PrefectHQ / **prefect** Public

Code Issues 818 Pull requests 39 Discussions Actions Security an

Commit `7c70ac5`

devin-ai-integration[bot] and **desertaxle** authored 2 weeks ago · 94 / 99 · Verified

```
Fix DNS rebinding TOCTOU bypass in validate_restricted_url (#21591)

Co-authored-by: Devin AI <158243242+devin-ai-integration[bot]@users.noreply.github.com>
Co-authored-by: Alexander Streed <alex.s@prefect.io>
Co-authored-by: alex.s <ajstreed1@gmail.com>
```

main (#21591) · prefect-redis-0.2.11 ••• 3.6.28.dev2

1 parent 3458636 commit 7c70ac5

4 files changed +640 -21

Top

Filter files…

- src/prefect
  - blocks
    - notifications.py
    - webhook.py
  - utilities
    - urls.py
- tests/utilities
  - test_urls.py

Search within code

src/prefect/blocks/notifications.py

```
@@ -13,7 +13,11 @@
13  13  from prefect.logging import LogEavesdropper
```

```
14  14    from prefect.types import SecretDict
15  15    from prefect.utilities.templating import apply_values, find_placeholders
16      - from prefect.utilities.urls import validate_restricted_url
    16  + from prefect.utilities.urls import (
    17  +     SSRFProtectedAsyncHTTPTransport,
    18  +     SSRFProtectedHTTPTransport,
    19  +     validate_restricted_url,
    20  + )
17  21
18  22    PREFECT_NOTIFY_TYPE_DEFAULT = "info"  # Use a valid apprise type as default
19  23
          @@ -987,13 +991,18 @@ async def anotify(self, body: str, subject: str |
          None = None) -> None:
987 991           import httpx
988 992
989 993           request_args = self._build_request_args(body, subject)
    994 +         client_kwargs: dict[str, Any] = {
    995 +             "headers": {"user-agent": "Prefect Notifications"},
    996 +         }
990 997           if not self.allow_private_urls:
991 998               validate_restricted_url(request_args["url"])
    999 +             # Re-validate at connection time and pin the resolved IP to close
   1000 +             # the DNS-rebinding TOCTOU window.
   1001 +             client_kwargs["transport"] = SSRFProtectedAsyncHTTPTransport()
992 1002          cookies = request_args.pop("cookies", dict())
   1003 +         client_kwargs["cookies"] = cookies
993 1004          # make request with httpx
994     -         async with httpx.AsyncClient(
995     -             headers={"user-agent": "Prefect Notifications"}, cookies=cookies
996     -         ) as client:
    1005 +         async with httpx.AsyncClient(**client_kwargs) as client:
997 1006              resp = await client.request(**request_args)
998 1007          resp.raise_for_status()
999 1008
          @@ -1002,13 +1011,16 @@ def notify(self, body: str, subject: str | None =
          None) -> None:
1002 1011          import httpx
1003 1012
1004 1013          request_args = self._build_request_args(body, subject)
    1014 +         client_kwargs: dict[str, Any] = {
```

```
                1015 +             "headers": {"user-agent": "Prefect Notifications"},
                1016 +         }
1005            1017           if not self.allow_private_urls:
1006            1018               validate_restricted_url(request_args["url"])
                1019 +             client_kwargs["transport"] = SSRFProtectedHTTPTransport()
1007            1020           cookies = request_args.pop("cookies", dict())
                1021 +         client_kwargs["cookies"] = cookies
1008            1022           # make request with httpx
1009            -              with httpx.Client(
1010            -                  headers={"user-agent": "Prefect Notifications"}, cookies=cookies
1011            -              ) as client:
                1023 +         with httpx.Client(**client_kwargs) as client:
1012            1024               resp = client.request(**request_args)
1013            1025           resp.raise_for_status()
1014            1026
```

