DNS Extensions
Internet-Draft
Intended status: Standards Track
Expires: September 13, 2012

S. Krishnaswamy
R. Story
SPARTA, Inc.
A. Hayatnagarkar

March 12, 2012

DNSSEC Validator API
draft-hayatnagarkar-dnsext-validator-api-09

Abstract

The DNS Security Extensions (DNSSEC) provide origin authentication and integrity of DNS data. However, the current resolver Application Programming Interface (API) does not specify how a validating stub resolver should communicate results of DNSSEC processing back to the application. This document describes an API between applications and a validating stub resolver that allows applications to control the DNSSEC validation process and obtain results of DNSSEC processing.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 13, 2012.

Copyright Notice

Table of Contents

1. Introduction

The DNS Security Extensions ([refs.RFC4033], [refs.RFC4034], [refs.RFC4035]) enable DNS resolvers to test the origin authenticity and integrity of data returned by the DNS. A DNSSEC validator, or more formally, a validating stub resolver, is a piece of software that performs these tests by constructing an authentication chain [refs.RFC4033] from a locally configured DNSSEC trust anchor [refs.RFC4033] to a cryptographic signature that covers the DNS information in question. This document presents an API between an application and a DNSSEC validator, which enables applications to control the DNSSEC validation process and enables applications to obtain DNSSEC validation results upon which to base program behavior.

The API can be broadly divided into three groups: the high-level DNSSEC validator API, the low-level DNSSEC validator API and the DNSSEC validator-context API. Section 3, Section 4, and Section 6 describe these groups in greater detail.

The high-level DNSSEC validator API is designed for ease of use and mirrors existing DNS-related functions. This API is best suited for existing applications that use legacy DNS functions such as gethostbyname() and getaddrinfo() [refs.IEEE.1003.1-2004] and have no requirement for detailed DNSSEC validation status information.

The low-level DNSSEC validator API enables applications to examine the DNSSEC validation details for each element of the DNSSEC authentication chain.

The DNSSEC validator-context API enables applications to control the DNSSEC policies that are used for validating DNS responses.

The range of functions provided in this API supports different classes of applications, ranging from those that are only interested in basic DNSSEC results to more sophisticated applications that can look for specific errors in an authentication chain as a sign of some abnormality or attack.

2. Terminology

Some of the terms used in this specification are defined below:

Legacy DNS Functions: existing functions, such as gethostbyname() and getaddrinfo(), which are not capable of returning DNSSEC validation status codes for DNS responses.

DNSSEC Validator Policy: a set of configuration parameters for the DNSSEC validator, which can influence the eventual outcome of the DNSSEC validation process.

DNSSEC Validator Context: the application's run-time handle to the DNSSEC validator policy.

3. High-level DNSSEC Validator API

The high-level DNSSEC validator API defines DNSSEC-aware substitutes for commonly used legacy DNS functions. It provides an easy path for applications already using legacy DNS functions to transition towards becoming DNSSEC-aware. This API also defines the val_get_rrset() function, which enables applications to obtain data for an arbitrary DNS name, class and type, and inspect the corresponding DNSSEC validation status value(s).

A number of legacy DNS functions exist; however, some of these functions (such as gethostbyname_r and gethostbyname2) are only available on a subset of Operating Systems and are not part of any official standard. Also, some functions are defined as minor extensions of other well-known legacy DNS functions. For example, gethostbyname2 differs from gethostbyname_r only by virtue of having the extra argument to explictly specify the address family. Further, some functions differ from others only by virtue of being able to support a re-entrant and thread-safe implementation. Instead of providing an exhaustive list of DNSSEC-capable replacement functions for all known resolver function calls, the high-level DNSSEC validator API in this document only describes DNSSEC extensions for the canonical set of function calls specified in [refs.IEEE.1003.1-2004]. DNSSEC replacement functions for other legacy DNS functions are expected to mirror, to a large extent, other functions described in this document.

The ctx parameter in the functions described in this API points to a DNSSEC validator context object (see Section 6). Applications MUST either supply a reference to a valid DNSSEC validator context object created using the functions specified in Section 6 or supply a NULL value for this parameter. Libraries that implement the DNSSEC Validator API MUST internally use a default DNSSEC validator context when the application supplies a NULL value for ctx.

