via callback parameter manipulation. The subscription plan update server action was tested for BYOC-specific business logic bypass in both org contexts.

Client-side feature gating — where the application uses a `localStorage` flag to determine what features to show — was tested for bypass, checking whether the server enforced the same entitlement independently of what the client reported.

BYOC as a Dedicated Attack Surface

The BYOC (Bring Your Own Cloud) feature — which allows enterprise customers to connect their own cloud infrastructure — was treated as a distinct, high-value target throughout the engagement.

Beyond the SSRF and XSS testing already described, the BYOC authentication model was assessed separately: core, path-fuzz, and setup BYOC API routes were tested for authentication bypass. JWT tokens used in BYOC context were tested for cross-org substitution. Credential values submitted during BYOC configure and setup flows were assessed for exposure risk. Cross-tenant IDOR was tested specifically for the BYOC configuration: can OrgA read or modify OrgB's BYOC settings?

API Protocol Security (GraphQL & tRPC)

Both of the platform's non-REST API layers received dedicated attention.

The GraphQL API was tested for introspection availability without authentication — a misconfiguration that exposes the full schema to unauthenticated attackers. Schema enumeration was performed. Authenticated GraphQL access was tested from OrgA, and cross-tenant data retrieval was attempted from OrgB. Batch abuse via aliased queries was tested as a rate limit bypass technique. The tRPC API was tested for procedure enumeration (discovering available procedures without documentation), cross-tenant IDOR, and batch request abuse.

Information Disclosure

Throughout all phases, testers remained alert to information disclosure — sensitive data surfacing in places it should not.

Specific disclosures assessed included: API keys embedded in JavaScript bundles, secrets exposed via source maps, configuration and environment files accessible at predictable paths, session tokens leaking through response headers, internal route structures revealed in callback URL parameters, invite error messages leaking whether a user exists or which org they belong to, error pages and debug endpoints revealing stack traces or internal service behavior, RSC enumeration of billing settings and resource data, email service API key extraction and full inbox enumeration, internal parameter data exposure, and pending authentication token leakage into analytics pipelines.

Supporting Test Categories

Several additional categories rounded out the engagement. CSRF was tested broadly across every state-changing operation in the application — API key creation, admin actions, invite flows, magic links, resource operations, dashboard terms, org settings, integrations, OAuth

callbacks, logout, and the email subscribe endpoint. Rate limiting was evaluated on magic links, OTP, feature flags, and GraphQL batch operations. Race conditions were tested on registration and org-join flows. Webhooks and integrations were tested for SSRF, XSS, CSRF, and cross-tenant IDOR. The identity provider webhook endpoint was tested for forgery. Internal cron endpoints were tested for authentication bypass and cross-org billing manipulation. The email testing service was assessed for API key exposure and inbox enumeration. The analytics ingest proxy received a comprehensive standalone assessment.

Closing Note

The breadth of this engagement reflects the complexity of the target platform's attack surface: a modern JavaScript framework application with a sophisticated multi-tenant architecture, cloud-native infrastructure integrations, a novel BYOC deployment model, and a billing system with real financial consequences for logic flaws. Testing was methodical and covered each subsystem both in isolation and in combination, with particular depth applied to cross-tenant isolation, SSRF, and billing logic — the areas where vulnerabilities tend to carry the most impact for a platform of this type.

Threat Ranking Methodology

RunSybil testing and vulnerability threat rankings are aligned to industry-proven Common Vulnerability Scoring System (CVSS) threat rankings methodology. The following section outlines the CVSS scoring methodology applied to the assessment findings:

Common Vulnerability Scoring System (CVSS)		
Severity Rating	**Score**	**Definitions**
Critical	9.0 - 10.0	Vulnerabilities that could lead to catastrophic consequences, such as full system compromise, unauthorized root/administrator-level access to critical systems, exposure or modification of large volumes of highly sensitive data (e.g., financial records, credentials, PII of all users), or complete destruction/disruption of essential services.
High	7.0 - 8.9	Vulnerabilities that could cause significant damage or disruption, such as privilege escalation to a high-level user, exposure or modification of significant amounts of sensitive data, denial of service affecting major application features, or serious authorization flaws allowing access to unauthorized functions/data.
Medium	4.0 - 6.9	Vulnerabilities with moderate potential impact or requiring more complex exploitation scenarios. This could include disclosure of moderately sensitive information (e.g., some business data, limited user details), tampering with non-critical data, partial denial of service, or flaws requiring significant user interaction to exploit.
Low	0.1 - 3.9	Vulnerabilities with minor potential impact or that are very difficult to exploit. These typically involve exposure of non-sensitive information, minor application misbehavior, limited denial of service, or require highly unlikely circumstances or significant prior access to exploit.
Informational	0.0	Findings that do not represent a direct, exploitable security vulnerability but are relevant observations for security posture, code quality, or defense-in-depth.

The severity levels correspond to the numerical score, helping prioritize vulnerabilities. Lower scores might require less immediate action, while higher scores indicate a need for urgent attention.

Findings Summary

The following chart provides an overview of CVSS scoring and a summary of the findings discovered during the assessment:

	Description	Severity	CVSS	Status
1	Unauthenticated Authentication Cookie Generation via /admin/Home/ExtendUserSession	critical	9.1	Open
2	Mass Assignment on Public Registration Endpoint Allows Unauthenticated Privilege Escalation via IsInternalUser Field	critical	9.8	Open
3	OS Command Injection in _testReward JSON Parameter	critical	9.8	Confirmed
4	GraphQL Introspection Bypass via Apollo Federation Service Endpoint	high	7.1	Open
5	Insecure Direct Object Reference in IAM Groups Endpoint Allows Cross-Tenant Data Access	high	7.5	Open
6	Path Traversal in Checkpoint Archive Endpoint Parameter	high	7.5	Fixed
7	SQL Injection in FeatureDemoLookup via platform parameter	high	8.1	Confirmed
8	Exposed phpinfo() Debug Page in /tmp/ Directory	medium	5.3	Confirmed
9	Reflected Cross-Site Scripting via javascript: URI in VideoPlayer iframe src Parameter	medium	6.1	Open

	Description	Severity	CVSS	Status
10	Business Logic Flaw: Multiple Tax IDs of Same Type Allowed Per Customer	low	2.1	Open

Critical Threat Assessment Findings

Unauthenticated Authentication Cookie Generation via /admin/Home/ExtendUserSession

Critical	CVSS Score: 9.1	Open

CWE IDs:

CWE-203, CWE-287

Affected Components:

https://rc.acmecorp.com/admin/Home/ExtendUserSession

Description

The `/admin/Home/ExtendUserSession` endpoint on `rc.acmecorp.com` (and all AcmeApp tenant hosts) exposes unauthenticated generation of valid `.DemoApp_AUTH` FormsAuthentication session cookies. When an unauthenticated attacker POSTs a numeric `userId` parameter to this endpoint, the server issues a fully valid `.DemoApp_AUTH` cookie scoped to that user account — with no credential verification, session validation, CSRF protection, or access control of any kind.

