PKI for Internet Address Space
Contract FA8750-07-C-0006

Final Report

July 31, 2007

1 Overview

1.1 Introduction

The Internet represents a critical component of the United States telecommunication infrastructure. The security of this infrastructure requires having accurate Border Gateway Protocol (BGP) interdomain routing. There is currently an ongoing effort to establish a global Resource PKI (RPKI) that will attest to IP address and Autonomous System (AS) resource holdings and support improved security for interdomain routing in the public Internet. The objective of the work funded by this contract is to facilitate the deployment of the RPKI. Once the RPKI is deployed, Internet Service Providers (ISPs) will be able to generate route filters based on the authoritative data that it provides. This represents a significant improvement over the current use of unverified Internet Routing Registry (IRR) data. The RPKI also will enable ISPs to detect attempts to trick them into abetting entities who try to "hijack" address space, by allowing the ISPs to demand proof of ownership of address space before advertising routes to it. Finally, the RPKI will pave the way for more sophisticated routing security technology such as S-BGP or soBGP.

1.2 Technical Approach

The RPKI attests to IP address and Autonomous System (AS) resource holdings. IP addresses are managed through a hierarchical system rooted at the Internet Assigned Numbers Authority (IANA). IANA allocates blocks of addresses to Regional Internet Registries (RIRs) and they, in turn, allocate address blocks to ISPs and to subscribers.[1] ISPs may allocate addresses to other (downstream) ISPs and to subscribers. In BGP, each ISP is identified by one or more AS numbers. AS numbers are allocated via the same hierarchy as address blocks (except that the allocation terminates at regional and national registries, i.e., ISPs do not sub-allocate AS numbers as they do addresses). The RPKI encompasses issuance of certificates for both address block and AS number holdings.

Creating a PKI that reflects the allocation of address blocks and AS numbers enables ISPs and subscribers to generate digitally signed data structures that express relationships among these resource holdings. Specifically, the holder of a block of addresses can sign a Route Origination Authorization (ROA) that identifies the AS numbers that are authorized to originate routes to the addresses in question. Securing route origination, a fundamental aspect of BGP operation, represents a critical first step toward securing BGP. Using the RPKI, ISPs can verify ROAs and use them to detect and reject bogus route origination advertisements transmitted through BGP UPDATE messages.

The RPKI consists of four major components: certificates and certificate revocation lists (CRLs), additional signed objects, a repository system for all RPKI signed objects, and software used by registries and ISPs to manage and process the signed objects. Each of the five RIRs will issue (X.509) certificates to its membership (primarily ISPs) attesting to the address blocks and AS

[1] In some regions, RIRs allocate resources to a next tier of national registries that represent some of the countries in that region. For simplicity this summary ignores this detail.

numbers held by each member. In turn, the ISPs will issue certificates to their customers (e.g., subscribers or "down stream" ISPs) attesting to the sub-allocation of address space by the ISPs. This process creates a hierarchical PKI that parallels the resource allocation hierarchy. Since the RPKI parallels the existing allocation hierarchy, no new "trusted" entities need be introduced as Certification Authorities (CAs) within the RPKI.

Under this contract, BBN has developed software components essential to the deployment of the RPKI. The primary users of this software are ISPs, the relying parties in this PKI. This software verifies certificates and CRLs to ensure that they meet the RPKI's syntactic criteria, and also validates certification paths. This enables an ISP to create a table of verified route origination data, and to generate route filters, which function as access control lists for BGP advertisements in routers. The BBN-developed software performs these tasks in a highly efficient manner, using a carefully-managed, local cache of certificates, CRLs, and ROAs. The BBN software complements the work of the RIRs, who are developing software for use by registries and ISPs, in their roles as issuers of certificates.

BBN staff also have spoken at RIR conferences to educate the community about the RPKI, have met with RIR staff to provide technical assistance, and are authoring RFCs within the IETF Secure Inter-Domain Routing (SIDR) working group. These activities have been critical for promoting adoption of the RPKI.

1.3 Meetings and Presentations

Under this contract, BBN

- held a project kickoff meeting on 1/25/07 attended by Dr. Douglas Maughan, at which we presented accomplishments to date and plans for the remaining work.
- attended AfriNIC's 5th public policy meeting (11/27/06 to 12/1/06)

1.4 Accomplishments

BBN completed the following tasks.

- Updated Rule Editor and Certificate/CRL Syntax Checking Software – BBN had previously developed software for use in other PKIs to verify the format of certificates and CRLs before they are issued. Under this contract, BBN modified the software for use by registries and ISPs within the RPKI. (That is, BBN updated it to deal with RFC 3779 syntax, tailored it to the Address/AS# PKI certificate policy, and performed additional Q/A testing.) The software was ported to a Unix/Linux environment and released to APNIC (the Asia/Pacific RIR) and ARIN (the North American RIR)
- Developed Certificate and ROA Validation Software – BBN developed the following components and released them to the RIR community.
    - RSYNC wrapper software that interacts with the distributed repository (RSYNC-based) system being developed by the RIRs to acquire the latest updates

    - The local repository (MYSQL-based) and the software infrastructure that sits between this repository and the various client software
    - Client software that will handle update transactions (adds, changes, deletions) for the local MYSQL repository
    - ROA generation and processing software
- Developed Ancillary Software for a Distributed PKI Root – BBN located and tested prototype software to provide the crypto functions needed to implement a distributed RPKI root. BBN also produced a report on how to use this software to perform distributed certificate (and CRL) signing in the RPKI context.
- Provide Outreach – Under this contract, BBN worked with the RIRs to educate their staff and their communities about the RPKI and provide technical assistance with respect to the RPKI.

2 Software

2.1 Code Development

As noted above, under this contract the following software was developed and/or extended for RPKI use:

- Rule Editor – Tool for generating the rules used by the Certificate/CRL syntax checking software (the Rule Engine)
- Rule Engine – Software for checking Certificate/CRL syntax
- Certificate and ROA Validation Software
    - RSYNC wrapper software that interacts with the distributed repository (RSYNC-based) system being developed by the RIRs to acquire the latest updates
    - Local repository (MYSQL-based) and the software infrastructure that sits between this repository and the various client software
    - The client software that will handle update transactions (adds, changes, deletions) for the local MYSQL repository
    - ROA generation and processing software
- Distributed Root – software that provides the crypto functions needed to support a distributed PKI root.

2.2 Licensing

The software systems constructed under this contract are a combination of newly developed software and software that was modified and/or reused from past work at BBN and other organizations. The licensing for the software was addressed as follows:

1. All code that was newly created using project funds is copyright by BBN Technologies and delivered to the Government with unrestricted rights (as defined in the FAR).

2. All code that was developed at BBN under internal funds and is reused for this project (such as the rule editor code) is copyright by BBN Technologies and delivered to the Government with restricted rights (as defined in the FAR).
3. All code that is modified and/or reused from other sources carries a license based on the original source license. For example, the modified threshold signature software (used in the distributed root) is provided under the Mozilla Public License (MPL), version 1.1, as this was the license selected for the original code by its developer, Steven Weiss.

All software source code files contain a license block in the header comment that specifies the licensing status.

This software will be made available to all interested parties without a fee. Consideration was given to placing the BBN software as open source, but as the intended audience is limited to just a few organizations, it was deemed more flexible to interact with the RIRs and ISPs directly to address their specific needs.

2.3 Security Code Review

We expect that sections of the code developed under this contract will be lifted and integrated with other applications. However, many RPKI users will not have the time or resources to perform a security analysis of the source code. Accordingly, BBN performed a security analysis of the project code to mitigate the risk of propagating any security flaws that may exist in the current code into new projects. A description of what this analysis included and, just as importantly, what this analysis did not address is provided in Appendix G.

2.4 Documentation

The following software documentation was produced under this contract and is included in the subsequent sections:

- Appendix A. Rule Engine/Editor User Guide
- Appendix B. Local Repository -- Design
- Appendix C. Local Repository – User Documentation and Installation Guide
- Appendix D. Local Repository -- Test Plan
- Appendix E. Local Repository – System and Performance Testing
- Appendix F. Distributed Root – Background and Procedures
- Appendix G. Security Code Review

3 Future Directions

Under this contract, several critical software components have been completed and made available to staff at the RIRs. The next steps should include

- tuning the software to address feedback from the initial users

- releasing the software to some major ISPs and addressing their feedback
- continuing to refine the software as various aspects of the RPKI evolve, e.g., the Route-Origination-Authorization data structure (SIDR working group), the architecture of the distributed RPKI repository system, the implementation of the RPKI "root", and integration of the RPKI into ISP operations.

In addition, the software could be enhanced as follows:

Detecting Overlapping IP Address Space Allocations.
Every certificate within the RPKI contains an allocation of IP addresses and AS numbers, as specified in RFC3779. The set of all such certificates forms a hierarchy, with the RFC 3779 allocations of a certificate forming a subset of those appearing in the parent certificate. The RPKI software developed by BBN rigorously verifies all subset relationships as part of its validations. It would also be desirable to detect the situation in which two components of the PKI have inappropriate overlapping allocations. Such overlaps can be innocuous, or they can be created by human error or malicious activity. It would be useful to develop an overlap detection mechanism for address space allocations and to integrate it with the BBN RPKI software suite. As part of this effort, a method could be included that enables users to annotate situations in which overlaps are permitted. Such an annotation scheme would allow for valid overlaps to be excluded from the report generated by the detection algorithm. The overlap detection software would be distributed as an (optional) upgrade to the RPKI users.

A Graphical User Interface for the RPKI
The RPKI repository/database system developed by BBN manages a variety of digital objects (ROAs, certificates, and CRLs) using a database and a set of programs that interface with that database. These programs are currently command line driven programs that perform validation, query and maintenance tasks on the set of objects in the PKI. The development of graphical front end for this system would significantly improve the experience of the end user. Such a graphical front end should provide: (a) a display of the database contents showing all valid ROAs and certificates as nodes in a tree, with the appropriate parent-child relationships and the ability to query a node for more details; (b) an operations console that would act as a front-end to the database client programs, with support for “one-click” methods to accomplish common tasks, such as performing the daily rsync run to synchronize the local repository with its remotes; (c) the ability to parse all the logfiles and output files created by the RPKI suite, and generate clickable reports based on their content, e.g., detailed analysis of a rejected certificate; (d) support for an annotation capability.

Appendix A -- Rule Engine/Editor User Guide

1 Introduction

The Rule Editor a Java program that provides a GUI to allow a user to create rules for certificates and CRLs issued by CAs. For this release of the Rule Editor, the rules must follow the profile for X.509 resource certificates (draft-ietf-sidr-res-certs-02.txt). The certificate and rule files used by the Rule Editor are in ASN.1 syntax. Each file can store only one set of rules or one certificate. The Rule Editor can read both BER and DER ASN.1 encoded files and produces DER ASN.1 encoded files.

As illustrated in Figure 1, the Rule Editor should be executed on a workstation separate from the CA workstation, for security reasons. A CA staff member begins by loading into the Rule Editor the certificate of the CA for which the rules are to be generated. He then creates one rule file for CA certificates and one for CRLs (and optionally, one for end entity certificates). These rule files are imported into the Rule Engine, operating on a CA workstation. The generated rules are used to check whether certificates or CRLs that are to be signed by a CA meet the syntactic requirements established by the CA. Before being signed, each certificate or CRL is passed into the Rule Engine along with the appropriate rule file. The Rule Engine then determines if the offered certificate or CRL meets the requirements as expressed in the rule file. Figure 1 below illustrates the relationship between the Rule Editor, Rule Engine, and certificate or CRL processing by a CA.

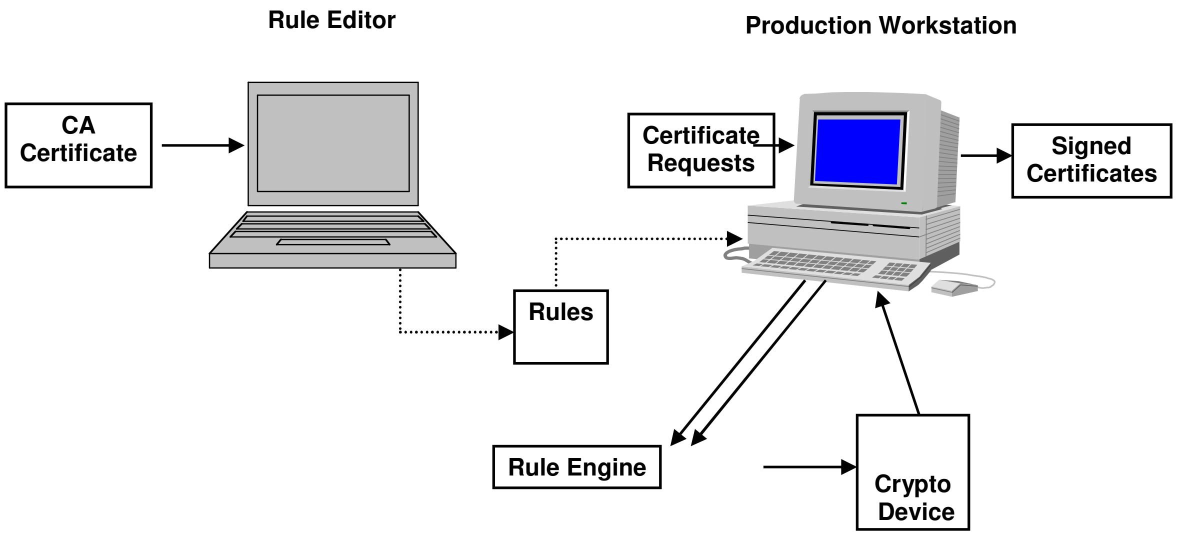


Figure 1. System Overview

2 Compiling the Rule Editor

There is a Makefile in the src directory. Type “make” at a command line in the src directory to compile the Java code in each of the subdirectories. The resulting class files are put into the lib directory.

3 Starting the Rule Editor

Open a command window, enter the lib directory, and type “java RuleEditor” to start the Rule Editor. The user is prompted to supply a CA certificate. A file browser window opens to allow the user to search for a file containing the desired CA certificate. Alternatively, the user may type “java RuleEditor myCAFilename” where the relative path name to the CA certificate file is myCAFilename. The Rule Editor imports this CA file and extracts relevant values to be used in constructing the rules. (If there are any invalid IP Address Blocks in this CA certificate, an error window appears listing them. In addition, the program lists in the command window additional information identifying the error, e.g. “V4: 0 2 6” means that entries 0, 2 and 6 in the V4 section have errors.)

If there are no errors I the supplied CA certificate, the Rule Editor Window then appears. All rules created will be based on this CA certificate until a new CA certificate file is opened, as described in Section 9.

The “examples” directory contains sample certificates, CRLs, and rules. Filenames contain “Ctf”, “CRL”, or "rule" in the filename to indicate if the file contains a CA certificate, a CRL, or a rule set, respectively. A filename may have more than one of these strings. For example, the filename “NIRCRLrules” indicates a set of rules for generating CRLs. Additionally, filenames with a “.txt” extension are a textual representation of the corresponding ASN.1 (binary) file. The text file corresponding to “NIRCRLrules” is “NIRCRLrules.txt”. These “.txt” files can be opened in any text editor for inspection, but cannot be re-imported into the RuleEditor. Filenames containing “Bad” in the name, such as “NIRBad1CRL” which contains a bad CRL, are erroneous files. Certificates are often created manually, resulting in many potential errors. These errors are most commonly found in the IP Address Block and Autonomous System Identifier extensions as these extensions can be quite cumbersome to create. The Rule Editor will not open erroneous files, but will give an error message indicating the problem that was process. Finally, files ending with “results” are text files that illustrate the Rule Engine output when the corresponding rule file and certificate (or CRL) file were passed into the Rule Engine.

4 Rule Editor Window

The Rule Editor window, shown in Figure 2, has three tabs. These are labeled “End-Entity Certificate Rule”, “CA Certificate Rule”, and “CRL Rule” and are used for creating rule sets for end entity certificates, CA certificates, and CRLs issued by the current CA, respectively. Two or three rule sets typically will be generated based on the same CA certificate. If a CA issues both end-entity certificates and CA certificates, as well as CRLs, that CA’s certificate would form the basis for three rule sets.

At the bottom of the Rule Editor Window are three buttons called “Retrieve CA’s Certificate File”, “Retrieve Rule File”, and “Create Rule and Save to File”. The use of these buttons is described in sections 7, 8, and 9.

The full path to the (current) imported CA certificate file is displayed in the title bar of the window.

