Secure Inter-Domain Routing
Internet-Draft
Intended status: Standards Track
Expires: October 2, 2013

M. Reynolds
IPSw
S. Kent
BBN
M. Lepinski
BBN

Apr 5, 2013

Local Trust Anchor Management for the Resource Public Key Infrastructure <draft-ietf-sidr-ltamgmt-08.txt>

Status of this Memo

Copyright and License Notice

Abstract

This document describes a facility to enable a relying party (RP) to manage trust anchors (TAs) in the context of the Resource Public Key Infrastructure (RPKI). It is common in RP software (not just in the RPKI) to allow an RP to import TA material in the form of self-signed certificates. However, this approach to incorporating TAs is potentially dangerous. (These self-signed certificates rarely incorporate any extensions that impose constraints on the scope of the imported public keys, and the RP is not able to impose such constraints.) The facility described in this document allows an RP to impose constraints on such TAs. Because this mechanism is designed to operate in the RPKI context, the most important constraints are the Internet Number Resources (INRs) expressed via RFC 3779 extensions. These extentions bind address spaces and/or autonomous system (AS) numbers to entities. The primary motivation for the facility described in this document is to enable an RP to ensure that INR information that it has acquired via some trusted channel is not overridden by the information acquired from the RPKI repository system or by the putative TAs that the RP imports. Specifically, the mechanism allows an RP to specify a set of overriding bindings between public key identifiers and INR data. These bindings take precedence over any conflicting bindings acquired by the putative TAs and the certificates downloaded from the RPKI repository system. This mechanism is designed for local use by an RP, but any entity that is accorded administrative control over a set of RPs may use this mechanism to convey its view of the RPKI to RPs within its jurisdiction. The means by which this latter use case is effected is outside the scope of this document.

Table of Contents

1 Introduction

The Resource Public Key Infrastructure (RPKI) [RFC6480] is a PKI in which certificates are issued to facilitate management of Internet Resource Numbers (INRs). Such resources are expressed in the form of X.509v3 "resource" certificates with extensions defined by RFC 3779 [RFC6487]. Validation of a resource certificate is preceded by path discovery. In a PKI path discovery is effected by constructing a certificate path between a target certificate and a trust anchor (TA). No IETF standards define how to construct a certificate path; commonly such paths are based on a bottom-up search using Subject/Issuer name matching, but top-down and meet-in-the-middle approaches may also be employed [RFC4158]. In contrast, path validation is top-down, as defined by [RFC5280].

In the RPKI, certificates can be acquired in various ways, but the default is a top-down tree walk as described in [RFC6481], initialized via a Trust Anchor Locator [RFC6490]. Note that the process described there is not path discovery per sem but the collecting of certificates to populate a local cache. Thus, the common, bottom-up path discovery approach is not inconsistent with these RFCs. Morevoer, a bottom-up path discovery approach is more general, accommodating certificates that might be acquired by other means, i.e., not from an RPKI repository. There are circumstances under which an RP may wish to override the INR specifications obtained through the RPKI distributed repository system [RFC6481]. This document describes a mechanism by which an RP may override any conflicting information expressed via putative TAs and the certificates downloaded from the RPKI repository system. Thus the algorithms described in this document adopt a bottom-up path discovery approach.

To effect this local control, this document calls for a relying party to specify a set of bindings between public key identifiers and INRs through a text file known as a constraints file. The constraints expressed in this file then take precedence over any competing claims expressed by resource certificates acquired from the distributed repository system. (The means by which a relying party acquires the key identifier and the RFC 3779 extension data used to populate the constraints file is outside the scope of this document.) The relying party also may use a local publication point (the root of a local directory tree that is made available as if it were a remote repository) as a source of certificates and CRLs (and other RPKI signed objects, e.g., ROAs and manifests) that do not appear in the RPKI repository system.

In order to allow reuse of existing, standard path validation mechanisms, the RP-imposed constraints are realized by having the RP itself represented as the only TA known in the local certificate validation context. To ensure that all RPKI certificates can be validated relative to this TA, this RP TA certificate must contain

all-encompassing resource allocations, i.e. 0/0 for IPv4, 0::/0 for IPv6 and 0-4294967295 for AS numbers. Thus, a conforming implementation of this mechanism must be able to cause a self-signed certification authority (CA) certificate to be created with a locally generated key pair. It also must be able to issue CA certificates subordinate to this TA. Finally, a conforming implementation of this

mechanism must process the constraints file and modify certificates as needed in order to enforce the constraints asserted in the file.

The remainder of this document describes in detail the types of certificate modification that may occur, the syntax and semantics of the constraints file, and the implications of certificate modification on path discovery and revocation.

1.1 Terminology

It is assumed that the reader is familiar with the terms and concepts described in "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile" [RFC5280] and "X.509 Extensions for IP Addresses and AS Identifiers" [RFC3779].

The key words "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.

2 Overview of Certificate Processing

The fundamental aspect of the facility described in this document is one of certificate modification. The constraints file, described in more detail in the next section, contains assertions about INRs that are to be specially processed. As a result of this processing, certificates in the local copy of the RPKI repository are transformed into new certificates satisfying the INR constraints so specified. This enables the RP to override conflicting assertions about resource holdings as acquired from the RPKI repository system. Three forms of certificate modification can occur. (Every certificate is digitally signed and thus cannot be modified without "breaking" its signature. In the context of this document we assume that certificates that are modified have been validated previously. Thus the content can be modified, locally, without the need to preserve the integrity of the signature. These modified certificates are referred to as paracertificates (see section 2.4 below).)