Source code root cause confirmed. The vulnerable method in `AcmeApp.Admin/Controllers/HomeController.cs` carries no `[Authorize]` attribute, and no global authorization filter is registered in `AcmeApp.Admin/App_Start/FilterConfig.cs`. The implementation calls `SetAuthCookie()` in `BaseController.cs`, which unconditionally writes a fully encrypted `FormsAuthenticationTicket` containing the target user's serialized `UserData` object (`UserID`, `UserName`, `Email`, `RoleID`, `SiteID`, `AuthKey`) to `FormsAuthentication.FormsCookieName` — which is `.DemoApp_AUTH`, the storefront's authentication cookie name as configured in `AcmeApp.Web/Web.config`. This design choice — writing to the storefront's cookie name from an admin endpoint — only makes operational sense if both IIS applications share a common `<machineKey>` configuration, which is the standard deployment pattern for co-hosted Admin and Web projects.

Testing confirmed the following differential behavior:

- **Valid user IDs** cause the server to return a `.DemoApp_AUTH` cookie (approximately 1,800+ hex characters, with `secure`, `HttpOnly`, and `path=/` flags, expiring ~20 minutes in the future) alongside a load balancer cookie.
- **Invalid user IDs** return only the load balancer cookie — no `.DemoApp_AUTH` cookie is present.

User ID enumeration at scale was confirmed: a scan of IDs 1–100,050 identified 94,000+ valid accounts, forming a dense continuous block from approximately ID 6,000 onward, with isolated valid entries in the 150–5,999 range.

The endpoint accepts both `application/json` and `application/x-www-form-urlencoded` content types, and the parameter name is case-insensitive (`userId` and `UserID` are both accepted).

Phase 2 (cookie reuse) in the RC environment: The `.DemoApp_AUTH` cookie generated by the admin app (`/admin/`) cannot be consumed by the storefront (`/`) in the RC deployment because they are configured as separate IIS applications with independent auto-generated `<machineKey>` values. The admin's `FormsAuthentication.Encrypt()` uses one key; the storefront's `FormsAuthentication.Decrypt()` expects a different key — the ticket fails to decrypt and `[Authorize]` redirects to login. This is an infrastructure constraint of the RC environment only. In deployments where the Admin and Web applications share a `<machineKey>` (the standard configuration for single-server or shared-config deployments of the same Visual Studio solution), Phase 2 fully succeeds and the forged cookie provides complete account access on every `[Authorize]`-protected storefront endpoint.

Recommendation

The following remediation steps are recommended, ordered by priority:

1. **Enforce Authentication and Authorization Immediately:** Apply an `[Authorize]` attribute to the `ExtendUserSession` action in `HomeController.cs` and ensure it requires a valid authenticated administrator session before processing any request. This single change eliminates the unauthenticated cookie generation vector. The Admin application's OWIN/SSO authentication stack is already in place — `ExtendUserSession` simply needs to participate in it.
2. **Evaluate and Likely Remove the Endpoint:** Assess whether `ExtendUserSession` serves a legitimate, necessary function in the current application workflow. Its only documented purpose is to refresh a user's session from within the admin UI, but the same outcome can be achieved through the existing authenticated session lifecycle. If not strictly required, remove the endpoint entirely to eliminate the attack surface permanently.
3. **Audit Machine Key Configuration Across Deployments:** Determine whether any production AcmeApp instances host the Admin and Web IIS applications under a shared `<machineKey>`. If shared keys are confirmed in any environment, the unauthenticated cookie generation described in this finding becomes a complete, exploitable account-takeover path against production accounts. Document and enforce explicit `<machineKey>` isolation between the admin and storefront applications in all environments, or implement application-level session binding (e.g., tying the auth ticket to a server-side session record) that a forged ticket cannot satisfy.
4. **Implement Anti-CSRF Protections:** Add ASP.NET `AntiForgeryToken` validation to `ExtendUserSession` and all other state-changing administrative endpoints. This prevents cross-site request forgery attacks even after authentication is enforced.

5. **Normalize Error Responses to Eliminate Enumeration Oracle:** Ensure the endpoint returns an identical HTTP response — same status code, headers, and body structure — regardless of whether the supplied `userId` is valid or invalid. The current differential behavior (auth cookie present vs. absent) provides a reliable oracle for enumerating all 94,000+ user accounts in the database.
6. **Add Rate Limiting:** Implement rate limiting on `/admin/Home/ExtendUserSession` as a defense-in-depth measure to slow automated enumeration attempts, even after authentication enforcement is in place.
7. **Review Cookie Name Scoping:** The fact that `SetAuthCookie()` in the Admin app writes to `FormsAuthentication.FormsCookieName` (`.DemoApp_AUTH`) — the storefront's cookie — rather than the admin's own `.DemoAppAdmin_ssoauth` cookie is a design anomaly that presupposes shared machine keys. After remediating the authentication gap, refactor `SetAuthCookie()` in `AcmeApp.Admin/Controllers/BaseController.cs` to write to the admin-specific cookie name, eliminating the cross-application cookie dependency entirely.
8. **Audit Related Administrative Endpoints:** Review all endpoints under `/admin/` for similar missing authentication decorations, particularly session management functions. The absence of a global `[Authorize]` filter in `FilterConfig.cs` means each action method must individually enforce access control — audit all admin controllers for missing attributes.

Reproduction Steps

1. Open a tool capable of sending HTTP requests (e.g., `curl`, Burp Suite, or a browser developer console).
2. Send the following `POST` request to the `ExtendUserSession` endpoint with a confirmed valid user ID:

```
POST https://rc.acmecorp.com/admin/Home/ExtendUserSession HTTP/1.1
Host: rc.acmecorp.com
Content-Type: application/json

{"userId": 142073}
```

1. Observe the response headers. The server returns HTTP 200 with a `Set-Cookie` header containing a `.DemoApp_AUTH` cookie:

```
Set-Cookie: .DemoApp_AUTH=<~1800 hex chars>; expires=...; path=/; secure;
HttpOnly
```

This cookie contains a fully encrypted `FormsAuthenticationTicket` with the serialized account data for the target user account.

