ADA LOGICS

Python ECH Implementation Security Audit

In collaboration with OSTIF and the DEfO project

Adam Korczynski, David Korczynski, Ada Logics

February 2026

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

This report is licensed under Creative Commons Attribution Share-Alike 4.0 International

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

Contents

Audit contacts

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org

Introduction

In December 2025 and January 2026, Ada Logics carried out a security audit of the Python integration of defo-projects' ECH module. In particular, we audited this branch: https://github.com/irl/cpython/tree/issue-89730 at commit 4fbb9cc7a674e93b0fd562c2da5610e8fedd483d which had an open PR here: https://github.com/python/cpython/pull/135435. The security audit consisted of threat modelling, manual code review and SAST-assisted auditing. Initially, we created a Docker container in which we could run the updated ECH client both to familiarize ourselves with the work from a user perspective and to verify that it could actually run. After that, we ran a series of SAST tools and reviewed all findings, and finally we audited the code manually. We also reviewed the comments in the code in scope.

The audit resulted in 9 findings ranging from syntax errors to possible memory corruptions. Since the code is open in a PR at this moment and unmerged, the authors of the code did not require any secrecy around findings, and as such, we are sharing the findings with this report with the authors without reporting any of the findings in private.

Despite this being new code, we found no critical security issues in the code, however, we did find issues that could lead to memory corruptions in certain edge cases and issues that could lead to memory leaks of sensitive data. While some of the findings are informational, due to the security-sensitive nature of the implementation, we encourage the developers to remediate all before this PR gets merged - even the issues in the comments.

Risk scoring

During the audit we used a simplified risk scoring system that considers risk exposure and risk impact. Exposure is the level at which an issue is exposed to an attacker. Impact is the level of privilege escalation an attacker can obtain by exploiting the security issue. We score both on a scale of 1-5 and add the two scores together for a final combined score. This score determines the severity of security issues. We assign this severity to the issues we find.

Risk Exposure

- 5: The security issue exists in core component(s) and is exposed in all use cases to untrusted input.
- 4: The security issue exists in widely used component(s) and is enabled by default. Users of the component(s) expose the issue by default to untrusted input.
- 3: The issue is exposed to authenticated and/or authorized users only.
- 2: The issue exists in component(s) that users need to enable to be affected.
- 1: The issue is only exposed to trusted users.

Risk Impact

- 5: An attack will have the highest possible impact.
- 4: An attack will have high impact with some constraints or limitations.
- 3: An attack can cause partial harm.
- 2: An attack can result in privilege escalation that will cause limited harm.
- 1: An attack can result in limited privilege escalation but requires further privilege escalation to cause harm.

We score each issue on both scales and then add the scores for a combined total score. The total score is the basis for the overall severity of found issues.

- 10: Critical
- 9 - 8: High
- 7 - 6: Moderate
- 5 - 4: Low
- 3 - 1: Informational

Scope

The audit covered the changes made in this branch: https://github.com/irl/cpython/tree/issue-89730. At the time of the audit, the branch had an open PR here: https://github.com/python/cpython/pull/135435.

Threat Model

What ECH Protects and Why It Matters

When a user opens a TLS connection, the very first message the client sends – the ClientHello – contains a Server Name Indication (SNI) field that tells the server which hostname the client wants to reach. In traditional TLS (including TLS 1.3 without ECH), this field is sent in cleartext before encryption is established, which means any observer on the network path – an ISP, a corporate proxy, a government firewall, or a malicious WiFi operator – can read the exact hostname and know which website the user is visiting. As a result, an employer can see that an employee is visiting a job-search site, a government can detect access to a dissident news outlet, or an ISP can build a detailed browsing profile for advertising. This has been a well-known privacy gap in TLS since its inception: the protocol encrypts the data exchanged after the handshake, but the initial negotiation – including the destination hostname – has historically been visible to anyone who can observe the connection.

Encrypted Client Hello (ECH) solves this problem by splitting the ClientHello into two layers. The outer layer contains a benign cover hostname (the "outer SNI"), such as `cloudflare-ech.com`, which is visible to network observers. The inner layer contains the real destination hostname (the "inner SNI"), such as `dissent-news.org`, and is encrypted using a public key obtained from DNS. A network observer sees only the cover hostname and a blob of encrypted data; the real destination is revealed only to the terminating server, which holds the corresponding private key.

This design introduces a fundamental tension. ECH must bootstrap its own security from DNS, which is itself an insecure channel in most deployments. The ECH configuration – including the public key used to encrypt the inner ClientHello – is retrieved from DNS HTTPS or SVCB records, which can be spoofed, poisoned, or stripped by an attacker who controls the network. Similarly, when ECH fails, the server sends a retry configuration that the client is expected to trust and use for a subsequent connection attempt. Both of these inputs cross a critical trust boundary: they arrive from the very network infrastructure that ECH is designed to defend against.

Sensitive Data in an ECH Connection

The following fields are carried inside the encrypted inner ClientHello and are considered explicitly sensitive. Leaking any of them – whether through cleartext fallback, memory leaks, or logging – undermines the privacy guarantees that ECH is designed to provide.