2.1 Target Certificate Processing

If a certificate is acquired from the RPKI repository system and its Subject key identifier (SKI) is listed in the constraints file, it will be reissued directly under the RP TA certificate, with (possibly) modified RFC 3779 extensions. (The SKI is used as a compact reference to the public key in a target certificate.) The modified extensions will include any RFC 3779 data expressed in the constraints file. Other certificate fields may also be modified to maintain consistency. (These fields are enumerated in Table 1, and discussed in Section 3.3.) In Section 4.2, target certificate processing corresponds to stage one of the algorithm. (When a target certificate is re-parented, all subordinate signed products will still be valid, unless the set of INRs in the targeted certificate is reduced.)

2.2 Perforation

When a target certificate is re-issued directly under the RP's TA, its INRs MUST be removed from all of its parent (CA) certificates. (If these INRs were not removed, then conflicting assertions about INRs could arise and undermine the authority of the RP TA.) Thus, every certificate acquired from the RPKI repository MUST be examined to determine if it contains an RFC 3779 extension that intersects the resource data in the constraints file. If there is an intersection the certificate will be reissued directly under the RP TA, with modified RFC 3779 extensions. We refer to the process of modifying the RFC 3779 extension in an affected certificate as "perforation" (because the process will create "holes" in these extensions). The

modified extensions will exclude any RFC 3779 data expressed in the constraints file. In the certificate processing algorithm described in Section 4.2, perforation corresponds to stage two of the algorithm ("ancestor processing") and also to stage three of the algorithm ("tree processing").

2.3 TA Re-parenting

All valid, self-signed certificates offered as TAs in the public RPKI certificate hierarchy, e.g., self-signed certificates issued by IANA or RIRs, will be re-issued under the RP TA certificate. This processing is done even though all but one of these certificates might not intersect any resources specified in the constraints file. We refer to this reissuance as "re-parenting" since the issuer (parent) of the certificate has been changed. The issuer name is changed from that of the certificate subject (this is a self-signed certificate) to that of the RP TA. In the certificate processing algorithm described in Section 4.2, TA re-parenting corresponds to stage four of the algorithm. (In a more generic PKI context, re-parenting enables an RP to insert extensions in these certificates to impose constraings on path processing in a fashion consistent with RFC 5280. In this fashion an RP can impose name constraints, policy constraints, etc.)

2.4 Paracertificates

If a certificate is subject to any of the three forms of processing just described, that certificate will be referred to as an "original" certificate and the processed (output) certificate will be referred to as a paracertificate. When an original certificate is transformed into a paracertificate all the fields and extensions from the original certificate will be retained, except as indicated in Table 1, below.

Original Certificate Field	Action
Version	unchanged
Serial number	created per note A
Signature	replaced if needed with RP's signing alg
Issuer	replaced with RP's name
Validity dates	replaced per note B
Subject	unchanged
Subject public key info	unchanged
Extensions	
Subject key identifier	unchanged
Key usage	unchanged
Basic constraints	unchanged
CRL distribution points	replaced per note B
Certificate policy	replaced per note B
Authority info access	replaced per note B
Authority key ident	replaced with RP's
IP address block	modified as described
AS number block	modified as described
Subject info access	unchanged
All other extensions	unchanged
Signature Algorithm	same as above
Signature value	new

Table 1 Certificate Field Modifications

Note A. The serial number will be created by concatenating the current time (the number of seconds since Jan 1, 1970) with a count of the certificates created in the current run. Because all paracertificates are issued directly below the RP TA, this algorithm ensures serial number uniqueness.

Note B. These fields are derived (as described in Section 3.3 below) from parameters in the constraints file (if present); otherwise, they take on values from the certificates from which the paracertificates are derived.

3 Format of the constraints file

This section describes the syntax of the constraints file. (The syntax has been defined to enable creation and distribution of constraint files to a set of RPs, by an authorized third party.) The model described below is nominal; implementations need not match all details of this model as presented, but the external behavior of implementations MUST correspond to the externally observable characteristics of this model in order to be compliant. It is RECOMMENDED that the syntax described herein be supported, to facilitate interoperability between creators and comsumers of constraints files.

The constraints file consists of four logical subsections: the replying party subsection, the flags subsection, the tags subsection and the blocks subsection. The relying party subsection and the blocks subsection are REQUIRED and MUST be present; the flags and tags subsections are OPTIONAL. Each subsection is described in more detail below. Note that the semicolon (;) character acts as the comment character, to enable annotating constraints files. All characters from a semicolon to the end of that line are ignored. In addition, lines consisting only of whitespace are ignored. The subsections MUST occur in the order indicated. An example constraints file is given in Appendix A.

3.1 Relying party subsection

The relying party subsection is a REQUIRED subsection of the constraints file. It MUST be the first subsection of the constraints file, and it MUST consist of two lines of the form: (RECOMMENDED)

```
PRIVATEKEYMETHOD      value [ ... value ]
TACERTIFICATE         value
```

The first line provides a pointer (including an access method) to the RP's private key. This line consists of the string literal PRIVATEKEYMETHOD, followed by one or more whitespace delimited string values. These values are passed to the certificate processing algorithm as described below. Note that this entry, as for all entries in the constraints file, is case sensitive.

The second line of this subsection consists of the string literal TACERTIFICATE, followed by exactly one string value. This value is the name of a file containing the relying party's TA certificate. The file name is passed to the certificate processing algorithm as described below.

3.2 Flags subsection

The flags subsection of the constraints file is an OPTIONAL

subsection. If present it MUST immediately follow the relying party

subsection. The flags subsection consists of one or more lines of the form

```
CONTROL  flagname  booleanvalue
```

Each such line is referred to as a control line. Each control line MUST contain exactly three whitespace delimited strings. The first string MUST be the literal CONTROL. The second string MUST be one of the following three literals:

```
resource_nounion
intersection_always
treegrowth
```

The third string denotes a Boolean value, and MUST be one of the literals TRUE or FALSE. Control flags influence the global operation of the certificate processing algorithm; the semantics of the flags is described in Section 4.2. Note that each flag has a default value, so that if the corresponding CONTROL line does not appear in the constraints file, the algorithm flag is considered to take the corresponding default value. The default value for each flag is FALSE. Thus, if any flag is not named in a control line it takes the value FALSE. If the flags subsection is absent, all three flags assume the default value FALSE.