1. Confirm user ID enumeration by sending the same request with an invalid user ID (e.g., `1`):

```
POST https://rc.acmecorp.com/admin/Home/ExtendUserSession HTTP/1.1
Host: rc.acmecorp.com
Content-Type: application/json

{"userId": 1}
```

Observe that the response contains only a load balancer cookie — no `.DemoApp_AUTH` cookie is present. This differential response is the enumeration oracle.

1. Repeat with additional invalid IDs (e.g., `11111`, `999999`) to confirm consistent behavior. A scan of IDs 1–100,050 confirms this oracle identifies 94,000+ valid accounts.
2. Confirm the endpoint also accepts form-encoded bodies:

```
POST https://rc.acmecorp.com/admin/Home/ExtendUserSession HTTP/1.1
Host: rc.acmecorp.com
Content-Type: application/x-www-form-urlencoded

userId=142073
```

1. **(Shared machine key deployments — Phase 2):** On an AcmeApp deployment where the Admin and Web IIS applications share an explicit `<machineKey>` configuration (standard for single-server or shared-config production deployments), supply the `.DemoApp_AUTH` cookie from step 3 to any `[Authorize]`-protected storefront endpoint:

```
GET https://<host>/Profile/Home/Index HTTP/1.1
Host: <host>
Cookie: .DemoApp_AUTH=<value from step 3>
```

The server returns HTTP 200 and renders the authenticated profile page for the target user, confirming full account takeover. In the RC environment, this step returns a 302 redirect due to machine key isolation between the two separately-configured IIS applications; the cookie-generation (Phase 1) vulnerability is confirmed regardless.

1. **Source code reference:** The vulnerable method with no `[Authorize]` attribute is at `AcmeApp.Admin/Controllers/HomeController.cs:220`. The `SetAuthCookie()` implementation writing to `.DemoApp_AUTH` is at `AcmeApp.Admin/Controllers/BaseController.cs:53`. The absence of a global authorization filter is confirmed in `AcmeApp.Admin/App_Start/FilterConfig.cs`.

Mass Assignment on Public Registration Endpoint Allows Unauthenticated Privilege Escalation via IsInternalUser Field

Critical	CVSS Score: 9.8	Open

CWE IDs:

CWE-915

Affected Components:

https://api.u1.demo.acmecloudtest.com/useradmin/registrations

RegisterOrganizationAndUserRequest DTO

AccessPolicyHandler

Description

The public (unauthenticated) `POST /useradmin/registrations` endpoint is vulnerable to mass assignment. The `RegisterOrganizationAndUserRequest` data transfer object (DTO) includes a `public bool IsInternalUser` property that has no protective attributes such as `[BindNever]` or `[JsonIgnore]`, and no server-side validation or override logic. When a registration request is submitted, the ASP.NET Core model binder deserializes all client-supplied JSON fields — including `IsInternalUser` — directly onto the model. The controller then copies this value to a `CreateUserRequest` object at line 153 without any sanitization, passing the attacker-controlled value through to the service layer.

The root cause is a shared DTO pattern: the registration model includes the `IsInternalUser` property because it is also used in the authenticated administrative user creation flow, where an internal-admin guard exists. The public registration controller, however, has no `[Authorize]` attribute and no equivalent check, leaving the property fully controllable by any unauthenticated caller.

ASP.NET Core's `System.Text.Json` deserializer uses **case-insensitive property matching** by default, which means multiple casing variations of the field name — `IsInternalUser`, `isInternalUser`, `isinternaluser`, and `ISINTERNALUSER` — all successfully bind to the backend property. Testing confirmed that only the JSON boolean literal `true` is accepted; string representations (`"true"`, `"True"`, `"1"`) and integer values (`1`) are rejected by System.Text.Json's strict type checking with HTTP 400 deserialization errors. Field names with underscores (`Is_Internal_User`) or missing prefixes (`InternalUser`) are silently ignored.

Baseline control tests confirmed that the HTTP 500 response observed during testing originates from a downstream SSO integration failure and is unrelated to `IsInternalUser`

validation — the same 500 occurs with or without the field. The critical observation is the differential response: invalid type values produce a specific HTTP 400 error referencing `$.IsInternalUser` and `System.Boolean`, proving the server recognizes and attempts to bind the property. Valid boolean `true` values pass model binding entirely and reach controller logic.

Recommendation

1. **Immediate fix — Attribute-based binding restriction:** Add the `[JsonIgnore]` attribute (or `[BindNever]` for MVC model binding) to the `IsInternalUser` property on the `RegisterOrganizationAndUserRequest` model to prevent it from being deserialized from user-supplied JSON during registration:

```
[JsonIgnore]
public bool IsInternalUser { get; set; }
```

1. **Server-side override:** As a defense-in-depth measure, explicitly set `IsInternalUser = false` in the registration controller before passing the request object to the service layer, ensuring the value cannot be influenced by client input regardless of binding configuration.
2. **Separate DTOs for public and administrative flows:** Refactor the registration endpoint to use a dedicated DTO that does not include the `IsInternalUser` property. The current shared model between the public registration flow and the authenticated admin user creation flow is the root cause of this mass assignment vulnerability. Applying the DTO projection pattern ensures that sensitive administrative properties are structurally unavailable on public endpoints.
3. **Comprehensive model audit:** Review all properties on `RegisterOrganizationAndUserRequest` and other public-facing DTOs for additional mass assignment risks. Any property that controls privileges, roles, organization membership, or access levels should be explicitly excluded from client-controlled binding on unauthenticated or low-privilege endpoints.
4. **Integration testing:** Add automated tests that verify the public registration endpoint rejects or ignores privilege-escalation fields such as `IsInternalUser`, ensuring this class of vulnerability is caught in CI/CD pipelines going forward.