- **Inner SNI (Server Name Indication).** When a browser connects to a website, it must tell the server which site it wants – much like addressing a letter. The SNI is that address. The inner SNI is the real destination hostname the client wants to reach (e.g., `dissent-news.org`). This is the primary datum ECH was created to protect and the most privacy-critical field in the handshake.
- **Inner ALPN (Application-Layer Protocol Negotiation).** Before exchanging data, the client and server agree on a "language" to speak – for example, HTTP/2 or HTTP/1.1. ALPN is the mechanism that negotiates this choice. The inner ALPN is the list of application protocols the client is willing to use for the real connection (e.g., `h2`, `http/1.1`). Because the outer ClientHello may advertise a different ALPN list as cover traffic, exposing the inner ALPN reveals the true protocol intent.
- **Inner TLS extensions.** TLS extensions are optional features that the client and server can request during connection setup – things like which encryption versions they support or what cryptographic keys they prefer. Any extension placed in the inner ClientHello – including `supported_versions`, `key_share`, `psk_key_exchange_modes`, `server_name`, and others – is encrypted by ECH. These extensions can fingerprint the client or reveal negotiation capabilities that the outer ClientHello deliberately conceals.
- **ECH retry configuration.** Sometimes the first ECH attempt fails because the server's encryption settings have changed – similar to arriving at a building and finding the locks have been rekeyed. The server then hands back new settings so the

client can try again. This retry configuration contains the server's ECH public key, the cover hostname (`public_name`), and supported cipher suites. If leaked, an attacker learns the server's current ECH deployment parameters, which may help them craft a targeted downgrade attack.

- **The ECH payload itself (the encrypted inner ClientHello).** The ECH payload is the encrypted package that wraps all of the sensitive fields listed above into a single blob that only the destination server can open. Although encrypted on the wire, an implementation that logs, caches, or fails to free the decrypted plaintext after use creates an additional exposure surface beyond what the protocol intends.

Any implementation handling these fields must treat them with the same care as session keys: they should be freed immediately after use, never logged at default verbosity, and never returned to callers in error paths where they are no longer needed.

Threat Actors

We identify four categories of adversaries relevant to the ECH threat model. The first three are network-level attackers that ECH is designed to defend against; the fourth targets the implementation itself.

1. Passive Network Observer. This actor can monitor traffic but cannot modify it. ISPs, corporate network administrators, government surveillance programmes, and public WiFi operators fall into this category. They can read DNS queries, observe the cleartext SNI in TLS ClientHello messages, and collect traffic metadata such as packet sizes and timing. Their goal is to identify which websites a user visits, build browsing profiles, or enforce content policies. ECH directly defeats this actor by encrypting the inner SNI so that only the cover hostname is visible on the wire.

2. Active Network Attacker (Man-in-the-Middle). This actor has all the capabilities of a passive observer but can also inject, modify, or drop packets in transit. Nation-state firewalls, compromised routers, and rogue WiFi access points are examples. Their goals include censorship (blocking access to specific sites), forcing ECH to fail so that the SNI is exposed in cleartext, and stripping the ECH extension from the ClientHello entirely. ECH's retry mechanism and GREASE mode are designed to resist this actor, but the implementation must fail securely – falling back to an unprotected connection defeats the purpose.

3. Malicious or Compromised DNS Infrastructure. Because ECH configurations are distributed via DNS, an attacker who controls or can poison DNS responses occupies a privileged position. They can supply malformed ECH configurations designed to trigger parsing bugs or memory corruption in the client, craft configurations that cause ECH to fail predictably (forcing a fallback to cleartext SNI), or withhold ECH records entirely so that the client never attempts ECH. This actor is particularly dangerous because DNS is fundamentally insecure without DNSSEC, which has limited real-world deployment. ECH must obtain its bootstrap material from this untrusted source, creating a chicken-and-egg problem that the protocol acknowledges but cannot fully solve.

4. Malicious or Compromised Server. A server that terminates the TLS connection can supply crafted retry configurations, force repeated ECH failures to exhaust client resources, or return unexpected data in ECH status fields. While a malicious server can already see the decrypted inner SNI for its own connections, its ability to trigger client-side vulnerabilities – particularly resource exhaustion or memory safety issues that persist across connections to other servers – makes it a relevant threat.

Out of scope. Endpoint compromise (attacker with code execution on the client or server), certificate authority compromise, attacks against TLS 1.3 cryptography itself, and side-channel attacks (timing, power analysis) are outside ECH's threat model and are not considered further.

Implementation Risks

The audited commit introduces five new C functions and one documentation example. For each, we assess whether a network attacker can influence the function's behaviour, and then evaluate the severity of three impact categories — remote code execution, denial of service, and leaking sensitive data — in the context of ECH's privacy guarantees. This analysis is independent of the specific findings reported later in this document; it describes the risk landscape of each function regardless of whether we found a concrete bug in that category.

`set_ech_config()`

Attacker Reachability This function is directly reachable by a network attacker. It receives a raw byte buffer that the application obtained from DNS or from a server's retry response. Although the application mediates the call, the binary content is opaque — the application has no practical way to validate it before passing it through. The actual binary parsing is delegated to OpenSSL's `OSSL_ECHSTORE_read_echconfiglist()`, but the glue code that prepares and cleans up around that call handles the untrusted data directly. An attacker who controls DNS or acts as a man-in-the-middle can craft the bytes that this function processes.