src/prefect/blocks/webhook.py

```
                @@ -8,12 +8,21 @@
8    8
9    9      from prefect.blocks.core import Block
10   10     from prefect.types import SecretDict
11   -      from prefect.utilities.urls import validate_restricted_url
          11 + from prefect.utilities.urls import (
          12 +     SSRFProtectedAsyncHTTPTransport,
          13 +     validate_restricted_url,
          14 + )
12   15
13   16     # Use a global HTTP transport to maintain a process-wide connection pool for
14   17     # interservice requests
15   18     _http_transport = AsyncHTTPTransport()
16   19     _insecure_http_transport = AsyncHTTPTransport(verify=False)
          20 + # Separate pools for calls that must be protected from DNS-rebinding SSRF.  The
          21 + # protected transport validates the resolved IP at connection time and connects
          22 + # to the pre-resolved address, closing the TOCTOU window exploited by DNS
          23 + # rebinding attacks.
          24 + _safe_http_transport = SSRFProtectedAsyncHTTPTransport()
          25 + _safe_insecure_http_transport = SSRFProtectedAsyncHTTPTransport(verify=False)
17   26
18   27
```

```
19  28    class Webhook(Block):
          @@ -53,10 +62,13 @@ class Webhook(Block):
53  62        )
54  63
55  64        def block_initialization(self) -> None:
56      -         if self.verify:
57      -             self._client = AsyncClient(transport=_http_transport)
        65  +         if self.allow_private_urls:
        66  +             transport = _http_transport if self.verify else
            _insecure_http_transport
58  67            else:
59      -             self._client = AsyncClient(transport=_insecure_http_transport)
        68  +             transport = (
        69  +                 _safe_http_transport if self.verify else
            _safe_insecure_http_transport
        70  +             )
        71  +         self._client = AsyncClient(transport=transport)
60  72
61  73        async def call(self, payload: dict[str, Any] | str | None = None) ->
          Response:
62  74            """
```