3.1. val_gethostbyname, val_gethostbyaddr

```
#include <validator/validator.h>

struct hostent *val_gethostbyname( val_context_t *ctx,
                                   const char    *name,
                                   val_status_t  *val_status );

struct hostent *val_gethostbyaddr( val_context_t *ctx,
                                   const char    *addr,
                                   int            len,
                                   int            type,
                                   val_status_t  *val_status );
```

The val_gethostbyname() and val_gethostbyaddr() functions are DNSSEC-aware versions of the gethostbyname() and gethostbyaddr() legacy DNS functions.

The new functions have an additional parameter, val_status, which enables applications to check the DNSSEC validation status codes for the address-to-name and name-to-address translations. The other arguments to these functions and their return values have identical semantics to the corresponding legacy DNS functions as described in [refs.IEEE.1003.1-2004]. The val_gethostbyname() and val_gethostbyaddr() functions SHOULD only be used when retrofitting DNSSEC in existing applications that use the gethostbyname() and gethostbyaddr() functions. For new applications that need to perform these translations, the functions described in Section 3.2 and Section 3.4 SHOULD be used instead.

The DNSSEC validation status is returned in the val_status parameter. When evaluating the validity of a DNS response, applications SHOULD use the functions described in Section 8.2 instead of directly inspecting the DNSSEC validation status code returned in val_status.

The status code returned in val_status is determined by the following rules.

- A DNSSEC validation status of VAL_OOB_ANSWER MUST be returned if the complete answer is returned using some out-of-band mechanism (for example, from a local configuration store such as /etc/hosts or its equivalent) without any DNSSEC validation being performed. However, if local DNSSEC validator policy defines out-of-band answers to be trustworthy, a DNSSEC validation status of VAL_TRUSTED_ANSWER SHOULD be returned instead.
- A DNSSEC validation status of VAL_VALIDATED_ANSWER MUST be returned if all addresses and canonical names within the hostent structure are validated successfully.

- A DNSSEC validation status of VAL_TRUSTED_ANSWER MUST be returned if at least one address or canonical name within the hostent structure is not validated by the DNSSEC validation process, but all answers are still considered trustworthy (see Section 6) by way of the configured local DNSSEC validator policy.
- A DNSSEC validation status of VAL_UNTRUSTED_ANSWER MUST be returned if at least one address or canonical name within the hostent structure is neither validated through the DNSSEC validation process nor considered to be trusted according to the configured local DNSSEC validator policy.
- A DNSSEC validation status of VAL_NONEXISTENT_NAME or VAL_NONEXISTENT_TYPE MUST be returned if the DNSSEC validation process is able to prove non-existence for the name or type being queried for. A DNSSEC validation status of VAL_NONEXISTENT_NAME_NOCHAIN or VAL_NONEXISTENT_TYPE_NOCHAIN MUST be returned if a DNS response with an RCODE reflecting type or name non-existence is returned, and local DNSSEC validator policy is configured to treat such answers as trustworthy. If the previous two conditions for non-existence are not satisfied, val_status MUST be set to VAL_UNTRUSTED_ANSWER.

3.2. val_getaddrinfo, val_freeaddrinfo, val_getnameinfo

```
#include <validator/validator.h>

int val_getaddrinfo( val_context_t          *ctx,
                     const char             *nodename,
                     const char             *servname,
                     const struct addrinfo  *hints,
                     struct addrinfo       **res ,
                     val_status_t           *val_status);

void val_freeaddrinfo( struct addrinfo     *res);

int val_getnameinfo( val_context_t         *ctx,
                     const struct sockaddr *sa,
                     socklen_t              salen,
                     char                  *host,
                     size_t                 hostlen,
                     char                  *serv,
                     size_t                 servlen,
                     int                    flags,
                     val_status_t          *val_status );
```

These functions are DNSSEC-aware versions of the getaddrinfo(), freeaddrinfo() and getnameinfo() legacy DNS functions ([refs.RFC3493]) respectively.

The val_getaddrinfo() function returns the address and service information for the specified domain name and service. The val_freaddrinfo() function releases the memory used by the struct addrinfo returned in the res parameter when calling val_getaddrinfo. The val_getnameinfo() function performs an address-to-name translation in a protocol independent manner.