Denial of Service The denial-of-service risk is severe. A crash in this function means the ECH configuration is never loaded, so the privacy-protected connection is never established. If the application falls back to a non-ECH connection (a common pattern), the real destination hostname is sent in cleartext — exactly the outcome ECH was deployed to prevent. Because the attacker controls DNS, they can serve a different malformed configuration on every attempt, ensuring the client never successfully negotiates ECH. In a multi-user service, one poisoned DNS record crashes the entire process, disrupting privacy protection for all users — not just those connecting to the targeted domain.

Remote Code Execution Remote code execution would be very severe in principle. The function runs before any TLS authentication has occurred, so an attacker who gains code execution at this point controls the client process entirely: they can read all subsequent plaintext traffic, extract credentials, or pivot to other systems. However, the function's own glue code does not parse the binary ECH configuration directly; it delegates that to OpenSSL's `OSSL_ECHSTORE_read_echconfiglist()`. The glue code's failure modes are triggered by allocation failures rather than by crafted input, making the realistic exploitation path a crash (denial of service) rather than controlled code execution.

Data Leakage Not applicable. This function processes ECH configuration data (public keys and cipher suite metadata), which is not sensitive — it is distributed via public DNS records. There is no sensitive user data flowing through this function.

`get_ech_status()`

Attacker Reachability This function is indirectly reachable by a network attacker. It takes no external input, but it reads values from the OpenSSL `SSL` object that were determined by the remote server during the TLS handshake. A malicious server cannot control *which* code path the function takes, but it can influence the values returned (inner SNI, outer SNI, status code), and it can attempt to create conditions (such as memory pressure via concurrent connections) that trigger error paths.

Denial of Service The denial-of-service risk is moderate. A crash in this function occurs after the handshake has completed, so the privacy-protected connection was already established — the real hostname was never exposed on the wire for that connection. The severity comes from collateral damage: in a multi-user service, a crash terminates the entire process, disrupting ECH protection for all other users. If the service restarts and those users' connections fall back to non-ECH, their

hostnames are exposed. However, the attacker must be the server the client is connecting to, which limits the attack surface — the attacker already knows the hostname for their own connection.

Remote Code Execution The remote code execution risk is severe but difficult to achieve. A malicious server that gains code execution on the client can read all subsequent traffic and credentials. However, the memory corruption paths in this function require the attacker to induce allocation failures on the client, which is difficult to do reliably from the server side. A crash is the realistic outcome.

Data Leakage The data leakage risk is severe. This function handles the inner SNI — the real destination hostname that ECH exists to protect. If the inner SNI is not freed on an error path, it persists in process memory indefinitely. Over many connections where error paths are triggered, leaked hostnames accumulate into a partial plaintext browsing history. An attacker who later gains read access to the process (via a core dump, a separate memory-disclosure vulnerability, or a debug interface) can recover these hostnames. ECH encrypted them on the wire, but the client's own process stores them unencrypted. The severity is high because the inner SNI is the most privacy-critical datum in the ECH protocol — it is the exact piece of information that ECH was designed to hide.

`get_ech_retry_config()`

Attacker Reachability This function is indirectly reachable by a network attacker. It reads a retry configuration buffer from the OpenSSL `SSL` object. This buffer was supplied by the server during the handshake. A malicious server controls the content and can force the client into repeated retry loops by sending a new retry configuration on every attempt.

Denial of Service The denial-of-service risk is moderate. Each call to this function leaks memory. A malicious server that forces repeated retries causes leaked allocations to accumulate until the process crashes. The attacker's cost is minimal (each retry is a small TLS message), while the client bears the full cost. In a long-running service such as a proxy or crawler, this gradually degrades and eventually destroys the process that provides privacy protection for all users — not just the connection to the attacker's server. However, the degradation is slow (one small leak per retry), so it requires sustained interaction with the malicious server.

Remote Code Execution Not applicable. The function contains no memory corruption paths — it reads a buffer, copies it into a Python object, and returns. The only bug is a missing free, which leads to a leak, not corruption.

Data Leakage The data leakage risk is low. The leaked retry configurations contain the server's ECH public key and the server's public_name (the cover hostname), along with cipher suite preferences. This is metadata about the server's ECH deployment, not the client's real destination hostname — the inner SNI is not present in the retry configuration. An attacker who gains read access to the process can learn the server's current ECH parameters, but this is largely the same information available via DNS. The severity is low because the leaked data does not directly reveal the client's browsing intent.

`_set_outer_alpn_protocols()`

Attacker Reachability This function is not reachable by a network attacker. It receives its input entirely from the application — a byte buffer containing the outer ALPN protocol list. A network attacker cannot influence what the application passes to this function. However, a bug in this function affects what a network observer can infer from the resulting connection.

Denial of Service A traditional crash-based denial of service does not apply, but a bug in this function can cause ECH-protected connections to fail silently rather than crash. If the outer ALPN list is used where the inner list was intended, the server may fail to negotiate a working protocol for the real connection. The user sees a generic TLS error with no indication that the root cause is an ALPN mismatch. Because the failures are intermittent and opaque, developers may disable ECH entirely as a troubleshooting step — a self-inflicted denial of privacy protection.

Remote Code Execution Not applicable. A network attacker cannot influence this function's input.

