Package 'ssrfr'

November 23, 2026

Type Package

Title Server-Side Request Forgery Guard for Outbound Requests

Version 1.0.0

Description Protects R applications that fetch URLs an attacker can influence from server-side request forgery. Each redirect hop is parsed once, resolved once, and every resolved address is classified before the connection is pinned to a validated address, so a hostname cannot be rebound between check and connect. Refusals carry stable reason codes. Provides the guarded fetch of each hop of a redirect chain, a helper that follows a whole chain, structural inspection of a URL without network access, and resolved inspection of every address a name resolves to.

License MIT + file LICENSE

URL https://bart-turczynski.gitlab.io/ssrfr/, https://gitlab.com/bart-turczynski/ssrfr, https://bart-turczynski.r-universe.dev/ssrfr

BugReports https://gitlab.com/bart-turczynski/ssrfr/-/work_items

X-schema.org-keywords ssrf, server-side-request-forgery, url-validation, ip-address, dns-rebinding, cloud-metadata, redirects, libcurl, security

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports curl (>= 7.0.0), pslr (>= 1.2.2), raddr (>= 0.1.2), rurl (>= 3.1.0)

Suggests callr, httpuv, knitr, oysteR, pkgload, processx, rmarkdown, rosv, testthat (>= 3.0.0), webfakes (>= 1.4.0), withr

SystemRequirements libcurl (>= 7.73.0)

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Bart Turczynski [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0002-8788-7980>)

Maintainer Bart Turczynski <bartek@turczynski.pl>

Repository CRAN

Date/Publication 2026-11-23 15:00:02 UTC

Contents

ssrf_fetch 2
ssrf_fetch_chain 4
ssrf_inspect_url 6
ssrf_policy 8
ssrf_prepare_hop 10
ssrf_public_reason 13
ssrf_vocabulary 15

Index **16**

`ssrf_fetch` *Fetch through a guarded binding*

Description

Makes the request a binding from `ssrf_prepare_hop()` describes, connecting only to an address that binding validated. The name is never resolved again: the connection is pinned to the validated address while TLS, the `Host` header and certificate verification stay bound to the hostname. Certificate verification is never weakened, no proxy is used whatever the environment says, redirects are not followed, and the request goes over HTTP/1.1 on a new connection that is closed when the fetch returns.

Usage

```
ssrf_fetch(binding)
```

Arguments

`binding`	A binding from `ssrf_prepare_hop()` that has not been fetched.

Details

A binding is single-use. `ssrf_fetch()` spends it on entry, so a second call with the same binding is an error of class `ssrfr_error_spent_binding`, even when the first was interrupted. The spent binding stays readable: its `state` records the response status, the `Location` field, the address the connection was pinned to and each address tried. A recorded status alone does not mean the fetch succeeded: a size-limit failure and a `"redirect-limit"` refusal record one too.

When a name resolved to several addresses, they are tried in the order the resolver returned them, moving on only when a connection could not be established; a request is never sent twice. After connecting, `ssrfr` confirms from libcurl's trace that the connection went to the pinned address; if it cannot, the fetch fails as `"pin-mismatch"` without trying another address.

The whole response is read before `ssrf_fetch()` returns, within the policy's limits: `max_response_size` counts the body's bytes after any `gzip` or `deflate` decoding, as they arrive, so a compressed body cannot exceed it; `max_header_bytes` and `max_header_fields` bound the header, including any interim `1xx` responses and the trailer fields of a chunked body, as it arrives; `connect_timeout` bounds each connection attempt and `total_timeout` the time spent in `ssrfr` for the chain, decoding included.

A redirect is returned, not followed: its status and `Location` are in the response, and the next hop is prepared with `ssrf_prepare_hop(location, policy, from = binding)`. A caller that wants the whole chain followed calls `ssrf_fetch_chain()` from the first hop, in place of `ssrf_prepare_hop()` and `ssrf_fetch()`; it cannot take over a redirect already in hand. Once the chain has followed the policy's `max_redirects` redirects, a `3xx` response is refused as `"redirect-limit"` instead, with or without `Location`; under `max_redirects = 0` every `3xx` is. The refusal is decided at the status line and the transfer stopped there, so nothing after it changes it: not a second `Location`, nor a body over the limits or one that stalls. `ssrfr` guards only requests made through a binding. R's own `download.file()`, `url()`, `readLines()` on a URL, direct `curl` calls, and the packages that read URLs through them, such as `jsonlite::fromJSON(url)`, `data.table::fread(url)` and `xml2::read_xml(url)`, stay unguarded, as does the `curl` command-line tool.