The value of res MUST point to a valid addrinfo structure ([refs.RFC3493]) on a successful return from the val_getaddrinfo() function or NULL in case of error. Sufficient memory MUST be internally allocated to hold the linked list pointed to by res. This memory MUST be released when applications invoke the freeaddrinfo() function. ([refs.RFC3493]).

The DNSSEC validation status is returned in the val_status parameter. When evaluating the validity of a DNS response, applications SHOULD use the functions described in Section 8.2 instead of directly inspecting the DNSSEC validation status code returned in val_status. The syntax and semantics of other parameters in val_getaddrinfo() and val_getnameinfo() and their return values are identical to those specified for getaddrinfo() and getnameinfo() in [refs.RFC3493].

The status code returned in val_status is determined by the following rules.

- A DNSSEC validation status of VAL_OOB_ANSWER MUST be returned in val_status if the complete answer is returned using some out-of-band mechanism (for example, from a local configuration store such as /etc/hosts or its equivalent) without any DNSSEC validation being performed. However, if local DNSSEC validator policy defines out-of-band answers to be trustworthy, a DNSSEC validation status of VAL_TRUSTED_ANSWER SHOULD be returned instead.
- A DNSSEC validation status of VAL_VALIDATED_ANSWER MUST be returned in val_status if the hostname returned by val_getnameinfo(), or all addresses and canonical names returned by val_getaddrinfo(), are validated through the DNSSEC process.
- A DNSSEC validation status of VAL_TRUSTED_ANSWER MUST be returned in val_status if the hostname returned by val_getnameinfo(), or at least one address or canonical name returned by val_getaddrinfo(), is not validated by the DNSSEC validation process but all answers are still considered to be trustworthy through the local DNSSEC validator policy (see Section 6).
- A DNSSEC validation status of VAL_UNTRUSTED_ANSWER MUST be returned if at least one address or canonical name returned by val_getaddrinfo() within the addrinfo structure, or the returned hostname in val_getnameinfo(), is neither validated through the DNSSEC process nor considered to be trustworthy according to the local DNSSEC validator policy.

- A DNSSEC validation status of VAL_NONEXISTENT_NAME or VAL_NONEXISTENT_TYPE MUST be returned in val_status if the DNSSEC validation process is able to prove non-existence for the name or type being queried for. A DNSSEC validation status of VAL_NONEXISTENT_NAME_NOCHAIN or VAL_NONEXISTENT_TYPE_NOCHAIN MUST be returned if a DNS response with an RCODE reflecting type or name non-existence is returned, and local DNSSEC validator policy is configured to treat such answers as trustworthy (Section 6). If the previous two conditions for non-existence are not satisfied, val_status MUST be set to VAL_UNTRUSTED_ANSWER.

3.3. val_res_query

```
#include <validator/validator.h>

int val_res_query(val_context_t *ctx,
                  const char    *domain_name,
                  int            class,
                  int            type,
                  unsigned char *answer,
                  int            anslen,
                  val_status_t  *val_status);
```

The val_res_query() function is a DNSSEC-aware replacement for the res_query() function (currently not documented in any standard reference).

The val_res_query() function queries the DNS for the data associated with the given domain name, class and type, and returns the resulting resource record sets in a DNS-style response.

The val_res_query() function MUST return the actual size of the response packet on success and -1 on failure. On success, the response from the DNS MUST be copied to the user-allocated buffer in answer and MUST NOT exceed the buffer size specified in anslen.

The DNSSEC validation status is returned in the val_status parameter. When evaluating the validity of a DNS response, applications SHOULD use the functions described in Section 8.2 instead of directly inspecting the DNSSEC validation status code returned in val_status.

The status code returned in val_status is determined by the following rules.

- A DNSSEC validation status of VAL_VALIDATED_ANSWER MUST be returned if all resource record sets returned in the answer are validated by the DNSSEC validation process.

- A DNSSEC validation status of VAL_TRUSTED_ANSWER MUST be returned if at least one resource record set returned in the answer is not validated by the DNSSEC validation process, but all resource record sets are still considered to be trustworthy according to the configured local DNSSEC validator policy (see Section 6).
- A DNSSEC validation status of VAL_UNTRUSTED_ANSWER MUST be returned if at least one resource record set in the answer is neither validated through the DNSSEC validation process nor considered to be trustworthy according to the local DNSSEC validator policy.
- A DNSSEC validation status of VAL_NONEXISTENT_NAME or VAL_NONEXISTENT_TYPE MUST be returned if the DNSSEC validation process is able to prove non-existence for the name or type being queried for. A DNSSEC validation status of VAL_NONEXISTENT_NAME_NOCHAIN or VAL_NONEXISTENT_TYPE_NOCHAIN MUST be returned if a DNS response with an RCODE reflecting type or name non-existence is returned, and local DNSSEC validator policy is configured to treat such answers as trustworthy. If the previous two conditions for non-existence are not satisfied, val_status MUST be set to VAL_UNTRUSTED_ANSWER.