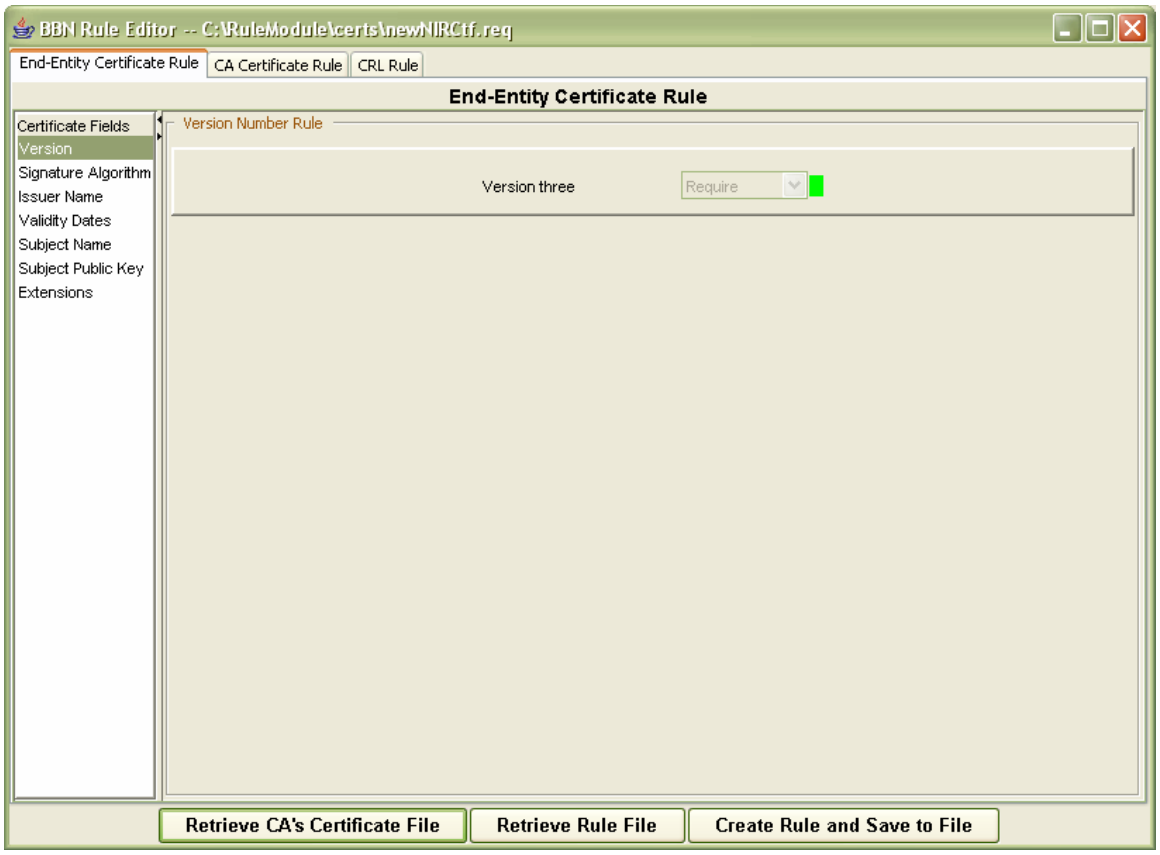


Figure 2. Rule Editor Main Window

5 Certificate Fields

If either of the certificate tabs is selected, a list of certificate fields appears on the left side of the window. The rule pertaining to the selected field appears on the right side of the window. Many of the certificate field values are fixed per the profile for X.509 resource certificates and therefore the user can only inspect these rules, not change them. For other fields, the user is allowed to impose constraints on the field values.

5.1 Version

The version number of the certificates is fixed at 3.

5.2 Signature Algorithm

The signature algorithm is fixed at sha256WithRSAEncryption.

5.3 Issuer Name

The Issuer name is imported from the Subject field of the current CA certificate, since that CA is the issuer of all certificates for which rules are being created.

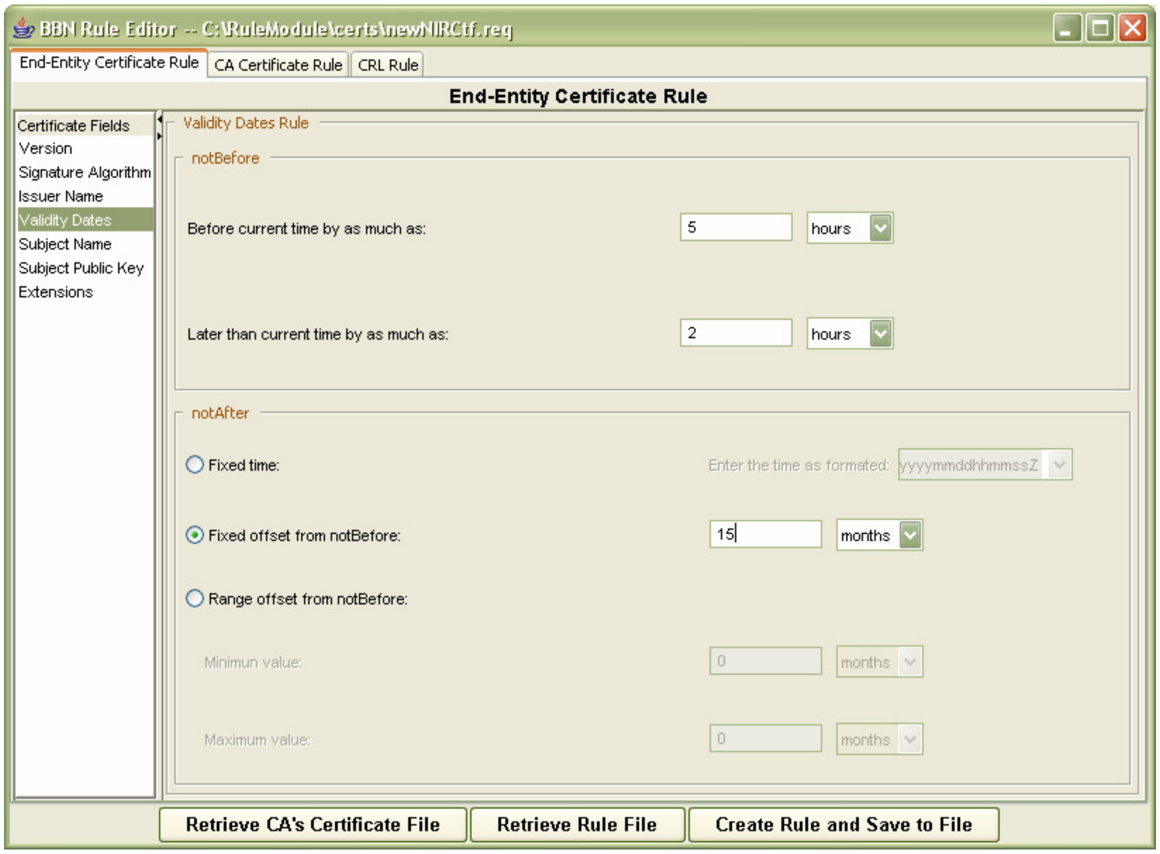
5.4 Validity Dates

Every certificate contains two dates that specify the time during which that certificate is valid. The validity dates rule allows the user to specify the window of allowable ranges for each end of the certificate validity period. In this rule the current time refers to the time that the CA signs the certificate, not the time when the rule is created.

The notBefore clause allows the user to specify a time window for the beginning of the certificate validity period.

The notAfter clause allows the user to specify allowed values for the end of the certificate validity period. This can be a fixed time, a fixed offset from the time in the notBefore clause, or a range offset from the time in the notBefore clause. As an example, for a rule to specify that all CA certificates issued by this CA should have a 15 month validity window, select the Fixed Offset radio button and enter 15 months to the right as shown in Figure 3. As another example, to specify that all CA certificates issued by this CA should be valid for anywhere between 15 and 18 months, choose the Range Offset radio button and enter 15 months for the first value and 18 months for the second value.

If a fixed date is entered, it is specified in a format :yyyymmddhhmmss. This indicates 4 digits for the year, and 2 each for the month, day, hours, minutes, and seconds. Each listed digit must be represented in order for the time to be accurately understood. Additionally, the time must be expressed as Greenwich Mean Time (as indicated by the use of the "Z" suffix).


BBN Rule Editor -- C:\RuleModule\certs\newNIRCtf.req

End-Entity Certificate Rule | CA Certificate Rule | CRL Rule

End-Entity Certificate Rule

Certificate Fields
Version
Signature Algorithm
Issuer Name
Validity Dates
Subject Name
Subject Public Key
Extensions

Validity Dates Rule

notBefore

Before current time by as much as: 5 hours

Later than current time by as much as: 2 hours

notAfter

Fixed time: Enter the time as formated: yyyymmddhhmmssZ

Fixed offset from notBefore: 15 months

Range offset from notBefore:

Minimun value: 0 months

Maximum value: 0 months

Retrieve CA's Certificate File | Retrieve Rule File | Create Rule and Save to File

Figure 3. Validity Dates Rule Window

5.5 Subject Name

The Subject Name rule allows the user to impose restrictions on the subject names of certificates issued by this CA. There is a list of four different types of distinguished name attributes (Country Name, Organization Name, Organizational Unit Name, and Common Name) that can be employed to create a Subject Name. Each type has a corresponding drop down box and each of the drop down boxes contains the possible values of Allow, Prohibit, and Require. “Allow” means that the attribute can optionally be present in the certificate, “Prohibit” means that the attribute cannot appear in the certificate, and “Require” means that the attribute must appear in the certificate.

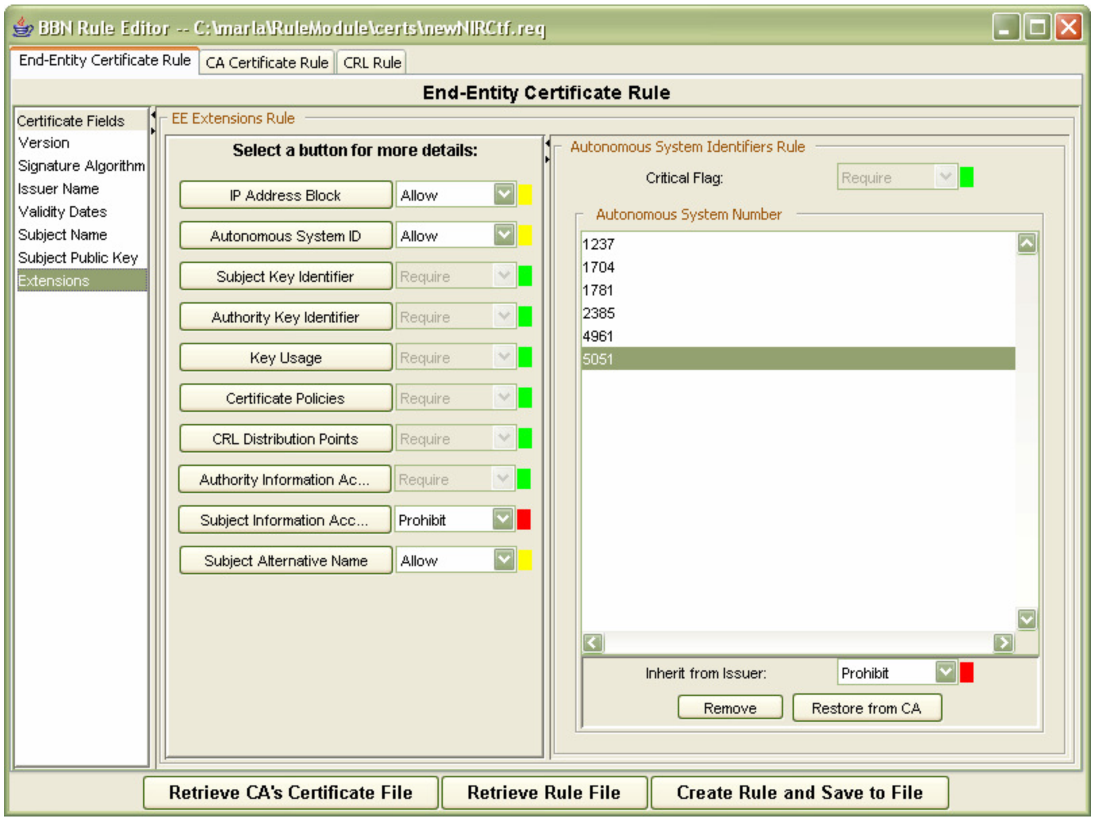
There is an optional text field to the right of each drop down box. This text box allows the user to impose restrictions on the allowable values for a particular field. For example, to specify that the Organizational Unit must be “Widget Dept.” for all end entity certificates issued by this CA, type “Widget Dept.” in the text field to the right. The drop down box can be set to either “Allow” or “Require.” Setting the drop down to “Allow” would indicate that field could be absent from the certificate, but if present must be “Widget Dept”. If the drop down is set to ”Prohibit,” the text field is cleared.

5.6 Subject Public Key

The subject public key algorithm is fixed at RSA (rsadsi-rsaEncryption), but the key size can be changed by the user.

5.7 Certificate Extensions

Selecting "Extensions" from the list at the left causes the right part of the window to break into two sections as seen in Figure 4. On the left is a list of buttons with a drop down box next to each. The buttons represent the list of extensions that are allowed to be in resource certificates per the profile for X.509 resource certificates. Clicking on a button changes the right side to display contents pertaining to the selected extension.


BBN Rule Editor -- C:\marla\RuleModule\certs\newNIRCtf.req
End-Entity Certificate Rule
CA Certificate Rule
CRL Rule
End-Entity Certificate Rule
Certificate Fields
Version
Signature Algorithm
Issuer Name
Validity Dates
Subject Name
Subject Public Key
Extensions
EE Extensions Rule
Select a button for more details:
Autonomous System Identifiers Rule
Autonomous System Number
Remove
Restore from CA
Retrieve CA's Certificate File
Retrieve Rule File
Create Rule and Save to File

IP Address Block	Allow	🟨
Autonomous System ID	Allow	🟨
Subject Key Identifier	Require	🟩
Authority Key Identifier	Require	🟩
Key Usage	Require	🟩
Certificate Policies	Require	🟩
CRL Distribution Points	Require	🟩
Authority Information Ac...	Require	🟩
Subject Information Acc...	Prohibit	🟥
Subject Alternative Name	Allow	🟨

Critical Flag: Require 🟩

Autonomous System Number: 1237, 1704, 1781, 2385, 4961, 5051

Inherit from Issuer: Prohibit 🟥

Figure 4. Certificate Extensions Window

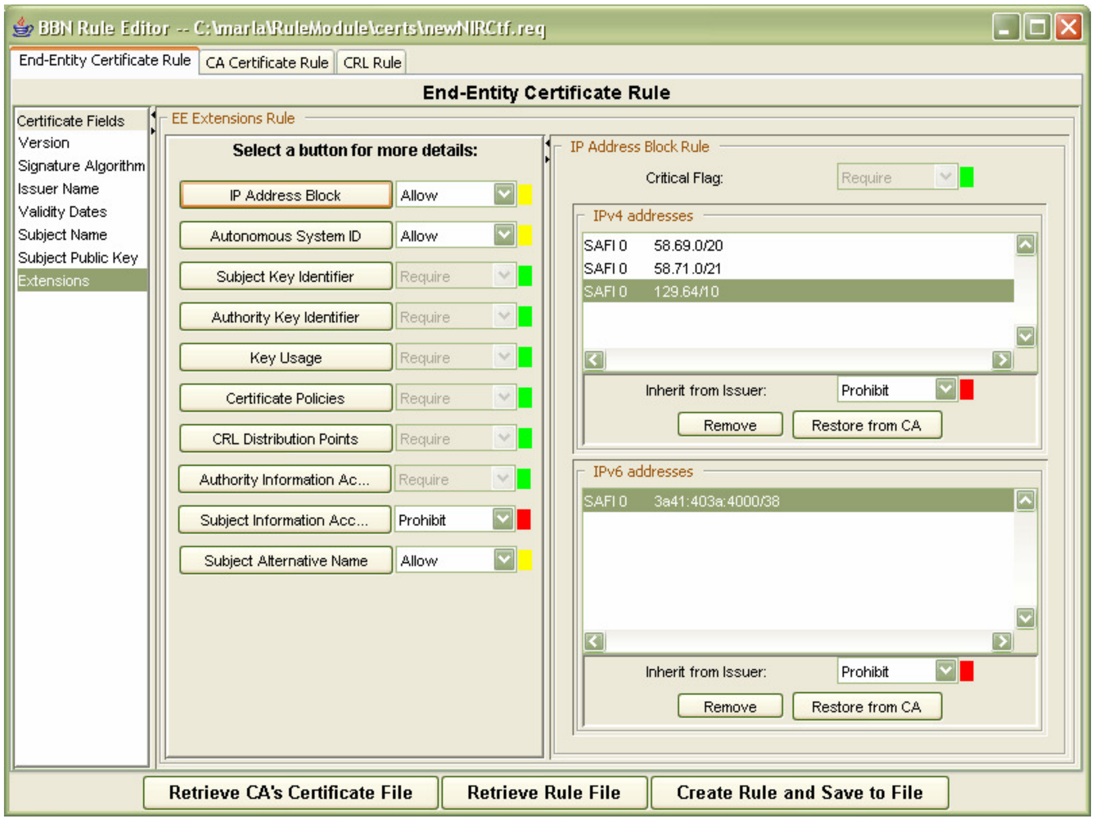
The drop down box next to each button determines if that extension is allowed, prohibited, or required to be in the certificate. A drop down box is disabled if that extension is required per the profile for X.509 PKIX resource certificates. If the drop down box is set to "Prohibit," then the right side of the window simply states that inclusion of the extension has been prohibited instead of showing contents pertaining to the extension.

Resource certificates must contain either an IP Address Block or an Autonomous System ID extension (or both). Both extensions start as being set to "Allow". If the user sets one to "Prohibit", the other will automatically be switched to "Require" and the drop down box will be disabled until the one that was prohibited is changed to either "Allow" or "Require."

Upon opening a CA certificate the Rule Editor performs a check to ensure that at least one of these extensions is present. If neither is there, then the Rule Editor indicates an error and asks the user to exit the Rule Editor or choose another CA certificate.

5.7.1 IP Address Blocks

The IP address block extension view shows the list of IPV4 list of address ranges on top and a list of IPV6 address ranges on the bottom. The list of addresses is taken directly from the current CA certificate . If this extension is missing from the CA certificate then this extension is prohibited from appearing in the rules. Figure 5 illustrates this interface.