Reproduction Steps

1. Send a `POST` request to `https://api.u1.demo.acmecloudtest.com/useradmin/registrations` with the `Content-Type: application/json` header. No authentication is required.
2. Include the following JSON body with `IsInternalUser` set to the boolean value `true`:

```
{
  "Username": "testuser",
```

```
  "FirstName": "Test",
  "LastName": "User",
  "Email": "testuser@example.com",
  "Phone": "5555550001",
  "PhoneType": 1,
  "OrgName": "TestOrg",
  "AcmeBusinessId": "99999",
  "OrganizationId": "99999",
  "UserType": 1,
  "EmailAsUsername": false,
  "IsInternalUser": true
}
```

1. Observe that the server responds with HTTP 500 and a message indicating a downstream SSO integration failure. This confirms that the **`IsInternalUser`** field was accepted by model binding and the request reached controller logic.
2. To confirm the server recognizes and type-checks the **`IsInternalUser`** property, send the same request but with **`IsInternalUser`** set to the string **`"true"`** instead of the boolean **`true`**:

```
{
  "Username": "testuser2",
  "FirstName": "Test",
  "LastName": "User",
  "Email": "testuser2@example.com",
  "Phone": "5555550002",
  "PhoneType": 1,
  "OrgName": "TestOrg",
  "AcmeBusinessId": "99999",
  "OrganizationId": "99999",
  "UserType": 1,
  "EmailAsUsername": false,
  "IsInternalUser": "true"
}
```

1. Observe that the server responds with HTTP 400 and the error **`"The JSON value could not be converted to System.Boolean. Path: $.IsInternalUser"`**. This differential response proves the server recognizes **`IsInternalUser`** as a bound boolean property — it rejects invalid types while accepting the boolean literal **`true`**.
2. Verify that case-insensitive binding also works by sending the request with **`isInternalUser: true`** (camelCase), **`isinternaluser: true`** (all lowercase), or **`ISINTERNALUSER: true`** (all uppercase). All variations produce the same HTTP 500 response, confirming successful model binding.
3. As a baseline control, send the same registration request with no **`IsInternalUser`** field at all and observe the identical HTTP 500 SSO error — this confirms the 500 is unrelated to the **`IsInternalUser`** value and that the field acceptance (vs. rejection with 400) is the meaningful indicator of successful binding.

OS Command Injection in _testReward JSON Parameter

Critical	CVSS Score: 9.8	Confirmed

CWE IDs:

CWE-78

Affected Components:

```
/acme_web_services/controller.DemoClientAffinity.php/com.Acme
Corp.services.DemoClientAffinity._testReward/

/controller.DemoClientAffinity.php/com.Acme
Corp.services.DemoClientAffinity._testReward
```

Description

The _testReward endpoint in the Acme Corp web service contains a critical command injection vulnerability. User-supplied JSON parameters are passed unsanitized to a shell_exec() function call. By injecting shell metacharacters and commands into JSON string values (specifically the "st" parameter within _systemJson), an attacker can break out of the intended JSON string context and execute arbitrary OS commands. The vulnerability occurs because the application constructs shell commands by directly concatenating user input without proper escaping or parameterization, allowing attackers to manipulate the command structure.

Recommendation

Immediate Actions: • Disable the _testReward endpoint in production environments immediately • Remove or disable debug output that exposes internal application state and command execution results

Code-Level Fixes: • Replace shell_exec() calls with safer alternatives like proc_open() or exec() with proper parameter arrays • Implement input validation and sanitization for all JSON parameters before processing • Use escapeshellarg() or escapeshellcmd() functions when shell execution is absolutely necessary • Avoid constructing shell commands through string concatenation

Security Controls: • Implement application-level input validation with strict allowlists for expected parameter values • Add logging and monitoring for suspicious parameter values or command injection attempts • Apply principle of least privilege to the web server process • Consider running the web application in a containerized or sandboxed environment to limit the impact of successful exploitation

Reproduction Steps

1. Send a GET request to the vulnerable endpoint: **`/acme_web_services/controller.DemoClientAffinity.php/com.acme-corp.services.DemoClientAffinity._testReward/`**
2. Include a crafted JSON payload in the URL path with the following structure, replacing the "lg" field value with the injection payload:

```
{"_systemJson":{"lg":"1033\";id;echoINJ;\"","gLg":"en-US","osC":"10.0","osB":"15063","is6":"1","GFPV":"385.41","gIsB":"0","go":"US"}}
```

1. URL-encode the JSON payload and append it to the endpoint path
2. Observe the response contains debug output showing the injected command execution
3. Look for evidence of command execution such as **`uid=48(apache) gid=48(apache) groups=48(apache)`** and the **`INJ`** marker in the response
4. Verify that other shell commands can be executed by modifying the injection payload

High Threat Assessment Findings

GraphQL Introspection Bypass via Apollo Federation Service Endpoint

High	CVSS Score: 7.1	Open

CWE IDs:

CWE-205, CWE-548, CWE-1295

Affected Components:

https://demo-uat.acme-portal.app/apigw/datatracker/v1/graphql

Description

The GraphQL API at `/apigw/datatracker/v1/graphql` has introspection disabled for standard `__schema` queries, returning an error message 'Introspection has been disabled for this request' with classification 'IntrospectionDisabled'. However, the API exposes Apollo Federation's `_service.sdl` endpoint, which returns the complete GraphQL schema definition in SDL (Schema Definition Language) format. This represents a common misconfiguration in Apollo Federation implementations where the service endpoint used for federation gateway discovery is left accessible in production environments. The `_service` query is a standard Apollo Federation field that federated services expose to advertise their capabilities to the gateway, but it should be disabled or restricted in production when introspection is intentionally blocked. The extracted schema contains 24 Query root type operations (including 8 not documented in the API documentation), 4 Mutation operations, over 25 custom types including a User type with sensitive fields (id, email, firstName, lastName, userName), 13 enum definitions revealing business logic and error handling structures, 4 custom scalars, and 13 input types for filtering and searching. This comprehensive schema exposure defeats the intended security control and provides attackers with complete API surface area documentation.

Recommendation

Immediate remediation is required to disable the Apollo Federation service endpoint in production or restrict it to internal administrative access only:

Disable Apollo Federation service queries in production:

For Apollo Server implementations, configure the server to disable federation service queries using `ApolloServerPluginInlineTrace` with `{ rewriteError }` or by explicitly removing the

`_service` field from the schema. Alternatively, set `apollo: { key: null }` to disable federation entirely if the gateway functionality is not required.

Implement authentication and authorization controls:

Specifically for the `_service` query field, add resolver-level middleware to check that requests for `_service.sdl` originate from trusted internal IP ranges or possess elevated administrative privileges. Return an authorization error for standard user requests.

Apply consistent introspection controls:

Ensure consistency across all schema discovery mechanisms. If introspection is disabled via `__schema` and `__type` queries, make sure that Apollo Federation service queries, GraphQL Playground/GraphiQL interfaces, and any other schema exposure mechanisms are similarly restricted.

