WS-Federation: Passive Requestor Profile

Version 1.0
July 8, 2003

Authors

Siddharth Bajaj, VeriSign
Brendan Dixon, Microsoft
Mike Dusche, Microsoft
Maryann Hondo, IBM
Matt Hur, Microsoft
Chris Kaler (Editor), Microsoft
Hal Lockhart, BEA
Hiroshi Maruyama, IBM
Anthony Nadalin (Editor), IBM
Nataraj Nagaratnam, IBM
Andrew Nash, RSA Security
Hemma Prafullchandra, VeriSign
Yordan Rouskov, Microsoft
John Shewchuk, Microsoft
Jeff Spelman, Microsoft

Copyright Notice

(c) 2001-2003 IBM Corporation, Microsoft Corporation, BEA Systems, Inc., RSA Security, Inc., Verisign, Inc. All rights reserved.

BEA, IBM, Microsoft, RSA Security and VeriSign (collectively, the "Authors") hereby grant you permission to copy and display the WS-Federation: Passive Requestor Specification, in any medium without fee or royalty, provided that you include the following on ALL copies of the WS-Federation: Passive Requestor Specification, or portions thereof, that you make:

1. A link or URL to the Specification at this location

2. The copyright notice as shown in the WS-Federation: Passive Requestor Specification.

EXCEPT FOR THE COPYRIGHT LICENSE GRANTED ABOVE, THE AUTHORS DO NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY, INCLUDING PATENTS, THEY OWN OR CONTROL.

THE WS-Federation: Passive Requestor SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE WS-Federation: Passive Requestor SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE WS-Federation: Passive Requestor SPECIFICATION.

The WS-Federation: Passive Requestor Specification may change before final release and you are cautioned against relying on the content of this specification.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Specification or its contents without specific, written prior permission. Title to copyright in the WS-Federation: Passive Requestor Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Abstract

This profile specification describes how the cross trust realm identity, authentication and authorization federation mechanisms defined in WS-Federation can be utilized used by passive requestors such as Web browsers to provide Identity Services. Passive requesters of this profile are limited to the HTTP protocol.

Modular Architecture

By using the XML, SOAP and WSDL extensibility models, the WS* specifications are designed to be composed with each other to provide a rich Web services environment. WS-Federation: Passive Requestor by itself does not provide a complete security solution for Web services. WS-Federation: Passive Requestor is a building block that is used in conjunction with other Web service and application-specific protocols to accommodate a wide variety of security models.

Status

This WS-Federation Passive Requestor Profile Specification is an initial public draft release and is provided for review and evaluation only. BEA, IBM, Microsoft, RSA Security and VeriSign hope to solicit your contributions and suggestions in the near future. BEA, IBM, Microsoft, RSA Security and VeriSign make no warrantees or representations regarding the specifications in any manner whatsoever

Table of Contents

1. Introduction

The WS-Federation specification defines an integrated model for federating identity, authentication and authorization across different trust realms and protocols. This specification defines how the WS-Federation model is applied to passive requestors such as Web browsers that support the HTTP protocol.

For the passive mechanisms to work seamlessly and provide a single or reduced sign-on, there needs to be a service that will verify that the claimed requestor is really the requestor. Initial verification MUST occur in a secure fashion, for example, using SSL/TLS or HTTP/S.

Subsequent verifications of passive requestors MAY use custom mechanisms or cookies to optimize the flow. However, use of cookies may suffer from the certain security risks. It is strongly RECOMMENDED that if cookies are used, that the *discard* attribute as defined in RFC 2965 be used.

Aside from the discard issue, artifacts and cookies still suffer from replay attacks. Passive requesters SHOULD consider using stronger methods of authentication such as digest authentication (RFC 2617) and the HTTP Security Extensions. Such mechanisms MAY be used to authenticate to the Web server, if it supports such mechanisms.

1.1. Goals and Requirements

The primary goal of this specification is to define a profile for passive requesters to federate identity, authentication, and authorization information.

1.1.1 Requirements

The following list identifies the key driving requirements for this specification:

- Enable sharing of identity, authentication, and authorization data from and through passive requestors
- Enable the brokering of trust and security token exchange in a passive requestor environment
- Allow optional hiding of identity information and other attributes in a passive requestor environment

1.1.2. Non-Goals

The following topics are outside the scope of this document:

- Definition of message security or trust establishment/verification protocols
- Specification of new security token formats
- Modifying existing browsers to provide support for additional protocols and functionality

1.2. Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

When describing abstract data models, this specification uses the notational convention used by the XML Infoset. Specifically, abstract property names always appear in square brackets (e.g., [some property]).

When describing concrete XML schemas, this specification uses the notational convention of WS-Security. Specifically, each member of an element's [children] or [attributes] property is described using an XPath-like notation (e.g., /x:MyHeader/x:SomeProperty/@value1). The use of {any} indicates the presence of an element wildcard (<xs:any/>). The use of @{any} indicates the presence of an attribute wildcard (<xs:anyAttribute/>).

1.3. Namespaces

The following namespaces are used in this document:

Prefix	Namespace
S	http://www.w3.org/2002/06/soap-envelope
wsse	http://schemas.xmlsoap.org/ws/2003/07/secext
wsu	http://schemas.xmlsoap.org/ws/2002/07/utility

wp	http://schemas.xmlsoap.org/ws/2002/12/policy

1.4. Terminology

The following definitions outline the terminology and usage in this specification.

Passive Requestor – A *passive requestor* is an HTTP browser or application capable of broadly supported HTTP (e.g. HTTP/1.1).

Claim – – A *claim* is a declaration made by an entity (e.g. name, identity, key, group, privilege, capability, etc).

Security Token – A *security token* represents a collection of claims.

Signed Security Token – A *signed security token* is a security token that is asserted and cryptographically signed by a specific authority (e.g. an X.509 certificate or a Kerberos ticket)

Proof-of-Possession – – *Proof-of-possession* is authentication data that is provided with a message to prove that the message was sent and or created by a claimed identity.

Proof-of-Possession Token – A *proof-of-possession token* is a security token that contains data that a sending party can use to demonstrate proof-of-possession. Typically although not exclusively, proof-of-possession information is encrypted with a key known only to the sender and recipient parties.

Digest – A *digest* is a cryptographic checksum of an octet stream.

Signature - A *signature* is a value computed with a cryptographic algorithm and bound to data in such a way that intended recipients of the data can use the signature to verify that the data has not been altered since it was signed by the signer.

Security Token Service (STS) - A *security token service* is a Web service that issues security tokens (see WS-Security). That is, it makes assertions based on evidence that it trusts, to whoever trusts it. To communicate trust, a service requires proof, such as a security token or set of security tokens, and issues a security token with its own trust statement (note that for some security token formats this can just be a re-issuance or co-signature). This forms the basis of trust brokering.

Trust - *Trust* is the characteristic that one entity is willing to rely upon a second entity to execute a set of actions and/or to make set of assertions about a set of subjects and/or scopes.

Trust Domain/Realm - A *Trust Domain/Realm* is a security space in which the target of a request can determine whether particular sets of credentials from a source satisfy the relevant security policies of the target. The target may defer trust to a third party thus including the trusted third party in the Trust Realm.

Direct Trust – *Direct trust* is when a relying party accepts as true all (or some subset of) the claims in the token sent by the requestor.

Direct Brokered Trust – *Direct Brokered Trust* is when one party trusts a second party who, in turn, trusts or vouches for, a third party.

Indirect Brokered Trust – *Indirect Brokered Trust* is a variation on direct brokered trust where the second party negotiates with the third party, or additional parties, to assess the trust of the third party.

Signature validation – *Signature validation* is the process of verifying that the message received is the same as the one sent.

Sender Authentication – *Sender authentication* is corroborated authentication among Web service actors/roles indicating the sender of a Web service message (and its associated data). Note that it is possible that a message may nave multiple senders if authenticated intermediaries exist. Also not that it is application-dependent (and out of scope) as to how it is determined who first created the messages as the message originator might be independent of, or hidden behind an authenticated sender.

Realm or Domain – A *realm* or *domain* represents a single unit of security administration or trust.

Federation – A *federation* is a collection of realms that have established trust. The level of trust may vary, but typically includes authentication and may include authorization.

Identity Provider (IP) – *Identity Provider* is an entity that acts as a peer entity authentication service to end users and data origin authentication service to service providers (e.g. security token service)

Single Sign On – *Single Sign On* is an optimization of the authentication sequence to remove the burden of repeating actions placed on the end user. To facilitate SSO, an element called an Identity Provider can act as a proxy on a user's behalf to provide evidence of authentication events to 3rd parties requesting information about the requestor. These Identity Providers are trusted 3rd parties and need to be trusted both by the requestor (to maintain the requestor's identity information as the loss of this information can result in the compromise of the requestors identity) and the Web services which may grant access to valuable resources and information based upon the integrity of the identity information provided by the IP.

Identity Mapping – *Identity Mapping* is a method of creating relationships between identity properties. Some Identity Providers may make use of id mapping.

