SSRF Beyond HTTP: Egress-Path-Incomplete Guards in AI Agent Platforms

Pavan Nallamothu
Independent Security Research
pavanchow

***Abstract*—Server-side request forgery defenses in AI agent platforms are built around one egress chokepoint: a hardened HTTP client that validates every URL it touches. A platform can validate that client perfectly and still ship a second network client, `smtplib.SMTP()`, a raw socket, an FTP dialer, that opens outbound connections through a call site the blocklist never sees. The guard is not weak. It is incomplete in the egress dimension. We define the *Egress-Path-Incomplete* (EPI) SSRF class as a coverage defect, $P \subsetneq E$, where E is the set of outbound clients and P the subset on which the guard actually runs, and we show that any client in $E \setminus P$ is a full-strength SSRF primitive regardless of guard quality. CVE-2026-33234 in AutoGPT (170k+ GitHub stars) is the instance: `validate_url_host()` governs `backend/util/request.py`, while `SendEmailBlock` reaches the network through `smtplib.SMTP(smtp_server, smtp_port)` and calls none of it. A protocol-parser mismatch upgrades the bug from blind to non-blind: `smtplib` expects a `220` greeting, so a non-SMTP service's banner (`SSH-2.0-OpenSSH_8.9p1 Ubuntu`) is placed into an `SMTPConnectError` and echoed to the user. A blind SSRF becomes a non-blind internal port scanner leaking exact version strings at roughly $\log_2 3 \approx 1.58$ bits of topology per API call. We give the fix (guard the socket, not the URL), note that the shipped 0.6.52 patch re-resolves the hostname at connect and so leaves a DNS-rebinding TOCTOU open, and describe a dominator-analysis coverage audit that finds the class mechanically.**

***Index Terms*—SSRF, AI agents, egress control, DNS rebinding, TOCTOU, cloud metadata, CWE-918**

I. Introduction

An AI agent platform is a machine for turning user intent into outbound network calls. Users compose graphs of "blocks," and a block fetches a web page, calls an API, reads a file, or sends an email. Every one of those is an egress path, and each is a place where a server-side application connects somewhere on behalf of a low-privilege user. That is the classic setup for server-side request forgery.

Platforms know this and defend it. AutoGPT ships `validate_url_host()` and a `BLOCKED_IP_NETWORKS` list covering loopback, RFC1918, link-local, and the cloud metadata range. The list is correct. The IDNA handling is correct. The problem is where the guard is wired: into the HTTP client, and nowhere else. A platform has many outbound clients and one guard, and the guard sits at the URL-parsing layer of a single library.

CVE-2026-33234 is what that gap costs. AutoGPT's `SendEmailBlock` takes an SMTP server and port as per-execution input fields, not admin credentials, and opens `smtplib.SMTP(smtp_server, smtp_port)` straight to whatever the user names. The HTTP guard never runs on this path because `smtplib` takes a `(host, port)` pair, not a URL, so a URL-shaped guard cannot even be applied without refactoring. The attacker points the block at `10.0.0.5:22` and the platform connects to internal SSH.

There is a second, sharper problem. A raw SSRF through an unguarded TCP client is normally blind; the attacker learns state only from timing and connection success. `smtplib` removes the blindness for free. It reads the first bytes the target sends, expects a `220` SMTP greeting, and on anything else raises `SMTPConnectError` carrying the offending bytes. Point it at SSH and the exception contains `SSH-2.0-OpenSSH_8.9p1 Ubuntu`; the platform returns that string to the user. The blind primitive becomes a non-blind internal scanner that leaks exact service versions.

Contributions:

- We formalize EPI-SSRF as a coverage defect ($P \subsetneq E$) distinct from a guard-logic defect, and show the guarded set and the client set must be equal for the guard to be sound.
- We give the full anatomy of CVE-2026-33234, including the parser-differential banner leak that amplifies blind to non-blind, with exact code, payload, and output.
- We enumerate sibling egress paths (raw TCP, FTP, DNS, WebSocket, DB drivers) that a URL-centric guard misses.
- We describe a dominator-analysis coverage audit over the taint-to-connect subgraph that flags every unguarded egress mechanically.
- We show the shipped 0.6.52 fix is check-then-connect rather than connect-to-checked-IP, leaving a DNS-rebinding TOCTOU.