3.4. val_get_rrset

```
#include <validator/validator.h>

int val_get_rrset(val_context_t            *ctx,
                  const char               *name,
                  int                       class,
                  int                       type,
                  unsigned int              flags,
                  struct val_answer_chain **answers);

void val_free_answer_chain(struct val_answer_chain *answers);
```

The val_get_rrset() function queries the DNS for the data associated with the given domain name, class and type. The flags argument specifies a list of options to the validation process, logically OR'd to each other. This possible flags are defined in Section 4.1. val_get_rrset() MUST return 0 on success and an error code from Section 7 on failure.

val_get_rrset() MUST return its results in the val_answer_chain structure after allocating sufficient memory for this structure. Applications MUST release this memory after use by invoking the val_free_answer_chain() function.

The val_answer_chain structure is defined below.

```
struct val_answer_chain {
      val_status_t             val_ans_status;
      char                    *val_ans_name;
      int                      val_ans_class;
      int                      val_ans_type;
      struct rr_rec           *val_ans;
      struct val_answer_chain *val_ans_next;
};

struct rr_rec {
      size_t         rr_length;
      unsigned char *rr_data;
      struct rr_rec *rr_next;
};
```

val_ans_name MUST be set to the DNS name of the actual resource record set returned. This value may differ from the name argument in val_get_rrset() if the resource record is returned after following a CNAME ([refs.RFC1034]) or DNAME ([refs.RFC2672]) alias. val_ans_class and val_ans_type MUST be set to the actual class and type for the returned resource record. These values may differ from the class and type arguments in val_get_rrset() if the query type or class has the value 255 (ANY). The resource record sets MUST be returned in val_ans as a linked list of rr_rec structures, with each element containing the rr_length and rr_data tuple for a resource record in the resource record set. val_ans MUST be set to NULL if no answer was returned for the given query or if a proof of non-existence was returned.

The DNSSEC validation status code is returned in the val_ans_status field. Since validation status codes returned by val_get_rrset() are available per resource record set, the set of possible values for val_ans_status is more granular than that possible for the val_status field in other high-level API functions. The list of possible codes for val_ans_status are listed in Section 8.1. When evaluating the validity of a DNS response, applications SHOULD use the functions described in Section 8.2 instead of directly inspecting the DNSSEC validation status code returned in val_ans_status.

4. Low-level DNSSEC Validator API

The low-level DNSSEC validator API provides applications with greater control and visibility into the DNSSEC validation process. The functions and data structures defined in the low-level DNSSEC validator API are summarized below.

4.1. val_resolve_and_check, val_free_result_chain

```
#include <validator/validator.h>

int val_resolve_and_check( val_context_t              *ctx,
                           const char                 *domain_name,
                           int                         class,
                           int                         type,
                           unsigned int                flags,
                           struct val_result_chain    **results);

void val_free_result_chain(struct val_result_chain *results);
```

The val_resolve_and_check() function queries the DNS for the <domain_name, class, type> tuple and then performs the DNSSEC validation operation for the responses received. The val_free_result_chain() function releases the resources allocated for the returned result.

The flags argument specifies a list of options to the validation process, logically OR'd to each other. The following flags are defined; their values are implementation-specific:

VAL_QUERY_AC_DETAIL: If this flag is specified, the complete authentication chain MUST be returned for each answer within the val_result_chain structure. If this flag is not set then only the rrset information corresponding to the answer or proof of non-existence for the queried name, domain_name SHOULD be returned within the val_result_chain structure.

VAL_QUERY_DONT_VALIDATE: If this flag is specified, validation processing MUST be ignored for the given name. Any resulting answer will have a validation status value of VAL_IGNORE_VALIDATION.

VAL_QUERY_IGNORE_SKEW: If this flag is specified, signature inception and expiration times MUST be ignored for RRSIGs in the authentication chain.