Data Leakage Traditional data leakage does not apply, but a bug in this function can silently undermine ECH's privacy guarantee. If the inner and outer ALPN lists are conflated, a passive network observer can see the server negotiating a protocol that reveals the client's true intent — for example, a specialised protocol that the cover hostname would never use. The user believes their connection is privacy-protected, but a distinguishing signal in the protocol negotiation tells any observer that the outer hostname is a decoy. This is particularly insidious because neither the client nor the server produces any error — the privacy failure is silent and invisible to the user.

`set_outer_server_name()`

Attacker Reachability This function is not reachable by a network attacker. It receives a hostname string entirely from the application. A network attacker cannot influence the input. However, the function's lack of validation means an application developer can inadvertently undermine ECH by choosing an implausible cover hostname.

Denial of Service Not applicable. The function delegates directly to OpenSSL, checks the return value, and has no failure modes that a network attacker can trigger.

Remote Code Execution Not applicable. A network attacker cannot influence this function's input, and the function's own code is minimal.

Data Leakage No sensitive data flows through this function, so traditional data leakage does not apply. However, if the application sets an implausible outer hostname, a network observer can trivially identify the connection as ECH-protected and infer that the user is deliberately hiding their real destination. This is a misconfiguration risk rather than a vulnerability — ECH becomes a signal that something is being concealed, rather than a shield that hides it.

Documentation Example — ECH Usage Pattern in `ssl.rst`

Attacker Reachability This is a code example, not a runtime function, so a network attacker cannot reach it directly. However, if the example contains a bug, every developer who copies it deploys an application with that bug. The attacker's "reach" is through the developer who trusts the documentation.

Impact The risk is a silent security downgrade. If the canonical example disables the retry mechanism, applications that copy it will silently fall back to cleartext SNI whenever the server rotates its ECH keys — a routine operational event. A passive network observer who knows the rotation schedule can simply wait and read the exposed hostname of every affected client. The attacker needs no active capability. For a security-sensitive protocol, a flawed official example has outsized impact because developers treat documentation as authoritative and are reluctant to deviate from it.

Findings

The following table summarises the issues found in this audit.

ID	Title	Severity	Status
ADA-ECH-1	Memory Leak in `get_ech_status()` Error Path	Low	Reported
ADA-ECH-2	Syntax Error in `set_ech_config()`	Informational	Reported
ADA-ECH-3	Memory Leak in `get_ech_retry_config()`	Low	Reported
ADA-ECH-4	Missing NULL Check in `get_ech_status()`	Moderate	Reported
ADA-ECH-5	Double-Free in `get_ech_status()`	Moderate	Reported
ADA-ECH-6	Uninitialized Variable in `set_ech_config()`	Low	Reported
ADA-ECH-7	Double `PyBuffer_Release` in `set_ech_config()`	Moderate	Reported
ADA-ECH-8	Shared `alpn_protocols` Field Corruption	Low	Reported
ADA-ECH-9	Missing `()` in Documentation ECH Example	Informational	Reported

Memory Leak in `get_ech_status()` Error Path

Severity	Low
Status	Reported
ID	ADA-ECH-1
Component	`Modules/_ssl.c`, lines 2100-2115

The `get_ech_status()` function in Python's SSL module leaks a partially constructed Python tuple when errors occur during ECH status retrieval. The function builds a 3-element tuple containing an ECH status code, the inner SNI (the real destination hostname), and the outer SNI (the cover hostname). If `PyTuple_New(3)` succeeds but a subsequent operation fails – such as `PyUnicode_FromString()` returning NULL – execution jumps to **`goto fail`**, which frees the OpenSSL-allocated C strings but does not free the tuple itself. The tuple remains allocated on the heap with no reference.

Whether the leaked tuple contains sensitive data depends on which error path is taken. If the failure occurs during outer SNI conversion (line 2115), the inner SNI has already been successfully converted to a Python string and inserted into the tuple at index 1. In this case, the leaked tuple contains a Python string copy of the real destination hostname – the datum ECH exists to protect. If the failure occurs earlier (during inner SNI conversion at line 2102), the tuple exists but does not yet contain the inner SNI. In either case, the tuple object itself is leaked.

The error handling code properly frees OpenSSL-allocated strings but forgets to free the Python tuple:

Code pointer:

File: `Modules/_ssl.c`, Lines 2125-2131

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L2125-L2131

```
2125   fail:
2126       if (inner_sni != NULL)
2127           OPENSSL_free(inner_sni);
2128       if (outer_sni != NULL)
2129           OPENSSL_free(outer_sni);
2130       return NULL;  // BUG: retval tuple LEAKED!
```

The tuple is never freed on the error path. If the failure occurred after the inner SNI was inserted (i.e., during outer SNI conversion), the leaked tuple contains a Python string copy of the sensitive destination hostname, which persists in process memory.

Required Fix:

```
2125   fail:
2126       Py_XDECREF(retval);  // Free the tuple if it was allocated
2127       if (inner_sni != NULL)
2128           OPENSSL_free(inner_sni);
2129       if (outer_sni != NULL)
2130           OPENSSL_free(outer_sni);
2131       return NULL;
```

Syntax Error in `set_ech_config()`

Severity	Informational
Status	Reported
ID	ADA-ECH-2
Component	`Modules/_ssl.c`, line 4791

The `set_ech_config()` function in Python's SSL module contains a syntax error that prevents the code from compiling correctly: an extra closing parenthesis appears in the line `es = OSSL_ECHSTORE_new(NULL, NULL));`. This is a trivial typo that would be caught by the compiler.