II. Background

Server-side request forgery (CWE-918 [1], CAPEC-664 [2]) is the general weakness of an application making an attacker-influenced outbound request. The canonical high-value target is the cloud instance metadata service

at `169.254.169.254`, reachable from inside a VPC and not from the public internet, whose responses can include credentials. Orange Tsai's work established that SSRF is not HTTP-only: URL-parser confusion can smuggle FTP, SMTP, Redis, and Memcached traffic through a fetch primitive [3].

The amplifier in this paper is a parser differential, the same structure as HTTP request smuggling: two components disagree on how to parse the same bytes and the security property lives in the gap [4]–[7]. Here the two components are the target service and `smtplib`, and the leaked artifact is a banner rather than a smuggled request.

The residual bug in the fix is a time-of-check-to-time-of-use flaw of the DNS-rebinding family, first framed for browsers [8] and codified in SSRF guidance [11]. The same validate-then-act pattern produced CVE-2026-27826 in MCP-Atlassian [9] and a sibling advisory in AutoGPT's own HTTP wrapper [10].

The agent-security literature has concentrated elsewhere. Indirect prompt injection [12], agent attack benchmarks [13]–[16], and the OWASP LLM Top 10 [17] all model the LLM-instruction channel: content that manipulates what the agent decides to do. This paper targets the tool and egress channel: what a block's network code does once the agent decides to run it. That seam is largely unmodeled.

III. Threat Model

System under test. A self-hosted or shared multi-tenant AutoGPT deployment. The backend runs inside a private network segment that can reach internal services, Redis, SSH bastions, databases, and the metadata endpoint, that the public internet cannot. The public boundary is the block-execution API.

Attacker capability. Any authenticated low-privilege user, which on a shared or SaaS deployment is a self-service signup. The CVSS vector encodes this as `PR:L`. The attacker controls the `smtp_server` string and `smtp_port` integer of a `SendEmailBlock`, because `SMTPConfig` is a per-execution input field, and reads the block's output and error string, which the execution API returns and persists to the run log.

What the attacker does not need. No prompt injection: this is direct misuse of an exposed capability by the block's operator, not a payload smuggled through retrieved content. No admin role, and no working SMTP credentials, because the connection error fires at or before the greeting, before authentication.

```
[auth user] --HTTPS--> [block-exec API] --smtplib.SMTP()-->
     [internal:port]
                          |
          |
                    HTTP guard lives here              raw
    TCP, guard never runs
                    (validate_url_host)
    banner read as SMTP greeting
                          |
          |
                                  +<---- SMTPConnectError(banner
    ) <----+
```

Listing 1. Attacker to internal, with the guard on the wrong path.

Assets exposed. Internal host liveness, an open-port map, exact service-version banners from services that speak first (SSH, MySQL, FTP), reachability of services that speak second (HTTP, Redis, Memcached), and reachability confirmation of the metadata endpoint. Each leaked version string is a pivot into a CVE lookup.

IV. Methodology

The class is found by enumerating egress, not by fuzzing a URL. The audit lists every network-opening primitive in the codebase, `socket.`, `smtplib.`, `ftplib.`, `asyncio.open_connection`, `http.client`, and driver connects, then asks, for each, whether the guard function dominates every path from a user-tainted destination to the `connect`. This is a dominator analysis over the taint-to-connect subgraph. Any connect reachable from tainted input without a dominating guard call is an EPI finding. Applied to AutoGPT, the procedure is mechanical: the HTTP wrapper has `validate_url_host` on its path; `email_block.py` does not, and it would have surfaced automatically. Dynamically, instrumenting `getaddrinfo` and `connect` at the process boundary and running the agent's blocks against a honeypot subnet flags any connection to a blocked network that did not pass the guard.