Value

A response, class `ssrfr_response`; an operational failure, class `ssrfr_failure`; or, for a `3xx` response past the chain's redirect budget, a refusal, class `ssrfr_refusal`, with code `"redirect-limit"`.

A response is a plain list that owns no handle, connection or file: `status` (the HTTP status code), `headers` (the final response's header fields, a character vector named by lowercase field name, without the trailer fields of a chunked body; a value that is not valid UTF-8 is kept byte for byte and marked `"bytes"`) and `body` (the decoded body, a raw vector; `rawToChar(response$body)` reads text). Its `print()` and `format()` show only the status, the media type and the body size, never the body or another header value.

A failure carries `cause`, from `ssrf_vocabulary("causes")`: `"connect-failed"`, `"timeout"`, `"tls-failed"`, `"pin-mismatch"`, `"response-too-large"` or `"protocol-error"`. Its `detail` lists each address tried and how the attempt ended, and names the limit that was reached. A `"response-too-large"` failure whose `detail$check` is `"decoded-bytes"` (the body exceeded `max_response_size`) also carries `detail$status`, the final response's status, which `binding$state$status` records too, and `detail$observed_bytes`, the decoded body bytes reached, the delivery that exceeded the limit included. So a `404` with a large body stays distinguishable from a large `200`. The status is left out when libcurl's and the status line's disagree, and a header limit (`"header"`) records none. Such a binding records no response and no `Location`, so it is never a `from`.

Testing against a local server

A loopback server is a refused destination, so a test that fetches one through the guard reopens loopback in its own policy, with `ssrf_policy(allow_ranges = "127.0.0.0/8", allow_ports = <port>)`. `ssrfr` has no test mode and no switch that turns the guard off; keep such a policy out of production code.

See Also

`ssrf_prepare_hop()`, which makes the binding; `ssrf_public_reason()` before a failure reaches an untrusted party.

Examples

```
binding <- ssrf_prepare_hop(
  "https://93.184.216.34/",
  ssrf_policy(),
  request = list()
)
## Not run:
response <- ssrf_fetch(binding)
response
response$status
rawToChar(response$body)

# A binding is spent on first use.
try(ssrf_fetch(binding))
binding$state$status

## End(Not run)

# In a test, reach a local server by reopening loopback in the policy:
test_policy <- ssrf_policy(allow_ranges = "127.0.0.0/8", allow_ports = 8080)
ssrf_prepare_hop("http://127.0.0.1:8080/", test_policy, request = list())
```

---

ssrf_fetch_chain *Follow a redirect chain through the guard*

Description

Fetches a URL through the guard and follows its redirects, hop by hop, for a caller without a redirect loop of its own. It is the loop documented under `ssrf_prepare_hop()`, and nothing more: each hop is prepared with `ssrf_prepare_hop()` and fetched with `ssrf_fetch()`, so every hop is checked from the start, resolved once and pinned, the request plan is inherited and stripped across origins, and the chain's budgets are its first hop's.

Usage

```
ssrf_fetch_chain(url, policy, request)
```

Arguments

`url`	The URL of the chain's first hop, a single absolute URL string.
`policy`	The policy to decide every hop under, from `ssrf_policy()`. Its `max_redirects` and `total_timeout` bound the whole chain.
`request`	The request plan for the first hop, a list as described under `ssrf_prepare_hop()`; `list()` is a plain `GET`. It has no default, as a first hop's plan has none. Later hops inherit it, transformed for each redirect.

Details

Only a followed redirect is followed: a `301`, `302`, `303`, `307` or `308` response with exactly one `Location` field. The next hop is prepared with `from` set to the previous binding and `url` set to the `Location` value that binding recorded, byte for byte. Any other response ends the chain and is returned as it is, a `304` or a `302` without `Location` included. Once the chain has followed the policy's `max_redirects` redirects, `ssrf_fetch()` refuses the next `3xx` as `"redirect-limit"`, which ends the chain too; the helper keeps no count of its own.

It returns only the last outcome. The bindings of the hops it followed are not kept, so a caller that must log every hop, as OWASP's "log all accepted and blocked network flows" asks, runs the per-hop loop shown in the examples of `ssrf_prepare_hop()` and logs each binding and outcome itself.