3.3 Tags subsection

The tags subsection is an OPTIONAL subsection in the constraints file. If present it MUST immediately follow the relying party subsection (if the flags subsection is absent) or the flags subsection (if it is present). The tags subsection consists of one or more lines of the form

```
TAG  tagname  tagvalue [ ... tagvalue ]
```

Each such line is referred to as a tag line. Each tag line MUST consist of at least three whitespace delimited string values, the first of which must be the literal TAG. The second string value gives the name of the tag, and subsequent string(s) give the value(s) of the tag. The tag name MUST be one of the following four string literals:

```
Xvalidity_dates
Xcrldp
Xcp
Xaia
```

The purpose of the tag lines is to provide an indication of the means

by which paracertificate fields, specifically those indicated above under "Note B", of Table 1are constructed. Each tag has a default, so that if the corresponding tag line is not present in the constraints file, the default behavior is used when constructing the paracertificates. The syntax and semantics of each tag line is described next.

Note that the tag lines are considered to be global; the action of each tag line (or the default action, if that tag line is not present) applies to all paracertificates that are created as part of the certificate processing algorithm.

3.3.1 Xvalidity_dates tag

This tag line is used to control the value of the notBefore and notAfter fields in paracertificates. If this tag line is specified and there is a single tagvalue which is the literal string C, the paracertificate validity interval is copied from the original certificate validity interval from which it is derived. If this tag is specified and there is a single tagvalue which is the literal string R, the paracertificate validity interval is copied from the validity interval of the RP's TA certificate. If this tag is specified and the tagvalue is neither of these literals, then exactly two tagvalues MUST be specified. Each must be a Generalized Time string of the form YYYYMMDDHHMMSSZ. The first tagvalue is assigned to the notBefore field and the second tagvalue is assigned to the notAfter field. It MUST be the case that the tagvalues can be parsed as valid Generalized Time strings such that notBefore is less than notAfter, and also such that notAfter represents a time in the future (i.e., the paracertificate has not already expired).

If this tag line is not present in the constraints file the default behavior is to copy the validity interval from the original certificate to the corresponding paracertificate.

3.3.2 Xcrldp tag

This tag line is used to control the value of the CRL distribution point extension in paracertificates. If this tag line is specified and there is a single tagvalue that is the string literal C, the CRLDP of the paracertificate is copied from the CRLDP of the original certificate from which it is derived. If this tag line is specified and there is a single tagvalue that is the string literal R, the CRLDP of the paracertificate is copied from the CRLDP of the RP's TA certificate. If this tag line is specified and there is a single tagvalue that is not one of these two reserved literals, or if there is more than one tagvalue, then each tagvalue is interpreted as a URI that will be placed in the CRLDP sequence in the

paracertificate.

If this tag line is not present in the constraints file the default behavior is to copy the CRLDP from the original certificate into the corresponding paracertificate.

3.3.3 Xcp tag

This tag line is used to control the value of the policyQualifierId field in paracertificates. If this tag line is specified there MUST be exactly one tagvalue. If the tagvalue is the string literal C, the paracertificate value is copied from the value in the corresponding original certificate. If the tagvalue is the string literal R, the paracertificate value is copied from the value in the RP's top level TA certificate. If the tagvalue is the string literal D, the paracertificate value is set to the default OID. If the tagvalue is not one of these reserved string literals, then the tagvalue MUST be an OID specified using the standard dotted notation. The value in the paracertificate's policyQualifierId field is set to this OID. Note the RFC 5280 specifies that only a single policy may be specified in a certificate, so only a single tagvalue is permitted in this tag line, even though the CertificatePolicy field is an ASN.1 sequence.

If this tag line is not specified the default behavior is to use the default OID in creating the paracertificate.

This option permits the RP to convert a value of the policyQualifierId field in a certificate (that would not be in conformance with the RPKI CP) to a conforming value in the paracertificate. This conversion enables use of RPKI validation software that checks the policy field against that specified in the RPKI CP [RFC6484].

3.3.4 Xaia tag

This tag line is used to control the value of the Authority Information Access (AIA) extension in the paracertificate. If this tag line is present then it MUST have exactly one tagvalue. If this tagvalue is the string literal C, then the AIA field in the paracertificate is copied from the AIA field in the original certificate from which it is derived. If this tag line is present and the tagvalue is not the reserved string literal, then the tagvalue MUST be a URI. This URI is set as the AIA extension of the paracertificates that are created.

If this tag line is not specified the default behavior is to use copy the AIA field from the original certificate to the AIA field of the paracertificate.

3.4 Blocks subsection

The blocks subsection is a REQUIRED subsection of the constraints file. If the tags subsection is present, the blocks subsection MUST appear immediately after it. This MUST be the last subsection in the constraints file. The blocks subsection consists of one or more blocks, known as target blocks. A target block is used to specify an association between a certificate (identified by an SKI) and a set of resource assertions. Each target block contains four regions, an SKI region, an IPv4 region, an IPv6 region and an AS number region. All regions MUST be present in a target block.