Implement defense-in-depth security controls for the GraphQL API:

- Enable query complexity analysis with configurable thresholds to prevent resource exhaustion attacks
- Implement query depth limiting to prevent deeply nested query attacks
- Apply rate limiting at the GraphQL endpoint level to restrict reconnaissance attempts
- Enable comprehensive logging of all GraphQL operations, particularly `_service`, `__schema`, and `__type` queries
- Consider field-level authorization checks to validate user permissions for specific operations and data fields

Review exposed schema for sensitive information disclosure:

Since the complete schema is now exposed, conduct a security review of all operations to identify unnecessarily exposed internal operations, sensitive field names that leak implementation details, operations that should require elevated privileges, and opportunities to reduce API surface area by deprecating or removing unused operations.

Establish monitoring and alerting:

Set up monitoring for reconnaissance activity by creating alerts for queries to `_service`, `__schema`, `__type`, and other introspection-related fields. Monitor for patterns indicating schema enumeration attempts, such as sequential testing of operations or error-based field discovery.

Reproduction Steps

1. Verify that standard GraphQL introspection is blocked by sending a POST request to the target GraphQL endpoint with appropriate authentication headers (`Cookie`, `Content-Type: application/json`, and CSRF token headers). Include the request body `{"query":"{__schema{queryType{name}}}","operationName":null}`. Observe the HTTP 200 response with error: `{"errors":[{"message":"Introspection has been disabled for`

`this request","locations":[{"line":1,"column":28}],"extensions":{"classification":"IntrospectionDisabled"}}]}`.

2. Bypass the introspection control by querying the Apollo Federation service endpoint. Send a POST request to the same endpoint with the same authentication headers. Include the request body `{"query":"query{_service{sdl}}","operationName":null}`.

3. Observe the successful HTTP 200 response containing the complete GraphQL schema in SDL format. The response body will be approximately 19,142 characters and will contain the full schema definition starting with `{"data":{"_service":{"sdl":"<complete SDL schema>"}}}`.

4. Parse the SDL response to extract all available operations. The schema will reveal 24 Query operations including `dataTrackerDefinition`, `dataTrackerDefinitions`, `dataTrackerEntity`, `dataTrackerEntities`, `dataTrackerEntitiesPaginationInfo`, `dataTrackerEvents`, `dataTrackerEventsPaginationInfo`, `dataTrackerMetrics`, `dataTrackerMetricsPaginationInfo`, and 15 other operations. It will also show 4 Mutation operations: `dataTrackerDefinitionCreation`, `dataTrackerDefinitionUpdate`, `dataTrackerDefinitionDelete`, and `dataTrackerDefinitionRefresh`.

5. Test that discovered operations are functional by querying an undocumented operation. Send a POST request with body `{"query":"query{dataTrackerDefinition(id:\"<sample-uuid>\"){id name status}}"}` to confirm the operation exists and is accessible (returns valid data or an authorization error, not a 'field undefined' validation error).

Insecure Direct Object Reference in IAM Groups Endpoint Allows Cross-Tenant Data Access

High	**CVSS Score:** 7.5	Open

CWE IDs:

CWE-284, CWE-639

Affected Components:

`https://demo-uat.acme-portal.app/apigw/iam/v2/tenants/{id}/groups`

Description

The **`/apigw/iam/v2/tenants/{id}/groups`** endpoint contains an authorization bypass vulnerability that fails to validate tenant ownership. The endpoint accepts a tenant ID as a path parameter but does not verify that the authenticated user's tenant (extracted from JWT claims) matches the requested tenant ID. This allows any authenticated user to access the complete IAM group structure, user lists, email addresses, and role assignments of any tenant in the system by simply modifying the tenant ID in the URL.

The root cause is the absence of server-side authorization logic that compares the authenticated user's tenant context (TENANT_ID claim in the JWT) against the requested tenant ID path parameter. The application relies solely on authentication (valid JWT token) without implementing proper authorization controls to ensure users can only access resources within their own tenant boundary.

During testing, a user authenticated to tenant 21 (USER_ID: 969, TENANT_ID: 21) successfully accessed tenant 22's complete IAM structure by requesting **`/apigw/iam/v2/tenants/22/groups`**. The API returned an HTTP 200 response containing detailed information about 11 users organized across 4 groups, complete with email addresses, role assignments spanning 19 distinct role types, and group UUIDs. The response included sensitive details such as integration account credentials and deleted users still present in the system.

Recommendation

Primary Fix - Implement Server-Side Authorization:

Implement mandatory server-side authorization checks that validate the authenticated user's tenant ID (from JWT claims) matches the requested tenant ID in the URL path parameter. When a mismatch is detected, return HTTP 403 Forbidden with an appropriate error message. This validation must occur before any database queries or data retrieval operations.

```
if (authenticatedUserTenantId != requestedTenantId) {
    return HTTP 403 Forbidden
}
```

Additional Security Controls:

- **Comprehensive Endpoint Audit:** Review all tenant-scoped API endpoints (particularly those using `/tenants/{id}/` path patterns) for similar authorization bypass vulnerabilities. Implement consistent authorization middleware across all tenant-scoped resources.
- **Centralized Authorization Middleware:** Develop and enforce centralized authorization middleware that automatically validates tenant context for all requests. This prevents future endpoints from being deployed without proper authorization controls.
- **Security Logging and Monitoring:** Implement security logging for cross-tenant access attempts, recording the authenticated user's tenant ID, requested tenant ID, timestamp, and source IP. Configure alerts for repeated cross-tenant access attempts that may indicate reconnaissance or exploitation.
- **Soft-Delete Isolation:** Remove deleted users from API responses or implement proper soft-delete isolation that prevents deleted users from appearing in active tenant queries. Deleted user records were exposed in the response, indicating incomplete user lifecycle management.
- **PII Minimization:** Consider masking or removing email addresses from group listing endpoints unless explicitly required for the user's role. Implement field-level authorization to limit PII exposure to only users with legitimate need-to-know.