V. Formalization

A. The EPI class

Let a server-side application expose outbound clients $E = \{e_1, \ldots, e_n\}$. Each e_i accepts a destination $d = (host, port)$ derived from untrusted input and opens $\text{connect}(e_i, d)$. Let G be the SSRF guard, a predicate $G(d) \in \{\text{allow}, \text{deny}\}$ denying destinations that resolve into a blocked set B. Let $P \subseteq E$ be the egress paths on which G is evaluated before connect.

The EPI defect is $P \subsetneq E$: there exists $e_k \in E \setminus P$ that reaches attacker-influenced destinations without evaluating G. Every such e_k is a full-strength SSRF primitive regardless of how strong G is on the paths in P. This is a coverage defect, not a logic defect. The guard's predicate can be perfect, correct blocklist, correct IDNA handling, correct IPv4-mapped-IPv6 normalization, and the system is still exploitable because the guard is wired into the wrong number of call sites.

For CVE-2026-33234, $E \supseteq \{\text{HTTP wrapper}, \text{SMTP client}\}$, $G =$ `validate_url_host()` enforcing `BLOCKED_IP_NETWORKS` and wired only into the HTTP wrapper, so $P = \{\text{HTTP wrapper}\}$, and $e_k =$ `smtplib.SMTP(smtp_server, smtp_port)` $\in E \setminus P$.

B. The blind-to-non-blind amplifier

A raw SSRF through an unguarded TCP client is normally blind: the attacker infers state from timing and connection success. `smtplib` upgrades it to non-blind because of a parse mismatch between what `smtplib` expects and what the target speaks. Let s be the first bytes the target sends on connect. The target's intended parse $\text{parse}_{\text{target}}(s)$ treats `SSH-2.0-OpenSSH_8.9p1 Ubuntu-3ubuntu0.6\r\n` as a valid SSH banner. `smtplib`'s parse calls `getreply()`, expecting `NNN[ -]text` with a leading `220`; any s that is not a well-formed `2xx/3xx` greeting is an error, and CPython places the offending bytes into the raised `SMTPConnectError`.

The divergence input is any s such that $\text{parse}_{\text{target}}(s)$ is a valid banner but $\text{parse}_{\text{smtp}}(s)$ is an error carrying s. That set is every non-SMTP TCP service that speaks first: SSH, MySQL, FTP, and many databases send a greeting on connect, so all leak a banner. Services that speak second, HTTP, Redis, and Memcached, wait for a client command before sending anything; against them `smtplib.SMTP(host, port)` completes `connect()` and blocks in `getreply()` to its timeout, so they leak no banner but still confirm reachability through connect success versus `ConnectionRefusedError`, enough to fingerprint the metadata endpoint.

C. Real numbers

Attacker cost per probe is one authenticated block execution, one TCP connect and one banner read. The patched client uses `timeout=30`; a scanner tunes far lower, and `smtp_port` is an unrestricted integer in the vulnerable version. A /24 internal subnet times the roughly 20 top service ports is $256 \times 20 = 5{,}120$ probes to fingerprint a subnet, each a single API call, with no rate control on the vulnerable path. The signal per probe is a ternary oracle: a banner in `SMTPConnectError` means the host is up, the port is open, and the version is leaked; `ConnectionRefusedError` means the host is up and the port is closed; a timeout means the host is down or filtered. That is at most $\log_2 3 \approx 1.58$ bits of network-topology information per API call, an upper bound under a uniform ternary outcome that a real subnet does not achieve, with the open-port case additionally leaking a full version string. CVSS 5.0 Moderate, `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:L/I:N/A:N`; `S:C` because the vulnerable component reaches resources beyond its own authority, `C:L` because it leaks banners and topology rather than full data. CWE-918.