Sign-Out – A *sign-out* is the process by which a principal indicates that they will no longer be using their token and services in the realm can destroy their token caches for the principal.

2. Model

The WS-Federation specification defines a model and set of messages for brokering trust and federation of identity and authentication information across different trust realms and protocols. This additional profile shows how this Federations model is applied to passive requestors such as Web browsers.

The federation model described in WS-Federation builds on the foundation established by WS-Security and WS-Trust. Consequently, this specification profiles the mechanisms for requesting, exchanging, and issuing security tokens within the context of a passive requestor.

The Federation model as profiled in this specification allows for support of different but philosophically compatible message exchanges. For example, the resource might act as its own security token service (STS) and not use a separate service (or even URI) thereby eliminating some steps. It is expected that subsequent profiles can be defined to extend the passive profile to include additional exchange patterns.

2.1. Sign-On

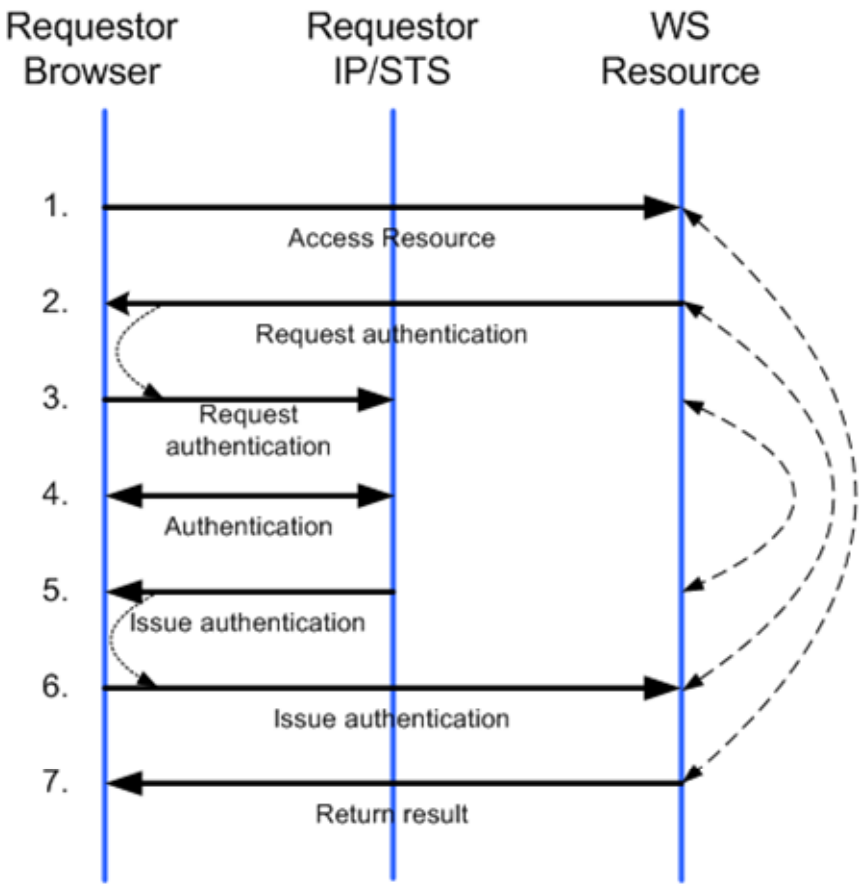
The primary issue for *passive browsers* is that there is no inherent way to directly alter the browser. Consequently, the processing must be performed within the confines of the base HTTP 1.1 functionality (GET, POST, redirects, and cookies) and conform as closely as possible to the WS-Trust protocols for token acquisition.

At a high-level, requestors are associated with an identity provider (IP) or security token service (STS) where they authenticate themselves. At the time/point of initial authentication an artifact/cookie MAY be created for the requestor at their identity provider so that every request for a resource doesn't require requestor intervention. At other times, authentication at each request is the desired behavior.

In this profile, there is a common pattern used when communicating with an IP/STS. In the first step, the requestor accesses the resource; the requestor is then redirected to an IP/STS if no token is cached (in this case the requestor's IP/STS). The IP/STS generates a security token for use by the federated party (the resource). In some cases the IP/STS has the requisite information cached, in other cases it must prompt the user, and in federated scenarios it may require communication with other IP/STS (which is described later).

As indicated all communication occurs with the standard HTTP GET and POST methods using redirects (steps 2 and 6) to automate the communication. In step 2 the resource may act as its own IP/STS so communication with an additional service isn't required. In step 3, a shared or third party IP/STS can also be avoided (depending on the configuration and established trust policies).

It should be noted that in step 4, the authentication protocol employed MAY be implementation-dependent.

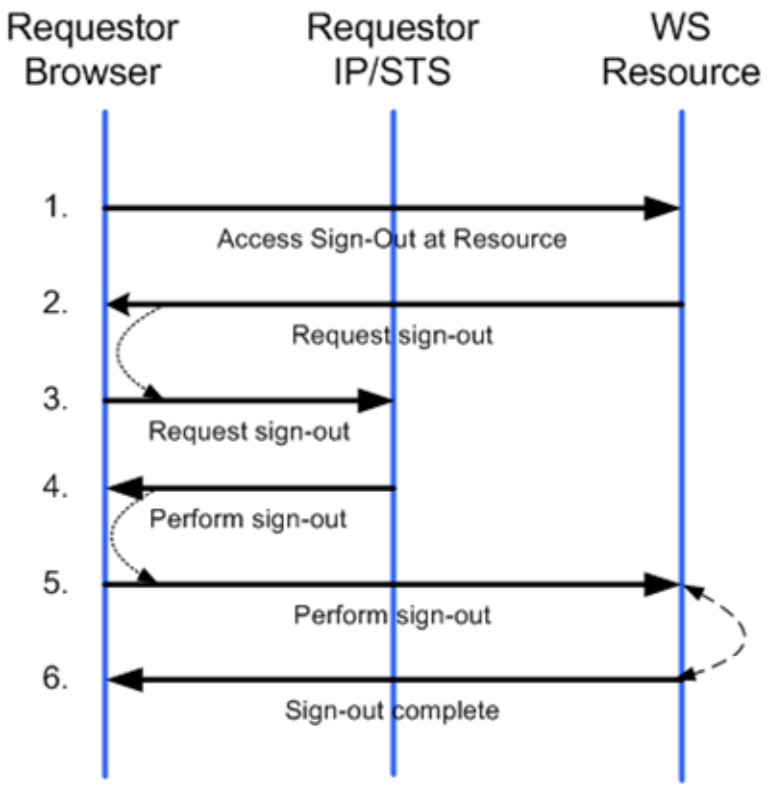


2.2. Sign-Out

For passive browsers, sign-out can be initiated by selecting the sign-out URL at a resource. In doing so, the browser will ultimately be redirected to the requestor's IP/STS indicating sign-out. Note that the browser MAY be first redirected to the resource's IP/STS and then to the requestor's IP/STS. Note that if multiple IP/STS services are used, and unaware of each other, multiple sign-outs may be required.

The requestor's IP/STS SHOULD keep track of the realms to which it has issued tokens where cleanup may be required – specifically the IP/STS for the realms (or resources if different). When the sign-out is received at the requestor's IP/STS, it is responsible for issuing HTTP GET requests against the tracked realms indicating a sign-out cleanup is in effect or it can use the sign-out mechanism described in WS-Federation if it is supported by the endpoints. The exact mechanism by which this occurs is up to the IP/STS. The only requirement is that a sign-out cleanup GET be performed against any realms that may have cached tokens. Optionally, the requestor's IP/STS can request that the sign-out cleanup GET redirect back to the requestor's IP/STS.

When a sign-out clean-up GET is received at a realm, the realm SHOULD clean-up any cached information and delete any associated artifacts/cookies. If requested, on completion the requestor is redirected back to requestor's IP/STS.

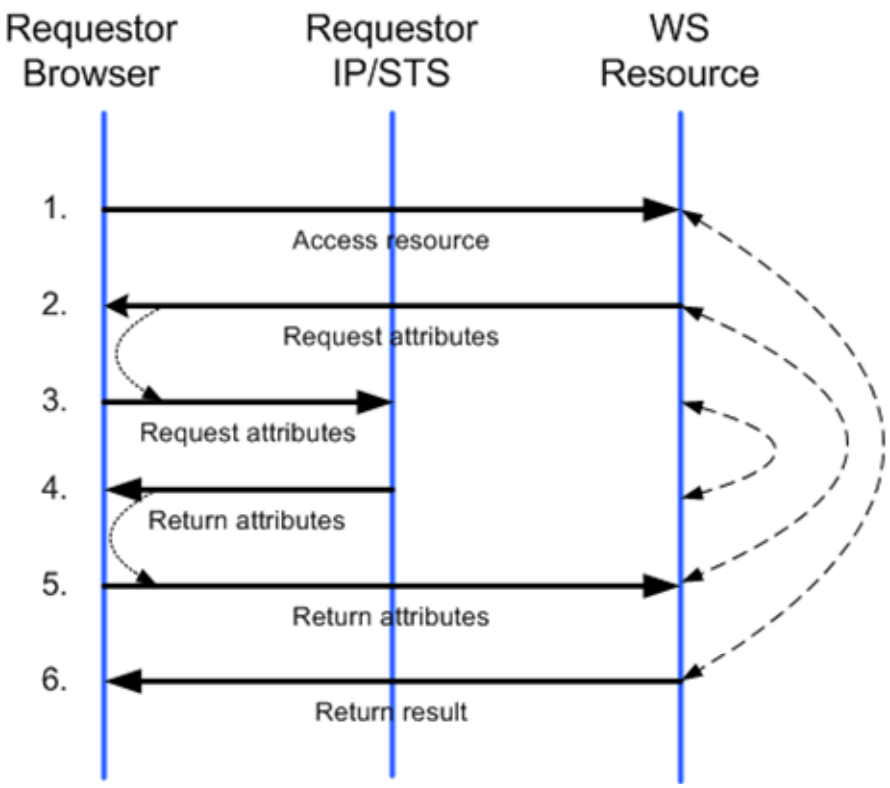


The figure above illustrates this process including calling out the redirection in steps 2 and 4 (optional) and the general *correlation* of messages.