IP Address Block	Allow
Autonomous System ID	Allow
Subject Key Identifier	Require
Authority Key Identifier	Require
Key Usage	Require
Certificate Policies	Require
CRL Distribution Points	Require
Authority Information Ac...	Require
Subject Information Acc...	Prohibit
Subject Alternative Name	Allow

Critical Flag: Require

IPv4 addresses:

```
SAFI 0    58.69.0/20
SAFI 0    58.71.0/21
SAFI 0    129.64/10
```

Inherit from Issuer: Prohibit

IPv6 addresses:

```
SAFI 0    3a41:403a:4000/38
```

Inherit from Issuer: Prohibit

Figure 5. IP Address Block Extension Window

The user can remove items from either the IPV4 or IPV6 list by selecting an IP address range and pushing the remove button. This could be used to allow the user to reserve some address ranges for later use, i.e., to ensure that the removed addresses are not permitted in any certificates issued by this CA.

Clicking the "Restore from CA" button resets the list contents to that shown in the current CA certificate. This is also true of a retrieved rule set (see section 8).

The critical flag is required.

5.7.2 Autonomous System Numbers

The list of autonomous system number ranges is taken directly from the current CA certificate. If this extension is missing from the CA certificate, this extension is prohibited from appearing in the rules. Figure 4 in Section 5.7 illustrates this interface.

The user can remove items from the list by selecting a number range and clicking the "Remove" button. This allows the user to reserve some ranges for later allocation.

Clicking the "Restore from CA" button will reset the list contents to that shown in the current CA certificate. This is also true of a retrieved rule set (see section 8).

The critical flag is required.

5.7.3 Subject Key Identifier

The Subject Key Identifier rule is fixed per the profile for X.509 resource certificates. The identifier will be produced using a 160 bit sha-1 hash of the subject's public key. The critical flag is prohibited.

5.7.4 Authority Key Identifier

The Authority Key Identifier is taken directly from the CA certificate. It is the subject key identifier of the CA certificate.

The critical flag is prohibited.

5.7.5 Key Usage

The Key Usage rule is fixed per the profile for X.509 resource certificates. Only the digital signature bit is set for end entity certificates and only the CRL sign and cert sign bits are set for CA certificates.

The critical flag is required.

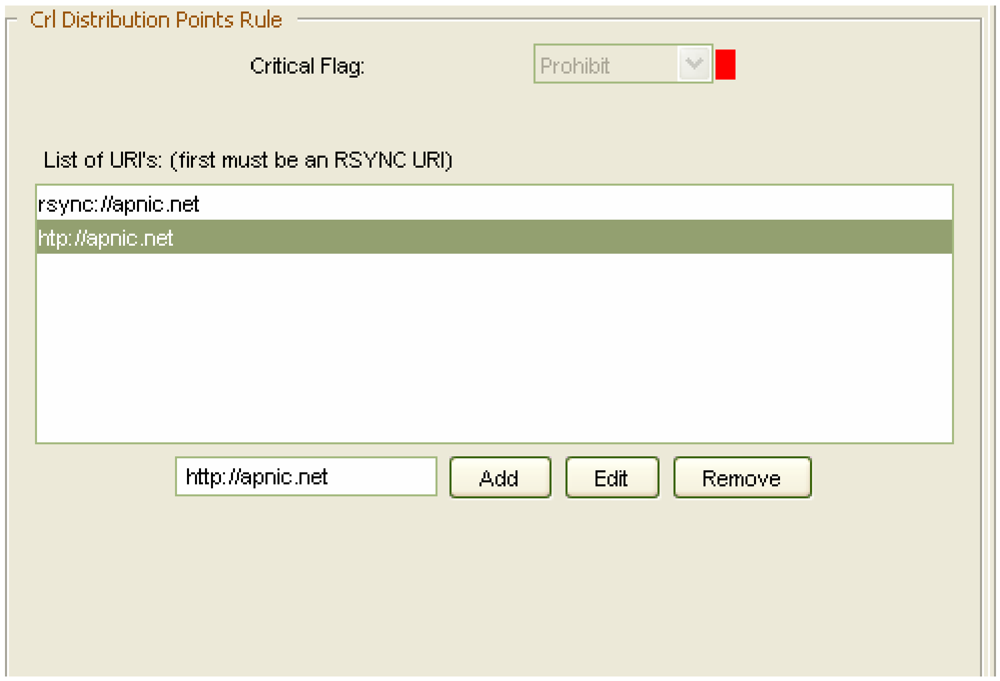
5.7.6 Certificate Policies

The certificate policy specifies the resource certificate policy, which is fixed at "1.3.6.1.5.5.7.14.2", as specified in the profile for X.509 resource certificates.

The critical flag is required.

5.7.7 CRL Distribution Points, Authority Information Access, Subject Information Access

The user interface for these three extensions is identical. For each the contents must be a list of one or more URI locations. The first URI in the list must begin with "rsync://", and there must be only one such entry.


Crl Distribution Points Rule
Critical Flag:
Prohibit
List of URI's: (first must be an RSYNC URI)
Add
Edit
Remove

rsync://apnic.net
htp://apnic.net

http://apnic.net

Figure 6. CRLDP URI List Window

An entry is added by typing the desired URI into the text field to the left of the three buttons, and clicking the "add" button. To edit an entry previously added, first select the entry with the mouse. The entry appears in the text field. Simply edit the contents and then click the "Edit" button to replace the highlighted entry with the contents of the text field. Figure 6 shows this interface with two URIs entered. The second has a misspelling of "http" so the entry was selected and then edited in the text field. The user can now push the "Edit" button to replace the selected value with the corrected value.

5.7.7.1 CRL Distribution Points

The CRL Distribution Points extension specifies the location where the CRL(s) associated with the issuer will be stored.

The critical flag is prohibited.

5.7.7.2 Authority Information Access

The Authority Information Access field specifies the name of a file containing the certificate of the issuer of the current CA's certificate.

The critical flag is prohibited.

5.7.7.3 Subject Information Access

The Subject Information Access field specifies the name of a directory containing files related to the subject of the certificate in which this extension appears, e.g., certificates signed by the currently active CA.

The critical flag is prohibited.

5.7.8 Subject Alternative Name

The subject alternative name is an optional extension. If this extension is allowed or required the interface shows four possible alternative name types. The user should set the drop down list for each to specify if that type of name is allowed, prohibited, or required to be in certificates issued. Note that at least one of the four must be allowed or required if the extension is allowed or required, so that if three types are set to "Allow" the remaining type will be automatically set to "Require." (If none are to be allowed, set the extension drop down box to "Prohibit.")

The critical flag is prohibited.

6 CRL Fields

6.1 Version

The version number of the CRLs is fixed at 2.

6.2 Signature Algorithm

The signature algorithm is fixed at sha256WithRSAEncryption.

6.3 Issuer Name

The Issuer Name is taken from the Subject field of the current CA certificate since that CA will be the issuer of all CRLs for which rules are being created.

6.4 This Update

The rule for this field allows the user to specify a time window around the time of signing the CRL. In this rule the current time refers to the time that the CA signs the CRL, not the time when the rule is created.

6.5 Next Update

The rule for this field allows the user to specify a time window for when the next CRL will be issued.

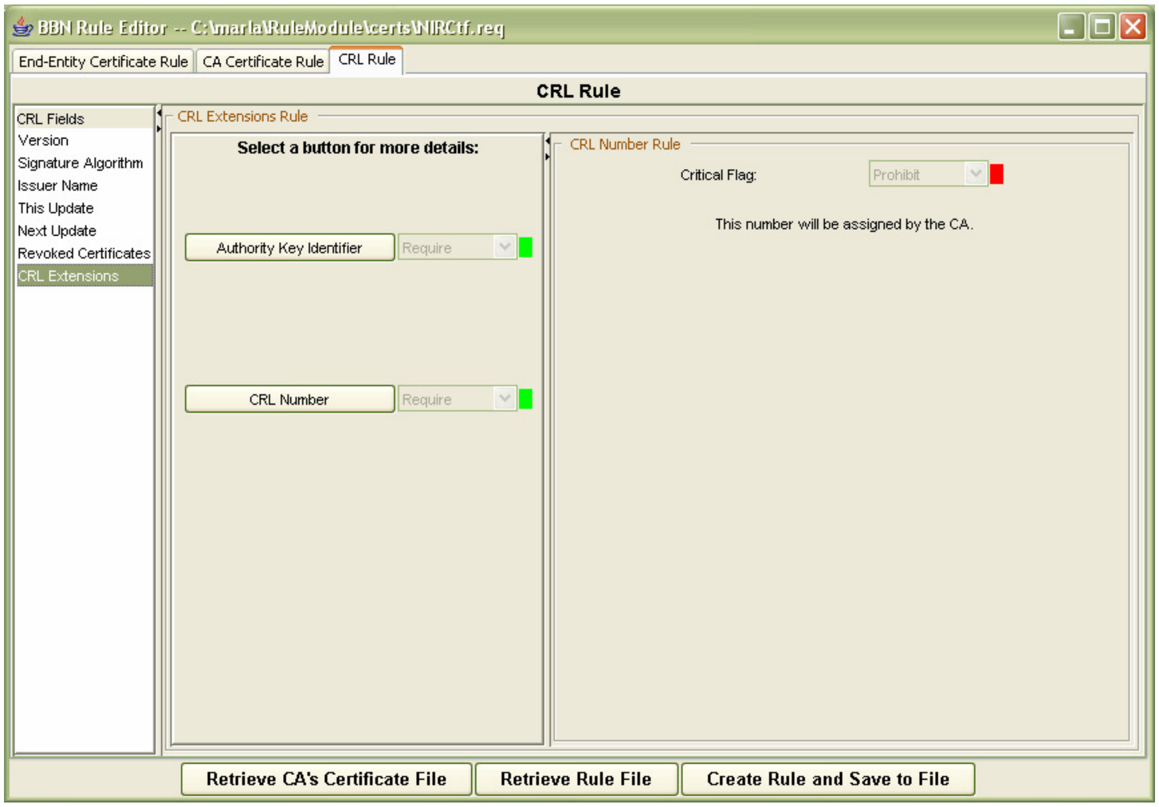
The rule may contain a fixed time, a fixed offset from the thisUpdateTime, or a range offset from thisUpdateTime. As an example, to specify that the next CRL will be issued every month, select the Fixed Offset radio button and enter 1 month to the right. As another example, to specify that the next CRL will be issued between 2 and 4 weeks after the current CRL, choose the range offset radio button and enter 2 weeks for the first value and 4 weeks for the second value.

6.6 Revoked Certificates

Revoked Certificates is a list of certificate number and revocation date pairs. Both values are required for each list entry. No rules can be specified for this portion of the CRL.

6.7 CRL Extensions

Selecting "CRL Extensions" from the list at the left causes the right part of the window to break into two sections as seen in Figure 7. On the left is a list of buttons with a drop down box next to each. The buttons represent the list of extensions that are allowed to be in CRLs per the profile for X.509 resource certificates. Clicking a button changes the right side to display contents pertaining to the selected extension.



Authority Key Identifier: Require 🟩

CRL Number: Require 🟩

Critical Flag: Prohibit 🟥

This number will be assigned by the CA.

Figure 7. CRL Extensions Window

6.7.1 Authority Key Identifier

The Authority Key Identifier must be present and is taken directly from the current CA certificate. It is the subject key identifier of the CA certificate.

The critical flag is prohibited.

6.7.2 CRL Number

The CRL number extension must be present and will be assigned by the CA. No rules can be specified.

The critical flag is prohibited.

7 Saving a Rule Set

Click the "Create Rule and Save to File" button to save the current set of rules to a file. If there are any errors in the rules, a dialog box will pop up to prompt you to fix them before saving. If there are no errors, then a browser dialog box will open. Find the desired directory, type in a file name, and click the save button. If there are errors, the Rule Editor will show a dialog box indicating where they are. These errors must be fixed before the rule set could be saved.

8 Editing an Existing Rule Set

Pushing the "Retrieve Rule File" button will open a file browsing window to select the desired rule file. Once a rule file is opened, the appropriate tab will be selected based on the type of rule set opened (i.e., End Entity, CA, or CRL rule), and the rule fields will be populated from the rules in the file. If the contents of the rule file conflict with the current CA certificate, an error pop up box appears indicating the error and the rule file will not be made available for editing. The rule fields will revert to values based on the contents of the current CA certificate.

9 Changing to Another CA Certificate

Clicking the "Retrieve CA's Certificate File" button opens a file browsing window to select the desired CA Certificate file. Once the file is opened, the rule fields will be either reset or populated from the certificate as appropriate. If there are any errors in the certificate or certificate extensions, an error box will pop up prompting the user either to either pick another certificate or to exit the Rule Editor. For example, if the CA certificate is missing both the IP Address Block and Autonomous System number extensions, or if there are overlaps in the ranges in the IP Address Block extension, or the ASN.1 does not parse properly, then the certificate is invalid and cannot be opened.

Appendix B -- Certificate and ROA Validation Software – Design

1 Introduction

This document describes the design of a database system for managing a set of digital objects – certificates; certificate revocation lists (CRLs); and route origination authorizations (ROAs) – in support of a PKI of IP address space and AS# allocations. The core requirement for this system is to manage the collection of ROA objects in a way that makes it straightforward for the user to identify those ROAs that are valid, and convert the IP address and AS# information in those ROAs into a form that can be postprocessed into BGP filter specifications.

Functionally, this core requirement can be broken down into three top-level steps: obtain a set of ROAs and their associated digital objects; identify those ROAs that are valid; and extract the information needed for BGP filter generation from those valid ROAs. The first part of this process, the actual mechanism used to obtain the digital objects, is not directly within the scope of our responsibility, although our software does interact with this mechanism. The second component, the ROA validation component, contains the overwhelming majority of the effort. In order to validate a ROA one must not only verify that the ROA meets the ROA specification, as well as conforms to the profile for this PKI, it also implies that the ROA must be validated against the certificate that signed it. This, in turn, implies that this certificate itself must be validated, which leads to the need to trace up the chain from this certificate to its parent and further predecessors until one reaches a self-signed trust anchor certificate. At each point in this process the certificate in question must be validated, which involves a number of separate tests (Has it expired? Has it been revoked? Does it actually have a parent?). The third and final component, the generation of the ROA output, presupposes the existence of a ROA parser, which would have to exist in order to perform the validation step. Given that parser it is straightforward to generate any output format one wants from the IP address and AS# information.

The remainder of this document will describe the software design that has been created in an effort to meet these requirements. The description given will take a process oriented form. The software consists of a number of components that perform various aspects of the overall work required. The work performed by each component is designed to satisfy a subset of the requirements for the whole task. There is a logic flow of activity starting from the point at which a new digital object arrives, and proceeding through the point at which the filter file(s) are created. Some of the components must be executed in a specific order, while others can be executed out of order. Despite this, it is still meaningful to speak of an overall process flow, namely the set of activities that must be executed in a given time interval, nominally a single day.

This document will describe these software components, how they integrate with one another and with the external processes also taking place, and most specifically, how they conspire to realize the overall requirements of the database portion of the PKI.

2 Component Design

The Resource PKI (RPKI) database system manages digital objects. These objects are stored in files of three logical types: certificates, certificate revocation lists (CRLs) and route origination authorizations (ROAs). Individual users of the RPKI system obtain their local files through a synchronization process with external sources of these files. The collection of local files is known as the local *repository cache*. In the general case, any process could be used for this synchronization. In the RPKI database system, however, we explicitly **assume** that the process will be performed using the rsync utility. This utility is a standard component of Unix/Linux systems and was not developed or modified in our RPKI system. The RPKI system is closely integrated with rsync, however. Should the synchronization process change (to use http, for example), there are two dependencies with the RPKI software that would need to be changed. These dependencies will be described below in subsections 2 and 6.

The RPKI database software consists of the following components: (1) the rsync wrapper; (2) the rsync_aur program; (3) the rcli database (DB) client; (4) the query client; (5) the garbage collection client; (6) the chaser client; (7) the RPKI database; (8) the database interface libraries; and (9) the ROA processing library. Each of these components will be described in turn.

2.1 The rsync wrapper

The rsync wrapper is a shell script that invokes (wraps) the rsync utility. It calls the rsync utility with a set of command line arguments that cause rsync to generate a logfile of all its actions. It also ensures that logfiles are rotated, i.e. that a new logfile is created each time rsync is run. This script can optionally specify a configuration file for rsync to use; if no such configuration file is provided, rsync uses a default configuration file. It is **assumed** that the user provides the default configuration file. The rsync wrapper may be invoked manually (by the user); it may also be invoked as a result of running the chaser client. After the rsync utility has done its work, the logfile will record the changes that rsync has made to the repository. Note carefully that the RPKI query software never modifies the repository itself: it is a read-only consumer of the information in the repository.

2.2 The rsync_aur program

When the rsync wrapper (and, by implication rsync) has run to completion, a logfile of actions will have been generated. This file is in a format that is specific to rsync. The rsync_aur program is launched by the rsync wrapper as its last action. The rsync_aur program parses the logfile and generates a series of "events" corresponding to each action. This is one of the dependencies on the use of rsync stated above. Should another synchronization protocol be used, such as http, a different logfile format would be generated, and therefore that portion of rsync_aur that parses the logfile would have to change. Each event is actually a message sent across a socket corresponding to an action. A full description of the socket protocol is beyond the scope of this document (it can be found in the file "apkiaur.txt.") For our purposes, three events (messages) are significant. When a file is added to the repository, rsync_aur sends an "A" message to the socket. When a file is updated in the repository (new contents are stored into an existing file), rsync_aur sends a "U" message to the socket. Finally, when a file is removed from the repository, rsync_aur sends an "R" message to the socket. From a process standpoint, rsync_aur performs three actions: it opens a socket to a port specified on the command line; it parses the rsync logfile and generates a number of messages (typically A, U or R messages), and then it closes its end of the socket. This entire process is known as a socket session.