VI. Case Studies

A. CVE-2026-33234: the vulnerable data flow

`SMTPConfig` is user input, not a credential, which is what makes the destination attacker-controlled:

```
class SMTPConfig(BaseModel):
    smtp_server: str = SchemaField(description="SMTP server
      address")
    smtp_port: int = SchemaField(default=25, description="
     SMTP port number")
    model_config = ConfigDict(title="SMTP Config")

class Input(BlockSchemaInput):
    ...
    config: SMTPConfig = SchemaField(description="SMTP
     Config")  # per-execution, user-controlled
```

The unguarded egress call opens a raw TCP socket straight to the user-supplied destination:

```
with smtplib.SMTP(smtp_server, smtp_port, timeout=30) as
     server:  # raw TCP, no blocklist check
    server.starttls()
    server.login(smtp_username, smtp_password)
    server.sendmail(smtp_username, to_email, msg.as_string
     ())
```

The guard that never runs on this path lives in `backend/util/request.py`:

```
BLOCKED_IP_NETWORKS = [
    ipaddress.ip_network("0.0.0.0/8"),       # "This"
     Network
    ipaddress.ip_network("10.0.0.0/8"),      # Private-Use
    ipaddress.ip_network("100.64.0.0/10"),   # CGNAT (RFC
     6598)
    ipaddress.ip_network("127.0.0.0/8"),     # Loopback
    ipaddress.ip_network("169.254.0.0/16"), # Link Local
     <-- IMDS lives here
    ipaddress.ip_network("172.16.0.0/12"),   # Private-Use
    ipaddress.ip_network("192.0.0.0/24"),
    ipaddress.ip_network("192.168.0.0/16"), # Private-Use
    ipaddress.ip_network("198.18.0.0/15"),
    ipaddress.ip_network("224.0.0.0/4"),
    ipaddress.ip_network("240.0.0.0/4"),
    ipaddress.ip_network("::1/128"),
    ipaddress.ip_network("fc00::/7"),
    ipaddress.ip_network("fe80::/10"),
    ipaddress.ip_network("ff00::/8"),
]
```

`validate_url_host()` parses the URL, restricts the scheme to http and https, IDNA-encodes the host, then calls `resolve_and_check_blocked()`. The SMTP path in the vulnerable version calls none of it. The advisory pins the leak to `email_block.py` line 184, which re-raises the unhandled `SMTPConnectError` with the banner intact.

B. Trigger and observed result

Block input:

```
{
  "block_id": "send_email_block",
  "inputs": {
    "smtp_server": "10.0.0.5",
    "smtp_port": 22,
    "to": "attacker@evil.com",
    "subject": "test",
    "body": "test"
  }
}
```

Direct API call:

TABLE I
ORACLE OUTPUT BY TARGET SERVICE.

Target	Port	Banner leaked	Oracle meaning
SSH	22	`SSH-2.0-OpenSSH_8.9p1`	up, open, version
MySQL	3306	`5.7.42-0ubuntu0.18.04.1`	up, open, version
FTP	21	`220 (vsFTPd 3.0.3)`	up, open, version
Redis	6379	none (server waits)	up, open, no banner
Closed	9999	`ConnectionRefusedError`	up, closed
Metadata	80	reachability confirmed	IMDS reachable

TABLE II
EGRESS PATHS AND GUARD COVERAGE.

Egress path	Client	Guarded?	Leak channel
HTTP/S	requests/httpx	Yes	response body
SMTP	smtplib.SMTP	**No**	`SMTPConnectError` banner
Raw TCP	socket	No	recv bytes / timing
FTP	ftplib.FTP	No	220 welcome banner
DNS	dns.resolver	No	timing, rebinding pivot
WebSocket	websockets	No	HTTP 101 / server header
DB drivers	redis/pymysql	No	handshake error strings

```
curl -X POST https://autogpt-platform/api/blocks/execute \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '@payload.json'
```

Illustrative response, reconstructed from the advisory and not captured on a live 0.6.51 instance:

```
{ "error": "SMTP connection error carrying the target's
    banner bytes, e.g. b'...OpenSSH_8.9p1 Ubuntu'" }
```

The exact wrapper depends on CPython internals: `smtplib.getreply()` strips the leading four bytes (`line[4:]`) when the reply is not a well-formed `NNN` code, so the leaked substring is the banner tail rather than the full `SSH-` prefix, and a first-byte disconnect surfaces as `SMTPServerDisconnected` rather than `SMTPConnectError`. The leak of the service banner holds in either case; the byte-exact string is advisory-derived. Table I is the target matrix from the write-up.