Code pointer:

File: `Modules/_ssl.c`, Line 4791

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L4788-L4791

```
4788        if (es_in == NULL) {
4789            goto fail;
4790        }
4791        es = OSSL_ECHSTORE_new(NULL, NULL));  // BUG: extra closing parenthesis
```

Memory Leak in `get_ech_retry_config()`

Severity	Low
Status	Reported
ID	ADA-ECH-3
Component	`Modules/_ssl.c`, line 2158

The `get_ech_retry_config()` function in Python's SSL module leaks memory every time it retrieves ECH retry configuration data from a TLS connection. The function calls OpenSSL's `SSL_ech_get1_retry_config()` API, which allocates memory and returns a pointer that the caller is responsible for freeing. After copying this data into a Python bytes object using `PyBytes_FromStringAndSize()`, the original OpenSSL-allocated buffer `ec` is never freed. Because `ec` is a local variable, it goes out of scope when the function returns – no caller ever receives the raw pointer, so the memory is irrecoverably leaked on every successful call.

The leak occurs because `PyBytes_FromStringAndSize()` does not take ownership of the source buffer. It allocates a new Python object and copies the data via `memcpy`, leaving the original untouched. This can be verified from CPython's own implementation:

File: `Objects/bytesobject.c`, Lines 154-160

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Objects/bytesobject.c#L154-L160

```
154      op = (PyBytesObject *)_PyBytes_FromSize(size, 0);  // allocates new object
155      if (op == NULL)
156          return NULL;
157      if (str == NULL)
158          return (PyObject *) op;
159
160      memcpy(op->ob_sval, str, size);  // copies data; source pointer NOT freed
```

The source pointer `str` (which is `ec` in the buggy code) is never freed, stored, or modified by `PyBytes_FromStringAndSize()`. The caller remains responsible for freeing it.

Code pointer:

File: `Modules/_ssl.c`, Lines 2144-2158

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L2144-L2158

```
2144  static PyObject *
2145  _ssl__SSLSocket_get_ech_retry_config_impl(PySSLSocket *self)
2146  /*[clinic end generated code: output=967da2032df9a37a input=0e51b545e93f43ba]*/
2147  {
2148  #ifdef HAVE_OPENSSL_ECH
2149      if (self->ssl == NULL) {
2150          Py_RETURN_NONE;
2151      }
2152      unsigned char *ec;
2153      size_t eclen;
2154      if (SSL_ech_get1_retry_config(self->ssl, &ec, &eclen) != 1) {
```

```
2155            _setSSLError(get_state_sock(self), NULL, 0, __FILE__, __LINE__);
2156            return NULL;
2157        }
2158        return PyBytes_FromStringAndSize((char *)ec, eclen);    // BUG: ec is never
                 freed! It goes out of scope here, leaked forever.
2159    }
```

The correct pattern is already used elsewhere in the same file. For example, `_certificate_to_der()` properly frees the OpenSSL buffer after copying it into a Python bytes object:

File: `Modules/_ssl.c`, Lines 1824-1826

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L1824-L1826

```
1824        retval = PyBytes_FromStringAndSize((const char *) bytes_buf, len);
1825        OPENSSL_free(bytes_buf);  // Correctly freed after copy
1826        return retval;
```

Suggested Fix:

Store the result, free the OpenSSL buffer, then return:

```
2158        PyObject *retval = PyBytes_FromStringAndSize((char *)ec, eclen);
2159        OPENSSL_free(ec);
2160        return retval;
```

`OPENSSL_free(ec)` is safe to call even if `PyBytes_FromStringAndSize()` returned NULL (i.e., failed to allocate), because `ec` is a valid OpenSSL-allocated pointer regardless of the Python allocation outcome.

Missing NULL Check in `get_ech_status()`

Severity	Moderate
Status	Reported
ID	ADA-ECH-4
Component	`Modules/_ssl.c`, line 2100

The `get_ech_status()` function in Python's SSL module contains unsafe error handling that can cause undefined behaviour. When constructing a status tuple, the code calls `PyLong_FromLong()` to create a Python integer object and immediately inserts the result into a tuple using `PyTuple_SET_ITEM()`, without checking whether the allocation succeeded. If memory allocation fails, `PyLong_FromLong()` returns NULL. This NULL value gets inserted directly into the tuple, creating an invalid data structure. When Python code or the garbage collector later attempts to access this tuple, the program may behave in an unexpected manner.

The buggy code inserts the result of `PyLong_FromLong()` directly into the tuple without checking if allocation succeeded:

Code pointer:

File: `Modules/_ssl.c`, Line 2099

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L2099

```
2099   PyTuple_SET_ITEM(retval, 0, PyLong_FromLong(status));  // BUG: No NULL check!
```

If `PyLong_FromLong()` fails and returns NULL, that NULL is inserted into the tuple, creating an invalid tuple that will crash when accessed.