It should be noted that as a result of the single sign-out request (steps 5 and 6), an IP/STS MAY send sign-out messages as described in WS-Federation.

2.3. Attributes

At a high-level, attribute processing uses the same mechanisms defined for security token service requests and responses. That is, redirection is used to issue requests to attribute services and subsequent redirection returns the results of the attribute operations. All communication occurs with the standard HTTP 1.1 GET and POST methods using redirects to automate the communication as shown in the example below.

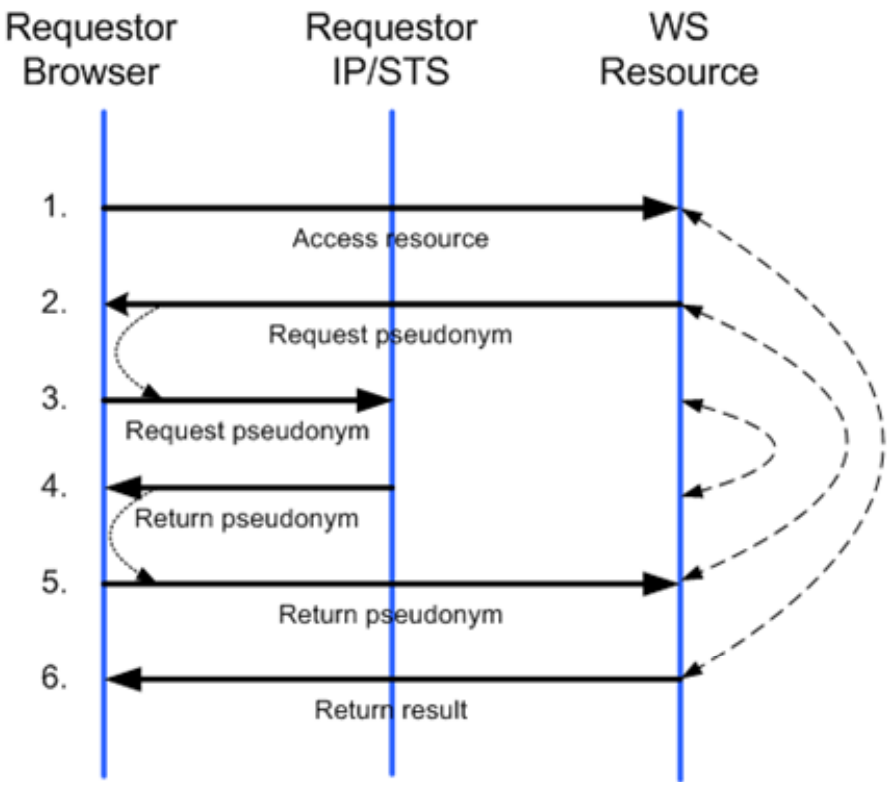


The figure above illustrates this process including calling out the redirection in steps 2 and 4 and the general *correlation* of messages.

As well, it should be noted that as a result of step 3 the IP/STS MAY prompt the user for approval before proceeding to step 4.

2.4. Pseudonyms

At a high-level, pseudonym processing uses the same mechanisms defined for attribute and security token service requests. That is, redirection is used to issue requests to pseudonym services and subsequent redirection returns the results of the pseudonym operations. All communication occurs with the standard HTTP GET and POST methods using redirects to automate the communication as in the example below.



The figure above illustrates this process including calling out the redirection in steps 2 and 4 and the general *correlation* of messages.

2.5. Artifacts/Cookies

In order to prevent requestor interaction on every request for security token, artifacts/cookies can be used by SSO implementations as they are used today to cache state and/or authentication information or issued tokens. However implementations MAY omit this caching if the desired behavior is to authenticate on every request. As noted in the Security Consideration section later in this document, there are security issues when using cookies.

There are no restrictions placed on artifacts/cookie formats – they are up to each service to determine. However, it is RECOMMENDED artifacts/cookies be encrypted or computationally hard to compromise.

3. HTTP Protocol Syntax

This section describes the syntax of the protocols used by passive requestors. This protocol typically uses the redirection facilities of HTTP 1.1. This happens using a standard HTTP 302 error code for redirects (as illustrated below) and HTTP POST to push the forms:

```
HTTP/1.1 302 Found
Location: url?parameters
```

The exact parameters and form fields are described in detail in the sub-sections that follow the detailed example.

In the descriptions below, some mechanisms are optional meaning they MAY be supported. Within an mechanism, certain parameters MUST be specified while others, noted using square brackets, are optional and MAY or MAY NOT be present.

3.1. Parameters

All HTTP 1.1 methods (both GET and POST) used in the redirection protocol allow query string parameters as illustrated below:

```
GET url?parameters

POST url?parameters
```

The GET and POST requests have required parameters and may have optional parameters depending on the operation being performed. For GET requests, these parameters are specified in the query string; for POST requests, these parameters are specified in the POST body (using the standard encoding rules for POST). The query string parameters of a POST request SHOULD be for extensibility only. The following describes the parameters used for messages in this profile:

```
wa=string
[wreply=URL]
[wres=URL]
[wctx=string]
[wp=URI]
[wct=timestring]
```

wa
: This required parameter specifies the action to be performed. By including the action, URIs can be overloaded to perform multiple functions. For sign-in, this string MUST be "wsignin1.0".

wreply
: This optional parameter is the URL to which responses are directed.

wres
: This optional parameter is the URL for the resource accessed.

wctx
: This optional parameter is an opaque context value that MUST be returned with the issued token if it is passed in the request.

wp
: This optional parameter is the URL for the policy which can be obtained using an HTTP GET and identifies the policy to be used related to the action specified in "wa", but MAY have a broader scope than just the "wa". Refer to WS-Policy and WS-Trust for details on policy and trust. This attribute is only used to reference policy documents.

wct
: This optional parameter indicates the current time at the recipient for ensuring freshness. This parameter is the string encoding of time using the XML Schema datetime time using UTC notation.

Note that any values specified in parameters are subject to encoding as specified in the HTTP 1.1 specification.

When an HTTP POST is used, any of the query strings can be specified in the form contents using the same name. Note that in this profile form values take precedence over URL parameters.

Parameterization is extensible so that cooperating parties can exchange additional information in parameters based on agreements or policy.

3.2. Requesting Security Tokens

The HTTP requests to an identity provider or security token services use a common syntax based on HTTP forms. Requests typically arrive using the HTTP GET method as illustrated below but MAY be issued using a POST method:

```
GET resourceSTS?parameters HTTP/1.1
POST resourceSTS?parameters HTTP/1.1
```

The parameters described in the previous section (wa, wreply, wres, wctx, wp, wct) apply to the token request. The additional parameters described below also apply. Note that any values specified in forms are subject to encoding as described in the HTTP 1.1 specification.

The following describes the additional optional parameters used for a token request:

```
[wtrealm=string]
[wreq=xml]
```

wtrealm
: This optional parameter is the URI of the requesting realm. This should be specified if it isn't obvious from the request (e.g. the `wreply` parameter).). The wtrealm SHOULD be a security realm of the resource in which nobody (except the resource or authorized delegates) can control URLs.

wreq
: This optional parameter specifies a token request using either a `<wsse:RequestSecurityToken>` element or a full request message as described in WS-Trust. If this parameter is not specified, it is assumed that the responding service *knows* the correct type of token to return.

In the event that the XML request cannot be passed in the form (due to size or other considerations), the following parameter MAY be specified and the form made available by reference:

```
wreqptr=url
```

wreqptr
: This optional parameter specifies a URL for where to find the request (`wreq` parameter).

It is strongly RECOMMENDED that the *resourceSTS* secure information be signed using XML Signature or use HTTP/S or some other transport-level security mechanism.