Value

What the chain's last call returned, told apart by class: the final response, class `ssrfr_response`, as `ssrf_fetch()` describes it; the refusal that ended the chain, class `ssrfr_refusal`, such as `"redirect-limit"`, `"downgrade"` or an address refusal on any hop; or the operational failure that ended it, class `ssrfr_failure`. A refusal and a failure name the hop, and the host and address where they are known, for the operator; project them with `ssrf_public_reason()` before an untrusted party sees them. A misuse, such as a request plan that breaks a header rule, is the error `ssrf_prepare_hop()` raises.

See Also

`ssrf_prepare_hop()` and `ssrf_fetch()`, the per-hop primitives it is built on, for a caller with its own loop or one that logs every hop.

Examples

```
policy <- ssrf_policy()

# Refused before any network I/O: the chain ends at its first hop.
refused <- ssrf_fetch_chain(
  "http://169.254.169.254/latest/",
  policy,
  request = list()
)
refused$code
ssrf_public_reason(refused)
```

```
## Not run:
result <- ssrf_fetch_chain(
  "https://example.com/start",
  ssrf_policy(max_redirects = 5),
  request = list(headers = c(Authorization = "Bearer secret"))
)
if (inherits(result, "ssrfr_response")) {
  result$status
  rawToChar(result$body)
} else {
  # A refusal's code or a failure's cause, for the operator only.
  if (inherits(result, "ssrfr_refusal")) result$code else result$cause
}

## End(Not run)
```

`ssrf_inspect_url` *Inspect a URL without fetching it*

Description

Reports what a URL is under a policy, without fetching it: whether it parses without ambiguity, what scheme, host and port the transport would act on, how an address host is classified and, at layer `"L1"`, every address a name resolves to and how each is classified. Inspection is a pre-filter and a way to lint a configuration, **not a defense**: what a name resolves to can change the moment this returns, so a URL it finds nothing wrong with may still be refused when fetched. Only the guarded fetch protects a request.

Usage

```
ssrf_inspect_url(url, policy = ssrf_policy(), base = NULL, layer = "L0")
```

Arguments

`url`	The URL, a single string.
`policy`	The policy to inspect it under, from `ssrf_policy()`.
`base`	The URL of the previous hop, a single string, when `url` came from a redirect: `url` may then be a relative reference, and an `https` `base` makes an `http` `url` a `"downgrade"`. `NULL`, the default, for a first hop, which must be an absolute URL.
`layer`	`"L0"` (the default) for structural inspection with no network I/O, or `"L1"` to also resolve a name and classify every address it resolves to.

Details

The URL is parsed as the guarded fetch parses it: `rurl` under the WHATWG URL Standard, then libcurl's own parser on the string libcurl would be handed. A URL the two parsers read differently is reported as `"parse"`. So is a host with an `xn--` label that is not a genuine A-label, such as `xn--a.example`, which `rurl` reports with its `domain-invalid-ace-label` diagnostic. So is a host with an empty label, such as `evil.example..`, which `rurl` reports with its `domain-empty-label` diagnostic, on the URL or on its serialization; a single trailing root dot is no empty label. Checks run in the order of a guarded hop, and the first one that applies names the reason code: the length limit and the parse, the scheme, embedded credentials, the port, an ambiguous numeric host spelling, the hostname rules, and the address rules. A relative reference, such as a redirect's `Location`, is resolved against `base` first.

Layer `"L0"`, the default, is structural and does no network I/O: it resolves no name, so for a name it reports no address. Layer `"L1"` also asks the system resolver, once, for every address of a name that passed the earlier checks, and classifies each one with its embedded addresses. The name is looked up as absolute, with a trailing root dot, so no DNS search domain is appended: `http://intranet/` is never looked up as `intranet.corp.example`. One refused address refuses the whole set. A resolver error, an empty answer or an answer that is not an address is the operational cause `"unresolvable"`. An address host, or a URL an earlier check refuses, is never resolved.

Value

An object of class `ssrfr_inspection`, a list of facts:

`code` The reason code (see `ssrf_vocabulary("reason_codes")`) that the first applicable check reports, or `NA`. It is derived from the facts below, not a verdict: `NA` is not an approval. At L0 a name is still to be resolved; at L1 the addresses may differ by the time a fetch resolves the name again.