Suggested Fix:

```
2099   PyObject *status_obj = PyLong_FromLong(status);
2100   if (status_obj == NULL) {
2101       goto fail;
2102   }
2103   PyTuple_SET_ITEM(retval, 0, status_obj);
```

Double-Free in `get_ech_status()`

Severity	Moderate
Status	Reported
ID	ADA-ECH-5
Component	`Modules/_ssl.c`, lines 2106-2131

The `get_ech_status()` function contains a possible double-free scenario on the `inner_sni` pointer. On the success path, after `PyUnicode_FromString(inner_sni)` succeeds, the code calls `OPENSSL_free(inner_sni)` to release the OpenSSL-allocated string–but does not set `inner_sni` to NULL. If the subsequent `PyUnicode_FromString(outer_sni)` call fails (e.g., due to memory pressure), execution jumps to **`goto fail`**. The fail path checks **`if`** `(inner_sni != NULL)` - which evaluates to true because the pointer was never zeroed - and calls `OPENSSL_free(inner_sni)` a second time. This is a double-free on heap memory managed by OpenSSL.

`OPENSSL_free` cannot set the caller's pointer to NULL on its own. It is a macro that expands to `CRYPTO_free`:

File: `include/openssl/crypto.h.in`, Lines 131-132 (in OpenSSL)

URL: https://github.com/openssl/openssl/blob/a7b0d678ab4272d8bd48910c8a28c727bb6d2510/include/openssl/crypto.h.in#L131-L132

```
131  #define OPENSSL_free(addr) \
132      CRYPTO_free(addr, OPENSSL_FILE, OPENSSL_LINE)
```

`CRYPTO_free` receives the pointer by value and calls `free`:

File: `crypto/mem.c`, Lines 323-332 (in OpenSSL)

URL: https://github.com/openssl/openssl/blob/a7b0d678ab4272d8bd48910c8a28c727bb6d2510/crypto/mem.c#L323-L332

```
323  void CRYPTO_free(void *str, const char *file, int line)
324  {
325      INCREMENT(free_count);
326      if (free_impl != CRYPTO_free) {
327          free_impl(str, file, line);
328          return;
329      }
330
331      free(str);
332  }
```

Because `str` is a local copy of the pointer, `free(str)` deallocates the memory but the caller's variable still holds the original (now dangling) address. Setting the pointer to NULL after calling `OPENSSL_free` is therefore the caller's responsibility.

Code pointer:

File: `Modules/_ssl.c`, Lines 2100-2132

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L2100-L2132

```
2100      if (inner_sni != NULL) {
2101          v = PyUnicode_FromString(inner_sni);
2102          if (v == NULL) {
2103              goto fail;
2104          }
2105          PyTuple_SET_ITEM(retval, 1, v);
2106          OPENSSL_free(inner_sni);
2107      }
2108      else {
2109          PyTuple_SET_ITEM(retval, 1, Py_None);
2110          Py_INCREF(Py_None);
2111      }
2112      if (outer_sni != NULL) {
2113          v = PyUnicode_FromString(outer_sni);
2114          if (v == NULL) {
2115              goto fail;
2116          }
2117          PyTuple_SET_ITEM(retval, 2, v);
2118          OPENSSL_free(outer_sni);
2119      }
2120      else {
2121          PyTuple_SET_ITEM(retval, 2, Py_None);
2122          Py_INCREF(Py_None);
2123      }
2124      return retval;
2125  fail:
2126      if (inner_sni != NULL) {
2127          OPENSSL_free(inner_sni);
2128      }
2129      if (outer_sni != NULL) {
2130          OPENSSL_free(outer_sni);
2131      }
2132      return NULL;
```

Elsewhere in the same file, `get_ciphers_impl()` uses `Py_CLEAR()` which is a macro that decrements the reference count and sets the pointer to NULL before jumping to a shared cleanup label. This prevents the cleanup path from touching an already-freed object:

File: `Modules/_ssl.c`, Lines 3519-3529

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L3519-L3529

```
3519          if (dct == NULL) {
3520              Py_CLEAR(result);   // frees AND nullifies -- safe for cleanup
3521              goto exit;
3522          }
3523          PyList_SET_ITEM(result, i, dct);
3524      }
3525
3526    exit:
3527      if (ssl != NULL)
3528          SSL_free(ssl);          // cleanup path; result is already NULL
3529      return result;
```

`Py_CLEAR` is CPython's idiomatic solution to exactly this problem for Python objects. For raw C pointers like `inner_sni`, the equivalent discipline is to set the pointer to NULL immediately after freeing.

Suggested Fix:

Set the pointer to NULL immediately after freeing:

```
2100  if (inner_sni != NULL) {
2101      v = PyUnicode_FromString(inner_sni);
2102      if (v == NULL) {
2103          goto fail;
2104      }
2105      PyTuple_SET_ITEM(retval, 1, v);
2106      OPENSSL_free(inner_sni);
2107      inner_sni = NULL;              // ADD THIS LINE
2108  }
```

The same pattern should be applied to `outer_sni` for defensive consistency.

Uninitialized Variable in `set_ech_config()`

Severity	Low
Status	Reported
ID	ADA-ECH-6
Component	`Modules/_ssl.c`, lines 4788-4808

The `set_ech_config()` function declares a local variable `OSSL_ECHSTORE *es;` without initializing it. If the preceding `BIO_new_mem_buf()` call returns NULL (e.g., due to memory allocation failure), execution immediately jumps to `goto fail`. The fail path then reads the uninitialized `es` variable with `if (es != NULL)`. Since `es` was never assigned, it contains whatever garbage value was on the stack. If that garbage value happens to be non-NULL, the code calls `OSSL_ECHSTORE_free(es)` on an arbitrary pointer, resulting in heap corruption or a crash.