3.3. Returning Security Tokens

Security tokens are returned by passing an HTTP form. To return the tokens, this profile embeds a <wsse:RequestSecurityTokenResponse> element as specified in WS-Trust.

```
POST resourceURI?parameters HTTP/1.1

GET resourceURI?parameters HTTP/1.1
```

In many cases the IP/STS to whom the request is being made, will prompt the requestor for information or for confirmation of the receipt of the token. As a result, the IP/STS can return an HTTP form to the requestor who then submits the form using an HTTP POST method. This allows the IP/STS to return security token request responses in the body rather than embedded in the limited URL query string. However, in some circumstances interaction with the requestor may not be required (e.g. cached information). In these circumstances the IP/STS have several options:

1. Use a form anyway to confirm the action
2. Return a form with script to automate and instructions for the requestor in the event that scripting has been disabled
3. Use HTTP GET and return a pointer to the token request response (unless it is small enough to fit inside the query string)

This specification RECOMMENDS using the POST method as the GET method requires additional state to be maintained and complicates the cleanup process whereas the POST method carries the state inside the method.

Note that when using the POST method, any values specified in parameters are subject to encoding as described in the HTTP 1.1 specification. The standard parameters apply to returning tokens as do the following additional form parameters:

```
wresult=xml

[wctx=string]
```

wresult
: This required parameter specifies the result of the token issuance. This can take the form of the <wsse:RequestSecurityTokenResponse> element, a SOAP security token request response (that is, a <S:Envelope>) as detailed in WS-Trust, or a SOAP <S:Fault> element.

wctx
: This optional parameter specifies the context information (if any) passed in with the request. It should be noted that this parameter specifies the context information (if any) passed in with the original request.

In the event that the token/result cannot be passed in the form, the following parameter MAY be specified:

```
wresultptr=url
```

wresultptr

This parameter specifies a URL to which an HTTP GET can be issued. The result is a document of type `text/xml` that contains the issuance result. This can either be the `<wsse:RequestSecurityTokenResopnse>` element, a SOAP response, or a SOAP `<S:Fault>` element.

3.4. Sign-Out Request Syntax

This section describes how sign-out requests are formed and redirected by passive requestors. For modularity, it should be noted that support for sign-out is optional.

The following describes the parameters used for the sign-out request:

```
wa=string
wreply=URL
```

wa
: This required parameter specifies the action to be performed. By including the action, URIs can be overloaded to perform multiple functions. For sign-out, this string MUST be "wsignout1.0".

wreply
: This optional parameter specifies the URL to return to once clean-up (sign-out) is complete.

The following describes the parameters used for the sign-out cleanup request:

```
wa=string
wreply=URL
```

wa
: This required parameter specifies the action to be performed. By including the action, URIs can be overloaded to perform multiple functions. For sign-out cleanup, this string MUST be "wsignoutcleanup1.0".

wreply
: This optional parameter specifies the URL to return to once clean-up (sign-out) is complete. If this parameter is specified, the requestor is redirected to the URL after cleanup completes. If this parameter is not specified, then after cleanup the GET completes by returning any realm-specific data such as a string indicating cleanup is complete for the realm.

3.5. Attribute Request Syntax

This section describes how attribute requests are formed and redirected by passive requestors. For modularity, it should be noted that support for attributes is optional. Additionally it should be noted that security considerations may apply. While the structure described here can be used with any attribute service supporting passive clients, the actual attribute request and response XML syntax is undefined and specific to the attribute store.

The following describes the valid parameters used within attributes requests:

```
wa=string
[wreply=URL]
wattr=xml-attribute-request
wresult=xml-result
```

wa

This required parameter specifies the action to be performed. By including the action, URIs can be overloaded to perform multiple functions. For attribute requests, this string MUST be "wattr1.0".

wreply
: This optional parameter specifies the URL to return to when the attribute result is complete.

wattr
: This required parameter specifies the attribute request. The syntax is specific to the attribute store being used and is not mandated by this specification. This attribute is only present on the request.

wresult
: This required parameter specifies the result as defined by the attribute store and is not mandated by this specification. This attribute is only present on the responses.

3.6. Pseudonym Request Syntax

This section describes how pseudonym requests are formed and redirected by passive requestors. For modularity, it should be noted that support for pseudonyms is also optional. As well, it should be noted that security considerations may apply.

The following describes the valid parameters used within pseudonym requests:

```
wa=string
[wreply=URL]
wpseudo=xml-pseudonym-request
wresult=xml-result
```

wa
: This required parameter specifies the action to be performed. By including the action, URIs can be overloaded to perform multiple functions. For pseudonym requests, this string MUST be "wpseudo1.0".

wreply
: This optional parameter specifies the URL to return to when the pseudonym result is complete.

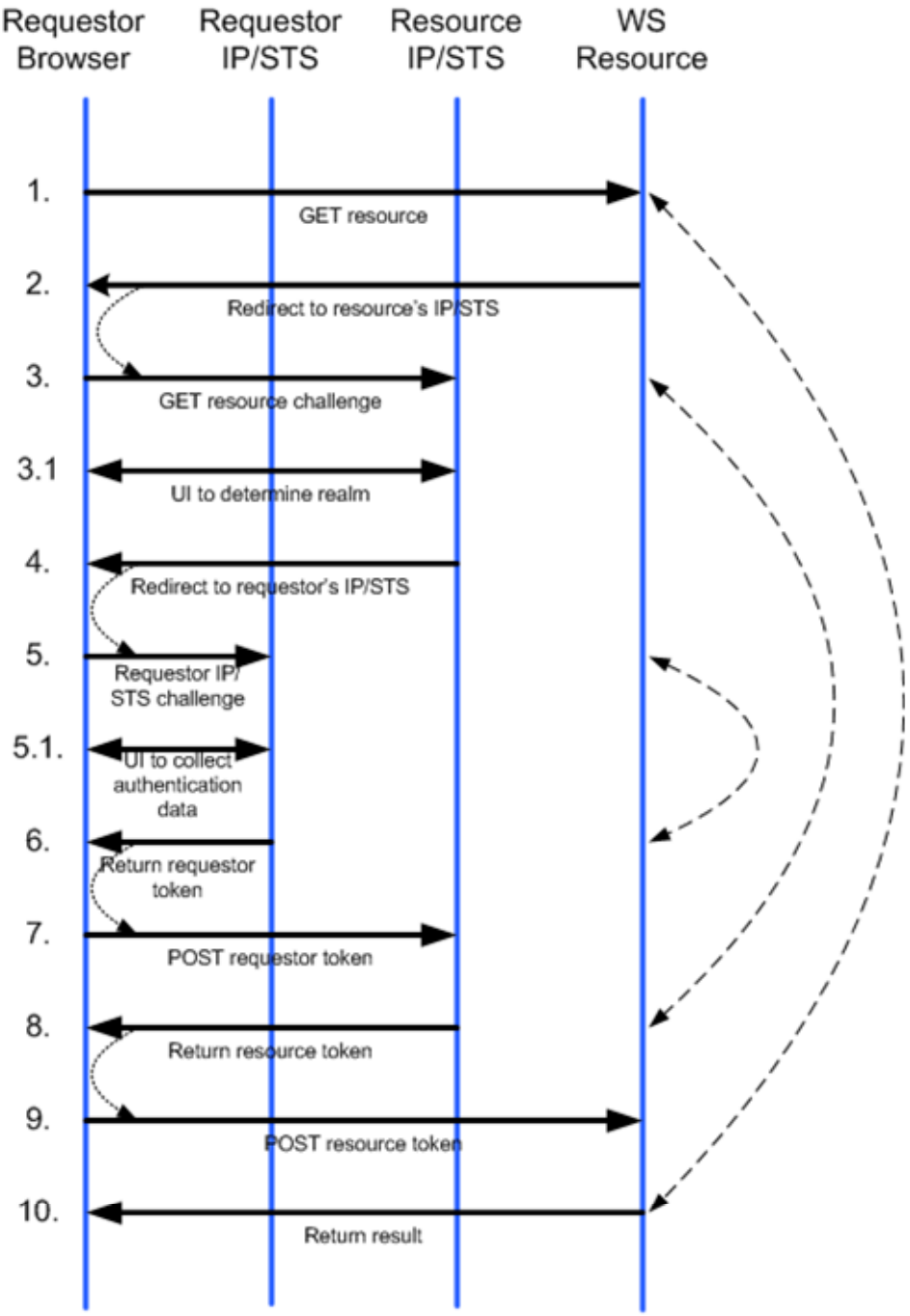
wpseudo
: This required parameter specifies the pseudonym request and either contains a SOAP envelope or an attribute request, such as `<wsse:GetPseudonym>`. This attribute is only present on the request.

wresult
: This required parameter specifies the result as either a SOAP envelope or a pseudonym response, such as a `<wsse:GetPseudonymResponse>`. This attribute is only present on the responses.

4. Detailed Example of Passive Requester Syntax

This section provides a detailed example of the profile defined in this specification. The exact flow for Web sign-in scenarios can vary significantly; however, the following diagram and description depict a *common* or basic sequence of events.

In this scenario, the user at a requestor browser is attempting to access a resource which requires security authentication to be validated by the resource's security token service.