src/prefect/utilities/urls.py

```
        ...   @@ -1,13 +1,18 @@
1   1     import inspect
2   2     import ipaddress
3   3     import socket
      4   + import time
4   5     import urllib.parse
      6   + from collections.abc import Iterable
5   7     from logging import Logger
6   8     from string import Formatter
7   9     from typing import TYPE_CHECKING, Any, Literal, Optional, Union, cast
8   10    from urllib.parse import urlparse
9   11    from uuid import UUID
10  12
      13  + import anyio.to_thread
      14  + import httpcore
```

```
      15  + import httpx
11    16    from pydantic import BaseModel
12    17
13    18    from prefect import settings
            @@ -71,7 +76,14 @@ def validate_restricted_url(url: str) -> None:
71    76        Validate that the provided URL is safe for outbound requests.  This
            prevents
72    77        attacks like SSRF (Server Side Request Forgery), where an attacker can make
73    78        requests to internal services (like the GCP metadata service, localhost
            addresses,
74        -     or in-cluster Kubernetes services)
          79  +     or in-cluster Kubernetes services).
          80  +
          81  +     This is a pre-flight check that validates every address the hostname
            resolves
          82  +     to via `getaddrinfo`.  Because DNS can change between this check and the
          83  +     actual HTTP connection, callers that need hardened SSRF protection should
          84  +     also use `SSRFProtectedAsyncHTTPTransport` / `SSRFProtectedHTTPTransport`,
          85  +     which re-validate at connection time and connect to the pre-resolved IP to
          86  +     close the TOCTOU window exploited by DNS rebinding attacks.
75    87
76    88        Args:
77    89            url: The URL to validate.
            @@ -100,17 +112,227 @@ def validate_restricted_url(url: str) -> None:
100   112            raise ValueError(f"{url!r} is not a valid URL.")
101   113
102   114        try:
103       -         ip_address = socket.gethostbyname(hostname)
104       -         ip = ipaddress.ip_address(ip_address)
          115  +         _validate_resolved_hostname(hostname)
          116  +     except _RestrictedHostError as exc:
          117  +         raise ValueError(f"{url!r} is not a valid URL.  {exc}")
          118  +
          119  +
          120  + class _RestrictedHostError(Exception):
          121  +     """Internal exception raised when a hostname resolves to a restricted
            address."""
          122  +
```

```
123 +
124 + def _validate_resolved_hostname(hostname: str) -> list[str]:
125 +     """Resolve `hostname` and validate every returned address.
126 +
127 +     Returns the list of resolved IPs (as strings) in resolution order.  Raises
128 +     `_RestrictedHostError` if any resolved address is private, or if the
129 +     hostname cannot be resolved.
130 +
131 +     Using `getaddrinfo` (rather than `gethostbyname`, which returns only the
132 +     first A record) closes an SSRF bypass where a hostname publishes both a
133 +     public and a private A/AAAA record: every resolved address is checked.
134 +     """
135 +     # IP literal: validate directly.
136 +     try:
137 +         ip = ipaddress.ip_address(hostname)
138 +     except ValueError:
139 +         pass
140 +     else:
141 +         if ip.is_private:
142 +             raise _RestrictedHostError(f"It resolves to the private address
    {ip}.")
143 +         return [str(ip)]
144 +
145 +     try:
146 +         addrinfos = socket.getaddrinfo(hostname, None, type=socket.SOCK_STREAM)
105 147         except socket.gaierror:
148 +         raise _RestrictedHostError("It could not be resolved.")
149 +
150 +     resolved: list[str] = []
151 +     for addrinfo in addrinfos:
152 +         sockaddr = addrinfo[4]
153 +         ip_str = sockaddr[0]
154 +         # Strip IPv6 zone identifier if present (e.g. "fe80::1%eth0")
155 +         ip_str = ip_str.split("%", 1)[0]
106 156             try:
107 -               ip = ipaddress.ip_address(hostname)
157 +               ip = ipaddress.ip_address(ip_str)
108 158             except ValueError:
109 -               raise ValueError(f"{url!r} is not a valid URL.  It could not be
    resolved.")
```

```
                159  +             continue
                160  +         if ip.is_private:
                161  +             raise _RestrictedHostError(f"It resolves to the private address
                     {ip}.")
                162  +         if ip_str not in resolved:
                163  +             resolved.append(ip_str)
110             164
111             -     if ip.is_private:
112             -         raise ValueError(
113             -             f"{url!r} is not a valid URL.  It resolves to the private address
                     {ip}."
                165  +     if not resolved:
                166  +         raise _RestrictedHostError("It could not be resolved.")
                167  +
                168  +     return resolved
                169  +
                170  +
                171  + class _SSRFProtectedAsyncBackend(httpcore.AsyncNetworkBackend):
                172  +     """An `httpcore.AsyncNetworkBackend` that validates resolved addresses.
                173  +
                174  +     Wraps an existing backend and, on each `connect_tcp` call, resolves the
                175  +     hostname itself, rejects any resolved address that is private, and then
                176  +     connects to the validated IP directly (rather than the hostname) so that
                177  +     the underlying backend cannot re-resolve to a different address.
                178  +
                179  +     TLS SNI / certificate validation is unaffected because httpcore passes the
                180  +     original hostname to `start_tls` via `server_hostname`, independently of
                181  +     the host used for the TCP connection.
                182  +     """
                183  +
                184  +     def __init__(self, wrapped: httpcore.AsyncNetworkBackend) -> None:
                185  +         self._wrapped = wrapped
                186  +
                187  +     async def connect_tcp(
                188  +         self,
                189  +         host: str,
                190  +         port: int,
                191  +         timeout: Optional[float] = None,
                192  +         local_address: Optional[str] = None,
                193  +         socket_options: Optional[Iterable[Any]] = None,
```

```