The SKI region contains a single line beginning with the string literal SKI and followed by forty hexadecimal characters giving the subject key identifier of a certificate, known as the target certificate. The hex character string MAY contain embedded whitespace or colon characters (included to improve readability), which are ignored. The IPv4 region consists of a line containing only the string literal IPv4. This line is followed by zero or more lines containing IPv4 prefixes in the format described in RFC 3779. The IPv6 region consists of a line containing only the string literal IPv6, followed by zero or more lines containing IPv6 prefixes using the format described in RFC 3513. (The presence of the IPv4 and IPv6 literals is to simplify parsing of the constraints file.) Finally, the AS number region consists of a line containing only the string literal AS#, followed by zero or more lines containing AS numbers (one per line). The AS numbers are specified in decimal notation as recommended in RFC 5396. A target block is terminated by either the end of the constraints file, or by the beginning of the next target block, as signaled by its opening SKI region line. An example target block is shown below. (The indentation used below is employed to improve readability and is not required.) See also the complete constraints file example in Appendix A. Note that whitespace, as always, is ignored.

```
SKI 00:12:33:44:00:BA:BA:DE:EB:EE:00:99:88:77:66:55:44:33:22:11
IPv4
  10.2.3/24
  10.8/16
IPv6
  1:2:3:4:5:6/112
AS#
  123
  567
```

The blocks subsection MUST contain at least one target block. Note that it is OPTIONAL that the SKI refer to a certificate that is known

or resolvable within the context of the local RPKI repository. Also, there is no REQUIRED or implied ordering of target blocks within the block subsection. Since blocks may occur in any order, the outcome of processing a constraints file may depend on the order in which target blocks occur within the constraints file. The next section of this document contains a detailed description of the certificate processing algorithm.

4 Certificate Processing Algorithm

The section describes the certificate processing algorithm by which paracertificates are created from original certificates in the local RPKI repository. For the purposes of describing this algorithm, it will be assumed that certificates are persistently associated with state (or metadata) information. This state information is nominally represented by an array of named bits associated with each certificate. No specific implementation of this functionality is mandated by this document. Any implementation that provides the indicated functionality is acceptable, and need not actually consist of a bit field associated with each certificate.

The following state bits used in certificate processing are

NOCHAIN
ORIGINAL
PARA
TARGET

If the NOCHAIN bit is set, this indicates that a full path between the given certificate and a TA has not yet been discovered. If the ORIGINAL bit is set, this indicates that the certificate in question has been processed by some part of the processing algorithm described in Section 4.2. If it was processed as part of stage one processing, as described in section 4.2.2, the TARGET bit also will be set. Finally, every paracertificate will have the PARA bit set.

At the beginning of algorithm processing each certificate in the local RPKI repository has the ORIGINAL, PARA and TARGET bits clear. If a certificate has a complete, validated path to a TA, or is itself a TA, then that certificate will have the NOCHAIN bit clear, otherwise it will have the NOCHAIN bit set. As the certificate processing algorithm proceeds, the metadata state of original certificates may change. In addition, since the certificate processing algorithm may also be creating paracertificates, it is responsible for actively setting or clearing the state of these four bits on those paracertificates.

The certificate processing algorithm consists of two sub-algorithms:

"proofreading" and "TA processing". Conceptually, the proofreading algorithm performs syntactic checks on the constraints file, while the TA processing algorithm performs the actual certificate transformation processing. If the proofreading algorithm does not succeed in parsing the constraints file, the TA processing-algorithm is not executed. Note also that if the constraints file is not present, neither algorithm is executed and the local RPKI repository is not modified. Each of the constituent algorithms will now be described in detail.

4.1 Proofreading algorithm

The proofreading algorithm checks the constraints file for syntactic errors, e.g., missing REQUIRED subsections, or malformed addresses. Implementation of this algorithm is OPTIONAL. If it is implemented, the following text defines correct operation for the algorithm. The proofreading algorithms performs a set of heuristic checks, such as checking for prefixes that are too large (e.g., larger than /8). The proofreading algorithm also SHOULD examine resource regions (IPv4, IPv6 and AS# regions) within the blocks subsection, and reorder such resources within a region in ascending numeric order. On encountering any error the proofreading algorithm SHOULD provide an error message indicating the line on which the error occurred as well as informative text that is sufficiently descriptive as to allow the user to identify and correct the error. An implementation of the proofreading algorithm MUST NOT assume that it has access to the local RPKI repository (even read-only access). An implementation of the proofreading algorithm MUST NOT alter the local RPKI repository in any way; it also MUST NOT change any of the metadata associated with certificates in that repository. (Recall that the processing described here is creating a copy of that local repository.) For simplicity the remainder of this document assumes that the proofreading algorithm produces a transformed output file. This file contains the same syntactic information as the text version of the constraints file.

The proofreading algorithm performs the following syntactic checks on the constraints file:

- verifies the presence of the REQUIRED relying party subsection and the REQUIRED blocks subsection.
- verifies the order of the two, three or four subsections as stated above.
- verifies that the relying party subsection conforms to the specification given in Section 3.1 above.
- verifies that, if present, the tags and flags subsections conform to the specifications in Sections 3.2 and 3.3 above.

After these checks have been performed, the proofreading algorithm then checks the blocks subsection:

- splits the blocks subsection into constituent target blocks, as delimited by the SKI region line(s)
- verifies that at least one target block is present

- verifies that each SKI region line contains exactly forty hexadecimal digits and contains no additional characters other than whitespace or colon characters.

For each target the proofreading algorithm:

- verifies the presence of the IPv4, IPv6 and AS# regions, and verifies that at least one such resource is present.
- verifies that, for each IPv4 prefix, IPv6 prefix and autonomous system number given, that the indicated resource is syntactically valid according to the appropriate RFC definition, as described in Section 3.4.
- verifies that no IPv4 resource has a prefix larger than /8.
- optionally performing reordering within each of the three resource regions so that stated resources occur in ascending numerical order.

(If the proofreading algorithm has performed any reordering of information it MAY overwrite the constraints file. If it does so, however, it MUST preserve all information contained within the file, including information that is not parsed (such as comments). If the proofreading algorithm has performed any reordering of information but has not overwritten the constraints file, it MAY produce a transformed output file, as described above. If the proofreading algorithm has performed any reordering of information, but has neither overwritten the constraints file nor produced a transformed output file, it MUST provide an error message to the user indicating what reordering was performed.)