2.3 The rcli DB client

The rcli program is a database client. It is known as a "client" because it is a client (user) of the database. (Throughout this document "client" will always have this specific meaning.) This program can run in two modes. In a manual scenario, rcli can be invoked on the command line to add or delete objects from the database. In a deployed configuration, rcli runs as a daemon (detached process), waiting for socket connections from rsync_aur and processing the messages it receives over the socket in an attempt to bring the database up to date with respect to changes in the repository that have been caused by rsync and reported by rsync_aur. When rcli is run as a daemon it runs perpetually, processing socket sessions. Fundamentally, rcli performs only two actions: attempting to add something to the database and attempting to delete something from the database (updates are processed as a delete operation followed by an add operation).

In order for a digital object to be added to the database it must be valid at the time it is added. The rcli program therefore makes extremely heavy use of the database infrastructure libraries. In order to validate certificates and CRLs it also makes use of the OpenSSL library. In order to validate ROAs it makes use of the ROA processing library. The rcli program is thus the primary validation engine for everything in the PKI. It will not add any object known to be invalid to the database. Objects known to be valid or those whose validity cannot be determined (e.g. due to the object being loaded before its parent certificate) are both placed in the database; flags in the database are set to distinguish between these two types of objects.

The only assumption that the rcli program makes about the structure of the repository is that there is a single top-level directory that contains the trust anchor certificates whose name starts with the string "TRUST". This allows rcli to know which certificates are allowed to be self-signed.

The rcli program uses three top level algorithms from the database infrastructure libraries: retrieve_tdir(), add_object() and delete_object(). The retrieve_tdir() algorithm is an extremely simple algorithm that requests the value of the repository root directory from the metadata table of the database. (This value will have been stored into this table when the database was initialized.) This directory named is then checked against the initial portion of the fully qualified pathname of the object being added. If they do not match the user and this is an interactive invocation of rcli, the user is requested to verify that the add operation is to be performed. (In a non-interactive invocation of rcli this test is not performed.) This is a sanity check to ensure that the user truly wishes to add an object that is not stored within the directory tree rooted at the repository's top level directory. This situation might occur for the trust anchor certificates; it might also occasionally occur for objects received by an out-of-band mechanism, e.g. received by email rather than rsync.

The add_object() algorithm calls a number of subordinate algorithms in order to accomplish work. The body of add_object() is shown in the following pseudocode:

```
If  not isokfile() return error;
Typ = infer_filetype();
If ( typ is invalid or unrecognized ) return error;
If not findorcreatedir() return error;
If ( typ is certificate )
    Return add_cert();
Else if ( typ is CRL )
    Return add_crl();
Else if ( typ is ROA )
    Return add_roa();
```

The isokfile() algorithm determines if the file that is being added is a normal file, and not a socket, pipe, device file, or symbolic link. It also determines if the file is readable. The infer_filetype() algorithm infers the type of the file based on its extension. Files with extensions ".cer", ".crl" and ".roa" are assumed to be DER encoded binary files of type certificate, CRL and ROA, respectively. Note that CRLs must have an extension of ".crl" and ROAs must have an extension of ".roa". Certificates should have an extension of ".cer", but this is not mandatory. Any file without an extension, or with an unknown extension, will be processed as if it were a certificate. This means that if objects other than certificates are placed in the repository, an error will occur when those objects are being processed by the OpenSSL read routines. If the file has an additional extension of ".pem" it is instead assumed to be a PEM armored representation of a DER encoded file of the corresponding type. Note that this inferencing step only looks at the filename; it does not attempt to open the file and apply any kind of heuristic to the contents.

In general any error is logged. The log entry will contain the name of the object being processed, the operation that is being attempted, and a reason for the failure. This allows the users to examine the logfiles to determine why an operation on an object failed.

The next step is to associate the directory part of the pathname to a directory identifier (the reason why directory identifiers are used is explained below in the database design section). The directory is looked up in the directory table of the database. If it is already present, its id is returned. If it is not already present, a new entry is made for that directory, and its id is returned. A switch is now made based on the inferred file type, and one of three subordinate algorithms is called: add_cert(), add_crl() or add_roa().

The add_cert() algorithm performs seven steps. Its pseudocode looks like:

```
If not cert2fields() return error;
Set some flag bits based on cert fields and user input;
If not rescert_profile_chk() return error;
If not verify_cert() return error;
If cert_revoked() return error;
If not add_cert_internal() return error;
verify_children()
```

The subalgorithm cert2fields() uses parts of the OpenSSL library to extract the following fields from the certificate: the subject, the issuer, the serial number, the SKI, the AKI, the SIA, the AIA (optional for trust anchors), the CRLDP, the basic constraints extension, the RFC3779 extension field (as a binary blob) and the notBefore and notAfter fields. If this extraction fails an error is returned. (The extraction could fail because, for example, the putative certificate file is not a valid X509 certificate file, has bad ASN.1, has a missing field that is mandated by the profile, or any number of syntactic failure causes.) Note also that other fields from the certificate, such as the key usage extension and the certificate policies extension, are also extracted subsequently (during verify_cert()). The purpose of cert2fields() is solely to extract those fields that will be used to populate the database.

The next step in add_cert() is to possibly set bit fields in the flags field for the certificate. The only bit that can have already been set is the FLAG_CA bit, which would have been set in cert2fields() if the certificate was a CA certificate. Add_cert() compares the subject and issuer fields, and if they are identical it sets the FLAG_SS (self-signed) bit. Add_cert() then checks an argument that was passed in on the command line, and if this argument is present it sets the FLAG_TRUST bit, indicating that this is a trust anchor. It is an error to attempt to set this bit if the FLAG_SS bit is not also set. This convention implies that from the rcli command line one can declare that a certificate is a trust anchor. There is no corresponding mechanism for rsync_aur to make such an assertion. The implication is that it is **assumed** that the database has been manually seeded with the trust anchor certificates using the command line form of rcli, and that as new trust anchors are issued or deleted, the user must manually update the database with the new trust anchor information.

The next step is to call the profile validation algorithm rescert_profile_chk(). This subalgorithm checks a variety of aspects of the certificate to insure that it conforms to the resource certificate profile. Specifically, it checks to see that the basic constraints extension is present and marked critical; that the SKI is not marked critical; that the AKI is not marked critical, that its keyIdentifier is present except in trust anchors, that its authorityCertIssuer is not present, and that its authorityCertSerialNumber is not present; that the key usage extension is present and marked critical and that it has only keyCertSign and CRLSign set for a CA certificate and only digitalSignature set for an EE certificate; that the CRLDP, SIA and AIA (if present) are marked non-critical; that the certificate policy extension is present and marked critical and that it does not have PolicyQualifiers and that the PolicyIdentifier OID has the value 1.3.6.1.5.5.7.14.2; and, finally, that the RFC3779 extension is marked as critical. If any test fails, an error is returned.

The next step is to call the path validation algorithm verify_cert(). This subalgorithm uses a combination of OpenSSL and database infrastructure routines in order to perform validation of the certificate. The pseudocode for this algorithm is as follows:

```
Initialize stack SU of untrusted certificates
Initialize stack ST of untrusted certificates
C = input certificate
P = parent_cert(C)
While ( P does not have trusted bit set )
    Push P onto SU
    P = parent_cert(P);
    If P is NULL return error
Push P onto ST
Ask OpenSSL to verify(SU, ST, C); if error return error
```

The subalgorithm parent_cert(C) is a simple algorithm. It uses database infrastructure routines to find the set of database entries in the certificate table that have a subject equal to the issuer of C, and, from within that set, the subset of entries that have an SKI equal to the AKI of C. All such certificates are also checked to ensure that they have the valid bit set, and are not included if they do not. If parent_cert() returns zero matches at any point in this algorithm, then validation is deferred, as explained below under verify_children(). If a complete chain of certificates is found that terminates at a trust anchor, then OpenSSL is asked to perform the final validation, and its status returned.

The cert_revoked() subalgorithm checks to see if there are any valid CRLs that are currently revoking this certificate. It does this by querying the database for any CRL with the same AKI and issuer as the certificate. For each such CRL, it checks to see if any serial number in its list matches the certificate's serial number, and if so concludes that the certificate is revoked.

If verify_cert() returns TRUE and also indicates that it found a complete chain to a trust anchor, the FLAG_VALID bit is set in the flags field; if a complete chain was not found then the FLAG_NOCHAIN bit is set instead. At this point the add_cert_internal() subalgorithm is called. This subalgorithm uses database infrastructure routines to construct the appropriate SQL statement that will insert the information for this certificate into the certificate table.

The verify_children() algorithm attempts to verify all those objects whose validity was previously undetermined because this certificate was previously not available and hence there was no chain to a trust anchor. First, it attempts to verify all ROAs for which this certificate is the parent. For each ROA, if it is verified, then the flag is set to indicate that it is valid; otherwise, it is deleted from the database. Next, it attempts to verify all CRLs that are children.

If the CRL is not verified, it is deleted from the database. Otherwise, the valid flag is set, and the whole process by which all the specified certificates are revoked is carried out. For each revoked certificate, the subalgorithm revoke_cert_and_children(), which is described in the garbage collection section, is called in order to invalidate the certificate's descendants as appropriate. Finally, it attempts to verify the certificates that are children, deleting those that fail and setting the valid flag of those that succeed. For all newly valid certificates, the verify_children() procedure is performed recursively.

The add_crl() algorithm performs four steps. Its pseudocode looks like:

```
If not crl2fields() return error;
If not verify_crl() return error;
If not add_crl_internal() return error;
revoke_certs();
```

The subalgorithm crl2fields() uses parts of the OpenSSL library to extract the following fields from the CRL: the issuer, the AKI, the serial number list (as a binary blob), the CRL number, and the thisUpdate and nextUpdate fields. If this extraction fails an error is returned. The extraction could fail because, for example, the putative CRL file is not a valid X509 CRL file, has bad ASN.1, has a missing field that is mandated by the profile, or any number of syntactic failure causes.

The next step is to call the validation algorithm verify_crl(). This subalgorithm uses a combination of OpenSSL and database infrastructure routines in order to perform validation of the CRL. Essentially, once the signing cert has been fetched from the database, and is determined to be valid, OpenSSL routines are used to validate the CRL. It is important to note that in this context the word "valid" as applied to the signing certificate has a more relaxed meaning than it did in the case of the full path validation performed during add_cert(). Since every certificate in the database was fully validated at the time it was added, it is only necessary to ensure that the signing certificate is still valid. In this case a reduced form of verify_cert(), known as verify_cert2(), is called. This function verifies that the certificate is in the database, has not expired and has its valid bit set, and that each of its parents is also in the database, has not expired, has its valid bit set, and that this chain terminates in a trust anchor. (A note on terminology: when the word "parents" is applied to a certificate or ROA it refers to the complete upward chain of parent certificate, grandparent certificate, etc, until the terminal trust anchor certificate.)

If verify_crl() returns TRUE, the FLAG_VALID bit is set in the flags field of the CRL being added and the next subalgorithm in add_crl() is called. This is the add_crl_internal() subalgorithm. This subalgorithm uses database infrastructure routines to construct the appropriate SQL statement that will insert the information for this CRL into the CRL table.

The final step is to do revoke_certs(). For each serial number in the list of serial numbers, it attempts to find a certificate in the database with that serial number and the same AKI and issuer as the CRL. If any such are found, it invokes the subalgorithm revoke_cert_and_children(), which is described in the section on garbage collection, in order to delete the certificate and invalidate its descendants.

The add_roa() algorithm performs three steps. Its pseudocode looks like:

```
If not roa2fields() return error;
If not verify_roa() return error;
Return add_roa_internal()
```

The subalgorithm roa2fields() uses the ROA processing library to extract the following fields from the ROA: the SKI, the IP address information and the AS#. If this extraction fails an error is returned. The extraction could fail because, for example, the putative ROA file is not a valid ROA file, has bad ASN.1, has a missing field that is mandated by the profile, or any number of syntactic failure causes.

The next step is to call the validation algorithm verify_roa(). This subalgorithm uses a combination of ROA library, OpenSSL and database infrastructure routines in order to perform validation of the ROA. This process can be broken down into the following components. First, the signing certificate for the ROA is fetched from the database. If this certificate is not present, then the deferred validation algorithm, described above, will be used; the ROA is immediately entered into the database (using add_roa_internal(), below) and is validated later, as described above under verify_children(). If the signing certificate was found, the IP address information and AS# in the ROA is compared with the IP address information in the signing certificate. If this fails to match an error is returned. Next valid_cert2() is called on the signing certificate to ensure that it has remained valid.

If verify_roa() returns TRUE, the FLAG_VALID bit is set in the flags field of the ROA being added and the final subalgorithm in add_roa() is called. This is the add_roa_internal() subalgorithm. This subalgorithm uses database infrastructure routines to construct the appropriate SQL statement that will insert the information for this ROA into the ROA table.

The delete_object() algorithm is the last of the three top level algorithms used by the rsync client. This algorithm calls the infer_filetype() algorithm to infer the type of object whose deletion is requested. If this algorithm cannot identify the file type, then an error is returned. Next, a lookup is performed on the directory portion of the filename. If this directory is found in the directory table in the RPKI database, its id is returned. If the directory is not found, the delete_object() function is not found, an error is returned. Finally, the database lookup on the pair {filename, directory-id} is performed in the database table that corresponds to the file type. If a match is found, it is deleted using a database infrastructure library call. If no match is found an error is returned.

When a valid certificate is deleted, there is one additional step, which is a call to revoke_cert_and_children(), a subroutine described in the section on garbage collection. This invalidates the certificate's descendants.

2.4 The query client

The query client is a database client program. It performs two types of operations based on command line parameters supplied by the user. It is run manually to generate the ROA output file(s), or to ask generalized questions about the database. The former operation is known as the "comprehensive query." The query client uses the database infrastructure libraries to ask for the set of all ROAs in the database by calling the database infrastructure routine that enumerates the contents of the ROA table. For each ROA in this list, it then must revalidate that ROA using a process similar to the one used by verify_roa(), as described in section 3.

The query client, however, has different requirements than the rsync client. In particular, it must keep track of two bits in the flags field of each of the parents of this ROA, namely the valid bit and the unknown bit. Therefore it must use a different algorithm, verify_roa2(). This algorithm is almost identical to the algorithm used for verify_roa(), except this algorithm takes note if any certificate encountered during its path traversal had its unknown bit set. (The unknown bit is set during garbage collection, as described below.) The complete set of ROAs is thus subdivided into three distinct subsets: invalid ROAs, ROAs for which the unknown bit was not set anywhere in the traversal and ROAs for which the unknown bit was set somewhere in the traversal.

Invalid ROAs are discarded. This action is logged with a reason code so that users can subsequently examine the logfile to determine which ROAs were deemed invalid and why. The contents of the second subset, the unambiguously valid ROAs, are presented to a subalgorithm print_roa() from the ROA library, to be described shortly. The contents of the third subset (the "ambiguous ROAs") are processed conditionally based on command line arguments to the query client. The user may make one of three choices: process the ambiguous ROAs as if they were valid; process the ambiguous ROAs as if they were invalid, or process the ambiguous ROAs specially.

In the first case, the set of ambiguous ROAs is also presented to the print_roa() algorithm. In the second case, the set of ambiguous ROAs is discarded (and logged). In the third case, the set of ambiguous ROAs is presented to print_roa() but with an alternate output filename.

The print_roa() algorithm converts the contents of a ROA into human readable ASCII text. The intent of this routine is to provide a simple output format containing one or more lines of output for each ROA. Each line contains the AS# and one of the IP address blocks contained within the ROA. This implies that a single ROA can produce one or more lines of output. The print_roa() algorithm accepts an argument specifying the output file to which the ROA contents are sent. There are at most two such files. The nominal output file is always present and contains at least the display form of all the unambiguous ROAs. The secondary output file may be present if the user requested special handling for ambiguous ROAs. If present, it contains the display form for all the ambiguous ROAs.

The query client can also be used to ask the database simple questions. Questions such as "what objects are in the database with FIELD=X", where "FIELD" and "X" are both supplied on the command line, can be asked. The user can thus use the query client to determine the set of all certificates in the DB that have been issued by a particular issuer, the set of all CRLs whose nextUpdate field has passed, and other such structured queries. The database infrastructure libraries are used for this activity.

2.5 The garbage collection client

The garbage collection (GC) client is a database client program. This client is a daemon program that typically runs in the background and performs its actions without user intervention. The GC client has three responsibilities: certificate validity interval checking, certificate expiration processing, and CRL staleness checking.

Certificate validity interval checking uses the algorithm certificate_validity() from the database infrastructure libraries. This algorithm traverses the database and examines the validity interval dates on every certificate. There are three possible outcomes of this interval check: the certificate appears to be ok; the certificate has a notBefore date in the future, and the certificate has a notAfter date in the past, i.e. it has expired.

If the certificate's validity dates are valid, then the subalgorithm certmaybeok() is called. This extremely simple algorithm merely checks to see if the FALG_NOTYET bit is set in the certificate's flags field. If it is set, then it clears this flag and sets the valid flag.

If a certificate has somehow been entered into the database but is not yet valid, the subalgorithm certtoonew() is called. This subalgorithm clears the validity bit in the flags field of the certificate and sets its FLAG_NOTYET bit.