194 +     ) -> httpcore.AsyncNetworkStream:
195 +         # Resolve in a worker thread so the event loop is not blocked during
196 +         # DNS lookups (which can be slow or intermittently failing).
197 +         validated_ips = await anyio.to_thread.run_sync(
198 +             _resolve_and_validate_for_connect, host
199 +         )
200 +         last_exc: Optional[BaseException] = None
201 +         deadline = time.monotonic() + timeout if timeout is not None else None
202 +         for ip in validated_ips:
203 +             remaining = _remaining_timeout(deadline)
204 +             if remaining == 0.0:
205 +                 break
206 +             try:
207 +                 return await self._wrapped.connect_tcp(
208 +                     ip,
209 +                     port,
210 +                     timeout=remaining,
211 +                     local_address=local_address,
212 +                     socket_options=socket_options,
213 +                 )
214 +             except (httpcore.ConnectError, httpcore.ConnectTimeout, OSError) as
      exc:
215 +                 last_exc = exc
216 +         assert last_exc is not None
217 +         raise last_exc
218 +
219 +     async def connect_unix_socket(
220 +         self,
221 +         path: str,
222 +         timeout: Optional[float] = None,
223 +         socket_options: Optional[Iterable[Any]] = None,
224 +     ) -> httpcore.AsyncNetworkStream:
225 +         return await self._wrapped.connect_unix_socket(
226 +             path, timeout=timeout, socket_options=socket_options
227 +         )
228 +
229 +     async def sleep(self, seconds: float) -> None:
230 +         await self._wrapped.sleep(seconds)
231 +
232 +
```

```
233 + class _SSRFProtectedSyncBackend(httpcore.NetworkBackend):
234 +     """Synchronous counterpart of `_SSRFProtectedAsyncBackend`."""
235 +
236 +     def __init__(self, wrapped: httpcore.NetworkBackend) -> None:
237 +         self._wrapped = wrapped
238 +
239 +     def connect_tcp(
240 +         self,
241 +         host: str,
242 +         port: int,
243 +         timeout: Optional[float] = None,
244 +         local_address: Optional[str] = None,
245 +         socket_options: Optional[Iterable[Any]] = None,
246 +     ) -> httpcore.NetworkStream:
247 +         validated_ips = _resolve_and_validate_for_connect(host)
248 +         last_exc: Optional[BaseException] = None
249 +         deadline = time.monotonic() + timeout if timeout is not None else None
250 +         for ip in validated_ips:
251 +             remaining = _remaining_timeout(deadline)
252 +             if remaining == 0.0:
253 +                 break
254 +             try:
255 +                 return self._wrapped.connect_tcp(
256 +                     ip,
257 +                     port,
258 +                     timeout=remaining,
259 +                     local_address=local_address,
260 +                     socket_options=socket_options,
261 +                 )
262 +             except (httpcore.ConnectError, httpcore.ConnectTimeout, OSError) as
    exc:
263 +                 last_exc = exc
264 +         assert last_exc is not None
265 +         raise last_exc
266 +
267 +     def connect_unix_socket(
268 +         self,
269 +         path: str,
270 +         timeout: Optional[float] = None,
271 +         socket_options: Optional[Iterable[Any]] = None,
```

```
272 +     ) -> httpcore.NetworkStream:
273 +         return self._wrapped.connect_unix_socket(
274 +             path, timeout=timeout, socket_options=socket_options
275 +         )
276 +
277 +     def sleep(self, seconds: float) -> None:
278 +         self._wrapped.sleep(seconds)
279 +
280 +
281 + def _remaining_timeout(deadline: Optional[float]) -> Optional[float]:
282 +     """Return the time left until `deadline`, or `None` if no deadline is set.
283 +
284 +     Clamps to `0.0` when the deadline has already passed.  Callers use this to
285 +     share a single connect-timeout budget across multiple per-IP retries so
286 +     that connect time stays bounded (roughly) by the caller's timeout rather
287 +     than scaling with the number of resolved addresses.
288 +     """
289 +     if deadline is None:
290 +         return None
291 +     return max(0.0, deadline - time.monotonic())
292 +
293 +
294 + def _resolve_and_validate_for_connect(host: str) -> list[str]:
295 +     """Resolve `host` and return all safe IPs to connect to.
296 +
297 +     Every returned IP has been validated against the private-address blocklist;
298 +     callers iterate them in order and retry on connect failures so that dual-
299 +     stack hostnames still work in single-stack environments.
300 +
301 +     Raises `httpcore.ConnectError` if any resolved address is private or if the
302 +     hostname cannot be resolved.  The returned IPs are passed to the underlying
303 +     network backend as IP literals, so it will not perform further DNS
304 +     resolution — eliminating the DNS rebinding TOCTOU window.
305 +     """
306 +     try:
307 +         return _validate_resolved_hostname(host)
308 +     except _RestrictedHostError as exc:
309 +         raise httpcore.ConnectError(f"Refusing to connect to {host!r}: {exc}")
      from None
310 +
```

```
      311 +
      312 + class SSRFProtectedAsyncHTTPTransport(httpx.AsyncHTTPTransport):
      313 +     """An `httpx.AsyncHTTPTransport` that guards against DNS rebinding SSRF.
      314 +
      315 +     Behaves identically to `httpx.AsyncHTTPTransport` except that, for every
      316 +     request, the hostname is resolved, every resolved address is checked
      317 +     against the private-address blocklist, and the connection is made to the
      318 +     specific validated IP.  This closes the TOCTOU window between a pre-flight
      319 +     `validate_restricted_url` check and the actual HTTP connection.
      320 +     """
      321 +
      322 +     def __init__(self, *args: Any, **kwargs: Any) -> None:
      323 +         super().__init__(*args, **kwargs)
      324 +         self._pool._network_backend = _SSRFProtectedAsyncBackend(
      325 +             self._pool._network_backend
      326 +         )
      327 +
      328 +
      329 + class SSRFProtectedHTTPTransport(httpx.HTTPTransport):
      330 +     """Synchronous counterpart of `SSRFProtectedAsyncHTTPTransport`."""
      331 +
      332 +     def __init__(self, *args: Any, **kwargs: Any) -> None:
      333 +         super().__init__(*args, **kwargs)
      334 +         self._pool._network_backend = _SSRFProtectedSyncBackend(
      335 +             self._pool._network_backend
  114 336           )
  115 337
  116 338
```