Reproduction Steps

1. Authenticate to the application as any valid user in any tenant (e.g., testuser+1@example.com in tenant 21)
2. Capture the authentication cookies from the login response:
3. `XSRF-TOKEN` cookie
4. `Demo-Authorization` cookie (contains JWT with your TENANT_ID)
5. Send a GET request to access your own tenant's groups endpoint to establish a baseline:

```
GET /apigw/iam/v2/tenants/21/groups HTTP/1.1
Host: demo-uat.acme-portal.app
Cookie: XSRF-TOKEN={your_xsrf_token}; Demo-Authorization={your_jwt_token}
Accept: application/json
```

1. Observe the HTTP 200 response containing your tenant's group structure, user lists, and role assignments

2. Modify the tenant ID in the URL path to access another tenant's data:

```
GET /apigw/iam/v2/tenants/22/groups HTTP/1.1
Host: demo-uat.acme-portal.app
Cookie: XSRF-TOKEN={your_xsrf_token}; Demo-Authorization={your_jwt_token}
Accept: application/json
```

1. Observe that the request returns HTTP 200 with complete group structure, user information, email addresses, and role assignments for tenant 22, despite being authenticated as tenant 21
2. Repeat with different tenant IDs (1, 23, 24, etc.) to enumerate additional tenants and their IAM structures
3. Extract sensitive information from responses including:
4. User IDs and full names
5. Email addresses for phishing campaigns
6. Role assignments and permission hierarchies
7. Group UUIDs for potential further exploitation
8. Integration account patterns

Path Traversal in Checkpoint Archive Endpoint Parameter

High	CVSS Score: 7.5	Fixed

CWE IDs:

CWE-22

Affected Components:

`https://model.acmecorp.dev/services/model-intern/api/v1/training_runs/{training_run_id}/checkpoints/{checkpoint_path}/archive`

Description

The checkpoint archive endpoint at **`/api/v1/training_runs/{training_run_id}/checkpoints/{checkpoint_path}/archive`** fails to properly validate the checkpoint path parameter before processing it. The application accepts URL-encoded path traversal sequences (**`%2F..%2F`** representing **`/../`**) and resolves them to construct the final storage path, rather than rejecting requests containing such sequences. Testing confirmed that three different checkpoint paths—a legitimate path (**`sampler_weights/000009`**), a path with an arbitrary prefix and traversal (**`xxxxxxx%2F..%2Fsampler_weights/000009`**), and another traversal variation (**`test%2F..%2Fsampler_weights/000009`**)—all successfully returned HTTP 302 redirects to the identical cloud storage URL. This behavior indicates the application normalizes or resolves the path after accepting the input, rather than implementing strict input validation that would reject malformed paths. The root cause is insufficient input validation on user-controlled path parameters. While the current storage implementation appears to limit exploitation scope (attempts to access system files and other users' training runs were unsuccessful), the lack of proper input validation creates potential for future exploitation if storage architecture or access controls change.

Recommendation

1. Implement strict input validation on the **`checkpoint_path`** parameter that explicitly rejects any requests containing path traversal sequences, including **`../`**, **`..\`**, and all URL-encoded variants (**`%2E%2E%2F`**, **`%2E%2E%5C`**, **`..%2F`**, etc.)
2. Use an allowlist approach for the checkpoint path parameter, permitting only alphanumeric characters, forward slashes, underscores, and hyphens that match expected checkpoint naming conventions
3. After any path construction, validate that the resolved absolute path remains within the expected checkpoint directory for the specific training run before generating the signed storage URL

4. Implement path validation using a secure path joining function that inherently prevents directory traversal (e.g., Python's `pathlib.Path.resolve()` with strict validation, or equivalent in your technology stack)
5. Add security logging and monitoring to detect and alert on path traversal attempts in the checkpoint parameter, enabling detection of exploitation attempts
6. Consider implementing rate limiting on failed or suspicious path access attempts to mitigate potential automated exploitation

Reproduction Steps

1. Authenticate to the API using a valid API key in the `X-Api-Key` header
2. Identify a valid training run ID and checkpoint path (e.g., training run `aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa` with checkpoint `sampler_weights/000009`)
3. Send a baseline request to establish the legitimate checkpoint access pattern:

```
GET /api/v1/training_runs/aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa/checkpoints/
sampler_weights/000009/archive
X-Api-Key: {valid_api_key}
```

1. Observe a 302 redirect response with a `Location` header pointing to cloud object storage
2. Send a request with a path traversal sequence using an arbitrary prefix:

```
GET /api/v1/training_runs/aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa/checkpoints/
xxxxxxx%2F..%2Fsampler_weights/000009/archive
X-Api-Key: {valid_api_key}
```

1. Observe that this request also returns a 302 redirect to the same cloud storage URL
2. Send another request with a different traversal pattern:

```
GET /api/v1/training_runs/aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa/checkpoints/
test%2F..%2Fsampler_weights/000009/archive
X-Api-Key: {valid_api_key}
```

1. Observe that this request also returns a 302 redirect to the identical cloud storage URL
2. Compare the `Location` headers from all three responses—they will be identical, confirming the application processes path traversal sequences rather than rejecting them

SQL Injection in FeatureDemoLookup via platform parameter

High	**CVSS Score:** 8.1	Confirmed

CWE IDs:

CWE-89

Affected Components:

`/services_toolkit/services/com/acme-corp/services/AjaxDemoService.php`

Description

The FeatureDemoLookup function in AjaxDemoService.php contains a SQL injection vulnerability in the platform parameter. The application directly concatenates user-supplied input into a SQL query without proper sanitization or parameterization. When a malicious payload is injected into the platform parameter, it is executed as part of the SQL statement, as evidenced by the detailed error message revealing the underlying query structure. The vulnerability occurs in a query that joins multiple internal database tables to retrieve feature information, and the platform parameter is inserted directly into the WHERE clause without escaping.

Recommendation

1. **Implement parameterized queries**: Replace the current string concatenation approach with parameterized queries or prepared statements to prevent SQL injection. Use the database driver's parameter binding functionality to safely handle user input.
2. **Input validation and sanitization**: Implement strict input validation for the platform parameter, allowing only predefined valid values from a whitelist of supported platforms.
3. **Error handling**: Implement proper error handling to prevent detailed SQL error messages from being exposed to users. Log detailed errors server-side while returning generic error messages to clients.
4. **Code review**: Conduct a comprehensive security review of all database queries in the AjaxDemoService.php file and related components to identify and remediate similar vulnerabilities.
5. **Database permissions**: Ensure the database user account used by the application has minimal necessary privileges to limit the potential impact of successful SQL injection attacks.