If a certificate has expired, the GC client calls the subalgorithm certtooold(). This subalgorithm immediately calls another subalgorithm, revoke_cert_and_children(). The revoke_cert_and_children() subalgorithm performs the following actions. First, it deletes the expired certificate from the database. Next it iterates (recursively) over all the children of that certificate. For each child it calls the subalgorithm countvalidparents() on the certificate. This subalgorithm is very similar to the parent_cert() algorithm described previously, except that instead of retrieving the parent certificate, it returns a count of the number of possible certificates that could be valid parents of the given certificate. If this returned count is greater than zero then revoke_cert_and_children() does nothing more to that child (or any of its descendents). The purpose of this test is to handle the case where a certificate expired, but one or more of the children of that certificate were reparented. This typically occurs when two CA certificates are issued with overlapping validity dates, have the same subject and issuer and contain the same key.

If countvalidparents() returns zero on a certificate, that certificate is deleted and revoke_cert_and_children() is called on all of its children. Thus, the impact is that whenever a certificate expires it is deleted, and all of its children (and grandchildren, etc) are also deleted unless they have been reparented. Note also that this recursive deletion operation not only applies to entries in the certificate table, it also applies to entries in the ROA table. If the certificate that signed a ROA has been deleted, the ROA must be checked to see if it has been reparented by another certificate; if not, it is deleted too.

Certificate revocation processing is primarily handled during initial loading of the data by the rcli client. As a safeguard, this is done again by the algorithm iterate_crl(). In this algorithm the GC client iterates over all CRLs in the database, calling a helper function for each CRL that is found and that has its valid bit set. This helper function extracts the issuer, AKI and serial number (SN) list of its CRL. For each triple {issuer, AKI, SN} it attempts to find a matching certificate in the database. If any such are found the revoke_cert_and_children() algorithm is then called on the matching certificate. As just described, this will delete the revoked certificate itself and then perform the reparenting check recursively on all its children (and grandchildren, etc) – both certificate children and ROA children.

CRL staleness checking is performed by the algorithm stale_crl(). In this algorithm the GC client again iterates over all CRLs in the database. If a CRL has a nextUpdate field that is in the past the CRL is said to be "stale." If a CRL is stale, then all certificates to which it might apply, that is all certificates that match its {issuer, AKI} fields, are marked as "unknown" by setting their FLAG_UNKNOWN bit. Note that this operation does not clear the valid bit on these certificates. Conversely, if a CRL is not stale, then all certificates to which it might apply are checked to see if they do have the unknown bit set. If so, this bit is cleared.

The GC client may also be invoked manually, on the command line, to perform an immediate garbage collection.

The GC client makes heavy use of the database infrastructure libraries.

2.6 The chaser client

The chaser client is a database client program. The purpose of the chaser client is to gather a set of URIs that rsync might not currently know about and tell rsync to also synchronize the repository with those URIs in addition to the ones it is already using for synchronization. The reason this needs to be done is twofold: to synchronize the local repository with newly discovered external repositories, and to refresh the set of CRLs from their distribution points with the intent of eliminating stale CRLs in the DB.

The chaser client uses the following series of steps. It first gathers the values of all non-empty SIA, AIA and CRLDP fields from the certificate table in the DB. In then sorts this list of URIs to eliminate any that are not rsync URIs (this is the second dependency on rsync that was mentioned earlier). It then culls this list to eliminate duplicates. It then examines the set of URIs that it obtained from CRLDPs. If any such URIs refer to a directory, a pattern "/*.crl" is appended to ensure that only CRL files within that directory are fetched. It next further culls this list to eliminate file or directory URIs that are implied by a URI that defines an initial segment. For example, if both rsync://a/b/c and rsync://a/b/c/d/e are on the list the latter is deleted since it will inevitably be fetched by rsync when the former is fetched. This final culled list is then compared against the default set of rsync URIs, stored in rsync's default configuration file (or an alternate configuration file provided on the command line). The chaser client then generates an output file containing those URIs that are on its internally generated list but are not on rsync's list. The chaser client then invokes the rsync wrapper with this new (delta) configuration file as a command line argument.

Note that it is currently possible for the chaser client to fetch the same object more than once. This can occur if the same object is stored in more than one remote repository, and if those objects end up being stored to different destination directories in the local repository. Note that this is not an error, but is inefficient.

2.7 The RPKI database

The RPKI database is a MySQL database that holds pertinent information about managed digital objects. Any object will have been validated before being inserted into the database. Note that the database does not hold the entire contents of any object. Instead, it holds abstracted information about those objects together with information on where the actual underlying object (file) may be found in the repository. A design decision was made to not put the entire object in the database for the sake of efficiency, since a substantial part of each object will not be used by any of the DB clients once that object has actually been entered into the DB.

The database consists of five tables: the certificate table, the CRL table, the ROA table, the directory table and the metadata table. The certificate table contains various fields derived directly from the contents of the certificate, including the subject, issuer, serial number, SKI, AKI, SIA, AIA, CRLDP and the notBefore and notAfter fields. It also contains a binary field (known as a blob) containing the RFC3779 information block extracted from the certificate. It also contains other fields that describe the certificate, including a {filename, directory-id} pair that can be used to locate the certificate file itself; a flags field that contains the valid, nochain, unknown, CA, SS and notyet bits; a unique local identifier that is generated when the certificate is first inserted into the DB; and a timestamp that is also set when the certificate is first inserted.

The CRL table contains the issuer, AKI, thisUpdate, nextUpdate and CRL number fields from the CRL. It also contains a binary blob containing the serial number list from the CRL. It also contains a {filename, directory-id} pair of fields used to locate the CRL file; a unique local identifier field; and a flags field. In the case of a CRL only the valid bit in the flags field is used.

The ROA table contains the SKI and AS# fields from the ROA. It also contains a {filename, directory-id} pair of fields used to locate the ROA file; a unique identifier field and a flags field. In the case of a ROA only the valid bit in the flags field is used.

The directory table contains directory-name and directory-id fields. This table is used in combination with any of the three previously described tables. Its presence is based on efficiency considerations. Any digital object must be associated with exactly one file in the repository. This file has a unique fully qualified pathname consisting of a directory and a filename. It may be desirable to search in the database based on the full pathname; however SQL searches based on long strings are not efficient. It is more efficient, therefore, to separate the full pathname into a filename and directory-identifier pair, and put the mapping between directory-identifiers and directory names in a separate table. It might seem counterintuitive that this is more efficient, since it would appear that two table lookups (and a string concatenation) are required in order to derive the full pathname, but in fact in SQL this type of indirect lookup can be directly supported in a single query.

The metadata table contains information about the RPKI database itself. It contains information about when each of the clients was last run; the name of the top level repository directory; and the maximum values of the unique identifiers used in each of the three main tables.

A complete definition of the database tables in SQL syntax can be found in the file "scmmain.h".

2.8 The database infrastructure libraries

The RPKI database is accessed via SQL statements. A large part of SQL programming involves a fairly repetitive set of operations that have a significant amount of common code. In addition, SQL programming through a standard interface library, a so-called Open DataBase Connector (ODBC), is a fairly specialized knowledge domain. For all these reasons it was deemed good programming practice to encapsulate access to the database through a set of interface abstractions. As a result, two database libraries were created that all four clients use. The lower level library is known as sqcon and the higher level library is known as sqhl.

The sqcon library contains abstractions for common database operations. Thus, it contains functions that insert entries in tables, delete entries from tables based on matching criteria, search for entries based on matching criteria, and so forth. This layer also contains functions for manipulating the metadata associated with table entries, such as finding the next available unique id for a row that is about to be inserted, and also manipulating the flags field of an entry.

The sqhl library contains the implementations of the actual algorithms used by the clients. For example, all the algorithms contained within sections 3-6 of this document are contained in the sqhl library.

2.9 The ROA processing library

The ROA processing library is a collection of functions that performs three tasks: generation of ROAs from an ASCII configuration file; parsing and validation of ROAs; and generation of the ROA output format from a validated (or declared-to-be-valid) ROA.

The ROA generating functions take an ASCII input file similar in form to an OpenSSL configuration file, validate the contents of that file, and generate a ROA as output. ROA generation is primarily a test tool for the RPKI.

The ROA parsing and validation functions must perform for ROAs the same set of operations that the OpenSSL library performs for certificates and CRLs. That is, these functions must verify that the information in the putative ROA file is valid ASN.1; that this ASN.1 conforms to the CMS definition of a ROA; that the signing certificate can be located in the DB based on the SKI found in the ROA (and therefore implicitly that this certificate is itself valid); that the ROA's signature is valid; and that the IP address and AS# in the ROA matches the corresponding information in the signing certificate. The interface to the ROA library was specified to be as independent of the DB infrastructure libraries as possible; some dependencies must exist, however, in order to fetch the signing certificate and perform signature verification.

The final task of the ROA processing library is to supply an output function that converts the IP address information and AS# information in a ROA into a user-friendly output format. At a very early stage it was decided not to support direct output in standard BGP filter syntax, as this syntax is somewhat complex. Instead, a decision was made to output the ROA information in a "lowest common denominator" format that each individual user installation could postprocess as it desired. The format chosen was ASCII text. Each ROA processed during a comprehensive query will result in one or more lines in one of the output file(s) generated by the query client. (Recall that the query client can be directed to generate either one or two output files – the “standard” output file for fully validated ROAs, and the “special” output file for ROAs that have certificates with the unknown bit in their validation path). Each line will contain an AS#, an IP address block, and an identifier for the ROA (currently, the SKI of the ROA).

Appendix C -- Local Repository – User Documentation and Installation Guide

1 Organization of This Document

This document describes how to install and use the BBN Address Space PKI database/repository software. Please refer to the design document, apkidesign.doc, for a detailed description of the various software components in this release. Section 2 of this document describes how to install that software, including all its dependencies. Section 3 provides a quick-start guide on how to get your system running. Section 4 describes the other documents in the deliverable package. Finally, Appendix C.1 provides troubleshooting tips for some common problems.

2 Installation

2.1 Prerequisites

2.1.1 OpenSSL

Install the latest released version of the OpenSSL package from http://www.openssl.org. It should be at least version 0.9.8d. Follow the installation instructions for that package. We recommend the default installation target directory /usr/local/ssl. Also, make sure that RFC3779 support is enabled (./config enable-rfc3779).

2.1.2 MySQL

Install the MySQL open source database from http://mysql.org. We recommend the default installation target of /usr/local/mysql. At least version 5.0 should be used; this is the version that our testing has used. Also install the Connector/ODBC driver from the same site. Insure that at least version 3.51 of the driver is installed. We strongly recommend that you give the root account a password in the ODBC initialization file (typically this is /usr/local/etc/odbc.ini).

Depending on your machine you may also need to install unixODBC as well. If you get a variety of link errors in the "proto" directory this is probably the problem.

2.1.3 cryptlib

Install Peter Gutmann's cryptlib from its default website, currently at http://www.cs.auckland.ac.nz/~pgut001/cryptlib. Insure that both the library itself and its include files are in accessible locations, e.g, /usr/local/lib and /usr/local/include. If you are on FreeBSD this can be done by issuing the command make search name="cryptlib" from the /etc/ports directory.

2.1.4 rsync

Your rsync should be at least version 2.6.9, as earlier versions do not necessarily support the necessary flags. Check your version with rsync --version, and install from rsync.samba.org.

2.2 RPKI database software

The RPKI software will have been delivered to you as a gzip-ed tarball named APKI.tgz. Gunzip using the command:

```
gunzip APKI.tgz
```

and then un-tar the resulting tarball using the command

```
tar xvpf APKI.tar
```

This will create a directory named RPKI which will contain various subdirectories.

3 Running the RPKI Software

The following are the steps required to run the RPKI software. Note that steps 1-4 only need to be executed once at the beginning if everything goes completely as planned. Steps 5-8 have to be run periodically even under ideal circumstances. It is recommended that the user create a script to run these four steps and have a cron job that executes this script periodically. All scripts for steps 3-8 are in the run_scripts directory.

3.1 make the executables

In the top-level directory, execute make on the appropriate makefiles; currently Makefile.linux24 (for Linux 2.4 kernels), Makefile.linux (for Linux 2.6 kernels) and Makefile.bsd are the only three variants available. This should create all the required executables in all the various directories.

3.2 set environment variables

There are three environment variables used throughout the scripts and the code:

- APKI_DB - the name of the MySQL database used to hold the RPKI data
- APKI_PORT - the port number used for the loader and feeder to talk
- APKI_ROOT - the full pathname of this directory (the top-level directory of the RPKI tree)

These environment variables will be set automatically to the following defaults by each of the scripts in the run_scripts directory:

- APKI_DB = apki
- APKI_PORT = 7344
- APKI_ROOT = current working directory, or one higher if in directory run_scripts

If you ever want to run the code directly instead of running the scripts, make sure to set these yourself.

3.3 initDB.sh

This script deletes the database, if any, with the name APKI_DB and sets up a new database with all the right tables. Be prepared to enter the root MySQL password twice.

3.4 loader.sh

This starts the process that receives data from the feeder and loads it into the database. Under ideal circumstances, this will continue to run forever, waiting for inputs from the feeder.

3.5 pull_and_feed.sh

pull_and_feed.sh is a script that optionally pulls the data from a set of repositories using rsync and then optionally loads them into the database by feeding them to the loader. pull_and_feed.sh takes a single argument which is the name of a configuration file. A sample configuration file called rsync_mock.config is included. The configuration file tells the name of all remote repositories to be downloaded, the top-level directory for the local repository (with the REPOSITORY directory included here the suggested one), to top-level directory for storing the log files of the downloads (with LOGS the suggested directory), whether or not to do the repository pull, and whether or not to do the database feed.

Note that you normally want to do the pull and the feed together. Only under special circumstances, such as an aborted feed, would you ever want to do one operation and not the other.

3.6 garbage.sh

This checks for certificates that may have expired due to the passage of time, or crls that may have become stale due to the passage of time, and takes the appropriate actions. Any children certificates or ROAs of an expired certificate that are not reparented are marked as invalid, with this propagating down signing chains. Certificates, and their descendant certificates and ROAs, that might potentially be revoked by the updated version of a stale crl are set to an intermediate state of validity (called "unknown"), but are set back to valid if a current crl is available.

3.7 chaser.sh

This chases down objects from repositories that may not have been loaded but are required. It checks the AIA, SIA and CRLDP fields of all certificates that arrived or have been modified since the last time this client was executed. A list of other repositories to check is compiled.

In the mode where the chaser automatically downloads all the other repositories, it requires only one argument, the full pathname of the configuration file used by rsync_pull.sh.

Note that this can potentially download large numbers of duplicate copies of objects due to caching. Therefore, there is a mode where you can execute this client to see what it would do without actually doing it. You specify this mode by providing a second argument "noexec" or "NOEXEC". It will print the name of the command it would execute along with creating a file chaser_rsync.config that is the configuration file it would feed to rsync_pull.sh. You can then view the file and modify it as appropriate.

3.8 query.sh

This is the way to pull information out of the database and local repository. There are two basic modes.

The comprehensive query, indicated by the argument "-a", pulls all the valid ROAs from the repository and outputs a set of BGP filters specified by these ROAs. To send the output to a file rather than the screen, include "-o <filename>" on the command line or just redirect the output. Those ROAs whose trust chain includes a certificate whose CRL is currently stale are in a state that is between valid and invalid that we call "unknown", and by default we send the filters from these ROAs instead to a file called unknown.out. To instead include them with the fully valid ROAs, include "-u valid" on the command line, and to completely ignore them, include "-u invalid" on the command line.

The informational query provides the user with a means to see what is in the database without directly executing MySQL commands. The basic informational query has the form ./query.sh -t <type> -d <field> [-d <field> ... -d <field> where type is either cert, crl or ROA, and the fields are all the fields of the objects to display. There are different possible fields for each object type, and the full list of possibilities is obtained with the command "query.sh -l <type>". Most of these fields also provide the capability for filtering the results based on a simple comparison. To do such filtering, add as many "-f <field>.<op>.<value>" arguments to the command line as desired, where op is one of (eq, ne, lt, gt, le, ge) and value is unquoted and contains # characters to replaces spaces. Additional arguments are given if you type ./query.sh without any arguments.

3.9 IMPORTANT NOTE

Aborting any of the scripts that change the database or repository in the middle of operation can leave the database in an inconsistent state. It is recommended that after any such abort, you re-initialize the database, clear the repository, and reload the data from scratch. In the future, we plan to provide less drastic means of recovering from an abort.

4 Document resources

api.txt	API for the database, ASN.1 and ROA libraries
testplan.doc	Test plan
apkidesign.doc	Top level design document
codereview.doc	Code review standards
apkiaur.txt	rsync_aur/rcli socket communication protocol

Appendix C.1. Troubleshooting

C.1.1 Database troubleshooting

If you are having database errors, particularly when you first attempt to use rcli, it is important to check and make sure that your MySQL installation is correct. Try the following command:

```
mysql --user=mysql
```

This should connect you to the database and give a "mysql> " prompt back. If it does not, then your MySQL and/or ODBC installation is not correct. Check the troubleshooting section of the MySQL documentation. Verify that your ODBC information, in /usr/local/etc/odbc.ini and /usr/local/etc/odbcinst.ini, is correct. Verify that the MySQL daemon process mysqld is running using "ps -ef". Verify that there is a mysql socket named mysql.sock, typically in /tmp.

Once you can successfully connect to the database as user "mysql" try connecting as root by issuing the command:

```
mysql --user=root --password=PWD
```

where PWD is the root password specified in your odbc.ini file. If this does not succeed, follow the steps given in the MySQL manual for resetting the root password, then stop and restart the mysqld process.

Appendix D – Certificate and ROA Validation Software – Unit Testing

This appendix describes the unit testing to be performed on the certificate and ROA validation software of the Address Space PKI contract. The goal of unit testing (aka single point testing) is to test low level functionality of the RPKI software in a comprehensive manner. Each unit test has been designed to test a single feature of the software in a manner orthogonal to other features. A separate description of the system and performance testing is included in Appendix E.