tests/utilities/test_urls.py

```
      ...    @@ -1,8 +1,13 @@
    1   1    import concurrent.futures
        2  + import socket
        3  + import threading
    2   4    import uuid
    3   5    from datetime import timedelta
    4      - from typing import Any, Literal
        6  + from typing import Any, Literal, cast
        7  + from unittest.mock import patch
```

```
5   8
        9   + import httpcore
        10  + import httpx
6   11      import pytest
7   12
8   13      from prefect.blocks.webhook import Webhook
            @@ -14,7 +19,14 @@
14  19      from prefect.settings import PREFECT_API_URL, PREFECT_UI_URL,
            temporary_settings
15  20      from prefect.states import StateType
16  21      from prefect.types._datetime import now
17      -   from prefect.utilities.urls import url_for, validate_restricted_url
        22  + from prefect.utilities.urls import (
        23  +     SSRFProtectedAsyncHTTPTransport,
        24  +     SSRFProtectedHTTPTransport,
        25  +     _SSRFProtectedAsyncBackend,
        26  +     _SSRFProtectedSyncBackend,
        27  +     url_for,
        28  +     validate_restricted_url,
        29  + )
18  30      from prefect.variables import Variable
19  31
20  32      MOCK_PREFECT_UI_URL = "https://ui.prefect.io"
            @@ -459,3 +471,364 @@ def
            test_url_for_with_additional_format_kwargs_raises_if_placeholder_not_replace
459 471             match="Unable to generate URL for worker because the following keys are
            missing: work_pool_name",
460 472         ):
461 473             url_for(obj="worker", obj_id="123e4567-e89b-12d3-a456-42661417400")
        474 +
        475 +
        476 + def _fake_getaddrinfo(ips: list[str]):
        477 +     """Return a `getaddrinfo` stub that yields `ips` as resolved addresses."""
        478 +
        479 +     def _impl(host: str, *_args: Any, **_kwargs: Any):
        480 +         results = []
        481 +         for ip in ips:
        482 +             try:
        483 +                 family = socket.AF_INET6 if ":" in ip else socket.AF_INET
        484 +             except Exception:
```

```
485 +                 family = socket.AF_INET
486 +             results.append((family, socket.SOCK_STREAM, 0, "", (ip, 0)))
487 +         return results
488 +
489 +     return _impl
490 +
491 +
492 + class TestValidateRestrictedUrlMultiRecord:
493 +     def test_rejects_when_any_resolved_ip_is_private(self):
494 +         """A hostname that resolves to both a public and a private address
495 +         must be rejected — `gethostbyname` returned only the first record and
496 +         could be bypassed by rotating records."""
497 +         with patch(
498 +             "prefect.utilities.urls.socket.getaddrinfo",
499 +             side_effect=_fake_getaddrinfo(["8.8.8.8", "10.0.0.1"]),
500 +         ):
501 +             with pytest.raises(ValueError, match="private address 10.0.0.1"):
502 +                 validate_restricted_url("https://example.com")
503 +
504 +     def test_accepts_when_all_resolved_ips_are_public(self):
505 +         with patch(
506 +             "prefect.utilities.urls.socket.getaddrinfo",
507 +             side_effect=_fake_getaddrinfo(["8.8.8.8", "1.1.1.1"]),
508 +         ):
509 +             validate_restricted_url("https://example.com")
510 +
511 +     def test_rejects_ipv6_mapped_private(self):
512 +         """A hostname that only resolves to an IPv6 address in a private
513 +         range is rejected."""
514 +         with patch(
515 +             "prefect.utilities.urls.socket.getaddrinfo",
516 +             side_effect=_fake_getaddrinfo(["fc00::1"]),
517 +         ):
518 +             with pytest.raises(ValueError, match="private address"):
519 +                 validate_restricted_url("https://example.com")
520 +
521 +
522 + class TestSSRFProtectedTransportTOCTOU:
523 +     """The protected transport must reject private IPs at connection time,
524 +     even when an initial validation via `validate_restricted_url` succeeded.
```

```
525 +    This simulates the DNS rebinding attack described in BOUNTY-422."""
526 +
527 +    async def test_async_transport_rejects_rebinding_to_private_ip(self):
528 +        transport = SSRFProtectedAsyncHTTPTransport()
529 +        # At connection time, DNS resolves to a private IP.
530 +        with patch(
531 +            "prefect.utilities.urls.socket.getaddrinfo",
532 +            side_effect=_fake_getaddrinfo(["127.0.0.1"]),
533 +        ):
534 +            async with httpx.AsyncClient(transport=transport) as client:
535 +                with pytest.raises(