This is undefined behavior per the C standard (C11 Section 6.3.2.1), i.e. reading an uninitialized variable of automatic storage duration has no defined semantics.

File: `Modules/_ssl.c`, Lines 4786-4809 (committed version)

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L4786-L4809

```
4786      BIO *es_in = BIO_new_mem_buf(ech_config->buf, ech_config->len);
4787      OSSL_ECHSTORE *es;          // UNINITIALIZED
4788      if (es_in == NULL) {
4789          goto fail;              // jumps with es containing stack garbage
4790      }
4791      es = OSSL_ECHSTORE_new(NULL, NULL));  // only assigned here
4792      if (es == NULL) {
4793          goto fail;
4794      }
4795      if (OSSL_ECHSTORE_read_echconfiglist(es, es_in) != 1) {
4796          goto fail;
4797      }
4798      if (SSL_CTX_set1_echstore(self->ctx, es) != 1) {
4799          goto fail;
4800      }
4801      OSSL_ECHSTORE_free(es);
4802      BIO_free_all(es_in);
4803      PyBuffer_Release(ech_config);
4804      Py_RETURN_NONE;
4805  fail:
4806      _setSSLError(get_state_ctx(self), NULL, 0, __FILE__, __LINE__);
4807      if (es != NULL) {           // READS UNINITIALIZED es!
4808          OSSL_ECHSTORE_free(es); // Frees garbage pointer!
4809      }
```

Suggested Fix:

Initialize `es` to NULL at declaration:

```
4787  OSSL_ECHSTORE *es = NULL;       // Initialize to NULL
```

This ensures the fail path's `if (es != NULL)` check is always safe.

Double `PyBuffer_Release` in `set_ech_config()`

Severity	Moderate
Status	Reported
ID	ADA-ECH-7
Component	`Modules/_ssl.c`, line 4801

The `set_ech_config()` function explicitly calls `PyBuffer_Release(ech_config)` on its success path before returning `Py_RETURN_NONE`. However, the clinic-generated wrapper function (in `Modules/clinic/_ssl.c.h`) also releases the same buffer in its `exit:` block. This means `PyBuffer_Release` is called twice on the same `Py_buffer` structure. The second release can corrupt the reference count of the underlying buffer object, potentially causing a use-after-free or crash.

This is a common pattern bug in CPython C extensions: the clinic framework automatically generates cleanup code for `Py_buffer` parameters, so the `_impl` function must **not** release them manually.

Impl function (receives pointer to the wrapper's local buffer):

File: `Modules/_ssl.c`, Lines 4781-4803 (committed version)

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L4781-L4804

```
4781  _ssl__SSLContext_set_ech_config_impl(PySSLContext *self,
4782                                       Py_buffer *ech_config)  // pointer to
                                             wrapper's local
4783  {
4784      // ...
4785      OSSL_ECHSTORE_free(es);
4786      BIO_free_all(es_in);
4787      PyBuffer_Release(ech_config);   // First release (SHOULD NOT BE HERE)
4788      Py_RETURN_NONE;
4789  }
```

The buffer is released twice: once in the impl function (via the pointer) and once in the clinic wrapper (via the address of the local). Both target the same `Py_buffer`.

Suggested Fix:

Remove the explicit `PyBuffer_Release` from the impl function and let the clinic wrapper handle it:

```
4799  // Success path -- do NOT release the buffer here:
4800      OSSL_ECHSTORE_free(es);
4801      BIO_free_all(es_in);
4802      // PyBuffer_Release(ech_config);  REMOVE THIS LINE
4803      Py_RETURN_NONE;
```

Shared `alpn_protocols` Field Corruption

Severity	Low
Status	Reported
ID	ADA-ECH-8
Component	`Modules/_ssl.c`, lines 3621-3627

CPython's `ssl` module has long provided `_set_alpn_protocols()`, which allows Python code to specify the list of application-layer protocols (e.g., `h2`, `http/1.1`) that a TLS context supports. This function stores the protocol list in `self->alpn_protocols` and registers it with OpenSSL via `SSL_CTX_set_alpn_protos()`. It also installs a server-side callback, `_selectALPN_cb`, which reads `self->alpn_protocols` to perform protocol selection during the handshake.

The audited commit introduces a new function, `_set_outer_alpn_protocols()`, whose purpose is to set the **outer** ALPN list – the cover protocol list that is visible to network observers in the outer ClientHello. ECH requires a separate outer ALPN list because the whole point of the protocol is that the outer ClientHello is cover traffic: if the outer and inner ALPN lists were always identical, a network observer could read the outer ALPN and learn the client's real protocol preferences, leaking information that ECH is designed to hide. For example, a specialised application might negotiate a custom protocol in the inner ALPN but advertise generic `h2` and `http/1.1` in the outer ALPN to blend in with normal HTTPS traffic.

The new function correctly calls `SSL_CTX_ech_set1_outer_alpn_protos()` to register the outer list with OpenSSL's ECH layer, which takes its own internal copy of the data. However, the implementation was seemingly copy-pasted from `_set_alpn_protocols()` and also writes the outer list into `self->alpn_protocols` – the same struct field that the pre-existing `_set_alpn_protocols()` uses to store the real (inner) protocol list. We consider this to be unnecessary: `self->alpn_protocols` exists to feed the server-side `_selectALPN_cb` callback during protocol negotiation, and that callback must see the real protocol list, not the outer cover list. The `SSL_CTX_ech_set1_outer_alpn_protos()` call handles the OpenSSL-level registration independently, so there is no reason for `_set_outer_alpn_protocols()` to touch `self->alpn_protocols` at all.