For purposes of the ROA PKI, single point testing is defined to be verification of whether the system properly interprets and handles single input files. Legitimate files include route origination authorizations (ROAs), certificates, and certificate revocation lists (CRLs) which are syntactically correct and conform to the proper profile.

The single point test procedure will be conducted as follows:

A fully-compliant and syntactically correct ROA, certificate, and CRL will be selected from a set of sample data or manufactured from scratch. Each of these will then be modified in the following ways:

- A number of legitimate variations in content (as appropriate to the particular file) and will be created. These are intended to pass all verification tests to ensure that different legal options and variations are recognized.
- For each file type, one or more incorrect samples will be created to test the syntax parsing and content verification of each field within the file. These errors will include incorrect OIDs, illegitimate content, missing and duplicated fields, incorrect signatures, etc.
- For each file type, one or more samples will be created that will be syntactically correct (relative to the normative RFCs, in particular RFC3280), but do not adhere to the PKI profile. These are intended to be rejected by the system even if verified to be syntactically valid.

A complete list of all the file variations (both correct and incorrect) is available in the next section.

Once the sample set has been created, all the files will be input into the system for checking. The tester will be responsible for verifying that the system accepts all the correct variations and rejects all malformed and non-compliant versions.

List of Single Point Failure Cases

```
General guide to bad certs:
```

```
Type 1: Missing Fields
  Null file
  Critical bits not set

Type 2: Extra/Duplicate Fields
  Various repeated fields, some the same, some different
  Critical bits set where they are not supposed to be

Type 3: Corrupted/bad Fields
  oid's incorrect
  Bad signatures
  Bad IP addresses
  Bad AS numbers
  Invalid dates

Type 4: Out of Order Fields
  IP addresses not in order
  AS numbers not in order

Original good cert: FLXmgA2Ff9X7hUPpD9NIPosEoE
Additional EE cert: BghcBcD2mv4nKeK_dd9MnRWJKK4

 1.1: Zero length file

 1.2: First ID (issuer) missing
 1.3: Second ID (subject) missing
 1.4: Date missing
 1.5: PK missing
 1.6: Basic Constraints field missing
 1.7: CA boolean set/not set (for non-CA/CA)

 1.8: Subject key ID field missing
 1.9: Authority key ID field missing for not self-signed cert
 1.10: Key usage field missing
 1.11: keyCertSign/crlSign bits missing (for CA cert with no other bits set)
```

```
1.12: digitalSignature bit missing (for EE cert)
1.13: Certificate policies field missing
1.14: CRL Dist point field missing
1.15: Auth info access field missing
1.16: Subj info access field missing
1.17: IP Addresses and AS number fields BOTH missing

1.18: Basic Constraints crit flag missing
1.19: Key usage crit flag missing
1.20: Certificate policies crit flag missing
1.21: IP Addresses/AS numbers crit flag missing


2.1: First ID (issuer) duplicated
2.2: Second ID (subject) duplicated
2.3: Date duplicated
2.4: PK info duplicated
2.5: Basic Constraints field duplicated
2.6: Subject key ID field duplicated
2.7: Authority key ID field duplicated
2.8: Key usage field duplicated
2.9: Certificate policies field duplicated
2.10: CRL Dist point field duplicated
2.11: Auth info access field duplicated
2.12: Subj info access field duplicated
2.13: IP Addresses or AS number field duplicated

2.14: Subject key ID critical bit set
2.15: Auth key ID critical bit set
2.16: CRL dist point critical bit set
2.17: Auth info access critical bit set
2.18: Subj info access critical bit set

2.19: Auth key ID subfield (authorityCertIssuer) present
2.20: Auth key ID subfield (authorityCertSerialNumber) present
```

```
2.21: CRL dist point subfield (Reasons) present
2.22: CRL dist point subfield (CRLIssuer) present

3.1: 1.2.840.113549.1.1.11 (PK algorithm) changed
3.2: 2.5.4.3 (issuer) changed
3.3: End date same as start date (expired)
3.4: End date before start date
3.5: Date before 2049 encoded as Generalized Time (should be UTC) (RFC3280)
3.6: Date in 2050 or later encoded as UTC (should be generalized)
3.7: 2.5.4.3 (subject) changed
3.8: 1.2.840.113549.1.1.1 (PK alg not RSA) changed
3.9: PK changed (short/long)
3.10: 2.5.29.19 (basic constr) changed
3.11: 2.5.29.14 (subj key ID) changed
3.12: 2.5.29.35 (auth key ID) changed
3.13: 2.5.29.15 (key usage) changed
3.14: 2.5.29.32 (cert policies) changed
3.15: 2.5.29.31 (CRL dist point) changed
3.16: 1.3.6.1.5.5.7.1.1 (auth info access) changed
3.17: 1.3.6.1.5.5.7.48.2 (auth info access data) changed
3.18: 1.3.6.1.5.5.7.1.11 (subj info access) changed
3.19: 1.3.6.1.5.5.7.48.5 (subj info access data) changed
3.20: 1.3.6.1.5.5.7.1.7 (ip addr) changed
3.21: 1.3.6.1.5.5.7.1.8 (AS num) changed

3.22: Cert version not 3 (i.e. != 0x02)
3.23: IP addresses with invalid values
3.24: AS numbers with invalid values

4.1: CRL dist has no rsync site
4.2: CRL dist uses "RSYNC"
4.3: Auth info access has no rsync siet
4.4: Auth info access uses "RSYNC"
4.5: Subj info access has no rsync site
4.6: Subj info access uses "RSYNC"
```

```
 4.7: IP addresses not in order
 4.8: AS numbers not in order

--------
ROA: mytest

 1.1: Empty file (pem header/footer only)
 1.2: Missing ContentType field
 1.3: Missing CMSVersion field
 1.4: Missing DigestAlgorithmIdentifiers field
 1.5: Missing EncapsulatedContentInfo field
 1.6: Missing SignerInfos field
 1.7: Missing eContentType field

 1.8: Missing SignerInfo version field
 1.9: Missing SignerIdentifier field
 1.10: Missing DigestAlgorithmIdentifier field
 1.11: Missing SignatureAlgorithmIdentifier field
 1.12: Missing SignatureValue field

 2.1: Duplicate ContentType field
 2.2: Duplicate CMSVersion field
 2.3: Duplicate DigestAlgorithmIdentifiers field
 2.4: Duplicate EncapsulatedContentInfo field
 2.5: Duplicate SignerInfos field
 2.6: Duplicate eContentType field
 2.7: Duplicate SignerInfo version field
 2.8: Duplicate SignerIdentifier field
 2.9: Duplicate DigestAlgorithmIdentifier field
 2.10: Duplicate SignatureAlgorithmIdentifier field
 2.11: Duplicate SignatureValue field

 3.1: Bad ContentType OID
 3.2: Bad CMSVersion number (not 3)
 3.3: Bad DigestAlgorithmIdentifiers (not SHA-256)
 3.4: Bad eContentType (not ROA)
 3.5: Bad eContentType info
 3.6: Bad SignerInfo version (not 3)
 3.7: Bad DigestAlgorithmIdentifier (not SHA-256)
 3.8: Bad SignatureAlgorithmIdentifier (not SHA-256 with RSA)
 3.9: Bad SignatureValue

[Not implemented:]
 4.1: Signed Data contents sequence out of order (24 possible variants)
 4.2: eContent contents out of order:
  4.2.1: asID after ipAddrBlocks
  4.2.2: addressFamily after addressesOrRanges
 4.3: signerInfo sequence contents out of order (120 possible variants)


-----------------------------------------------------------------------

CRL: s0Rk925AmjX-pu8WWN9FOXuHz8Q

 1.1: Blank file
 1.2: Missing issuer name
 1.3: Missing issue/next issue date
 1.4: Missing AKI
 1.5: Missing CRL number
 1.6: Missing PK

 2.1: Duplicate issuer
 2.2: Duplicate date
 2.3: Duplicate AKI
 2.4: Duplicate CRL number
 2.5: Duplicate PK

 2.6 CRL number field with crit bit set
```

```
3.1: Version number not 2
3.2: Cert OID corrupt
3.3: Issuer OID corrupt
3.4: AKI OID corrupt
3.5: AKI data corrupt
3.6: CRL number OID corrupt
3.7: CRL number data > 20 bytes
3.8: PK OID corrupt
3.9: PK data corrupt

3.10: Next issue date same as issue date
3.11: End date before start date
3.12: Date before 2049 encoded as Generalized Time (should be UTC) (RFC3280)
3.13: Date in 2050 or later encoded as UTC (should be generalized)
```

Appendix E -- Certificate and ROA Validation Software – System and Performance Testing

This document provides a summary of the goals and results of the system and performance tests as of 2007-06-21. Instructions on how to execute the tests and the outputs from the tests are contained in the directories testing (for system testing) and performance (for performance testing).

1 System Tests

This is a sequence of tests designed to exercise different functionality of the system in situations for which it is easy to tell whether the results are as expected. The data for the tests are

- Tests 1-6: There are 8 certs and 1 crl. One of the certs (call it C1) is a trust anchor, one (C2) is a child of C1, and six (C3-C8) are children of C2. The crl is a child of C2. Until July 31, 2007, all the certificates will be valid, and the signatures should validate. The crl revokes three of the six low-level certs (C6-C8). The crl is expired.
- Test 7: A ROA is added.
- Test 8: The two objects are the ROA and the cert that signed the ROA. Unfortunately, the cert is not self-signed, nor do we have the cert that signed it.

1.1 Test 1

This test examines the ability of certs and CRLs to be verified and CRLs to revoke certs in the process of loading these objects into the database. The test contains a cert (C3) that arrives out of order, loaded prior to its parent (C2). It also contains two certs (C6-C7) that arrive prior to the crl that revokes them and one crl that arrives afterwards.

The outputs verify that five certs made it into the database and are considered valid using the query.

1.2 Test 2

This test examines the ability of garbage collection to set the validity flags of certs to be unknown when the corresponding crl is expired. The three low-level certs still in the database (C3-C5) should be set to unknown state by the garbage collector. The query should handle these three certs depending on the command-line switch.

The outputs verify that the three certs were set to unknown and had their results written to the file unknown.out by the query when so specified on the command line.

1.3 Test 3

This test examines the ability to unset the unknown state flags when the associated crl is no longer expired. For this test, the next_update field of the crl is changed by directly manipulating the database, but in reality a new crl would arrive with a different expiration date. After the garbage collection is run, the states should no longer be unknown.

The outputs verfiy that all five certs are now fully valid.

1.4 Test 4

This test verifies that deleting a high-level cert causes all its descendants to become invalid. Conversely, re-adding the cert causes validity to propagate to its descendants. The trust anchor (C1) is the cert that is deleted and then re-added.

The outputs verify that after deletion of C1 the four remaining certs are all invalid. After re-adding C1, all five certs are once again valid.

1.5 Test 5

This test verifies that garbage collection expires certs whose expiration date has passed and propagates invalidity to all its descendants. The trust anchor (C1) has its expiration date changed in the database directly, and then the garbage collection is run.

The outputs verify that there are only four remaining certs, all of which

are invalid.

1.6 Test 6

This test verifies that when a crl has its verification delayed due to a trust anchor chain not existing when it first arrives, once it is verified it revokes all certs that have arrived prior to its verification. The database is cleared of certs and CRLs, and they are all reloaded, this time in an order such that C2 is the last object loaded, so nothing can be done until after this last cert is loaded.

The results show that there are five valid certs in the database.

1.7 Test 7

This test verifies certain ROA functionality, including garbage collection setting a ROA's state to unknown and back again, the comprehensive query printing out filter entries, and ROAs being invalidated due to the trust chain being broken. The ROA is loaded into the database, its state is changed to valid, and the ski of cert C5 is changed to be the same ski as that of the ROA. Garbage collection is run, which should set the ROA's state to unknown because cert C5 is also unknown. The comprehensive query is run, with unknown going to the file unknown.out. The expiration date on the CRL is changed so the crl is not expired and garbage collection rerun, with another query afterwards that should indicate that the ROA is now fully valid. The cert C5 is deleted from the database, which should invalidate the ROA, and another set of queries is run.

The results show the expected behaviors happening.

1.8 Test 8

This test shows that ROAs are validated whether they arrive in order or out of order. Because the cert that signed the ROA does not have a trust chain, we have to modify the code to do the test. We make it so that the results of the verification test for the cert is always true and hence the cert is always verified. We try the test both when loading the cert and then the ROA, and when loading the ROA before the cert. In both cases, the ROA should be validated.

The results show that the ROA is valid in both cases.

2 Performance Tests

We did two different tests to test the performance. The first works with real data downloaded from synch://apnic.mirin.apnic.net/mock. Whatever data is there is pulled down using rsync and then loaded into the database. Both sections are timed so that we can see the time spent on each operation.

The second test works with 10,000 copies of the ROA used in the system tests. The same ROA is put in 10,000 different files and each are loaded. The signing cert is put in the database prior to the ROAs being loaded.

The results are:

Cert/CRL Test:

- 22633 certs downloaded/loaded
- 10528 CRLs downloaded/loaded
- rsync download time: 169 seconds
- database load time: 757 seconds
- total time: 926 seconds
- memory usage stabilized

ROA Test:

- 10000 ROAs loaded
- database load time: 155 seconds
- memory usage stabilized

Appendix F -- Distributed Root – Background and Procedures

1 Introduction

Many proposals to achieve BGP [BGP] security ([S-BGP], [SOBGP], [PSBGP], [SPV]) call for authorities to issue certificates that attest to address spaces and autonomous system (AS) number holdings by Internet Service Providers (ISPs) and network subscribers. Different proposals have proposed different models for which organizations should act as certification authorities (CAs) for a PKI of this sort. However, ongoing work in the Internet Engineering Task Force (IETF) Secure Inter-Domain Routing (SIDR) Working Group has focused on a single PKI that parallels the extant allocation scheme used to manage address space and AS number assignments [ID-ARCH].

Ideally, a single CA would act as the trust anchor[2] (TA) for a PKI of this sort. A single TA would make life easier for relying parties, e.g., ISPs, when they verify certificates in the PKI. In the context of the address space and AS number PKI (aka Resource PKI or RPKI), the Internet Assigned Numbers Authority (IANA) is the obvious candidate to be the single TA, as it is the top level authority for all allocations of address space and AS numbers. However, political and/or operational considerations may preclude this solution.

An alternate solution involves having each of the five regional internet registries (RIRs), the next tier of allocation authorities below IANA, act as a separate TA. This would require that at least five public keys (one for each RIR) be distributed (via a secure, out-of-band means) to all relying parties. Since relying parties must obtain a TA in order to validate certificates, increasing the number of trust anchors greatly increases the complexity of the system for relying parties.

In order to obtain many of the benefits of a single TA, while allowing the RIRs to retain their autonomy, in this paper we outline procedures to allow multiple players to operate a *virtual* CA as the single TA for the PKI. This virtual CA would have a single public key, making it appear as a single CA to relying parties, but signing would be cooperatively performed by a threshold set of the entities that comprise the virtual CA. The RIRs are operationally suitable to form this virtual CA, e.g., they formed the Numbers Resource Organization (NRO) to act in concert on similar matters. We illustrate the process of creating a signature in Figure 1.

The virtual CA must be:

- **Secure**:

  If the private (signing) key is leaked or otherwise obtained by an adversary, that adversary can sign erroneous data. We refer to such keys as *compromised*. The virtual CA must be at

[2] A trust anchor generally is defined as a public key and associated data used as a starting point for certificate path validation in a PKI.

least as secure as a single CA, including the ability to utilize existing hardware security modules for protection of the secret key. In fact, our scheme for a virtual CA can be considered more secure, as more than one secret key must be compromised in order to break the scheme.