536 +                    httpx.ConnectError, match="private address 127.0.0.1"
537 +                ):
538 +                    await client.get("http://example.com")
539 +
540 +    def test_sync_transport_rejects_rebinding_to_private_ip(self):
541 +        transport = SSRFProtectedHTTPTransport()
542 +        with patch(
543 +            "prefect.utilities.urls.socket.getaddrinfo",
544 +            side_effect=_fake_getaddrinfo(["192.168.1.1"]),
545 +        ):
546 +            with httpx.Client(transport=transport) as client:
547 +                with pytest.raises(
548 +                    httpx.ConnectError, match="private address 192.168.1.1"
549 +                ):
550 +                    client.get("http://example.com")
551 +
552 +    async def test_async_transport_rejects_multi_record_with_private(self):
553 +        """Even if one resolved A record is public, a sibling private record
554 +        must fail the validation (defends against DNS responses that mix
555 +        public and private addresses)."""
556 +        transport = SSRFProtectedAsyncHTTPTransport()
557 +        with patch(
558 +            "prefect.utilities.urls.socket.getaddrinfo",
559 +            side_effect=_fake_getaddrinfo(["8.8.8.8", "10.0.0.1"]),
560 +        ):
561 +            async with httpx.AsyncClient(transport=transport) as client:
562 +                with pytest.raises(
563 +                    httpx.ConnectError, match="private address 10.0.0.1"
564 +                ):
```

```
565 +                     await client.get("http://example.com")
566 +
567 +     async def test_async_transport_rejects_unresolvable_host(self):
568 +         transport = SSRFProtectedAsyncHTTPTransport()
569 +         with patch(
570 +             "prefect.utilities.urls.socket.getaddrinfo",
571 +             side_effect=socket.gaierror("resolution failed"),
572 +         ):
573 +             async with httpx.AsyncClient(transport=transport) as client:
574 +                 with pytest.raises(httpx.ConnectError, match="could not be
    resolved"):
575 +                     await client.get("http://example.com")
576 +
577 +     async def test_async_transport_rejects_private_ip_literal(self):
578 +         transport = SSRFProtectedAsyncHTTPTransport()
579 +         async with httpx.AsyncClient(transport=transport) as client:
580 +             with pytest.raises(httpx.ConnectError, match="private address"):
581 +                 await client.get("http://127.0.0.1")
582 +
583 +     def test_transport_wraps_existing_network_backend(self):
584 +         """Ensure the transport replaces the pool's network backend with the
585 +         SSRF-protected wrapper while keeping the underlying backend
586 +         available for actual connections."""
587 +         atransport = SSRFProtectedAsyncHTTPTransport()
588 +         assert isinstance(atransport._pool._network_backend,
    _SSRFProtectedAsyncBackend)
589 +
590 +         stransport = SSRFProtectedHTTPTransport()
591 +         assert isinstance(stransport._pool._network_backend,
    _SSRFProtectedSyncBackend)
592 +
593 +
594 + class TestSSRFProtectedBackendPinsIP:
595 +     """The protected backends must connect to the resolved IP (a literal),
596 +     not the original hostname, so that the underlying backend cannot
597 +     re-resolve DNS and pick up a different (private) address."""
598 +
599 +     async def test_async_backend_connects_to_validated_ip(self):
600 +         class RecordingBackend(httpcore.AsyncNetworkBackend):
601 +             def __init__(self):
```

```
602 +                 self.connected_host: str | None = None
603 +
604 +             async def connect_tcp(
605 +                 self, host, port, timeout=None, local_address=None,
    socket_options=None
606 +             ):
607 +                 self.connected_host = host
608 +                 # Return a mock stream — we don't actually connect.
609 +                 raise httpcore.ConnectError("stop here")
610 +
611 +             async def connect_unix_socket(self, *args, **kwargs):
612 +                 raise NotImplementedError
613 +
614 +             async def sleep(self, seconds):
615 +                 pass
616 +
617 +         inner = RecordingBackend()
618 +         backend = _SSRFProtectedAsyncBackend(inner)
619 +
620 +         with patch(
621 +             "prefect.utilities.urls.socket.getaddrinfo",
622 +             side_effect=_fake_getaddrinfo(["8.8.8.8"]),
623 +         ):
624 +             with pytest.raises(httpcore.ConnectError, match="stop here"):
625 +                 await backend.connect_tcp("example.com", 443)
626 +