VAL_QUERY_NO_DLV: If this flag is specified, Dynamic Look-aside Validation (DLV) [refs.DLV] processing for this name MUST NOT be performed.

VAL_QUERY_ASYNC: If this flag is specified, this name MUST be resolved through the asynchronous lookup process (see Section 5).

VAL_QUERY_RECURSE: If this flag is specified, the name MUST be looked up by directly querying the authoritative name server for that name (which may involve iteratively querying various name servers in the delegation hierarchy) instead of requesting this information from any caching name server that may be specified as configuration data.

VAL_QUERY_NO_EDNS0_FALLBACK: If this flag is specified, re-trying the query with smaller EDNS0 advertised window sizes MUST NOT be attempted as a fallback strategy.
VAL_QUERY_SKIP_RESOLVER: If this flag is specified, only the cache should be consulted while looking up a name. New queries MUST NOT be sent on the wire as part of looking up and validating the cached answer.
VAL_QUERY_SKIP_CACHE: If this flag is specified, any locally cached information for the name being looked up MUST be ignored.

val_resolve_and_check() MUST return 0 on success and an error code from Section 7 on failure. Answers to the query MUST be returned in results, which is a linked list of val_result_chain structures, as defined below. val_resolve_and_check() MUST allocate sufficient memory to hold the contents of results. This memory MUST be released when applications invoke the val_free_result_chain() function.

```
#define MAX_PROOFS 4
struct val_result_chain {
      val_status_t                    val_rc_status;
      char                           *val_rc_alias;
      struct val_rrset_rec           *val_rc_rrset;
      struct val_authentication_chain *val_rc_answer;
      int                             val_rc_proof_count;
      struct val_authentication_chain *val_rc_proofs[MAX_PROOFS];
      struct val_result_chain         *val_rc_next;
};
```

Each element in the val_result_chain linked list MUST point to a distinct resource record set returned in the response. Multiple resource record sets can be returned in a response when the query is for the type code of 255 (ANY) or 46 (RRSIG). The val_rc_next field enables an application to iterate through the list of all results returned by the DNSSEC validator. For all val_result_chain elements that represent a name alias, val_rc_alias MUST point to the target name referenced by that alias.

val_rc_answer SHOULD point to a resource record in the answer portion; all associated proofs of non-existence (either in support of the answer in val_rc_answer or to prove the non-existence of a record) SHOULD be returned in val_rc_proofs. val_rc_proof_count MUST be set to the number of proof elements that are available. val_rc_answer and val_rc_proofs SHOULD be NULL if the VAL_QUERY_AC_DETAIL flag is not specified in the flags argument for val_resolve_and_check().

val_rc_rrset MUST point to resource record set information for the current element in the val_result_chain linked list. If no answers are returned (or a proof of non-existence is returned) in response to the query, val_rc_rrset MUST be set to NULL.

The DNSSEC validation status code is returned in the val_rc_status field. Since validation status codes are available per resource record set, it is possible to have a highly granular set of values for val_rc_status. Possible codes for val_rc_status are listed in Section 8.1. When evaluating the validity of a DNS response, applications SHOULD use the functions described in Section 8.2 instead of directly inspecting the DNSSEC validation status code returned in val_rc_status.

The val_authentication_chain structure represents a linked list whose elements comprise the DNSSEC authentication chain for an answer or proof of non-existence resource record set.

```
struct val_authentication_chain {
    val_astatus_t                    val_ac_status;
    struct val_rrset_rec            *val_ac_rrset;
    struct val_authentication_chain *val_ac_trust;
};
```

The DNSSEC validation status for the specified resource record set MUST be set in the val_ac_status field. Possible codes for this field are defined in Section 4.2. The val_ac_trust field MUST point to the next element in the authentication chain proceeding from the signed record towards a DNSSEC trust anchor. For an element with type DNSKEY, the next element MUST correspond to a DS record in the parent zone and for a DS record the next element MUST correspond to the DNSKEY in the same zone as the DS record. The value of val_ac_trust MUST be set to NULL if either the current element in the linked list points to a valid DNSSEC trust anchor or if an error condition is encountered. The validation status code stored in the val_ac_status field can be used to differentiate between different error conditions.