Simple Scenario:

This scenario depicts an initial federated flow. Note that subsequent flows from the requestor to the resource realm may be optimized. The steps below describe the above interaction diagram. Appendix I provides a set of sample HTTP messages for these steps.

Step 1: The requestor browser accesses a resource, typically using the HTTP GET method.

Step 2: At the resource, the requestor's request is redirected to the IP/STS associated with the target resource. The redirected URL MAY contain additional information reflecting agreements which the resource and its IP/STS have established; however, this (redirection target) URL MUST be used throughout the protocol as the URL for the resource's IP/STS. Typically, this occurs using a standard HTTP 302 error code. (Alternatively, the request for the token MAY be done using a HTTP POST method described in step 6).

It is RECOMMENDED that the resource STS provide confidentiality (e.g. using encryption or HTTP/S) of the information.

Step 3: Upon receipt of the redirection, the IP/STS must determine the requestor realm. This requestor realm could be cached in an artifact/cookie from an earlier exchange, it could be known to or fixed by the resource, or the requestor MAY be prompted to enter or select their realm (step 3.1).

Step 3.1: This is an optional step. If the resource IP/STS cannot determine the requestor's realm, then the IP/STS may prompt the requestor for realm information.

Step 4: The resource IP/STS redirects to the requestor's IP/STS in order to validate the requestor. Typically, this is done using a HTTP 302 redirect.

As in step 2, additional information may be passed to reflect the agreement between the two IP/STS's, and this request for the token MAY be done using a POST method (see syntax for details).

The requestor IP/STS SHOULD provide information confidentiality or use HTTP/S or some other transport-level security mechanism.

Step 5: The requestor's IP/STS now authenticates the requestor to establish a sign in.

Step 5.1: Validation of the requestor may involve displaying some UI in this optional step.

Step 6: Once requestor information has been successfully validated, a security token response (RSTR) is formatted and sent to the resource IP/STS.

Processing continues at the resource IP/STS via a redirect.

While an IP/STS MAY choose to return a pointer to token information using wresultptr, it is RECOMMENDED that, whenever possible to return the security token (RSTR) using a POST method to reduce the number of overall messages. This MAY be done using requestor-side scripting. The exact syntax is described in Appendix I.

Step 7: Resource's IP/STS receives and validates the requestor's security token (RSTR).

Step 8: The resource's IP/STS performs a federated authentication/authorization check (validation against policy). After a successful check, the resource's IP/STS can issue a security token for the resource. The resource IP/STS redirects to the resource.

It should be noted that the optional `wctx` parameter specifies the opaque context information (if any) passed in with the original request and is echoed back here. This mechanism is an optional way for the IP/STS to have state returned to it.

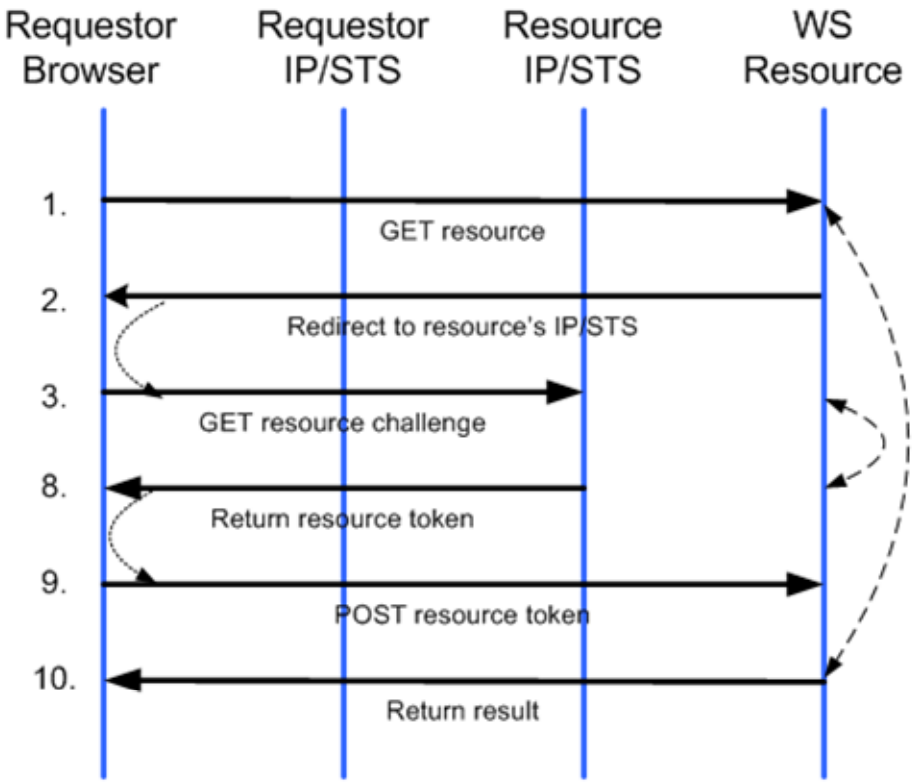
At this point the resource's IP/STS MAY choose to set an artifact/cookie to indicate the sign-in state of the requestor (which likely includes the requestor's realm).

Step 9: The resource receives the security token (RSTR) from the resource IP/STS. On successful validation the resource processes the request (per policy).

The security token SHOULD be passed using an HTML POST using the syntax previously described.

Step 10: The resource MAY establish a artifact/cookie indicating the sign-in state of the requestor when it returns the result of the resource request.

Optimized Scenario:



This scenario assumes that an initial federated flow has occurred. Note that many legs of the initial flow may be eliminated due to the presence of artifacts/cookies. For readability, the similar steps are number consistently with the previous non-optimized example.

Step 1: The requestor browser accesses a resource, typically using the HTTP GET method.

Step 2: At the resource, the requestor's request is redirected to the IP/STS associated with the target resource. The redirected URL MAY contain additional information reflecting agreements which the resource and its IP/STS have established; however, this (redirection target) URL MUST be used throughout the protocol as the URL for the resource's IP/STS. Typically, this occurs using a standard HTTP 302 error code. (Alternatively, the request for the token MAY be done using a HTTP POST method described in step 6).

It is RECOMMENDED that the resource STS provide confidentiality (e.g. using encryption or HTTP/S) of the information.

Step 3: Upon receipt of the redirection, the IP/STS must determine the requestor realm. This requestor realm could be cached in an artifact/cookie from an earlier exchange, it could be known to or fixed by the resource, or the requestor MAY be prompted to enter or select their realm (step 3.1).

Step 8: The resource's IP/STS performs a federated authentication/authorization check (validation against policy). After a successful check, the resource's IP/STS can issue a security token for the resource. The resource IP/STS redirects to the resource.

It should be noted that the optional `wctx` parameter specifies the opaque context information (if any) passed in with the original request and is echoed back here. This mechanism is an optional way for the IP/STS to have state returned to it.

At this point the resource's IP/STS MAY choose to set an artifact/cookie to indicate the sign-in state of the requestor (which likely includes the requestor's realm).

Step 9: The resource receives the security token (RSTR) from the resource IP/STS. On successful validation the resource processes the request (per policy).

The security token SHOULD be passed using an HTML POST using the syntax previously described.

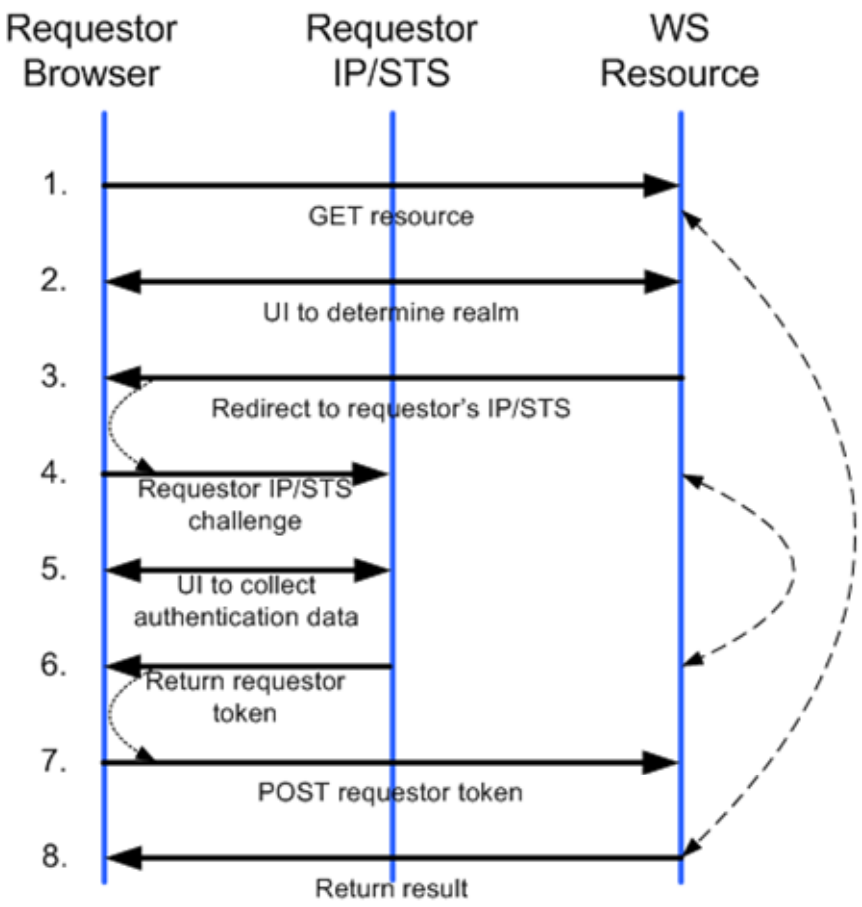
Step 10: The resource MAY establish a artifact/cookie indicating the sign-in state of the requestor when it returns the result of the resource request.