`cause` The operational cause (see `ssrf_vocabulary("causes")`), `"unresolvable"`, when resolution failed at L1, or `NA`. A URL carries a `code` or a `cause`, never both.

`step` The step of a guarded hop that reported `code` or `cause`, from 1 (parse) to 7 (resolution) and 8 (address classification), or `NA`.

`detail` Operator detail for `code` or `cause`: the gate and precedence tier, the address category, embedding kind or provider-endpoint kind behind a code, or which resolution failure.

`layer` `"L0"` or `"L1"`, the layer inspected.

`url` The URL as given, for display, without userinfo.

`scheme`, `host`, `port` The scheme, host and effective port libcurl reads, or `NA` when the URL did not parse that far.

`userinfo` Whether the URL carries credentials.

`host_kind` `"name"`, `"ipv4"` or `"ipv6"`, or `NA`.

`address` For an address literal, `raddr`'s facts about it: its canonical text, reachability, category, embedded addresses and provider-endpoint row. `NULL` otherwise.

`answers` At L1, for a name that was resolved, the addresses the resolver returned, in its order; `character(0)` for an empty answer. `NULL` when nothing was resolved or the resolver failed.

`addresses` At L1, when every answer is an address, one entry per answer in the resolver's order: a list of `address` (the answer), `code` (the reason code gates report for it, or `NA`), `detail` and `facts` (`raddr`'s facts, as in `address`). `NULL` otherwise.

See Also

`ssrf_policy()` for the rules; `ssrf_vocabulary()` for the reason codes, the causes, the provider endpoints and the metadata hostnames.

Examples

```
ssrf_inspect_url("http://127.0.0.1/admin")$code
ssrf_inspect_url("http://0177.0.0.1/")$code
ssrf_inspect_url("http://[::ffff:169.254.169.254]/")$code

# A name is not resolved at L0: nothing applies at this layer.
ssrf_inspect_url("https://example.com/")

# Lint a policy: does it reopen what it should, and nothing more?
policy <- ssrf_policy(allow_ranges = "10.0.0.0/8", deny_hosts = ".corp")
ssrf_inspect_url("http://10.1.2.3/", policy)$code
ssrf_inspect_url("http://api.corp/", policy)$code

# A redirect's Location, resolved against the previous hop.
ssrf_inspect_url("//169.254.169.254/", base = "http://example.com/a")$code

# At L1 a URL refused before resolution is never looked up.
ssrf_inspect_url("ftp://files.example/", layer = "L1")$code
ssrf_inspect_url("http://example.com:6379/", layer = "L1")$code
```

---

ssrf_policy *Build a guard policy*

Description

Builds the policy a guarded hop is evaluated under. Every field has a finite, conservative default, so `ssrf_policy()` alone is a complete policy. The policy is validated when it is built: an entry that could silently match nothing, or a limit that is not a usable number, is an error of class `ssrfr_error_invalid_policy`, never a policy that quietly admits or refuses more than it says.

Usage

```
ssrf_policy(
  allow_schemes = c("http", "https"),
  allow_ports = c(80, 443),
  deny_hosts = character(),
  allow_hosts = character(),
  deny_ranges = character(),
```

```
  allow_ranges = character(),
  allow_userinfo = FALSE,
  max_redirects = 20,
  connect_timeout = 3,
  total_timeout = 30,
  max_response_size = 10 * 1024^2,
  max_header_bytes = 16 * 1024,
  max_header_fields = 128,
  max_url_length = 8000,
  user_agent = default_user_agent()
)
```

Arguments

`allow_schemes`	Schemes a URL may use, a subset of `"http"` and `"https"`.
`allow_ports`	Ports a URL may use, whatever its scheme: an allowlist, never a denylist.
`deny_hosts`	Hostname rules that refuse a host.
`allow_hosts`	Hostname rules that override the built-in metadata hostname list for the matched name only.
`deny_ranges`	Address ranges that refuse an address.
`allow_ranges`	Address ranges that override the determinate built-in address refusals for the matched addresses only.
`allow_userinfo`	Whether a URL may carry embedded credentials (`user:password@`). `TRUE` or `FALSE`.
`max_redirects`	Redirects a chain may follow. `0` refuses any redirect.
`connect_timeout`	Seconds allowed for each connection attempt.
`total_timeout`	Seconds allowed for the whole redirect chain.
`max_response_size`	Bytes of decoded response body allowed per hop.
`max_header_bytes`	Bytes of response header allowed per hop.
`max_header_fields`	Response header fields allowed per hop.
`max_url_length`	Octets of URL string allowed per hop.
`user_agent`	The `User-Agent` header value. The default names the package version and repository. It must be a valid HTTP field value; CR, LF and other control characters are refused, never stripped.