The val_ac_rrset field in the val_authentication_chain structure MUST point to a val_rrset_rec structure holding the actual resource record set fields ([refs.RFC1034]) as described below.

```
struct val_rrset_rec {
    int                 val_rrset_rcode;
    char                *val_rrset_name;
    int                 val_rrset_class;
    int                 val_rrset_type;
    long                val_rrset_ttl;
    int                 val_rrset_section;
    struct sockaddr     *val_rrset_server;
    struct val_rr_rec   *val_rrset_data;
    struct val_rr_rec   *val_rrset_sig;
};
```

The information stored in the val_rrset_rec structure includes the DNS response error code in the val_rrset_rcode field, and the DNS response "envelope" comprising of the name, class, type and time-to-live tuple in the val_rrset_name, val_rrset_class, val_rrset_type and val_rrset_ttl fields respectively. Additionally, the name server from where this resource record set was received MUST be stored in the sockaddr data structure ([refs.IEEE.1003.1-2004]) pointed to by the val_rrset_server field. The section where the resource record set appeared in the DNS response MUST be saved in the val_rrset_section field within the val_rrset_rec structure, and MUST be set to one of the following values:

```
#define VAL_FROM_ANSWER 1  /* if the resource record set was present
   in the answer section of the DNS response. */

#define VAL_FROM_AUTHORITY 2  /* if the resource record set was
   present in the authority section of the DNS response. */

#define VAL_FROM_ADDITIONAL 3  /* if the resource record set was
   present in the additional section of the DNS response. */
```

The data returned for the resource record set MUST be queued to val_rrset_data. Any associated RRSIGs MUST be queued to val_rrset_sig. Both of these variables MUST point to lists of struct val_rr_rec elements, which specify the resource record data and the DNSSEC validation status for each resource-record within the resource record set as defined below.

```
struct val_rr_rec {
    size_t              rr_rdata_length;
    unsigned char       *rr_rdata;
    struct val_rr_rec   *rr_next;
    val_astatus_t       rr_status;
};
```

The rr_status member in val_rr_rec is only relevant for the

signatures present in val_rrset_sig or when val_rrset_data points to DNSKEY or DS resource records. In other cases the value of this field MUST be set to VAL_AC_UNSET. The rr_status field takes on a subset of all status codes possible for the val_astatus_t type and is further described in Section 4.2.

4.2. Authentication Chain Status Codes and p_ac_status()

For each authentication chain element in the val_authentication_chain structure, the val_ac_status field MUST contain one of the following codes:

VAL_AC_UNSET: The DNSSEC validation status for the resource record set could not be determined.

VAL_AC_IGNORE_VALIDATION: DNSSEC validation for the given resource record set was ignored on the basis of some configured DNSSEC validator policy.

VAL_AC_UNTRUSTED_ZONE: The resource record set belonged to a zone that the DNSSEC validator considered to be un-trusted, with no further DNSSEC validation being deemed necessary.

VAL_AC_PINSECURE: The resource record set belonged to a zone for which the DS record was provably absent.

VAL_AC_BARE_RRSIG: The resource record set contained only RRSIGs (in response to a query of type RRSIG). RRSIGs contain the cryptographic signatures for other DNS data and cannot themselves be validated.

VAL_AC_NO_LINK: No DNSSEC trust anchor was configured at or above the level of the authentication chain could be found above this point, therefore no validation could be performed.

VAL_AC_TRUST: At least one of the signatures covering the given resource record set was directly verified using a key that was configured as a DNSSEC trust anchor.

VAL_AC_RRSIG_MISSING: RRSIG data for the given resource record set could not be located.

VAL_AC_DNSKEY_MISSING: The DNSKEY data that generated signatures for the given resource record set could not be located.

VAL_AC_DS_MISSING: The DS data for the DNSKEY resource record set in question could not be located.

VAL_AC_DATA_MISSING: The returned resource record set was empty.

VAL_AC_DNS_ERROR: A DNS error was encountered during the query resolution process.

VAL_AC_NOT_VERIFIED: None of the RRSIGs covering the given resource record set could be verified.

VAL_AC_VERIFIED: At least one RRSIG covering the resource record set verified successfully.

For each signature val_rr_rec member within an authentication chain pointed to by val_ac_rrset, the DNSSEC validation status stored in the variable rr_status MUST be set to one of the following codes:

VAL_AC_UNSET: No DNSSEC validation status information could be obtained for the given signature.