4.2 TA processing algorithm

The TA processing algorithm acts on the constraints file (as processed by the proofreading algorithm) and the contents of the local RPKI repository to produce paracertificates for the purpose of enforcing the resource allocations as expressed in the constraints file. The TA processing algorithm operates in five stages, a preparatory stage (stage 0), target processing (stage 1), ancestor processing (stage 2), tree processing (state 3) and TA re-parenting (stage 4). Conceptually, during the preparatory stage the proofreader output file is read and a set of internal RP, tag and flag variables are set based on the contents of that file. (If the constraint file has not specified one or more of the tags and/or flags, those tags and flags are set to default values.) During target processing all certificates specified by a target block are processed, and the resources for those certificates are (potentially) expanded; for each target found a new paracertificate is manufactured with its various fields set, as shown in Table 1, using the values of the internal variables set in the preparatory stage and also, of course, the fields of the original certificate (and, potentially, fields of the RP's TA certificate). In stage 2 (ancestor) processing, all ancestors of the each target certificate are found, and the claimed resources are then removed (perforated). A new paracertificate with these diminished resources is crafted, with its fields generated based on internal variable settings, original certificate field values, and, potentially, the fields of the RP's TA certificate. In tree processing (stage 3), the

entire local RPKI repository is searching for any other certificates that have resources that intersect a target resource, and that were not otherwise processed during a preceding stage. Perforation is again performed for any such intersecting certificates, and paracertificates created as in stage 2. In the fourth (last) stage, TA re-parenting, any TA certificates in the local RPKI repository that have not already been processed are now re-parented under the RP's TA certificate. This transformation creates paracertificates; however, these paracertificates may have RFC 3779 resources that were not altered during algorithm processing. The final output of algorithm processing will be threefold:

- the metadata information on some (original) certificates in the repository MAY be altered.
- paracertificates will be created, with the appropriate metadata, and entered into the repository.
- the TA processing algorithm SHOULD produce a human readable log of its actions, indicating which paracertificates were created and why. The remainder of this section describes the processing stages of the algorithm in detail.

4.2.1 Preparatory processing (stage 0)

During preparatory processing, the output of the proofreader algorithm, is read. Internal variables are set corresponding to each tag and flag, if present, or to their defaults, if absent. Internal variables are set corresponding to the PRIVATEKEYMETHOD value string(s) and the TACERTIFICATE string. The TA processing algorithm is queried to determine if it supports the indicated private key access methodology. This query is performed in an implementation-specific manner. In particular, an implementation is free to vacuously return success to this query. The TA processing algorithm next uses the value string for the TACERTIFICATE to locate this certificate, again in an implementation-specific manner. The certificate in question may already be present in the local RPKI repository, or it may be located elsewhere. The implementation is free to create the top level certificate at this time, and then assign to this newly-created certificate the name indicated. It is necessary only that, at the conclusion of this processing, a valid trust anchor certificate for the relying party has been created or otherwise obtained.

Some form of access to the RP's private key and top level certificate are required for subsequent correct operation of the algorithm. Therefore, stage 0 processing MUST terminate if one or both conditions are not satisfied. In the error case, the implementation SHOULD provide an error message of sufficient detail that the user can correct the error(s). If stage 0 processing does not succeed, no further stages of TA processing are executed.

4.2.2 Target processing (stage 1)

During target processing, the TA processing algorithm reads all target blocks in the proofreader output file. It then processes each target block in the order specified in the file. In the description that follows, except where noted, the operation of the algorithm on a single target block will be described. Note, however, that all stage 1 processing is executed before any processing in subsequent stages is performed.

The algorithm first obtains the SKI region of the target block. It then locates (in an implementation-dependent manner) the certificate identified by the SKI. Note that this search is performed only against (original) certificates, not against paracertificates. If more than one original certificate is found matching this SKI, there are two possible scenarios. If a resource holder has two certificates issued by the same CA, with overlapping validity intervals and the same key, but distinct subject names (typically, by virtue of the SerialNumber parts being different), then these two certificates are both considered to be (distinct) targets, and are both processed. If, however, a resource holder has certificates issued by two different CAs, containing different resources, but using the same key, there is no unambiguous method to decide which of the certificates is intended as the target. In this latter case the algorithm MUST issue a warning to that effect, mark the target block in question as unavailable for processing by subsequent stages and proceed to the next target block. If no certificate is found then the algorithm SHOULD issue a warning to that effect and proceed to process the next target block.

If a single (original) certificate is found matching the indicated SKI, then the algorithm takes the following actions. First, it sets the ORIGINAL state bit for the certificate found. Second, it sets the TARGET state bit for the certificate found. Third, it extracts the INRs from the certificate. If the global resource_nounion flag is TRUE, the algorithm compares the extracted certificate INRs with the INRs specified in the constraints file. If the two resource sets are different, the algorithm SHOULD issue a warning noting the difference. An output resource set is then formed that is identical to the resource set extracted from the certificate. If, however, the resource_nounion flag is FALSE, then the output resource set is calculated by forming the union of the resources extracted from the certificate and the resources specified for this target block in the constraints file. A paracertificate is then constructed according to Table 1, using fields from the original certificate, the tags that had been set during

stage 0, and, if necessary, fields from the RP's TA certificate. The INR resources of the paracertificate are equated to the derived output resource set. The PARA state bit is set for the newly created paracertificate.

4.2.3 Ancestor processing (stage 2)

The goal of ancestor processing is to discover all ancestors of a target certificate and remove from those ancestors the resources specified in the target blocks corresponding to the targets being processed. Note that it is possible that, for a given chain from a target certificate to a trust anchor, another target might be encountered. This is handled by removing all the target resources of all descendants. The set of all targets that are descendants of the given certificate is formed. The union of all the target resources of the corresponding target blocks is computed, and this union in then removed from the shared ancestor.