The chain stated in the source: iterate IP ranges by common ports to map live hosts and services from block error types and timing; feed leaked version strings into CVE lookups; if an internal Redis is unauthenticated and unpatched, common in development, the "send an email" block becomes remote code execution on an internal service.

C. The fix and its residual gap

The 0.6.52 `run()` adds a port allowlist, a pre-connect resolve-and-check, and swallows the banner:

```
async def run(self, input_data, *, credentials, **kwargs):
    try:
        smtp_port = input_data.config.smtp_port
        if smtp_port not in self.ALLOWED_SMTP_PORTS:
            yield "error", (f"SMTP port {smtp_port} is not
     allowed. "
                            f"Allowed ports: {sorted(self.
     ALLOWED_SMTP_PORTS)}")
            return
        await resolve_and_check_blocked(input_data.config.
     smtp_server)  # guard now on SMTP path
        status = self.send_email(...)
        yield "status", status
    except smtplib.SMTPConnectError:
        yield "error", (f"Cannot connect to SMTP server "
                        f"'{input_data.config.smtp_server}'
      on port "
                        f"{input_data.config.smtp_port}.")
      # banner no longer echoed
```

Fixed in autogpt-platform-backend 0.6.52; affected $\geq$ 0.1.0, $<$ 0.6.52. The patch is check-then-connect, not connect-to-checked-IP. `resolve_and_check_blocked(host)` resolves and validates, then `send_email` calls `smtplib.SMTP(host, port)`, which re-resolves the same hostname. That is a validate-then-act TOCTOU. A DNS-rebinding attacker with a short TTL, answering public on the first lookup and `169.254.169.254` on the second, would still pass the check and connect internally, because the connection is not pinned to the validated IP. We identify this residual gap from the structure of the patched `run()` (validate then re-resolve), and do not claim a live demonstration against a 0.6.52 instance. This is the same residual bug that hit the project's HTTP wrapper [10] and the MCP-Atlassian fix [9]. The correct fix pins the socket to the validated address (Section VII).

D. Generalization: sibling egress paths

Table II enumerates E for a typical AI agent platform and marks which paths a single HTTP-centric guard misses. Each row is a candidate case study.

The rule the table implies: the guard must sit at the socket layer, or at a single egress proxy every client is forced through, not at the URL-parsing layer of one library. A guard at URL validation covers only clients that take a URL and route through the wrapped call site. `smtplib` takes `(host, port)`, so a URL-shaped guard cannot even be applied to it without a refactor.

VII. MITIGATION

Defense in depth, ordered.

1) **Single egress chokepoint.** Force every outbound client through one guarded interface or a dedicated egress proxy (the Smokescreen model) that resolves and connects atomically. No block gets a bare `socket`, `smtplib`, or `ftplib` handle. This closes the class, not the instance.
2) **Socket-level connection pinning.** Resolve once, validate the resulting IP against `BLOCKED_IP_NETWORKS`, then connect to that pinned IP. This defeats the DNS-rebinding TOCTOU the 0.6.52 check-then-connect fix leaves open.

3) **Port allowlist** for SMTP (25/465/587), present in 0.6.52, plus deny of metadata, loopback, and RFC1918.
4) **Error sanitization.** Never echo a transport-layer exception payload to the user; map `SMTPConnectError` to a generic message so even a bypass yields no banner.
5) **Config versus input separation.** Treat the SMTP server as an admin-scoped credential, not a per-execution input, so a low-privilege user cannot choose the destination at all.