Reproduction Steps

1. Send a GET request to the FeatureDemoLookup endpoint with valid series and productfamily parameters and append a SQL injection payload to the platform parameter: `GET /services_toolkit/services/com/acme-corp/services/AjaxDemoService.php?func=FeatureDemoLookup&series=Demo%20Series&productfamily=Demo&platform=Windows%2010%2064-bit'%20AND%20SLEEP(5)--+`
2. Observe the HTTP 404 response containing a detailed SQL error message
3. Note the error message reveals the complete SQL query structure and confirms that the injected payload was executed as part of the SQL statement
4. The error "'SLEEP' is not a recognized built-in function name" proves that the SQL injection was successful, as the database attempted to execute the injected SLEEP function

Medium Threat Assessment Findings

Exposed phpinfo() Debug Page in /tmp/ Directory

Medium	CVSS Score: 5.3	Confirmed

CWE IDs:

CWE-200

Affected Components:

/tmp/phpinfo.php

https://acme-portal-prdapplb-demo1-0000000000.ap-southeast-1.elb.amazonaws.com/tmp/phpinfo.php

Description

A phpinfo.php file is publicly accessible at `/tmp/phpinfo.php` and returns complete PHP configuration information through the phpinfo() function. This debug file exposes sensitive system details including PHP version 7.4.33, loaded modules (mysqli, pdo_mysql, openssl, etc.), file system paths (/var/www/html, /tmp), environment variables, and configuration directives. The file appears to be a development or debugging artifact that was inadvertently left accessible in the production environment.

Recommendation

Immediate Action: • Remove the phpinfo.php file from the `/tmp/` directory and all public web directories • Conduct a comprehensive audit to identify and remove any other debug, development, or test files from production environments

Long-term Measures: • Implement deployment procedures that exclude development and debug files from production releases • Configure web server to deny access to common debug file patterns (*.php files in /tmp/, phpinfo.php, test.php, debug.php) • If debugging information is required in production, secure it behind strong authentication and restrict access to authorized personnel only • Establish regular security reviews of deployed files to prevent similar exposures

Reproduction Steps

1. Navigate to `https://<target-host>/tmp/phpinfo.php` in a web browser
2. Observe the returned HTML page containing complete phpinfo() output

3. Review the exposed information including PHP version, loaded extensions, system paths, and configuration settings

Reflected Cross-Site Scripting via javascript: URI in VideoPlayer iframe src Parameter

Medium	CVSS Score: 6.1	Open

CWE IDs:

CWE-79

Affected Components:

https://smart.demo.democlouddtest.com/Redesign/VideoPlayer

VideoPlayer.cshtml (Razor View)

RedesignController.cs - VideoPlayer Action

Description

The `/Redesign/VideoPlayer` endpoint accepts a `url` query parameter that is rendered directly into an `<iframe>` element's `src` attribute in the Razor view (`VideoPlayer.cshtml`). The view uses standard Razor encoding (`src="@Model.URL"`), which HTML-encodes characters such as `<`, `>`, `&quot;`, `&#39;`, and `&` to prevent HTML injection.

However, this encoding is insufficient to prevent `javascript:` URI injection. The payload `javascript:alert(1)` consists entirely of alphanumeric characters, a colon, and parentheses — none of which are HTML special characters. As a result, Razor encoding passes the payload through unchanged, producing the following rendered HTML:

```
<iframe src="javascript:alert(1)">
```

When the browser processes this iframe, it executes the JavaScript within the security context of the application origin.

The root cause is a combination of two factors:

1. **Insufficient output encoding:** Razor HTML encoding does not protect against dangerous URI schemes in HTML attributes that accept URLs (such as `src`, `href`, and `action`). URL-context-aware encoding or validation is required.
2. **Missing server-side input validation:** The `RedesignController`'s `VideoPlayer` action accepts the `url` parameter from the query string and passes it to the `VideoViewModel`

without any URI scheme validation, domain allowlisting, or sanitization. Only authentication is enforced via `CustomAuthorizeAttribute`.

Note: The `title` parameter on the same endpoint was also analyzed and confirmed to be safe — it is rendered using standard Razor encoding (`@Model.Title`) within an `<h2>` element, where HTML encoding is effective.

Black-box validation of this finding was attempted but could not be completed due to persistent authentication session expiration during testing. The finding is based on definitive source code analysis confirmed by multiple independent reviews.

Recommendation

1. **Implement server-side URI scheme validation:** Before passing the `url` parameter to the view model, validate that it uses only safe URI schemes. Use `Uri.TryCreate()` to parse the URL and verify that `uri.Scheme` is either `http` or `https`. Reject `javascript:`, `data:`, `vbscript:`, and any other non-HTTP schemes. Example:

```
if (!Uri.TryCreate(url, UriKind.Absolute, out var uri) ||
    (uri.Scheme != "http" && uri.Scheme != "https"))
{
    return new HttpStatusCodeResult(400, "Invalid URL scheme");
}
```

1. **Implement a domain allowlist:** Since this endpoint is designed for embedding video content (e.g., Vimeo), restrict the `url` parameter to known, trusted video hosting domains such as `player.vimeo.com` and `vimeo.com`. Reject any URLs pointing to other domains.
2. **Add a Content-Security-Policy `frame-src` directive:** Configure the `Content-Security-Policy` response header to restrict which URLs can be loaded in iframes on this page. For example:

```
Content-Security-Policy: frame-src https://*.vimeo.com;
```

This provides defense-in-depth even if the server-side validation is bypassed.

1. **Add the `sandbox` attribute to the iframe:** Apply the `sandbox` attribute to the `<iframe>` element to restrict its capabilities. This limits the impact of any injection that bypasses other controls:

```
<iframe src="@Model.URL" sandbox="allow-scripts allow-same-origin">
```

Note that `sandbox` without `allow-scripts` would prevent `javascript:` URIs from executing, but may also break legitimate video embeds. Test the appropriate sandbox flags for your use case.

Reproduction Steps

1. Authenticate to the application as any valid user.
2. Navigate to the following URL in a browser:

```
https://smart.demo.democlouddtest.com/Redesign/VideoPlayer?
title=Video&url=javascript:alert(1)
```

1. Observe that the page renders an **`<iframe>`** element with the **`src`** attribute set to **`javascript:alert(1)`**.
2. Confirm that the JavaScript executes within the browser, demonstrating code execution in the context of the application origin (**`smart.demo.democlouddtest.com`**).
3. To verify the potential for session theft, use the following payload to display the current session cookies:

```
https://smart.demo.democlouddtest.com/Redesign/VideoPlayer?
title=Video&url=javascript:alert(document.cookie)
```

1. To verify via source code analysis, inspect the **`VideoPlayer.cshtml`** view and confirm that line 10 contains:

```
<iframe src="@Model.URL" ...>
```

1. Inspect the **`RedesignController.cs`** and confirm that the **`VideoPlayer`** action passes the **`url`** query parameter directly to the **`VideoViewModel`** without any URI scheme validation or domain allowlisting.