5. Additional Examples

This section presents interaction diagrams for additional passive requestor scenarios.

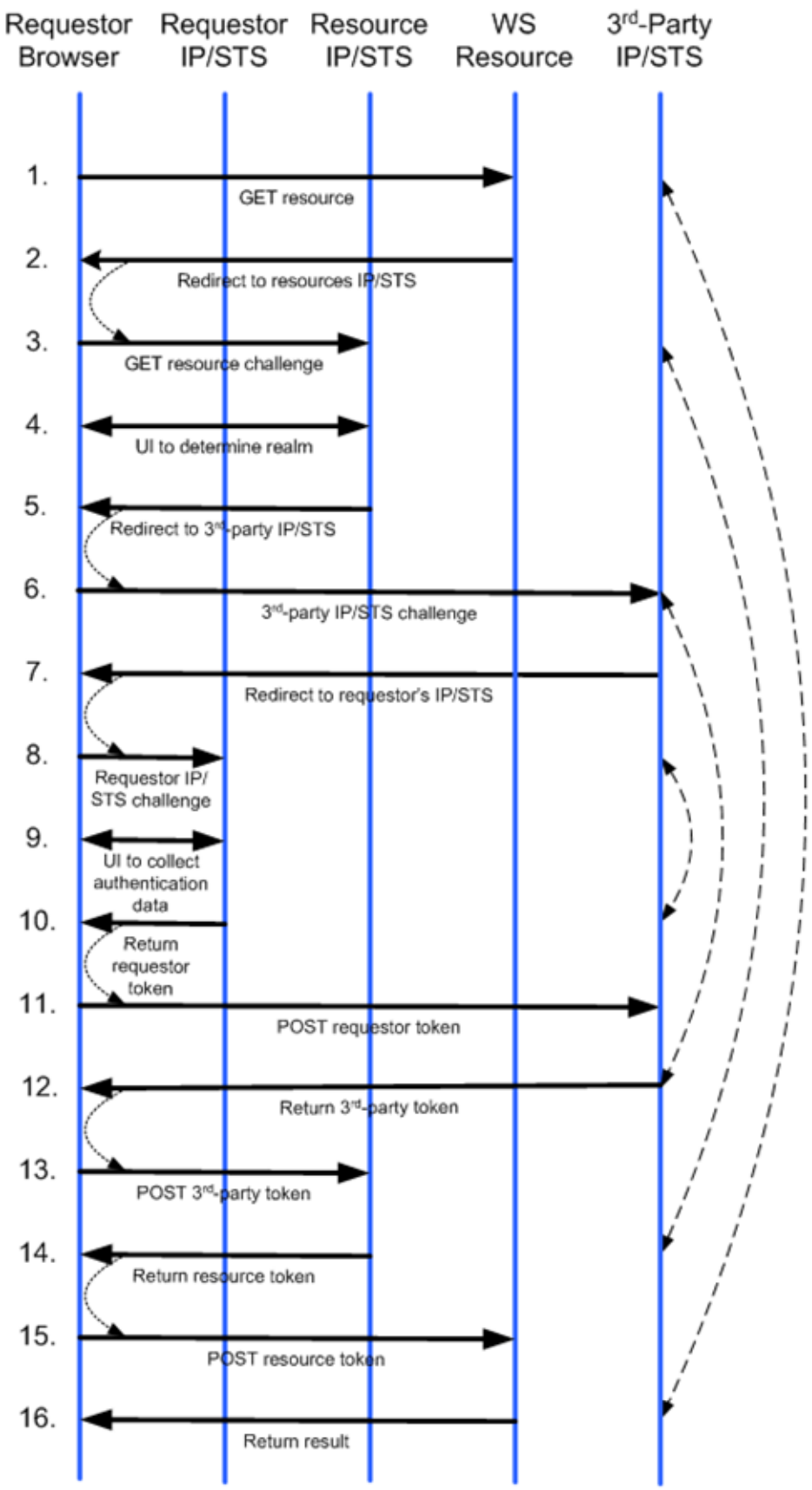
5.1. No Resource STS

The figure below illustrates the sign-in scenario above, but without a resource STS. That is, the requestor acts as its own STS:



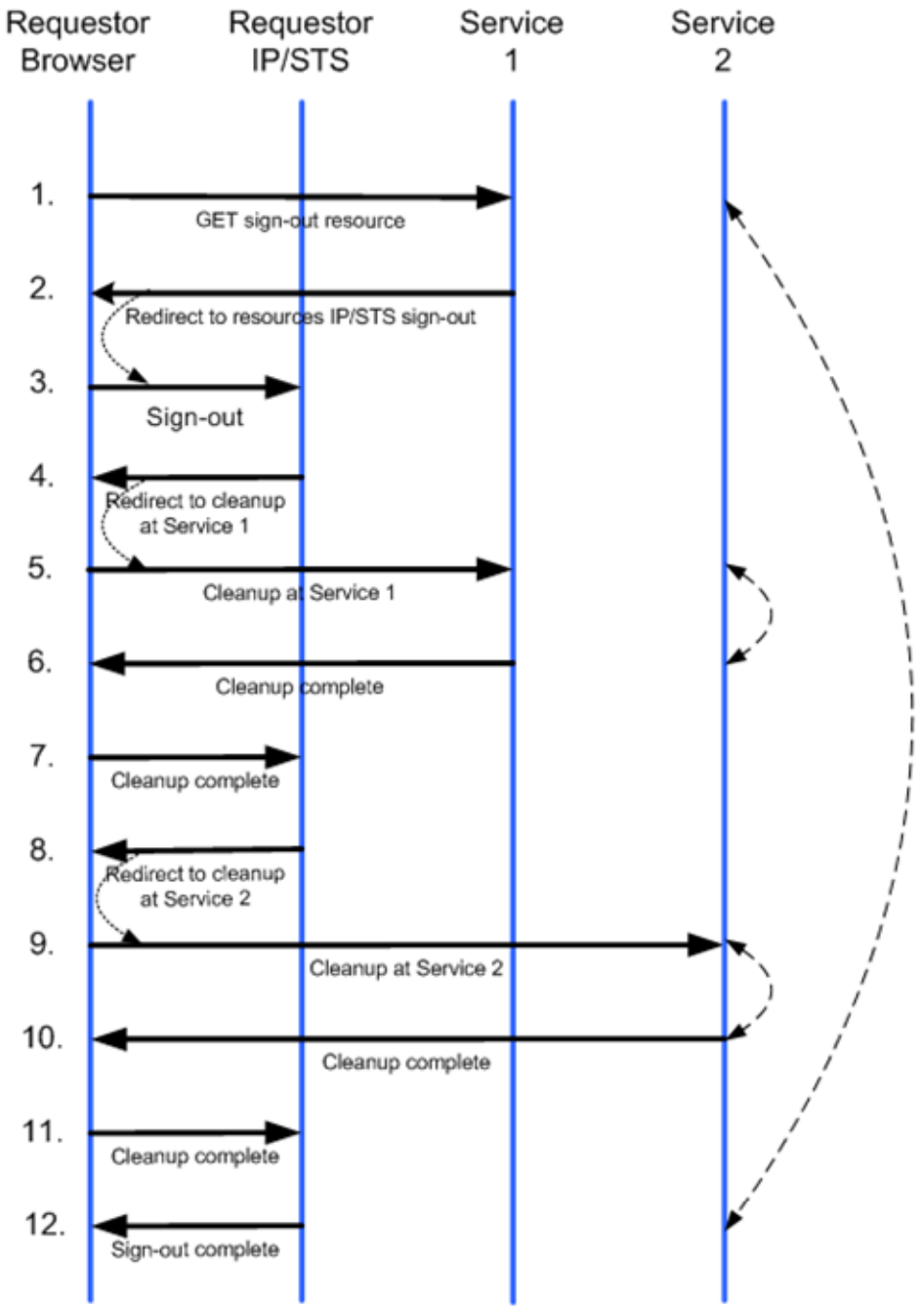
5.2. 3rd-Party STS

The figure below illustrates the sign-in scenario above, but trust is brokered through a 3rd-party STS:


Requestor Browser
Requestor IP/STS
Resource IP/STS
WS Resource
3rd-Party IP/STS
1.
GET resource
2.
Redirect to resources IP/STS
3.
GET resource challenge
4.
UI to determine realm
5.
Redirect to 3rd-party IP/STS
6.
3rd-party IP/STS challenge
7.
Redirect to requestor's IP/STS
8.
Requestor IP/ STS challenge
9.
UI to collect authentication data
10.
Return requestor token
11.
POST requestor token
12.
Return 3rd-party token
13.
POST 3rd-party token
14.
Return resource token
15.
POST resource token
16.
Return result