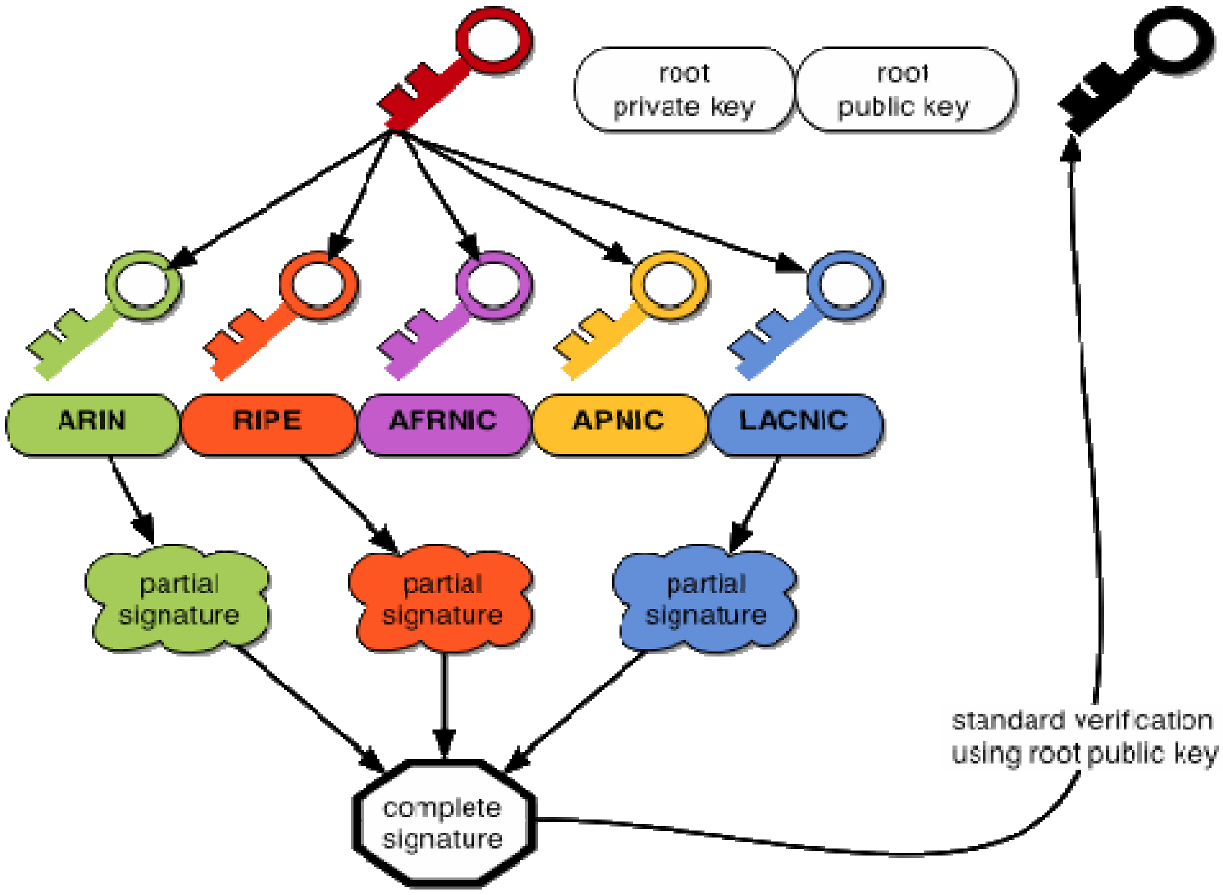


Figure 1: The distributed (root) TA of a PKI functions as follows: (1) a trusted dealer chooses a public and private key pair; (2) the dealer creates shares of the private key for each RIR; (3) to create a signature, at least t =3 RIRs create signature shares from their key shares and combine them into a complete signature; (4) the complete signature can be verified with the TA public key, just as if it were created directly from the TA private key.

- **Invisible**:

  No relying party should need to know the administrative arrangements that underlie the virtual CA. Signatures on certificates are verified using a single public key just as if there was a single physical CA instead of a virtual CA.

- **Efficient**.

  Signing certificates should not be a heavy burden on the members of the root collective. In particular, the amount of interactive communication during signing should be minimal: one player distributes data to be signed and the other players return *signature shares*, which can be combined into a complete signature, valid under the virtual signer's public key. In addition, our scheme for a virtual signing authority can actually increase availability over a single signer, because not all players in the root collective must be available to produce a signature.

2 Concept and Operation of a Distributed TA

The virtual CA signs certificates for its subordinates; in turn, these subordinate CAs sign certificates for their subordinates, etc.[3]. A TA acts as the starting point for certificate path validation within the PKI. If a PKI has one, generally-accepted TA, it is the starting point for all certificate path validation in the PKI. Such a TA is an extremely tempting target for attack. In a scenario where multiple parties represent the TA, the security of the corresponding TA private key can be increased by employing a threshold signature scheme to split this private key among several players. In the context of the RPKI, a quorum of RIR representatives, distributed over the globe, collectively act as the TA by jointly signing certificates to be verified with a single public key (TA).

All k participants representing the TA jointly generate key shares for each participant, such that $t \leq k$ players must collaborate to sign a certificate; no group of fewer than t players can generate a valid signature. (See Section 3 for details.) At any later time, they can generate a signature as follows: any t participants individually construct shares of the signature using their key shares; anyone may then combine these signature shares into a valid signature under the TA key.

Splitting the TA private key in this way protects against up to t-1 participants being compromised or attempting to cheat. In combination with strong physical security, the TA can achieve robust protection of the TA private key while simultaneously distributing control among the participants. To protect against undetected compromise of TA collective participants, we recommend that they refresh their key shares as we discuss in Section 7. Use of a threshold signature scheme can also increase availability, as k-t participants may be unavailable or slow to respond without hindering the signature process.

This paper is a companion to software (see Appendix F.2) that performs key generation, signing, and verification according to Victor Shoup's threshold RSA signature scheme [SHOUP]. In addition, we have added the capability to perform key refresh as we outline in Section 7. Operated according to the procedures described in this document, this code performs the core tasks necessary for a collective of players to operate a virtual signing authority.

[3] Subordinates may issues certificates only if they are permitted to do so by being specified as a CA in their X509 certificate.

3 Overview and Procedures

In order to operate a distributed TA based on threshold signatures, the k participants must first determine t, the number of participants required to create a signature. This must be chosen such that the (expected) maximum number of malicious or compromised participants is always less then t. For our proposal k=5 (since there are 5 RIRs), and a threshold (t) value of 3 seems appropriate. The TA members gather to create a common key pair; each participants gets one share of the secret key. In Section 4, we describe this process, as well as the physical and computational security precautions to be taken.

Each participant returns to his operations center with a share of the TA private (signing) key on removable media, which he loads into the crypto device that will construct signature shares. We discuss some of the issues involved in utilizing secure cryptographic signing devices for this purpose in Section 5.

Any t participants representing the TA can now collaborate to sign a certificate such that it can be verified under the TA. When an RIR wishes to sign a certificate, that RIR sends it to all other RIRs. Each RIR examines the certificate for correct structure and content (e.g., that the address space being allocated has been assigned to the RIR in question by IANA). Each signing RIR then uses its secret share of the TA private key to produce a signature share, then sends this share to the requesting RIR. Once the requesting RIR has gathered valid signature shares from at least t distinct RIR, the requesting RIR combines them into a signature valid under the TA public key. We show how these signature shares are constructed and combined in Section 6, as well as discuss procedures for producing signatures and identifying malicious or compromised RIRs.

Private key shares should be well protected. However, through administrative error or other means it is possible that a private key share might become known to a malicious entity. To protect against such an entity obtaining enough private key shares that it can sign data as the TA, we recommend that the members of the TA collective periodically refresh their private key shares. A refresh of private key shares produces new key shares for each of the participants without requiring a new public key to be issued. This removes the overhead and complexity involved in introducing a new TA unrelated to previously-issued certificates, while maintaining security with respect to known and unknown key share compromise. We discuss several approaches to refreshing key shares in Section 7.

4 How to Generate Keys

The first step in utilizing threshold signatures is to generate the root public key and corresponding private key shares. To ensure the security and integrity of these key shares, all players of the TA collective will physically gather for the key generation ceremony. We propose the security guidelines outlined in this section to prevent the key generation process from leaking private information or being biased in favor of any player.

4.1 Key Generation

Representatives of the RIRs already meet in person several times per year, at RIR meetings in each region. These conferences afford the opportunity for “out-of-band” creation of the TA public key and private key shares. Because the RIR representatives have previously met face-to-face, they can avoid the complex issues of remote identity verification by performing key generation during one of these gatherings. In addition, regular physical meetings provide an attractive setting for periodic key refresh, as outlined in Section 7.

To encourage equality and fair play, the TA collective first must agree upon a trusted ceremony official. The trusted official will operate the software and hardware involved in the key generation ceremony. Prior to the key generation ceremony, each RIR also creates a public/private RSA key pair exclusively for protecting their private key share during the ceremony.

During the ceremony, the trusted official presents the hardware and software to the RIR representatives for inspection. The official then generates the TA public key and the private key shares. To give each representative his RIR's key share, the official collects each RIR's RSA public key on removable media, uses each public key to encrypt one key share, and gives each representative the corresponding encrypted key share and TA public key on removable media. We recommend the use of encryption to protect the key shares; this prevents participants from obtaining access to other players' key shares. The trusted official then wipes all data from the key generation hardware. Details of this step will depend on the hardware selected.

Before concluding the key generation ceremony, the players may wish to verify that the collective can correctly sign data. Each player encrypts his key share and then transports the key share to his respective organization (See Section 4.2).

In our distributed signature scheme, we utilize the Shoup threshold RSA signature scheme, as noted earlier. Key generation for this scheme operates generally as follows:

Input -- Number of players k, number of players required to create a signature t, key length. (Key length should be chosen to create a secure RSA signature; we recommend 2048 bits, consistent with the SIDR certificate profile [HOUSTON].)

Output -- Root public key (e,n), secret key share s_i for each player i ($1 \leq i \leq k$). (In the full scheme, which we do not utilize, players may also receive verification keys v, $v_1, \ldots, v_k$. For discussion of proofs of verification for signature shares, see Section 6.)

Any t of the signature shares together define a polynomial that encodes the root secret key; given at least t of the shares, we can reconstruct this secret key (though we do not need to do so to produce signatures) [SHAMIR].

4.2 Key Transport

Each RIR must protect the confidentiality and integrity of its private key share during transport. It is extremely important that the private key shares not be disclosed either accidentally or to malicious adversaries actively attacking the protection measures. Thus, we recommend that the key shares remain encrypted during transit. (By having each RIR representative arrive with only its public transport key, secure separation between the public and private keys used for transport is easily achieved.) In addition, the key share should not reside on systems connected to public networks. Participants should make use of hardware cryptographic devices to protect both the private transport key and the TA key share.

If a participant's key is potentially compromised, he must inform the root collective of this fact. If more than t key shares are compromised, the TA key must be immediately replaced. If fewer than t keys are compromised, then the root collective can refresh their key shares and continue, without changing the TA key pair.

By refreshing their key shares, the TA collective creates new key shares for all participants without changing the TA public key. Because of the logistical difficulties entailed in distributing a TA key, the TA collective should perform key refresh regularly, e.g., semi-annually, to guard against both known and unknown key compromises.

5 Utilizing Crypto Hardware

For reasons of security and confidence, we recommend that participants employ cryptographic modules for the storage and use of their secret key shares. (Since the virtual CA operated by the RIRs would represent a TA, we recommend use of modules evaluated under FIPS 140-2 at level 3 [FIPS]. Once each participant has received his secret key share, the participant can transfer it to the crypto module that will be used to create signatures (see Section 6 for more information on creating signatures). Specifically, RIR_i ($1 \le i \le k$) who received secret key share s_i at the key generation ceremony (see Section 4), will create standard RSA signatures, using as its key the value $2^{(si)(k!)}$. (In this context k= 5, so k! = 120.)

Interfaces to crypto modules vary. For this reason, we recommend that the devices that are to be used be PKCS 11 compliant. An example of one such device is SafeNet's Luna SA. The RIR can load its signing key share into such a crypto module. (We give specific directions for SafeNet's Luna devices in Appendix F.3.) The player may then create signature shares as standard RSA signatures using the secret key in the crypto module.

6 How to Sign Data

After the RIRs have generated keys as described in Section 4, any t RIRs can jointly sign a certificate.

6.1 Creating signature shares

If one RIR wishes to sign a certificate, he sends it to all other RIRs and requests that they create signature shares for this certificate. (Note that the requesting RIR needs to receive at least t-1 correct responses.) Those RIRs then check the data according to their established security procedures[4].

If an RIR wishes to help create a signature on the certificate, that RIR constructs a signature share. In our code, we utilize Shoup's threshold RSA scheme, so RIR i ($1 \leq i \leq k$) with secret key s_i signs data x to produce signature share x_i as follows:

- $x_n = x \bmod n$
- $x_i = x_n^{2\, si\, k!}$

Note that this is a standard RSA signature calculation with secret key $2^{(si)(k!)}$. If the player is utilizing a crypto module as described in Section 5, then he simply requests that the box produce an RSA signature on data x with key $2^{(si)(k!)}$ to produce the signature share x_i.

6.2 Finding a misbehaving RIR

The TA collective can determine which RIR(s) are misbehaving (or malicious) by combining signature shares as we discuss in this section. This is based on the fact that if some RIR shares combine into a valid signature, the RIRs who created those shares created them correctly.

If all five participants respond then there are ten possible combinations of the private key shares. If there is exactly one misbehaving RIR, then one can create four valid combinations and six invalid combinations. Determining the misbehaving RIR involves identifying the member absent from the valid combinations and verifying that the suspect RIR is common within all invalid combinations. In the case of two misbehaving RIRs, where all participants have responded, there will be only one valid combination and nine invalid combinations. The RIRs not involved in the valid combination are identifiable as the misbehaving RIRs.

We encourage all RIRs to participate and contribute their signature shares even if t participants have already responded. If all RIRs participate, it becomes possible to identify problems or corrupt key shares at an earlier time.

In the full Shoup scheme, each player generates a small, non-interactive, zero-knowledge proof of correctness along with his signature share. These shares do not divulge secret information; they only allow any player to efficiently check that signature shares were constructed correctly. However, in this application, these proofs are not necessary, given the small number of players, and their use would interfere with our goal of employing standard crypto modules to effect the partial signatures.

[4] For example, the RIRs might institute a rule that they will sign an X.509 certificate only if the new allocation in the certificate is vouched for by IANA

6.3 Combining signature shares.

After each RIR produces its signature share, it sends its share to the requesting RIR. Once the requesting RIR has received these shares, he combines them into a joint signature and then checks that the signature is valid under the TA public key. If it is not valid, then one of the signature shares is invalid; the RIR that produced it is misbehaving, or an adversary may have interfered with the transmission of the signature share.

If the joint signature is not valid, the combining RIR should combine other sets of signature shares, as we described above, to discover which RIR (or RIRs) sent an invalid share. Generally, the combination procedure works as follows:

Input:

TA public key (e,n), signature shares x_1, …, x_t from distinct RIRs

i_1, …, i_t, respectively, data to be signed x.

Output:

Signature of x, verifiable under TA public key (e,n).

Essentially, the exponents of the signature shares are points from a polynomial that encodes the root secret key. Using polynomial interpolation, we can combine these shares in such a way that the exponent becomes the signing key. In this way, we produce a signature without revealing the secret key. Cryptographic details and mathematical intuition are discussed in Shoup's paper [SHOUP].

7 Issues Surrounding Refresh

Should it become known that one or more of the key shares (but fewer than the threshold required to perform signing) have become compromised, key refresh should be initiated. One must also take into account the fact that undetected key share compromise might have occurred, creating an unknown or undisclosed compromise scenario. To help protect against both known and unknown compromise we recommend that the RIRs refresh their key shares periodically in addition to the mandatory refresh that would be required in response to a known compromise.

A possible key refresh schedule is two or three times per year, at a subset of the RIR meetings that are attended by representatives from all of the RIRs. Key refresh creates new signing key shares for each RIR that create signatures valid under the same public key. By refreshing their key shares, the RIRs can protect against an adversary silently gaining access to fewer than t shares of the secret key in between refresh intervals. To successfully forge signatures, the adversary must collect at least t key shares that were created together.

To effect key share refresh, the RIRs generate new shares of the secret signing key. Then, they switch to signing with the new shares at an agreed-upon time. It is imperative that each participant destroy their old key shares as soon as possible, as their continued existence creates extra risk. Should an attacker gain access to t key shares created at the same time, the attacker could successfully sign under the existing public key.

The players physically gather and bring their secret keys to the key refresh meeting. Transporting these keys requires security measures similar to the key-transport protocols we describe in Section 4.

The trusted official distributes a public key to all players; only the official knows the corresponding secret key. Each RIR encrypts its key share s_i using this public key and loads it onto removable media, along with the public RSA key (which it generated) that will be required by the share transport procedure.

The RIRs run key refresh software in the same manner as key generation software was run in Section 4. The trusted official loads the encrypted key shares into the device used for the key refresh procedure and decrypts them. The official then utilizes signing share combining code to ensure that every key share is correct. (The procedure used here is the same as described in Section 6.2 for certificate signing.) The trusted official then generates new key shares using the key share generation code, destroys the old key shares, and distributes the new key shares as in Section 4.

8 Conclusion

To increase both availability and security of a single TA for the RPKI, a collection of responsible parties may share the TA signing key between them. They must carefully and jointly generate the key in order to ensure that each gains a secret share of this key. When a certificate must be signed, the requesting RIR sends it to all other RIRs. Those RIRs then decide whether it is a certificate that should be signed. If it is, each RIR creates a signature share from the certificate (really the hash of the certificate) and its secret key share, and returns the signature share to the requesting RIR. The requesting RIR then combines the signature shares into a complete RSA signature valid under the TA public key. In order to guard against the detected or undetected compromise of signature shares, the RIRs should refresh their key shares periodically. Key refresh allows the RIRs to obtain new key shares without changing the public key.

References

[BGP] Rekhter, Y., and Li, T., “A Border Gateway Protocol 4,” RFC 4271, 2006.

[FIPS] “Security Requirements for Cryptographic Modules,” Federal Information Processing Standard 140-2, December, 2003.

[HOUSTON] Houston, G., Michaelson, G., Loomis, R, “A Profile for X.509 PKIX Resource Certificates,” draft-ietf-sidr-res-certs-08.txt, July 2007.

[ID-ARCH]	Lepinski, M., Kent, S., and Barnes, R., “An Infrastructure to Support Secure Internet Routing,” draft-ietf-sidr-arch (work in progress), July 2007.
[PSBGP]	Kranakis, E., van Oorschot, P.C., and Wan, T., “On interdomain routing security and pretty secure BGP (psbgp),” Techical Report TR-05-08, Carleton University, 2005.
[SBGP]	Kent, S., Lynn, C., and Seo, K., “Secure Border Gateway Protocol (SBGP),” IEEE Journal on Selected Areas in Communications, 18(4):58-592, April 2000.
[SHAMIR]	Shamir, A., “How to Share a Secret,” *Communications of the ACM*, 22:612-613, 1979.
[SHOUP]	Shoup, V., “Practical Threshold Signatures,” in *Proceedings of Eurocrypt*, 2000.
[SOBGP]	White, R., “Securing BGP Through Secure Origin BGP,” Internet Protocol Journal, 6 (3), September, 2003.
[SOURCE]	Java threshold signature package project. https://sourceforge.net/projects/threshsig
[SPV]	Hu, Y., Perrig, A., and Sirbu, M., “Spv: Secure Path Vector Routing for Securing BGP,” in *Proceedings of ACM SIGCOMM*, August 2004.