Low Threat Assessment Findings

Business Logic Flaw: Multiple Tax IDs of Same Type Allowed Per Customer

Low	**CVSS Score:** 2.1	Open

CWE IDs:

CWE-840, CWE-841

Affected Components:

`https://acme-corp.demo.dev/api/v1/billing/stripe/tax-ids`

Description

The `/api/v1/billing/stripe/tax-ids` endpoint allows authenticated users to create multiple tax identification numbers of the same type for a single customer account. Standard business logic for tax identification systems typically enforces a constraint of one tax ID per jurisdiction/type per entity, as each business or individual should have only one official tax identifier for a given tax authority.

The API correctly validates that required fields are present and prevents exact duplicate tax IDs (same type and value combination). However, it does not enforce uniqueness constraints based on tax ID type alone. Testing demonstrated that a customer account successfully accumulated three separate `us_ein` (US Employer Identification Number) tax IDs with different values (000000000, 12-3456789, and 88-8888888).

This behavior suggests that the application's business logic validation is incomplete. While the system successfully prevents identical duplicates and validates input formats, it fails to implement the business rule that each customer should have at most one tax ID per type. The root cause appears to be missing uniqueness validation at the application layer that should check for existing tax IDs of the requested type before creating new ones.

Recommendation

Implement business logic validation to enforce tax ID type uniqueness per customer account:

Primary Remediation:

1. Add server-side validation that checks whether a tax ID of the requested type already exists for the customer before creating a new one

2. Return an HTTP 409 Conflict error with a descriptive message when attempting to add a duplicate type, such as: `"Customer already has a tax ID of type us_ein. Please update or remove the existing tax ID before adding a new one."`
3. Document the business requirement clearly: determine whether the system should allow only one tax ID per type globally, or if there are legitimate business cases for multiple tax IDs of the same type (document these exceptions explicitly)

Additional Improvements:

1. If legitimate business requirements exist for multiple same-type tax IDs, implement additional validation rules:
2. Designate one tax ID as primary/default per type
3. Require business justification or approval workflow for additional same-type IDs
4. Add metadata indicating the purpose or scope of each tax ID
5. Implement comprehensive audit logging for all tax ID operations (creation, updates, deletions) to track when multiple same-type IDs are added and by whom
6. Consider adding an update/replace operation (`PUT` or `PATCH`) that atomically replaces an existing tax ID rather than requiring separate delete and create operations
7. Review tax ID handling in downstream billing and reporting systems to ensure they can handle the current state gracefully while the fix is implemented

Reproduction Steps

1. Authenticate to the API using a valid API key in the `X-API-KEY` header
2. Send a POST request to `/api/v1/billing/stripe/tax-ids` with the following JSON body:

```
{"type": "us_ein", "value": "12-3456789"}
```

1. Observe a successful HTTP 200 response containing the newly created tax ID in the `tax_ids` array
2. Send another POST request to the same endpoint with a different value but same type:

```
{"type": "us_ein", "value": "88-8888888"}
```

1. Observe another successful HTTP 200 response
2. Examine the `tax_ids` array in the response body - it will contain multiple entries with `"type": "us_ein"` but different values
3. Both tax IDs are now successfully associated with the same customer account, confirming that multiple tax IDs of the same type can coexist

Appendix A: Assessment Scope Overview

Targets in Scope

- `*.example.com` (hostname) [WILDCARD]
- `sybildemo.com` (hostname)

Access Only Targets

Sybil may access these targets as needed to exercise the application, but active testing is not authorized.

No access-only targets were defined.

Targets Out of Scope

No specific targets were defined as out-of-scope.

Engagements Included

Name	Type	Status	Date
Engagement 2	Full	Cancelled	2026-07-04
Customer Portal Security Test	Full	Completed	2026-02-20

Appendix B: Manual Application Testing and OWASP Testing Methodology

The Open Web Application Security Project (OWASP) Testing Methodology is a framework for evaluating the security of web applications. This methodology provides a comprehensive approach to identifying and mitigating web application vulnerabilities. It is widely respected in the cybersecurity community and often used by security professionals, including penetration testers, to ensure thorough and standardized testing.

Overview of the OWASP Testing Methodology

The OWASP Testing Methodology is divided into several categories, each focusing on different aspects of web application security. These categories are designed to provide a structured approach to identifying security weaknesses.

Information Gathering: In this initial phase, information is gathered about the target application to understand its functionality, architecture, and technologies used. This includes mapping the application, identifying entry points, and understanding the application flow.

Configuration and Deployment Management Testing: This stage involves examining the configuration and deployment processes of the application to identify security issues related to improper configurations, outdated components, and insecure deployment practices.

Identity Management Testing: Mechanisms related to authentication and session management are evaluated. This includes testing for vulnerabilities like weak passwords, session fixation, and insufficient logout mechanisms.

Authentication Testing: This focuses on ensuring that authentication mechanisms are robust and resistant to common attacks such as brute force, credential stuffing, and bypassing authentication controls.

Authorization Testing: Here, the focus is on verifying that access controls are implemented correctly. Issues like privilege escalation, insecure direct object references, and permission bypasses are checked for.

Session Management Testing: This phase tests the application's session management mechanisms to ensure they are secure. This includes testing for session hijacking, session fixation, and cross-site request forgery.

Input Validation Testing: This crucial stage involves testing for vulnerabilities arising from improper user input handling, such as SQL injection, Cross-Site Scripting (XSS), and command injection.

Error Handling and Logging: The application is tested to ensure it handles errors and logs events correctly, looking for vulnerabilities that could leak sensitive information or allow for log tampering.

Data Protection: This involves ensuring that sensitive data is properly encrypted and protected, both in transit and at rest. It includes testing for vulnerabilities related to SSL/TLS, data leakage, and improper data storage practices.

Business Logic Testing: This stage involves testing the application's business logic for flaws that could be exploited. This includes bypassing business rules, abusing functionality, and testing for race conditions.

Client-Side Testing: The application is tested to ensure it handles client-side code such as JavaScript and HTML correctly, looking for vulnerabilities like DOM-based XSS, client-side logic manipulation, and CORS misconfigurations.

The OWASP Testing Methodology provides a structured and comprehensive approach to web application security testing. By following this methodology, a wide range of security vulnerabilities can be identified and mitigated, thereby enhancing the security posture of the web applications being assessed.