In detail, the algorithm is as follows. First, all (original) target certificates processed during stage 1 processing are collected. Second, any collected certificates that have the NOCHAIN state bit set are eliminated from the collection. (Note that, as a result of eliminating such certificates, the resulting collection may be empty, in which case this stage of algorithm processing terminates, and processing advances to stage 3.) Next, an implementation MAY sort the collection. The optional sorting algorithm is described in Appendix B. Note that all stage 2 processing is completed before any stage 3 processing.

Two levels of nested iteration are performed. The outer iteration is effected over all certificates in the collection; the inner iteration is over all ancestors of the designated certificate being processed. The first certificate in the collection is chosen, and a resource set R is initialized based on the resources of the target block for that certificate (since the certificate is in the collection, it must be a target certificate, and thus correspond to a target block). The parent of the certificate is then located using ordinary path discovery over original certificates only. The ancestor's certificate resources A are then extracted. These resources are then perforated with respect to R. That is, an output set of resources is created by forming the intersection I of A and R, and then taking the set difference A - I as the output resources. A paracertificate is then created containing resources that are these output resources, and containing other fields and extensions from the original certificate (and possibly the RP's TA certificate) according to the procedure given in Table 1. The PARA state bit is set on this paracertificate and the ORIGINAL state bit is set on A. If A is also a target certificate, as indicated by its TARGET state bit being set, then

there will already have been a paracertificate created for it. This previous paracertificate is destroyed in favor of the newly created paracertificate. In this case also, the set R is augmented by adding into it the set of resources of the target block for A. The algorithm then proceeds to process the parent of A. This inner iteration continues until the self-signed certificate at the root of the path is encountered and processed. The outer iteration then continues by clearing R and proceeding to the next certificate in the target collection.

Note that ancestor processing has the potential for order dependency, as mentioned earlier in this document. If sorting is not implemented, or if the sorting algorithm fails to completely process the collection of target certificates because the allotted maximum number of iterations has been realized, it may be the case that an ancestor of a certificate logically occurs before that certificate in the collection. Whenever an existing paracertificate is replaced by a newly created paracertificate during ancestor processing, the algorithm SHOULD alert the user, and SHOULD log sufficient detail such that the user is able to determine which resources were perforated from the original certificate in order to create the (new) paracertificate.

In addition, implementations MUST provide for conflict detection and notification during ancestor processing. During ancestor processing a certificate may be encountered two or more times and the modifications dictated by the ancestor processing algorithm may be in conflict. If this situation arises the algorithm MUST refrain from processing that certificate. Further, the implementation MUST present the user with an error message that contains enough detail so that the user can locate those directives in the constraints file that are creating the conflict. For example, during one stage of the processing algorithm it may be directed that resources R1 be added to a certificate C, while during a different stage of the processing algorithm it may be directed that resources R2 be removed from certificate C. If the resource sets R1 and R2 have a non-empty intersection, that is a conflict.

4.2.4 Tree processing (stage 3)

The goal of tree processing is to locate other certificates containing INRs that conflict with the resources allocated to a target, by virtue of the INRs specified in the constraints file. The certificates processed are not ancestors of any target. The algorithm used is described below.

First, all target certificates are collected. Second, all target certificates that have the NOCHAIN state bit set are eliminated from this collection. Third, if the intersection_always

global flag is set, target blocks that occur in the constraints file, but that did not correspond to a certificate in the local repository, are added to the collection. In tree processing, unlike ancestor processing, this collection is not sorted. An iteration is now performed over each certificate (or set of target block resources) in the collection. Note that the collection may be empty, in which case this stage of algorithm processing terminates, and processing advances to stage 4. Note also that all stage 3 processing is performed before any stage 4 processing.

Given a certificate or target resource block, each top level original TA certificate is examined. If that TA certificate has an intersection with the target block resources, then the certificate is perforated with respect to those resources. A paracertificate is created based on the contents of the original certificate (and possibly the RP's TA certificate, as indicated in Table 1) using the perforated resources. The ORIGINAL state bit is set on the original certificate processed in this manner, and the PARA state bit is set on the paracertificate just created. An inner iteration then begins on the descendants of the original certificate just processed. There are two ways in which this iteration may proceed. If the treegrowth global flag is clear, then examination of the children proceeds until all children are exhausted, or until one child is found with intersecting resources. If the treegrowth global flag is set, all children are examined. If a transfer of resources is in process, more than one child may possess intersecting resources. In this case, it is RECOMMENDED that the treegrowth flag be set. The inner iteration proceeds until all descendants have been examined and no further intersecting resources are found. The outer iteration then continues with the next certificate or target resource block in the collection. Note that unlike ancestor processing, there is no concept of a potentially cumulating resource collection R; only the resources in the target block are used for perforation.

4.2.5 TA re-parenting (stage 4)

In the final stage of TA algorithm processing, all TA certificates (other than the RP's TA certificate) that have not already been processed are now processed. At this stage all unprocessed TA certificates have no intersection with any target resource blocks. As such, in creating the corresponding paracertificates, the output resource set is identical to the input resource set. Other transformations as described in Table 1 are performed. The original TA certificates have the ORIGINAL state bit set; the newly created paracertificates have the PARA state bit set. Note that once stage four processing is completely, only a single TA certificate will remain in an unprocessed state, namely the relying party's own TA certificate.

4.3 Discussion

The algorithm described in this document effectively creates two coexisting certificate hierarchies: the original certificate hierarchy and the paracertificate hierarchy. Original certificates are not removed during any of the processing described in the previous section. Some original certificates may move from having no state bits set (or only the NOCHAIN state bit set) to having one or both of the ORIGINAL and TARGET state bits set. In addition, the NOCHAIN state bit will still be set if it was set before any processing. The paracertificate hierarchy, however, is intended to supersede the original hierarchy for ROA validation. The presence of two hierarchies has implications for path discovery, and for revocation.