Appendix F.1: Detailed Key Generation Procedures

In this section, we detail a list of requirements and procedures for the trusted ceremony official and the RIR participants.

The trusted official will be provided with the software required to load onto a laptop and perform the key generation in advance of the meeting. Each member will be given copies of the software in advance and will be permitted to verify that the software that the trusted official loads is the same as the software that they were provided for review.

The trusted official provides:

- A generic laptop computer with CD-ROM drive and USB connection[5], but without a hard disk
- Software CD-ROM with the requisite OS loader and software that the trusted official will use and that can be examined by the participants prior to start
- A crypto module such as SafeNet Luna PCMCIA

Each representative from an RIR provides the following:

- A suitably equipped laptop with a CD-ROM drive
- A USB flash drive
- An RSA public key on either a CD-ROM or USB flash drive (this key pair must be generated only for this ceremony; the corresponding private key must be secret)

Once the official and RIR representatives verify that all participants are present and have brought the appropriate resources, setup may commence. All hardware should not be connected to any network, all wireless capabilities should be turned off, and each computer should remain under the control of its owner at all times.

Each representative from an RIR boots his system. They are then provided with the CD-ROM containing the software that the trusted official[6] will be using[7]. The participants are permitted to copy the CD-ROM, examine its contents, and perform any checksum operations they feel necessary to foster a strong belief that the contents of the CD-ROM are correct and consistent with what they had previously been provided.

The official boots his system from the software CD-ROM, which contains a bootable Linux kernel that runs in RAM only. Additionally, the CD-ROM contains the required software for key generation, key share splitting, handling the key shares, encrypting them for transport with the public key supplied by each RIR representative, and sanitization. Sanitization software clears system memory and prevents adversaries from utilizing forensic capabilities to retrieve the private key shares.

The official loads each of the public keys presented by the RIR representatives onto the system. After this is accomplished the trusted official generates an RSA key pair and subsequently splits the RSA private key into key shares. The private key is then destroyed. The key shares, one for each RIR representative, are encrypted using the corresponding representative’s public key. The resulting encrypted key shares and public key are loaded onto their respective USB memory

[5] USB is presented as an example option for removable media. If there are security concerns regarding USB devices, the participants could elect to use one-time recordable CD-ROMs or any other mutually agreed upon media.

[6] The “trusted official” is an independent party trusted by all the RIRs to provide technology for the key generation process. He is not a key holder like the RIR representatives and must not be able to discover the private key or the share generated during this process.

[7] The CD-ROM should be **read-only**, alternatively the trusted official can make multiple copies of the CD-ROM in front of everyone (6 in this example), provide a CD-ROM to each participant and retain the final copy for use.

sticks and returned to the RIR representatives. After the keys have been transferred, the sanitization software is executed to remove any remaining traces of the private key or private key shares from the official's system and the system is powered down.

Appendix F.2: Code Summary

In this section, we give an overview of Java code [SOURCE] for a distributed root based on Shoup's threshold RSA signature scheme.

Key Generation

The trusted official runs the following code on a secure computer to generate key shares for each RIR.

1. Create a new Dealer object to create keys of size keysize.
   d = new Dealer(keysize)
2. Use the Dealer object to generate key shares for k players such that at least t players must collaborate to successfully sign data under the root key.
   d.generateKeys(t, k)
3. Retrieve the public key from the Dealer object. This key can be utilized to verify all valid signatures constructed by the RIRs; it is utilized just as any other RSA public key.
   gk = d.getGroupKey()
4. Retrieve the key shares from the Dealer object. Each RIR's key share is encrypted with a public key supplied by the RIR and then written to a separate piece of media, which is given to the corresponding RIR representative.
   KeyShare[] keys = d.getShares()

Constructing Signature Shares

Given their key share ei and root key modulus n, each player can calculate his signature share as in a standard RSA signature:

```
BigInteger bmod = new BigInteger(b).mod(n)

BigInteger sigshare = bmod.modPow(ei, n)
```

Each of the signing players sends sigshare and their identifier i to the party combining signature shares into a valid signature.

Note that the code supports the full Shoup scheme, including non-interactive zero-knowledge proofs, even though we do not recommend use of this feature, as noted above.

Combining Signature Shares

Given the data to be signed b, an array of t valid signature shares sigs, the number of signers t, and total number of players k, as well as the public key object gk, one combines the signature shares into a signature on b as follows:
SigShare.combine(b,sigsq,k,l,gk.getModulus(),gk.getExponent()).

Verifying Joint Signature

Given the data that was to be signed b, the signature to be checked sig, and the public key e,n, one may verify the signature as with any RSA signature:

sig.modPow(e,n).compareTo(b) == 0.

Appendix F.3: API and Implementation Details

Hardware Key Generation

While use of a hardware module for key generation entails greater logistical and security complexity, such a module can generate cryptographic keys using better entropy sources than is available on most computers. The following directions were provided by SafeNet for SafeNet's Luna line of hardware security modules[8]:

1. Generate the key using the PKCS call

   C_GenerateKeyPair (... CKM_RSAX_9_31_KKEY_PAIR_GEN, ...)
2. Extract the private key from the hardware device.

   C_GenerateKey (... CKM_AES_KEY_GEN, ... pTemplate)

   (This should set the CKA_WRAP and CKA_DECRPYT attributes to true and returns the handle of the AES key used to wrap the RSA key.)
3. C_WrapKey (... CKM_AES_CBC_PAD, wrapping key handle, handle of key to wrap) (This returns the wrapped key blob.)
4. C_DecrypInit (... CKM_AES_CBC_PAD, wrapping key handle)

[8] This information is courtesy of Robert Woodward, Alan Boyd, Mark Yakabuski, and Ben Hanrahan at SafeNet, Inc.

{ C_Decrypt (... wrappedKeyBlob, ... , pData, ...) (pData will now contain the decrypted key in DER-encoded form.)

Note that in order to export a key, the key export bit must be set on the hardware generation device. Once set, this bit cannot be cleared without wiping the device. Thus, we recommend using a separate device from the ones used for signing by the RIRs.

Importing Keys

Players can import a secret key into SafeNet's Luna devices as follows[9]:

1. Generate a 3DES or AES wrapping key on the module with both its CKA_ENCRYPT and CKA_UNWRAP bits set.
2. DER encode (PKCS-8) the RSA key and encrypt it (C_Encrypt) on the module with the wrapping key; this call returns the encrypted blob.
3. Unwrap via C_Unwrap the encrypted blob on the module using the same wrapping key; this call returns the key handle of the imported RSA key

The public key may be imported into the crypto module using the C_CreateObject call from PKCS-11.

From this point onward, the player may create signature shares as standard RSA signatures using the secret key in the crypto module.

[9] This information is courtesy of Robert Woodward, Alan Boyd, Mark Yakabuski, and Ben Hanrahan at SafeNet, Inc.

Appendix G – Security Code Review

This appendix describes the security code review that was done by Peiter Zatko (aka "Mudge"), a Division Scientist and Technical Director at BBN. He is a globally-recognized expert in security analysis and design of complex networked systems. He has developed advanced models for network data traffic analysis and interpretation; created models of attack scenarios for various life cycles in wireless networking schemas, developed models of selective and broad-scale jamming techniques, and created scenarios for a new class of attack not currently discussed in existing research.

1 Introduction / Overview

From the beginning of the work embarked upon for the RPKI Repository, it was determined that a security code review was to be included as one of the project steps. The rationale for this is straightforward and simple but all too often overlooked, even in security relevant programs.

Frequently the end user of an application or service is in the situation of needing or wanting security to such an extent that they will require new tools and solutions to provide this security for them. Such is the case for ensuring the validity of ownership for autonomous system (AS) numbers and the address ranges an organization is permitted to advertise. However, if the user is not able to perform a security analysis of the source code for this new tool, or if the solution is provided without source, then the user is left trusting that the tool or solution: a) solves the problem they are attempting to address, and b) does not introduce new security problems with the introduction of new code. Sadly, case "b" is often incorrect.[10] [11] [12]

If the program is being released with access to the source code, code sections will commonly be lifted and reused in other applications. Thus, it behooves the developers to consider secure coding guidelines in their work so as to minimize the propagation of errors that might exist in the initial code base to new projects.

It is for these reasons that BBN decided to include some form of security source code review of the RPKI Repository program. What follows is a description of what this analysis included and, just as importantly, what this analysis did not address.

[10] Kerberos4 – L0pht Security Advisory Nov 22, 1996. http:// http://attrition.org/security/advisory/l0pht/l0pht.96-11-22.kerberos

[11] DataLynx suGuard – L0pht Security Advisory Jan 3, 1999. http://attrition.org/security/advisory/l0pht/l0pht.99-01-03.suguard

[12] "For the fifth time in two months, security researchers have publicized a serious flaw in a widely used virus-scanning program.". http://news.zdnet.co.uk/security/0,1000000189,39191831,00.htm

1.1 Security Analysis

As with any subcomponent within a larger project or activity, one must determine what the correct level of effort for the subcomponent should be in order to maximize the value to the overall project without adversely affecting the project at large. Security audits are no exception.

As this project is in many ways a prototype and initial reference base, a full security source code review would be cost prohibitive in both time and finances. Conversely, not performing a security analysis would be remiss. If the code is to be referenced by others then it needs to show a security conscientious approach to coding to encourage others to engage in similar practices and to increase trust in the underlying code.

From these constraints the appropriate level of effort has been determined to be an automated lexical security audit followed by human analysis of the key areas identified.

Automated Lexical Analysis

For the automated analysis, tools previously authored by Mudge were used. These tools take C source and header files as input and parse the contents looking for a number of trigger code segments. These code segments are either commonly misused system function calls, common library calls that have security risks associated with them, or text that has been discovered empirically to be involved with or within close proximity to security coding errors of varying types. Over ninety (90) functions are identified as being security risk areas within the automated analysis tool and around twenty (20) tags endemic to security issues are also included.

The goal of the automated analysis tools is not to determine which functions and routines are secure and which ones are not. Rather the goal is to identify routines that are commonly insecure and routines that are designed to be secure but frequently used in insecure fashions. By identifying these areas that have a higher probability of containing security vulnerabilities, the amount of code that a human needs to manually inspect is greatly reduced (in comparison to the total code base of the project). As the automated tools are designed to err on the side of identifying a source file and code fragment and displaying it even if there might be other mechanisms in place to protect the code from being exploited, it is hoped that the tools provide too much information rather than omitting important data.

Automated analysis was performed on 35,790 lines found within ninety-six (96) files of the RPKI Repository code. From this 2,619 lines of code were highlighted as being potential security concerns.

Human Analysis

With the identified lines of code from the parsed files, the next step involved a human looking not only at the lines identified, but also tracing input and output variables that interact with or are created by the suspect code fragments and examining the code surrounding these suspect sections.

The human is then able to determine which code segments are actually exploitable and which are not. It should be noted that even if the code segment is not exploitable, good practices would encourage that the code still be modified to make it blatantly obvious that this is not the case.

Such activities show the reader of the code that the authors are performing security conscious coding practices and eases future security code auditing activities.

While this effort does not embody full code coverage (and further it does not guarantee that all security problems have been brought to light), it does a responsible job of identifying common errors. The result is an identification of a substantial amount of vulnerabilities that would require either low or moderate sophistication to exploit. This was the effort deemed appropriate given the project environment and constraints.

Highlights and Summary

Unbounded copies

The vast majority of errors identified were unbounded string copies. These were often operating on addresses located on the program stack or heap. These included standard vulnerable unbounded calls such as strcpy, sprintf, strcat, and others of similar nature. Some of these were exploitable whereas the majority were not. It was recommended that all of these be replaced with the bounded version of the function in question (e.g. strncpy, snprintf, strncat, etc.), even if the code in question was not obviously exploitable.

Bounded copies

In a few locations bounded string copies were found to be suspect and potentially vulnerable due to incorrect termination tests and improper length specifications. In the case of incorrect termination tests the majority were found to be within loops utilizing pointer arithmetic under the assumption that a marker value would be contained within the source data. In the event that the marker value was not found there was no subsequent test to ensure that the destination memory address would not be overrun. For situations where improper length specifications were found these took the form of inadvertent specification of either the size of the source data to be used as the maximum value to place in the destination data location, or a specification of external sizes (often in #defines) where the size was greater than the allocated memory in the destination referenced for storage. The latter being a somewhat common coding area where a function such as strncpy(dest, src, len) has the len value specified as the sizeof(src) rather than the sizeof(dst). These situations were pointed out to the appropriate authors for remediation.

Known Problematic Functions

A small number of errors were identified where function calls that previously had been considered safe but later found not to be so were used. Examples of such calls are mktemp(3) and stat(2). In the case of mktemp(3), the filename that will be returned by the call can be guessed. As the filename produced is only guaranteed not to exist at the point of creation of the string, a race condition exists that can oftentimes lead to varying degrees of system exploitation depending upon the program purpose and context. Good security practice encourages the use of mkstemp(3) in place of mktemp(3). For stat(2) a similar race condition exists in that the values returned from the call are only valid at the exact time that the call is made and may not be correct when an action is later taken on the file in question. If possible, it is recommended security practice to use fstat(2) in place of stat(2).

Format String Attacks

There was one case where input that could be affected by external sources, and that was trusted to specify format characteristics, was detected. The result is that unfiltered user input is provided as the format string parameter for certain functions. The attacker can utilize format tokens such as %x to print data from program memory and tokens such as %n to write data to program memory. The code identified could result in a format string attack and was brought to the attention of the developer for remediation.

Permissions and File Creation Errors

A limited number of locations where unnecessarily lax permissions were used in the creation of files were identified and pointed out to the developers. Similarly, there were a small number of places within the code where the third argument to open(2) was omitted. In this latter case of file creation, open would use values located on the program stack as input for the file creation mode with the result being arbitrary permission values.

A common error in the creation of files was also found. If a file is created via the open(2) call with the flag of O_CREAT specified, a new file will be created. In many cases where file creation is desired, the flag O_EXCL should be used in addition to O_CREAT. This additional flag ensures that the file that is about to be created does not already exist and that the last component in the pathname provided to open(2) does not reference a symbolic link (even if that link points to a non-existent file). The absence of this flag would enable the attacker to cause the program to create files with names and paths specified by the attacker. If the program is running with elevated privileges this can cause system compromise and denial of system wide services.

Shell Execution

The most interesting error found through these processes was that of task handoff between the running program and the underlying system. In particular that of handing off tasks to be executed by system command interpreters.

In the case identified, part of the data comprising the command being passed to the external interpreter was retrieved from remote sources and thus should have been treated as potentially tainted. If a malicious person were to encode this data with various shell meta-characters it would have been possible to remotely compromise the RPKI Repository system. As the account running the program is expected to have complete access to the database, at the very least total database compromise could occur along with user level compromise of the host system. If the account running the program was operating with elevated privileges, total remote system compromise could have been attained.

The developers were alerted to this and provided several suggestions for solutions. The first was to introduce several hundred new lines of code from the open source application rsync(1). This was the program and argument string that was being passed to the command interpreter. This would be the safest solution but would also require the most amount of re-work and effort. The next solution was to introduce string parsing functions that would be used to sanitize the arguments being passed to the command interpreter. This would not require as much effort as the first suggested solution with the caveat that it would not provide as much potential protection and such efforts have historically been shown to be difficult to implement correctly. The final suggestion was to document this security issue in a highly visible fashion so as to dissuade people from recreating such situations elsewhere and to demonstrate that this is an understood issue that is being tolerated for the time being due to the prototype and proof of concept nature of the program. In this case, the second option was chosen with comments introduced to the source code to make other developers aware of the potential security caveat.

Level of Effort and Completeness

This analysis attempted to approach the task of identifying problem areas within the code base in a fashion similar to that performed by many attackers. Out of thousands of lines of code within the total project, the sections most likely to contain vulnerabilities were identified. From this point, the majority of effort could be spent addressing these areas rather than focusing on sections that were unlikely to contain vulnerabilities. This approach attempted to address the most commonly found coding errors that could be leveraged to compromise the integrity of the program and of the system within which the program operates.

This effort does not guarantee that all security vulnerabilities were identified. Nor does this effort provide full code coverage and discovery of complex logic vulnerabilities.

1.2 Modifications and Corrective Measures Taken

The majority of issues identified were corrected using the suggestions provided during the security analysis. These included such practices as replacing unbounded string copies with bounded versions, ensuring the most restrictive yet usable permissions are applied to file creation, protecting against race conditions, and other similar solutions. There were however a few areas where corrections were not made. The reasons behind these being a combination of time and effort required given the available funding and the low potential risk.

For example, the *casn* library received modifications due to its potential for ubiquitous use. Preventing malicious acts in this code base was deemed important. *asn_gen*, on the other hand, is a tool that is only used when ASN.1 definitions change. This is further only used to generate “C” code in such situations. Given the difficulty of implementing some of the suggested bounds checking within this code, and the fact that said code base is not part of the running application and cannot be used for exploitative purposes the modification of *asn_gen* code was omitted.

As the system is specific in purpose, it is believed that real world use would primarily be performed on dedicated systems. Thus attempts were made to minimize certain race conditions inherent in common system calls such as stat(). However, as these are low risk vulnerabilities, replacing stat() with fstat() and subsequently passing and maintaining file descriptors between functions was viewed as too large an effort for small gain.