Details

Allow fields are exceptions to the built-in refusals, not a destination allowlist: `allow_ranges` reopens addresses the built-in rules refuse, and `allow_hosts` reopens names the built-in metadata hostname list refuses. A caller deny rule always wins over a caller allow rule.

Hostname rules match the host's normalized form: IDNA A-label, ASCII-lowercased, one trailing root dot removed. A rule matches that name exactly, unless it begins with `.`, in which case it matches every proper subdomain and not the bare name: `".corp"` matches `api.corp`, not `corp`. No other wildcard exists. An address literal is not a hostname rule; write it as a range. A rule with an empty label (`"a..b"`), or an `xn--` label that is not a genuine A-label, is an error. The policy stores the normalized rules.

Ranges are CIDR blocks read by `raddr`, such as `"10.0.0.0/8"` or `"2001:db8::/32"`. A single address is a `/32` or `/128`, and a block with host bits set is an error.

Every limit is finite and may be raised without a ceiling. `0` means zero, never unlimited, and only `max_redirects` accepts it, meaning "refuse any redirect".

Value

An object of class `ssrfr_policy`: a list holding every field above, with hostname rules normalized and ports and limits as numbers.

Examples

```
ssrf_policy()

# Reach an internal service by name: authorize its address range.
ssrf_policy(allow_ranges = "10.0.0.0/8", deny_hosts = ".corp")

# A malformed entry is an error, not a rule that matches nothing.
try(ssrf_policy(deny_hosts = "com, ru"))
```

`ssrf_prepare_hop` *Prepare one guarded hop*

Description

Decides whether a URL may be fetched under a policy and, when it may, returns the exact connection it may use. This is the first half of the guarded fetch, the only part of `ssrfr` that is a defense against server-side request forgery; `ssrf_fetch()` is the second.

Usage

```
ssrf_prepare_hop(url, policy, request = NULL, from = NULL)
```

Arguments

`url`	The URL, a single string: an absolute URL on a first hop; on a redirect hop, the `Location` value `from` recorded, `from$state$location`, byte for byte, which may be a relative reference. Any other value on a redirect hop, even the same URL spelled another way, is an error of class `ssrfr_error_invalid_from`: a redirect goes where the server pointed, and a different URL starts a new chain.

policy	The policy to decide under, from ssrf_policy().
request	The request plan for a first hop, a list as described above; list() for a plain GET. Exactly one of request and from is required.
from	The binding of the previous hop, for a redirect hop: fetched, and holding a followed redirect. Its plan, origin, status and chain budgets decide the new hop's plan.

Details

The URL is parsed as the transport will parse it, checked for its scheme, embedded credentials, port, host spelling and hostname rules, then, for a name, resolved exactly once, with a trailing root dot so no DNS search domain applies. Every address it resolves to is classified with the addresses embedded in it, and one refused address refuses the whole set. A URL that passes becomes a binding pinned to those validated addresses: ssrf_fetch() connects only to them, never resolving the name again, so a name cannot be rebound between the check and the connection.

The request plan is part of the decision. It is a list of up to four fields, each optional:

method The request method, "GET" by default. Any HTTP token is accepted; the application decides which methods an untrusted party may choose.

headers Header fields to send, as a named character vector or a named list of strings. Names must be HTTP tokens and values carry no CR or LF. Fields the transport owns (Host, Connection, Proxy-Connection, Keep-Alive, Transfer-Encoding, TE, Trailer, Upgrade, Content-Length, Accept-Encoding, Expect and User-Agent, which the policy sets) are refused, and so are the metadata-service request markers of ssrf_vocabulary("metadata_headers") unless the policy's allow_ranges names a provider endpoint exactly.

body The request body: a raw vector or a single string, sent as its UTF-8 bytes. NULL (the default) sends none.

carry Names of fields in headers the application asserts carry no credential, so a redirect to another origin may keep them. Every other field and the body are treated as secret. Authorization, Proxy-Authorization and Cookie can never be nominated.

list() is a plain GET. A plan that breaks one of these rules is an error of class ssrfr_error_invalid_request, raised before anything is parsed or resolved.