If one thinks of a certificate as being "named" by its SKI, then there can now be two certificates with the same name, an original certificate and a paracertificate. The next two sections discuss the implications of this duality in detail. Before proceeding, it is worth noting that even without the existence of the paracertificate hierarchy, cases may exist in which two or more original certificates have the same SKI. As noted earlier, in Section 4.2.2, these cases may be subdivided into the case in which such certificates are distinguishable by virtue of having different subject names, but identical issuers and resource sets, versus all other cases. In the distinguishable case, the path discovery algorithm treats the original certificates as separate certificates, and processes them separately. In all other cases, the original certificates should be treated as indistinguishable, and path validation should fail.

5 Implications for Path Discovery

Path discovery proceeds from a child certificate C by asking for a parent certificate P such that the AKI of C is equal to the SKI of P. With one hierarchy this question would produce at most one answer. With two hierarchies, the original certificate hierarchy and the paracertificate hierarchy, the question may produce two answers, one answer, or no answer. Each of these cases is considered in turn.

5.1 Two answers

If two paths are discovered, it SHOULD be the case that one of the matches is a certificate with the ORIGINAL state bit set and the PARA state bit clear, while the other match inversely has the ORIGINAL state bit clear and the PARA state bit set. If any other combination of ORIGINAL and PARA state bits obtains, the path discovery algorithm MUST alert the user. In addition, the path discovery algorithm SHOULD refrain from attempting to make a

choice as to which of the two certificates is the putative parent. In the no-error case, with the state bits are as indicated, the certificate with the PARA state bit set is chosen as the parent P. Note this means, in effect, that all children of the original certificate have been re-parented under the paracertificate.

5.2 One answer

If the matching certificate has neither the ORIGINAL state bit set nor the PARA state bit set, this certificate is the parent. If the matching certificate has the PARA state bit set but the ORIGINAL state bit not set, this certificate is the parent. (This situation would arise, for example, if the original certificate had been revoked by its issuer but the paracertificate had not been revoked by the RP.) If the matching certificate has the ORIGINAL state bit set but the PARA state bit not set, this is not an error but it is a situation in which path discovery MUST be forced to fail. The parent P MUST be set to NULL, and the NOCHAIN state bit must be set on C and all its descendants; the user SHOULD be warned. Even if the RP has revoked the paracertificate, the original certificate MAY persist. Forcing path discovery to unsuccessfully terminate is a reflection of the RP's preference for path discovery to fail as opposed to using the original hierarchy. Finally, if the matching certificate has both the ORIGINAL and PARA state bits set, this is an error. The parent P MUST be set to NULL, and the user MUST be warned.

5.3 No answer

This situation occurs when C has no parent in either the original hierarchy or the paracertificate hierarchy. In this case the parent P is NULL and path discovery terminates unsuccessfully. The NOCHAIN state bit must be set on C and all its descendants.

6 Implications for Revocation

In a standard implementation of revocation in a PKI, a valid CRL names a (sibling) certificate by serial number. That certificate is revoked and is purged from the local RPKI repository. The original certificate hierarchy and the paracertificate hierarchy created by applying the algorithms described above are closely related. It can thus be asked how revocation is handled in the presence of these two hierarchies. In particular do changes in one of the hierarchies trigger corresponding changes in the other hierarchy. There are four cases based on the state of the ORIGINAL and PARA bits. These are discussed in the subsections below. It should be noted that the existence of two hierarchies presents a particular challenge with respect to revocation. If a CRL arrives and is processed, that

processing can result in the descrution of one of the path chains. In the case of a single hierarchy this would mean that certain objects would fail to validate. In the presence of two hierarchies, however, a CRL revocation may force the preferred path to be destroyed. If the RP later determines that the CRL revocation should not have occurred, he is faced with an undesirable situation: the deprecated path will be discovered. In order to prevent this outcome, an RP MUST be able to configure one or more additional repository URIs in support of local trust anchor management.

6.1 No state bits set

If the CRL names a certificate that has neither the ORIGINAL state bit set nor the PARA state bit set, revocation proceeds normally. All children of the revoked certificate have their state modified so that the NOCHAIN state bit is set.

6.2 ORIGINAL state bit set

If the CRL names a certificate with the ORIGINAL state bit set and the PARA state bit clear, then this certificate is revoked as usual. If this original certificate also has the TARGET state bit set, then the corresponding paracertificate (if it exists) is not revoked; if this original certificate has the TARGET state bit clear, then the corresponding paracertificate is revoked as well. Note that since all the children of the original certificate have been re-parented to be children of the corresponding paracertificate, as described above, the revocation algorithm MUST NOT set the NOCHAIN state bit on these children unless the paracertificate is also revoked. Note also that if the original certificate is revoked but the paracertificate is not revoked, the paracertificate retains its PARA state bit. This is to ensure that path discovery proceeds preferentially through the paracertificate hierarchy, as described above.

6.3 PARA state bit set

If the CRL names a certificate with the PARA state bit set and the ORIGINAL state bit clear, this CRL must have been issued, perforce, by the RP itself. This is because all the paracertificates are children of the RP's TA certificate. (Recall that a TA is not revoked via a CRL; it is merely removed from the repository.) The paracertificate is revoked and all children of the paracertificate have the NOCHAIN state bit set. No action is taken on the corresponding original certificate; in particular, its ORIGINAL state bit is not cleared.

Note that the serial numbers of paracertificates are synthesized according to the procedure given in Table 1, rather than being assigned by an algorithm under the control of the (original) issuer.