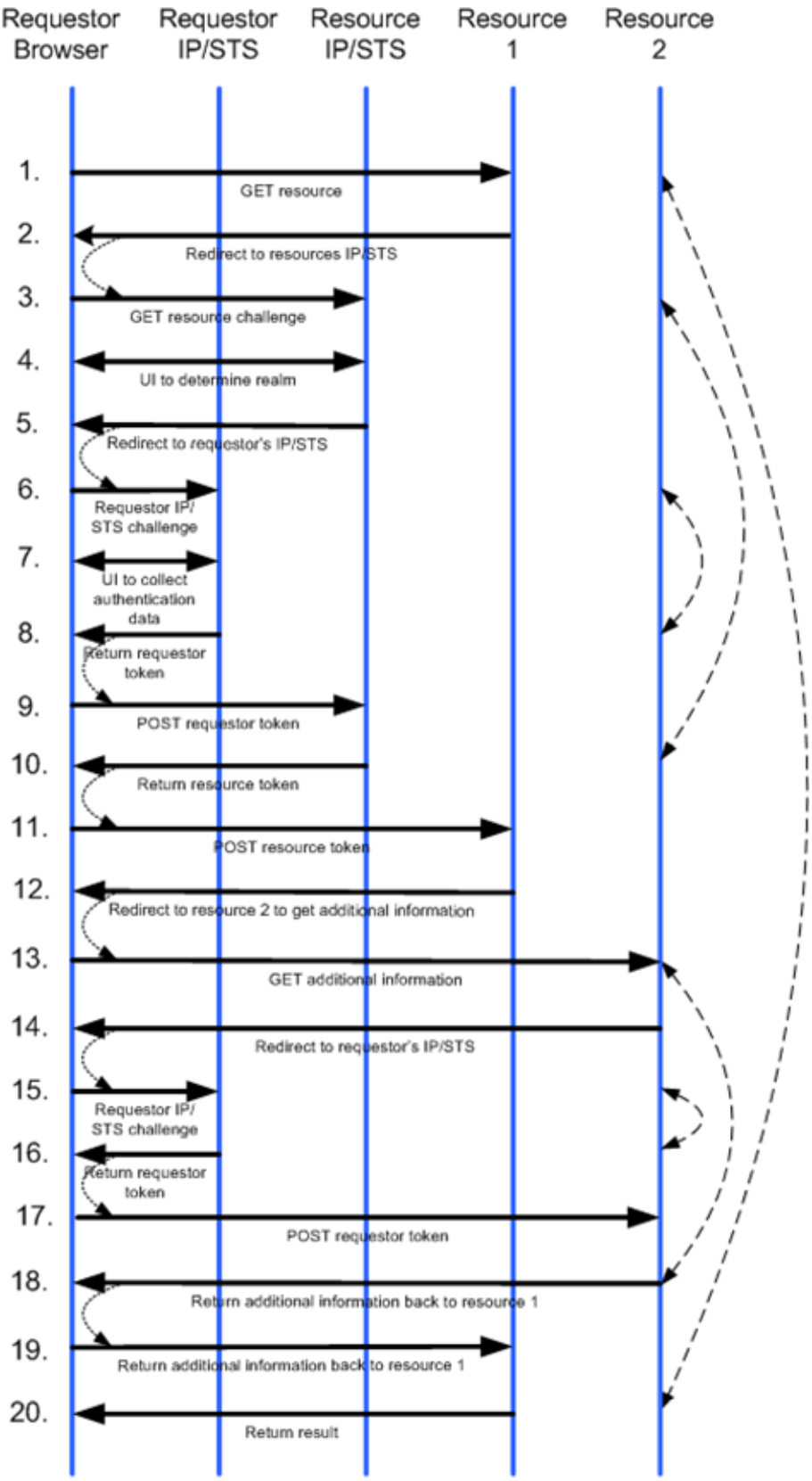
5.3. Sign-Out

The figure below illustrates the sign-out flow for a requestor that has signed in at two sites and requests that the sign-out cleanup requests redirect back to the requestor:



5.4. Delegated Resource Access

The figure below illustrates the case where a resource accesses data from another resource on behalf of the first resource and the information is returned through the requestor:


Requestor Browser
Requestor IP/STS
Resource IP/STS
Resource 1
Resource 2
1.
GET resource
2.
Redirect to resources IP/STS
3.
GET resource challenge
4.
UI to determine realm
5.
Redirect to requestor's IP/STS
6.
Requestor IP/ STS challenge
7.
UI to collect authentication data
8.
Return requestor token
9.
POST requestor token
10.
Return resource token
11.
POST resource token
12.
Redirect to resource 2 to get additional information
13.
GET additional information
14.
Redirect to requestor's IP/STS
15.
Requestor IP/ STS challenge
16.
Return requestor token
17.
POST requestor token
18.
Return additional information back to resource 1
19.
Return additional information back to resource 1
20.
Return result

6. Security Tokens

When accepting security tokens, recipients SHOULD:

- Verify the token is formatted correctly
- Verify STS signature
- Verify the token validity interval
- Verify properties requested by policy such as required authentication type, maximum time since authentication instant (e.g. a password must have been submitted within 1 hour), identity properties etc.

This chapter describes token format-specific requirements but it does not mandate usage of a particular token type.

6.1. X.509v3

This specification places the following requirements on X.509 tokens:

- Tokens MUST contain the name of the issuing authority and a signature of the issuing authority over the whole token unless a secure channel is used to communicate the token. That is, a signature element over the assertions. Note that it is RECOMMENDED that a signature be used even if a secure channel is used.
- Tokens MUST contain the subject identifier uniquely identifying the subject for whom the token was granted. X.509 does not specify rules for `Principal`field. X.509 tokens conformant with this specification SHOULD assure the principals issued are unique across realms and also the realm SHOULD be derivable from the principal name.
- Tokens MAY contain the time of initial authentication, validity interval and the type of authentication that was performed.
- Tokens MAY contain Certificate Revocation Information, such as a CRL distribution point
- X.509 certificates MUST be carried within a `wsse:BinarySecurityToken` element whose ValueType is `wsse:X509v3`.

6.2. Kerberos

This specification places the following requirements on Kerberos tokens:

- Kerberos ticket-granting tickets MUST be carried within a `wsse:BinarySecurityToken` element whose ValueType is `wsse:Kerberosv5TGT`.
- Kerberos service tickets MUST be carried within a `wsse:BinarySecurityToken` element whose ValueType is `wsse:Kerberosv5ST`.
- The symmetric key used SHOULD be derived from the desired realm.

6.3. XrML

This specification places the following requirements on XrML tokens:

- Processors MUST support the `xrml:issuer` element with and without contained signatures. Processors SHOULD NOT include a contained signature unless the `xrml:license` conveys the key (directly or indirectly).
- Tokens that contain signatures in one or more `xrml:issuer` elements MUST declare all XML namespaces on the `xrml:license` element.

- Processors MUST include an `xrml:issuer` element identifying the issuer under `xrml:details`.
- Processors MUST include within the `xrml:issuer` element an `xrml:validityInterval` when the xrml:license token conveys the key (directly or indirectly). The `xrml:validityInterval` MUST contain both `xrml:notBefore` and `xrml:notAfter` elements.
- Tokens SHOULD contain a recipient identifier indicating the scope of usage (such as the resource or realm) - this is represented by `grant resource`, with the tacit assumption that the realm is used.

6.4. SAML

This specification places the following requirements for SAML tokens:

- Tokens MUST contain a signature of the issuing authority over the whole token unless a secure channel is used to communicate the token. That is, a signature element over the SAML assertion. Note that it is RECOMMENDED that a signature be used even if a secure channel is used.
- Tokens MUST contain the subject identifier uniquely identifying the subject for whom the token was granted. SAML does not specify rules for `NameIdentifier` element. The SAML assertions conformant with this specification SHOULD assure the identifiers issued are unique across realms and also the realm SHOULD be derivable from the subject identifier.
- Tokens SHOULD contain a recipient identifier indicating the scope of usage (such as the resource or realm) - this is represented by the `AudienceRestriction` or `Recipient` elements in the SAML assertion.
- Tokens MUST contain the time of initial authentication, validity interval and the type of authentication that was performed. The validity interval in the SAML assertion is satisfied by the `NotBefore` and `NotOnOrAfter` attributes of the Conditions element. The initial authentication type and time are covered by the attributes of `AuthenticationStatement` element.
- Tokens MAY contain additional identity information. If they do, the schema describing the additional information MUST be understood by the recipient or the token MUST be rejected.

7. Error Handling

Errors are handled using the error mechanisms of HTTP as well as using the embedded Fault mechanisms. That is, HTTP-related errors are indicated using established HTTP errors. SOAP-related errors are handled using SOAP Fault elements are previously described.