A redirect is followed hop by hop: the caller fetches a binding, and when its response is a followed redirect (a 301, 302, 303, 307 or 308 with exactly one Location field) prepares the next hop with from = binding and the Location value as url. A relative Location is resolved against the previous hop's URL, and the new URL is checked from the start, as a first hop is. The request plan is inherited, never restated, and changes with the redirect:

- a 301 or 302 turns POST into GET without its body, and keeps any other method;
- a 303 keeps HEAD and turns every other method into GET, without a body;
- a 307 or 308 keeps the method and the body.

A redirect to another origin (another scheme, host or port) drops the body and every header field but those carry nominated; Authorization, Proxy-Authorization and Cookie never cross. Whenever the body is dropped, so are the fields that describe it, such as Content-Type, even when nominated. The binding's redirect field records what was dropped. The plan the redirect hop sends is

checked under that hop's policy before the new name is resolved, so a field it keeps and that policy refuses, such as a metadata-service marker, is an error of class `ssrfr_error_invalid_request`, whose message names the field by its place in `from$request`; a field the redirect dropped is not. An `https` hop that redirects to `http` is refused as `"downgrade"`. `ssrf_fetch_chain()` runs this loop for a caller without one of its own, and returns only the chain's last outcome; a caller that must log every hop runs the loop in the examples below instead.

The chain's budgets are its first hop's: `max_redirects` and `total_timeout` travel through `from`, with the time the chain has used, and a redirect hop's policy that states other values is an error of class `ssrfr_error_budget_change`. Every other policy field applies to the hop it is passed to. Once the chain has followed `max_redirects` redirects, `ssrf_fetch()` refuses the next `3xx` response as `"redirect-limit"`, whether or not it carries `Location`; under `max_redirects = 0` that is the first. A `from` that is not yet fetched, whose fetch failed or whose response is not a followed redirect is an error of class `ssrfr_error_invalid_from`. A spent binding stays usable as `from`, so a hop that failed can be prepared again from the same binding.

The binding is single-use: `ssrf_fetch()` spends it on entry. It prints without userinfo, header values or the body, and holds the policy by value, so a later change to the policy object does not reach it.

Value

One of three values, told apart by class:

`ssrfr_binding` The hop may proceed: pass it to `ssrf_fetch()`. Its fields can be read with `$`: `hop`, `url` (the URL the fetch requests, without fragment), `origin` (`scheme`, `host`, `port`), `validated` (the addresses a connection may use, in resolver order), `pin` (the first of them), `request` (the sanitized plan), `policy`, `budget` (the chain's `max_redirects` and `total_timeout`, and the seconds it had used), `redirect`, `tls` and `state`, the facts `ssrf_fetch()` records. On a redirect hop, `redirect` records the previous hop and status, whether the hop crossed an origin, the method before and after, whether the body was dropped and which fields were (by lowercase name); it is `NULL` on a first hop.

`ssrfr_refusal` The policy refuses the hop; `code` is a reason code from `ssrf_vocabulary("reason_codes")`.

`ssrfr_failure` The hop failed on the wire before a connection: `cause` is `"unresolvable"` when resolution failed, or `"timeout"` when resolution used up the policy's `total_timeout`.

A refusal and a failure name the hop, and the host and address where they are known, for the operator; project them with `ssrf_public_reason()` before an untrusted party sees them.

See Also

`ssrf_fetch()` to fetch through a binding; `ssrf_fetch_chain()` to follow a whole redirect chain; `ssrf_policy()` for the rules; `ssrf_inspect_url()` to lint a URL or policy without fetching.