6.4 Both ORIGINAL and PARA state bits set

This is an error. The revocation algorithm MUST alert the user and take no further action.

'

7 Security Considerations

The goal of the algorithm described in this document is to enable an RP to impose its own view of the RPKI, which is intrinsically a ecurity function. An RP using a constraints file is trusting the assertions made in that file. Errors in the constraints file used by an RP can undermine the security offered by the RPKI, to that RP. In particular, since the paracertificate hierarchy is intended to trump the original certificate hierarchy for the purposes of path discovery, an improperly constructed paracertificate hierarchy could validate ROAs that would otherwise be invalid. It could also declare as invalid ROAs that would otherwise be valid. As a result, an RP must carefully consider the security implications of the constraints file being used, especially if the file is provided by a third party.

8 IANA Considerations

[Note to IANA, to be removed prior to publication: there are no IANA considerations stated in this version of the document.]

9 Acknowledgements

The authors would like to acknowledge the significant contributions of Charles Gardiner, who was the original author of an internal version of this document, and who contributed significantly to its evolution into the current version.

10 References

10.1 Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3513] Hinden, R., and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003.

[RFC3779] Lynn, C., Kent, S., and K. Seo, "X.509 Extensions for IP Addresses and AS Identifiers", RFC 3779, June 2004.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key

Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5396] Huston, G., and G. Michaelson, "Textual Representation of Autonomous System (AS) Numbers", RFC 5396, December 2008.

[RFC6480] Lepinski, M. and S. Kent, "An Infrastructure to Support Secure Internet Routing", RFC 6480, February 2012.

[RFC6481] Huston, G., Loomans, R., and G. Michaelson, "A Profile for Resource Certificate Policy Structure", RFC 6481, Feburary 2012.

[RFC6487] Huston, G., Michaelson, G., and R. Loomans, "A Profile for X.509 PKIX Resource Certificates", RFC 6487, February 2012.

10.2 Informative References

None.

Authors' Addresses

Stephen Kent
Raytheon BBN Technologies
10 Moulton St.
Cambridge, MA 02138

Email: kent@bbn.com

Matthew Lepinski
Raytheon BBN Technologies
10 Moulton St.
Cambridge, MA 02138

Email: mlepinsk@bbn.com

Mark C. Reynolds
Island Peak Software
328 Virginia Road
Concord, MA 01742

Email: mcr@islandpeaksoftware.com

Appendix A: Sample Constraints File

```
   ;
   ; Sample constraints file for TBO LTA Test Corporation.
   ;
   ; TBO manages its own local (10.x.x.x) address space
   ; via the target blocks in this file.
   ;

   ;
   ; Relying party subsection. TBO uses ssh-agent as
   ; a software cryptographic agent.
   ;

   PRIVATEKEYMETHOD         OBO(ssh-agent)
   TACERTIFICATE            tbomaster.cer

   ;
   ; Flags subsection
   ;
   ; Always use the resources in this file to augment
   ;   certificate resources.
   ; Always process resource conflicts in the tree, even
   ;   if the target certificate is missing.
   ; Always search the entire tree.
   ;

   CONTROL  resource_nounion      FALSE
   CONTROL  intersection_always   TRUE
   CONTROL  treegrowth            TRUE

   ;
   ; Tags subsection
   ;
   ; Copy the original cert's validity dates.
   ; Use the default policy OID.
   ; Use our own CRLDP.
   ; Use our own AIA.
   ;

   TAG   Xvalidity_dates         C
   TAG   Xcp                     D
   TAG   Xcrldp       rsync://tbo_lta_test.com/pub/CRLs
   TAG   Xaia         rsync://tbo_lta_test.com/pub/repos

   ;
   ; Block subsection
   ;
```

```
   ;
   ; First block: TBO Corporate
   ;

   ; Resource Holder: TBO Corporation

   SKI 00112233445566778899998877665544332211
     IPv4
       10.2.3/24
       10.8/16
     IPv6
       2001:db8::/32
     AS#
       60123
       5507

   ;
   ; Second block: TBO LTA Test Enforcement Division
   ;

   ; Resource Holder: TBO Corporation

   SKI 653420AF758421CF600029FF857422AA6833299F
     IPv4
       10.2.8/24
       10.47/16
     IPv6
     AS#
       60124

   ;
   ; Third block: TBO LTA Test Acceptance Corporation
   ; Quality financial services since sometime
   ; late yesterday.
   ;

   ; Resource Holder: TBO Acceptance Corporation

   SKI 19:82:34:90:8b:a0:9c:ef:00:af:a0:98:23:09:82:4b:ef:ab:98:09
     IPv4
       10.3.3/24
     IPv6
     AS#
       60125

   ; End of TBO constraints file
```

Appendix B: Optional Sorting Algorithm for Ancestor Processing

Sorting is performed in an effort to eliminate any order dependencies in ancestor processing, as described in section 4.2.3 of this

document. The sorting algorithm does this by rearranging the processing of certificates such that if A is an ancestor of B, B is processed before A. The sorting algorithm is an OPTIONAL part of ancestor processing. Sorting proceeds as follows. The collection created at the beginning of ancestor processing is traversed and any certificate in the collection that is visited as a result of path discovery is temporarily marked. After the traversal, all unmarked certificates are moved to the beginning of the collection. The remaining marked certificates are unmarked, and a traversal again performed through this sub-collection of previously marked certificates. The sorting algorithm proceeds iteratively until all certificates have been sorted or until a predetermined fixed number of iterations has been performed. (Eight is suggested as a munificent value for the upper bound, since the number of sorting steps need not be any greater than the maximum depth of the tree.) Finally, the ancestor processing algorithm is applied in turn to each certificate in the remaining sorted collection. If the sorting algorithm fails to converge, that is if the maximum number of iterations has been reached and unsorted certificates remain, the implementation SHOULD warn the user.