627 +         assert inner.connected_host == "8.8.8.8"
628 +
629 +
630 + class TestSSRFProtectedBackendFallbackAcrossIPs:
631 +     """When `getaddrinfo` returns several safe IPs (e.g. a dual-stack AAAA+A
632 +     record set), the protected backends must try each validated IP in order
633 +     and only raise once every one has failed.  Pinning a single IP would
634 +     break dual-stack hostnames in single-stack environments."""
635 +
636 +     async def test_async_backend_falls_back_to_next_validated_ip(self):
637 +         class FlakyBackend(httpcore.AsyncNetworkBackend):
638 +             def __init__(self):
639 +                 self.connected_hosts: list[str] = []
640 +
```

```
641 +             async def connect_tcp(
642 +                 self, host, port, timeout=None, local_address=None,
    socket_options=None
643 +             ):
644 +                 self.connected_hosts.append(host)
645 +                 # First IP is unreachable, second succeeds.
646 +                 if len(self.connected_hosts) == 1:
647 +                     raise httpcore.ConnectError("network unreachable")
648 +                 return cast(httpcore.AsyncNetworkStream, object())
649 +
650 +             async def connect_unix_socket(self, *args, **kwargs):
651 +                 raise NotImplementedError
652 +
653 +             async def sleep(self, seconds):
654 +                 pass
655 +
656 +         inner = FlakyBackend()
657 +         backend = _SSRFProtectedAsyncBackend(inner)
658 +
659 +         with patch(
660 +             "prefect.utilities.urls.socket.getaddrinfo",
661 +             side_effect=_fake_getaddrinfo(["2606:4700:4700::1111",
    "93.184.216.34"]),
662 +         ):
663 +             await backend.connect_tcp("example.com", 443)
664 +
665 +         assert inner.connected_hosts == ["2606:4700:4700::1111",
    "93.184.216.34"]
666 +
667 +     async def test_async_backend_raises_last_error_when_all_fail(self):
668 +         class AlwaysFailingBackend(httpcore.AsyncNetworkBackend):
669 +             def __init__(self):
670 +                 self.attempts = 0
671 +
672 +             async def connect_tcp(
673 +                 self, host, port, timeout=None, local_address=None,
    socket_options=None
674 +             ):
675 +                 self.attempts += 1
676 +                 raise httpcore.ConnectError(f"attempt {self.attempts} failed")
```

```
677 +
678 +             async def connect_unix_socket(self, *args, **kwargs):
679 +                 raise NotImplementedError
680 +
681 +             async def sleep(self, seconds):
682 +                 pass
683 +
684 +         inner = AlwaysFailingBackend()
685 +         backend = _SSRFProtectedAsyncBackend(inner)
686 +
687 +         with patch(
688 +             "prefect.utilities.urls.socket.getaddrinfo",
689 +             side_effect=_fake_getaddrinfo(["2606:4700:4700::1111",
    "93.184.216.34"]),
690 +         ):
691 +             with pytest.raises(httpcore.ConnectError, match="attempt 2
    failed"):
692 +                 await backend.connect_tcp("example.com", 443)
693 +
694 +         assert inner.attempts == 2
695 +
696 +     def test_sync_backend_falls_back_to_next_validated_ip(self):
697 +         class FlakyBackend(httpcore.NetworkBackend):
698 +             def __init__(self):
699 +                 self.connected_hosts: list[str] = []
700 +
701 +             def connect_tcp(
702 +                 self, host, port, timeout=None, local_address=None,
    socket_options=None
703 +             ):
704 +                 self.connected_hosts.append(host)
705 +                 if len(self.connected_hosts) == 1:
706 +                     raise httpcore.ConnectError("network unreachable")
707 +                 return cast(httpcore.NetworkStream, object())
708 +
709 +             def connect_unix_socket(self, *args, **kwargs):
710 +                 raise NotImplementedError
711 +
712 +             def sleep(self, seconds):
713 +                 pass
```

```
714 +
715 +         inner = FlakyBackend()
716 +         backend = _SSRFProtectedSyncBackend(inner)
717 +
718 +         with patch(
719 +             "prefect.utilities.urls.socket.getaddrinfo",
720 +             side_effect=_fake_getaddrinfo(["2606:4700:4700::1111",
    "93.184.216.34"]),
721 +         ):
722 +             backend.connect_tcp("example.com", 443)
723 +
724 +         assert inner.connected_hosts == ["2606:4700:4700::1111",
    "93.184.216.34"]
725 +
726 +
727 + class TestSSRFProtectedBackendTimeoutBudget:
728 +     """The total connect timeout across all per-IP retries must stay bounded