Examples

```
policy <- ssrf_policy()

# Refused before any network I/O.
ssrf_prepare_hop("http://127.0.0.1/admin", policy, request = list())
ssrf_prepare_hop("http://0177.0.0.1/", policy, request = list())$code
```

```
# An address host needs no resolution: this binding is pinned to it.
binding <- ssrf_prepare_hop(
  "https://93.184.216.34/data?page=2",
  policy,
  request = list(headers = c(Authorization = "Bearer secret"))
)
binding

# A request plan that breaks a header rule is an error.
try(ssrf_prepare_hop(
  "https://example.com/",
  policy,
  request = list(headers = c(Host = "internal.example"))
))

## Not run:
# Following a redirect chain: prepare each hop from the previous binding.
result <- ssrf_prepare_hop(
  "https://example.com/old",
  policy,
  request = list(headers = c(Authorization = "Bearer secret"))
)
while (inherits(result, "ssrfr_binding")) {
  binding <- result
  result <- ssrf_fetch(binding)
  followed <- inherits(result, "ssrfr_response") &&
    result$status %in% c(301, 302, 303, 307, 308) &&
    identical(binding$state$location_count, 1L)
  if (!followed) {
    break
  }
  # A relative Location is resolved against the previous hop's URL, and
  # Authorization is dropped if the redirect leaves the origin.
  result <- ssrf_prepare_hop(binding$state$location, policy, from = binding)
}
# `result` is the chain's final outcome. Log a code or cause for the
# operator; show an untrusted party only ssrf_public_reason(result).
outcome <- if (inherits(result, "ssrfr_response")) {
  result$body # the body's bytes, a raw vector, whatever they hold
} else if (inherits(result, "ssrfr_refusal")) {
  result$code # "redirect-limit" once max_redirects redirects are followed
} else {
  result$cause # an operational failure
}

## End(Not run)
```

---

ssrf_public_reason *Project a refusal or failure for an untrusted party*

Description

Reduces a refusal or an operational failure to one fixed value that names no predicate, cause, address, hop or outcome class. Every refusal and every failure projects to the same value, so a party probing a service cannot tell a refused private address from a closed port, a missing name or a timeout.

Usage

```
ssrf_public_reason(x)
```

Arguments

x	A refusal or an operational failure.

Details

`ssrfr` cannot tell a trusted caller from an untrusted one, so it returns the full record to its caller. Applying this projection where an untrusted party receives the result is the application's job. Response timing still differs between outcomes; this projection hides only their shape.

A refusal (class `ssrfr_refusal`) carries `code`, a reason code from `ssrf_vocabulary("reason_codes")`; an operational failure (class `ssrfr_failure`) carries `cause`, from `ssrf_vocabulary("causes")`. Both also carry `hop`, `host`, `address`, `url` (a display form with any userinfo redacted) and `detail`. Their `print()` and `format()` methods show no userinfo, request header value, body or proxy value.

Value

The single string `"refused"`, whatever `x` holds.

Examples

```
# A loopback literal is refused at prepare, before any resolution.
refusal <- ssrf_prepare_hop(
  "http://127.0.0.1/admin",
  ssrf_policy(),
  request = list()
)
refusal$code
ssrf_public_reason(refusal)

# An operational failure, a timeout say, gives the same string.
```

`ssrf_vocabulary` *List a closed vocabulary*

Description

Enumerates one of the closed vocabularies `ssrfr` publishes: the reason codes a refusal carries, the causes an operational failure carries, the classes of the misuse conditions it raises, and the three tables of policy data it owns: the provider endpoints refused by address and the metadata hostnames refused by name, each row citing the vendor documentation that names it, and the metadata-service request headers a request plan may not carry. Each vocabulary is API, carries a version stamp, and changes only with a new version.

Usage

```
ssrf_vocabulary(domain = NULL)
```

Arguments

`domain`	The vocabulary to list: `"reason_codes"`, `"causes"`, `"condition_classes"`, `"provider_endpoints"`, `"metadata_hostnames"` or `"metadata_headers"`. `NULL`, the default, lists the vocabularies themselves.

Value

A data frame. For a named `domain`, one row per entry, keyed by the first column (`code`, `cause`, `class`, `address`, `hostname` or `header`), with the attributes `domain`, `version` (the vocabulary's version stamp) and `package_version`. For `NULL`, one row per vocabulary with its `domain`, `version` and `size`, and the `package_version` attribute.

Examples

```
ssrf_vocabulary()
ssrf_vocabulary("reason_codes")$code
attr(ssrf_vocabulary("causes"), "version")
ssrf_vocabulary("metadata_hostnames")$hostname
```

Index

ssrf_fetch, 2
ssrf_fetch(), *4, 5*, *10–12*
ssrf_fetch_chain, 4
ssrf_fetch_chain(), *3*, *12*
ssrf_inspect_url, 6
ssrf_inspect_url(), *12*
ssrf_policy, 8
ssrf_policy(), *5, 6*, *8*, *11, 12*
ssrf_prepare_hop, 10
ssrf_prepare_hop(), *2–5*
ssrf_public_reason, 13
ssrf_public_reason(), *4, 5*, *12*
ssrf_vocabulary, 15
ssrf_vocabulary(), *8*