The detection method is the dominator-analysis coverage audit of Section IV, complemented by process-boundary instrumentation of `getaddrinfo` and `connect` against a honeypot subnet, an approach that extends REST-API SSRF fuzzing [19] to the socket layer where existing SSRF detectors, which are HTTP-URL-centric [18], do not look.

VIII. Related Work

SSRF and protocol smuggling establish that the weakness is not HTTP-only [1]–[3]. The parser-differential literature supplies the banner-leak template: two components disagree on the same bytes [4]–[7]. DNS-rebinding and TOCTOU work underpins the residual-gap analysis [8]–[11]. Agent-security research models the LLM-instruction channel through injection studies and benchmarks [12]–[17]. SSRF detection work is HTTP-URL-centric [18], [19]. The gap this paper fills sits between them: agent-security work models what the agent is told to do, SSRF work models the HTTP-URL channel, and neither models multi-egress coverage in agent tool code.

IX. Conclusion

A URL-shaped guard can only defend URL-shaped clients. An agent platform has many outbound clients and typically one guard, so the sound property is not guard strength but guard coverage: the set of guarded egress paths must equal the set of egress paths. CVE-2026-33234 is one unguarded door, and the `smtplib` banner leak turns it from a blind primitive into a non-blind scanner. Guard the socket, audit coverage, and pin the connection to the address you validated.

References

[1] MITRE, "CWE-918: Server-Side Request Forgery (SSRF)," Common Weakness Enumeration.
[2] MITRE, "CAPEC-664: Server Side Request Forgery," Common Attack Pattern Enumeration and Classification.
[3] C.-D. Tsai, "A New Era of SSRF: Exploiting URL Parsers in Trending Programming Languages," *Black Hat USA*, 2017.
[4] J. Kettle, "HTTP Desync Attacks: Request Smuggling Reborn," *PortSwigger Research / Black Hat USA*, 2019.
[5] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: HTTP Request Smuggling with Differential Fuzzing," *ACM CCS*, 2021.
[6] "The HTTP Garden: Discovering Parsing Vulnerabilities in HTTP/1.1 Implementations by Differential Fuzzing of Request Streams," *arXiv:2405.17737*, 2024.
[7] "Gudifu: Guided Differential Fuzzing for HTTP Request Parsing Discrepancies," *RAID*, 2024.
[8] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting Browsers from DNS Rebinding Attacks," *ACM CCS*, 2007.
[9] "MCP-Atlassian: DNS-Rebinding TOCTOU Bypass of the SSRF Fix," GHSA-489g-7rxv-6c8q / CVE-2026-27826, 2026.
[10] Significant-Gravitas/AutoGPT, "SSRF via DNS Rebinding in the requests wrapper," GHSA-wvjg-9879-3m7w, 2026.
[11] OWASP, "A10:2021 — Server-Side Request Forgery (SSRF)," OWASP Top 10, 2021.
[12] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," *ACM AISec*, 2023.
[13] E. Debenedetti et al., "AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents," *NeurIPS*, 2024.
[14] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated LLM Agents," *ACL Findings*, 2024.
[15] Y. Ruan et al., "Identifying the Risks of LM Agents with an LM-Emulated Sandbox," *ICLR*, 2024.
[16] "Agent Security Bench (ASB)," *ICLR*, 2025.
[17] OWASP, "OWASP Top 10 for LLM Applications 2025 — LLM01: Prompt Injection," 2025.
[18] K. Al-talak and O. Abbass, "Detecting Server-Side Request Forgery (SSRF) Attack by Using Deep Learning Techniques," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 12, no. 12, 2021, doi:10.14569/IJACSA.2021.0121230.
[19] "Detecting Server-Side Request Forgery (SSRF) Vulnerabilities in REST API Fuzz Testing," *SBFT/ICSE Workshop*, 2026.