729 +     by the caller's `timeout`.  Without a shared budget, a hostname with N
730 +     resolved addresses could take up to N * timeout to fail."""
731 +
732 +     async def test_async_backend_shares_timeout_budget_across_retries(self):
733 +         class SlowFailingBackend(httpcore.AsyncNetworkBackend):
734 +             def __init__(self):
735 +                 self.timeouts: list[float | None] = []
736 +
737 +             async def connect_tcp(
738 +                 self, host, port, timeout=None, local_address=None,
    socket_options=None
739 +             ):
740 +                 self.timeouts.append(timeout)
741 +                 raise httpcore.ConnectError("nope")
742 +
743 +             async def connect_unix_socket(self, *args, **kwargs):
744 +                 raise NotImplementedError
745 +
746 +             async def sleep(self, seconds):
747 +                 pass
748 +
749 +         inner = SlowFailingBackend()
750 +         backend = _SSRFProtectedAsyncBackend(inner)
```

```
751 +
752 +         with patch(
753 +             "prefect.utilities.urls.socket.getaddrinfo",
754 +             side_effect=_fake_getaddrinfo(["2606:4700:4700::1111",
    "93.184.216.34"]),
755 +         ):
756 +             with pytest.raises(httpcore.ConnectError):
757 +                 await backend.connect_tcp("example.com", 443, timeout=5.0)
758 +
759 +         # Each attempt receives a timeout no greater than the caller's budget;
760 +         # retries inherit the *remaining* budget, not a fresh one.
761 +         assert len(inner.timeouts) == 2
762 +         for t in inner.timeouts:
763 +             assert t is not None
764 +             assert t <= 5.0
765 +         assert inner.timeouts[1] <= inner.timeouts[0]
766 +
767 +     def test_sync_backend_shares_timeout_budget_across_retries(self):
768 +         class SlowFailingBackend(httpcore.NetworkBackend):
769 +             def __init__(self):
770 +                 self.timeouts: list[float | None] = []
771 +
772 +             def connect_tcp(
773 +                 self, host, port, timeout=None, local_address=None,
    socket_options=None
774 +             ):
775 +                 self.timeouts.append(timeout)
776 +                 raise httpcore.ConnectError("nope")
777 +
778 +             def connect_unix_socket(self, *args, **kwargs):
779 +                 raise NotImplementedError
780 +
781 +             def sleep(self, seconds):
782 +                 pass
783 +
784 +         inner = SlowFailingBackend()
785 +         backend = _SSRFProtectedSyncBackend(inner)
786 +
787 +         with patch(
788 +             "prefect.utilities.urls.socket.getaddrinfo",
```

```
+            side_effect=_fake_getaddrinfo(["2606:4700:4700::1111", "93.184.216.34"]),
+        ):
+            with pytest.raises(httpcore.ConnectError):
+                backend.connect_tcp("example.com", 443, timeout=5.0)
+
+        assert len(inner.timeouts) == 2
+        for t in inner.timeouts:
+            assert t is not None
+            assert t <= 5.0
+        assert inner.timeouts[1] <= inner.timeouts[0]
+
+
+class TestSSRFProtectedAsyncBackendNonBlockingDNS:
+    """`_SSRFProtectedAsyncBackend.connect_tcp` must not block the event loop
+    during DNS resolution; it runs `getaddrinfo` in a worker thread."""
+
+    async def test_async_backend_runs_getaddrinfo_off_the_event_loop(self):
+        loop_thread = threading.get_ident()
+        observed_threads: list[int] = []
+
+        def fake_getaddrinfo(*args, **kwargs):
+            observed_threads.append(threading.get_ident())
+            return [(socket.AF_INET, socket.SOCK_STREAM, 0, "", ("93.184.216.34", 0))]
+
+        class StopBackend(httpcore.AsyncNetworkBackend):
+            async def connect_tcp(self, *args, **kwargs):
+                raise httpcore.ConnectError("stop")
+
+            async def connect_unix_socket(self, *args, **kwargs):
+                raise NotImplementedError
+
+            async def sleep(self, seconds):
+                pass
+
+        backend = _SSRFProtectedAsyncBackend(StopBackend())
+
+        with patch(
```

```
826 +             "prefect.utilities.urls.socket.getaddrinfo",
    side_effect=fake_getaddrinfo
827 +         ):
828 +             with pytest.raises(httpcore.ConnectError, match="stop"):
829 +                 await backend.connect_tcp("example.com", 443)
830 +
831 +         assert observed_threads, "getaddrinfo should have been invoked"
832 +         assert observed_threads[0] != loop_thread, (
833 +             "getaddrinfo must run off the event loop thread"
834 +         )
```

Comments 0

Please sign in to comment.