8. Security Considerations

If a security token is not self-securing, it SHOULD be included in some form of message integrity mechanism.

If privacy is a concern, the security tokens MAY be encrypted for the authorized recipient(s).

The browser-based protocols described here suffer from the same vulnerabilities that exist for all browser-based interactions:

- **Spoofing:** Web based sign-in requests require that security tokens are submitted in a HTTP POST form. Generally, it is difficult for the requestors to identify rogue pages. For example a malicious Web site can redirect the requestor to a fake Web sign in page – a fake ATM attack. To mitigate the threat, it is strongly RECOMMENDED that sign in pages are served over secure connections. Also, it is strongly recommended that security token services employ other means for spoof protection such as presenting requestor specific token during authentication and drawing attention to the URL being accessed.
- **Replay attacks:** It is possible that requests for security tokens could be replayed. Consequently, it is RECOMMENDED that all communication between security token services and resources take place over secure connections. All cookies indicating sign-in state SHOULD be set as secure.
- **Cookie requirements:** Since cookies COULD be used to keep sign in state, it is computationally infeasible to randomly guess a cookie value. It is RECOMMENDED that cookies represent binary blobs encrypted with sufficiently strong keys – e.g., AES or 3DES.
- **Cross-site scripting**: To prevent some cross-site scripting attacks, security token services MUST NOT return content to the requestor that was specified by a third party.
- **Forged security tokens:** Security token services MUST guard their signature keys to prevent forging of tokens and requestor identities.
- **Privacy:** Security token services SHOULD NOT send requestors' personal identifying information without getting consent from the requestor. For example a Web site SHOULD NOT receive requestors' personal information without an appropriate consent process.
- **Compromised services:** If a security token service is compromised, all requestor accounts serviced SHOULD be assumed to be compromised as well (since an attacker can issue security tokens for any account within the compromised realm or into any realm that trusts the compromised realm).
- **Man-in-the-Middle attacks**: The `wtreply` must be in `wtrealm` (i.e., the same URL, or, e.g., `wtreply` is a host within the domain of `wtrealm`). It is strongly RECOMMENDED that the Identity Provider verifies this, and that `wtreply` is an valid HTTP/S address.
    - The `wtrealm` SHOULD be a security realm of the resource in which nobody can control URLs.
        - For Kerberos tokens the key distribution SHOULD distribute correct realms for the keys, so that Identity Providers know what the correct realms are for keys that they use.
        - For SAML tokens the resource SHOULD verify that exactly this realm is in one of the two (fix one!) fields of the ticket.
        - For other token types similar considerations SHOULD be made before using them.

It is strongly RECOMMENDED that the *resourceSTS* secure information or use HTTP/S or some other transport-level security mechanism for all communications.

9. Acknowledgements

This specification has been developed as a result of joint work with many individuals and teams, including:

Giovanni Della-Libera, Microsoft
Josh Gray, Microsoft
Tim Hahn, IBM
Heather Hinton, IBM
Ryan Johnson, Microsoft
Bronislav Kavsan, RSA Security
Anthony Moran, IBM
Birgit Pfitzmann, IBM
Robert Philpott, RSA Security
Shane Weeden, IBM

10. References

[KEYWORDS]
S. Bradner, "Key Words for Use in RFCs to Indicate Requirement Levels," RFC 2119, Harvard University, March 1997.

[HTTP]
R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, RFC 2616, "Hypertext Transfer Protocol -- HTTP/1.1". June 1999.

[SOAP]
W3C Note, "SOAP: Simple Object Access Protocol 1.1," 08 May 2000.
Draft, SOAP 1.2, http://www.w3.org/TR/soap12-part0/
Draft, SOAP 1.2, http://www.w3.org/TR/soap12-part1/
Draft, SOAP 1.2, http://www.w3.org/TR/soap12-part2/

[URI]
T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[WS-Federation]
"Web Services Federation Language", BEA, IBM, Microsoft, RSA Security, VeriSign, July 2003

[WS-Security]
"Web Services Security Language", IBM, Microsoft, VeriSign, April 2002.
"WS-Security Addendum", IBM, Microsoft, VeriSign, August 2002.
"WS-Security XML Tokens", IBM, Microsoft, VeriSign, August 2002

[WS-Policy]
"Web Services Policy Framework", BEA, IBM, Microsoft, SAP, December 2002

[WS-PolicyAttachment]
"Web Services Policy Attachment Language", BEA, IBM, and Microsoft, SAP, December 2002

[WS-PolicyAssertions]
"Web Services Policy Assertions Language", BEA, IBM, Microsoft, SAP, December 2002

[WS-Trust]

"Web Services Trust Language", IBM, Microsoft, RSA, VeriSign, December 2002

[WS-SecureConversation]
"Web Services Secure Conversation Language", IBM, Microsoft, RSA, VeriSign, December 2002

[WS-SecurityAssertions]
"Web Services Security Assertions Language", IBM, Microsoft, RSA, Verisign December 2002

[XML-ns]
W3C Recommendation, "Namespaces in XML," 14 January 1999.

Appendix I. Sample HTTP Flows for Detailed Example

This appendix provides sample HTTP messages for the detailed example previously described.

In this example, the following URLs are used:

Item	URL
Resource Realm	`Resource.com`
Resource	`https://res.resource.com/sales`
Resource's IP/STS	`https://sts.resource.com/sts`
Account	`Account.com`
Resource	`https://sts.account.com/sts`

Step 1 – GET resource

```
GET https://res.resource.com/sales HTTP/1.1
```

Step 2 – Redirect to resource's IP/STS

```
HTTP/1.1 302 Found ↵
Location:
https://sts.resource.com/sts?wa=wsignin1.0&wreply=https://res.resource.
com/sales&wct=2003-03-03T19:06:21Z
```

In addition, the resource could check for a previously written artifact/cookie and, if present, skip to Step 10.

Step 3 – GET resource challenge

```
GET https://sts.resource.com/sts?wa=wsignin1.0&wreply=
https://res.resource.com/sales&wct=2003-03-03T19:06:21Z HTTP/1.1
```

Step 3.1 – UI to determine realm (OPTIONAL)

```
[Implementation Specific Traffic]
```

Step 4 – Redirect to requestor's IP/STS

```
HTTP/1.1 302 Found ↵
Location: https://sts.account.com/sts?wa=wsignin1.0&wreply=
https://sts.resource.com/sts&wctx=
https://res.resource.com/sales&wct=2003-03-
03T19:06:22Z&wtrealm=resource.com
```

In addition, the Resource IP/STS may check for a previously written artifact/cookie and, if present, skip to Step 8.

Step 5 – Requestor IP/STS challenge

```
GET
https://sts.account.com/sts?wa=wsignin1.0&wreply=https://sts.resource.c
om/sts&wctx=https://res.resource.com/sales&wct=2003-03-
03T19:06:22Z&wtrealm=resource.com HTTP/1.1
```

Step 5.1 – UI to collect authentication data (OPTIONAL)

```
[Implementation Specific Traffic]
```

Step 6 – Return requestor token

```
HTTP/1.1 200 OK
...

<html xmlns="https://www.w3.org/1999/xhtml">
<head>
<title>Working...</title>
</head>
<body>
<form method="post" action="https://sts.resource.com/sts">
<p>
<input type="hidden" name="wa" value="wsignin1.0" />
<input type="hidden" name="wctx" value="https://res.resource.com/sales"
/>
```

```
<input type="hidden" name="wresult"
value="&lt;RequestSecurityTokenResponse&gt;...&lt;/RequestSecurityToken
Response&gt;" />
<button type="submit">POST</button> <!-- included for requestors that
do not support javascript -->
</p>
</form>
<script type="text/javascript">
setTimeout('document.forms[0].submit()', 0);
</script>
</body>
</html>
```

Step 7 – POST requestor token

```
POST https://sts.resource.com/sts HTTP/1.1 ↵
… ↵
↵
wa=wsignin1.0 ↵
wctx=https://res.resource.com/sales
wresult=<RequestSecurityTokenResponse>…</RequestSecurityTokenResponse>
```

Step 8 – Return resource token

```
HTTP/1.1 200 OK
…

<html xmlns="https://www.w3.org/1999/xhtml">
<head>
<title>Working...</title>
</head>
<body>
<form method="post" action="https://res.resource.com/sales">
<p>
<input type="hidden" name="wa" value="wsignin1.0" />
```

```
<input type="hidden" name="wresult"
value="&lt;RequestSecurityTokenResponse&gt;...&lt;/RequestSecurityToken
Response&gt;" />
<button type="submit">POST</button> <!-- included for requestors that
do not support javascript -->
</p>
</form>
<script type="text/javascript">
setTimeout('document.forms[0].submit()', 0);
</script>
</body>
</html>
```

Step 9 – POST Resource token

```
POST https://res.resource.com/sales HTTP/1.1 ↵
... ↵
↵
wa=wsignin1.0 ↵
wresult=<RequestSecurityTokenResponse>...</RequestSecurityTokenResponse
>
```

Step 10 – Return result

```
[Implementation Specific Traffic]
```