File: `Modules/_ssl.c`, Lines 296-316

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L296

```
295  typedef struct {
296      // ...
297      unsigned char *alpn_protocols;      // ONE field, shared
298      unsigned int alpn_protocols_len;     // ONE field, shared
299      // ...
300  } PySSLContext;
```

`_set_alpn_protocols` writes to the shared field (line 3588):

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L3588-L3594

```
3588      PyMem_Free(self->alpn_protocols);
3589      self->alpn_protocols = PyMem_Malloc(protos->len);
3590      if (!self->alpn_protocols) {
3591          return PyErr_NoMemory();
3592      }
3593      memcpy(self->alpn_protocols, protos->buf, protos->len);
3594      self->alpn_protocols_len = (unsigned int)protos->len;
```

`_set_outer_alpn_protocols` writes to the same field (line 3621):

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L3621C12-L3627

```
3621      PyMem_Free(self->alpn_protocols);
3622      self->alpn_protocols = PyMem_Malloc(protos->len);
3623      if (!self->alpn_protocols) {
3624          return PyErr_NoMemory();
3625      }
3626      memcpy(self->alpn_protocols, protos->buf, protos->len);
3627      self->alpn_protocols_len = (unsigned int)protos->len;
```

Suggested Fix

Remove the lines in `_set_outer_alpn_protocols()` that write to `self->alpn_protocols`. The function should only call `SSL_CTX_ech_set1_outer_alpn_protos()`, which takes its own copy of the data at the OpenSSL level. The `self->alpn_protocols` field should be left untouched so that `_selectALPN_cb` continues to see the real protocol list:

URL: https://github.com/irl/cpython/blob/4fbb9cc7a674e93b0fd562c2da5610e8fedd483d/Modules/_ssl.c#L3610

```
3611  static PyObject *
3612  _ssl__SSLContext__set_outer_alpn_protocols_impl(PySSLContext *self,
3613                                                  Py_buffer *protos)
3614  {
3615  #ifdef HAVE_OPENSSL_ECH
3616      if ((size_t)protos->len > UINT_MAX) {
3617          PyErr_Format(PyExc_OverflowError,
3618              "protocols longer than %u bytes", UINT_MAX);
3619          return NULL;
3620      }
3621      // Pass directly to OpenSSL; do NOT write to self->alpn_protocols
3622      if (SSL_CTX_ech_set1_outer_alpn_protos(self->ctx, protos->buf,
3623                                             (unsigned int)protos->len))
3624      {
3625          return PyErr_NoMemory();
3626      }
3627      Py_RETURN_NONE;
3628  #else
3629      PyErr_SetString(PyExc_NotImplementedError,
3630                      "OpenSSL does not have ECH support");
3631      return NULL;
3632  #endif
3633  }
```

Missing `()` in Documentation ECH Example Disables Retry Logic

Severity	Informational
Status	Reported
ID	ADA-ECH-9
Component	`Doc/library/ssl.rst`, line 2916

The ECH usage example in the official `ssl` module documentation contains a bug that silently disables the entire retry-config code path. The example compares `ssock.get_ech_status` (the bound method object) against `ECHStatus.ECH_STATUS_GREASE_ECH` (an **enum value**). Because the parentheses `()` are missing, the method is never called, instead, the expression evaluates the method object itself, which is always true but never equal to any `ECHStatus` member. The `if` condition is therefore always `False`, and the retry branch is dead code.

Developers who copy this example verbatim will silently skip ECH retry, falling back to an unprotected connection without any indication that something went wrong.

File: `Doc/library/ssl.rst`, Line 2916 URL to comment: https://github.com/python/cpython/pull/135435/changes#r2653103973.

```
2916  def connect_with_tls_ech(hostname: str, ech_configs: List[str],
2917                           use_retry_config: bool=True) -> ssl.SSLSocket:
2918      context = ssl.create_default_context()
2919      for ech_config in ech_configs:
2920          context.set_ech_config(ech_config)
2921      with socket.create_connection((hostname, 443)) as sock:
2922          with context.wrap_socket(sock, server_hostname=hostname) as ssock:
2923              if (ssock.get_ech_status == ECHStatus.ECH_STATUS_GREASE_ECH  # BUG!
2924                      and use_retry_config):
2925                  return connect_with_ech(hostname, [ssock.get_ech_retry_config()
                          ],
2926                                          False)
2927              return ssock
```

The expression `ssock.get_ech_status` without `()` evaluates to a bound method object, not the ECH status. Bound method objects are true but are never == to an `IntEnum` member, so the comparison is always `False`.

Suggested Fix:

Add the missing parentheses to call the method:

```
2923  # Before (wrong):
2924  if (ssock.get_ech_status == ECHStatus.ECH_STATUS_GREASE_ECH
2925
2926  # After (correct):
2927  if (ssock.get_ech_status() == ECHStatus.ECH_STATUS_GREASE_ECH
```

Also fix the recursive call to use the correct function name:

```
2926  # Before (wrong):
2927  return connect_with_ech(hostname, ...)
```

```
2928
2929  # After (correct):
2930  return connect_with_tls_ech(hostname, ...)
```