VAL_AC_RRSIG_VERIFIED: The RRSIG verified successfully.

VAL_AC_WCARD_VERIFIED: The RRSIG covering a resource record proved that the record was wildcard expanded.

VAL_AC_RRSIG_VERIFIED_SKEW: The RRSIG verified successfully only after clock skew was taken into consideration.

VAL_AC_WCARD_VERIFIED_SKEW: The RRSIG covering a resource record proved that the record was wildcard expanded, but only after clock skew was taken into consideration.

VAL_AC_WRONG_LABEL_COUNT: The number of labels on the signature was greater than the count given in the RRSIG resource record data.

VAL_AC_INVALID_RRSIG: The RRSIG could not be parsed.

VAL_AC_RRSIG_NOTYETACTIVE: The RRSIG's inception time was in the future.

VAL_AC_RRSIG_EXPIRED: The RRSIG's expiration time was in the past.

VAL_AC_ALGORITHM_NOT_SUPPORTED: The RRSIG algorithm was not supported.

VAL_AC_RRSIG_VERIFY_FAILED: The RRSIG could not be verified.

VAL_AC_RRSIG_ALGORITHM_MISMATCH: The keytag referenced in the RRSIG matched a DNSKEY but the algorithms were different.

VAL_AC_DNSKEY_NOMATCH: The DNSKEY that created the given signature could not be found in the zone DNSKEY resource record set.

For each val_rr_rec member of type DNSKEY (or DS where indicated) within an authentication chain structure pointed to by val_ac_rrset, the DNSSEC validation status stored in the variable rr_status MUST be set to one of the following codes:

VAL_AC_UNSET: No DNSSEC validation status information could be obtained for the given DNSKEY or DS record.

VAL_AC_TRUST_POINT: The given DNSKEY or DS record was configured as a DNSSEC trust anchor.

VAL_AC_SIGNING_KEY: The given DNSKEY was used for generating an RRSIG for a resource record in the authentication chain.

VAL_AC_VERIFIED_LINK: The given DNSKEY or DS resource record provided the link in the authentication chain from a DNSSEC trust anchor to the signed record.

VAL_AC_UNKNOWN_ALGORITHM_LINK: The DNSKEY chained up to a DS record but the DNSKEY algorithm was unknown.

VAL_AC_UNKNOWN_DNSKEY_PROTOCOL: The DNSKEY protocol number was unknown.

VAL_AC_ALGORITHM_NOT_SUPPORTED: The DNSKEY or DS algorithm was not supported.

VAL_AC_DS_NOMATCH: The given DNSKEY did not chain up to any DS record in the parent zone.

VAL_AC_INVALID_KEY: The given DNSKEY was invalid.

VAL_AC_INVALID_DS: The given DS was invalid.

The numerical values for the codes listed above are implementation-specific. The p_ac_status() function is used to convert the DNSSEC validation status code stored in struct val_authentication_chain to a

string representation.

```
#include <validator/validator.h>

const char *p_ac_status(val_astatus_t status);
```

The value returned MAY be the string conversion for the corresponding val_astatus_t identifier. For example, the return value from p_ac_status(VAL_AC_VERIFIED) MAY be "VAL_AC_VERIFIED".

5. Low-level Asynchronous DNSSEC Validator API

The low-level Asynchronous DNSSEC validator API allows an application to submit multiple requests which can be processed in parallel. In most cases, this will result in validation completing much sooner than a series of synchronous requests.

When submitting an asynchronous request, an application may specify a callback function to be called when the request completes.

Since DNSSEC validation of a domain name involves multiple queries, applications must periodically give time to the API for processing responses and sending additional queries. (See val_async_check_wait() in Section 5.1.3.)

The functions and data structures defined in the low-level Asynchronous DNSSEC validator API are summarized below.

5.1. Asynchronous Requests

5.1.1. val_async_submit

```
#include <validator/validator.h>

int val_async_submit(val_context_t *ctx, const char *domain_name,
                     int class, int type, unsigned int flags,
                     val_async_event_cb *callback,
                     void *user_context,
                     val_async_status **async_status);
```

The val_async_submit() function submits a request for asynchronous processing of DNS queries for the data associated with the given domain name, class and type.

If specified, the given callback function will be called when results become available. Some flags, defined below, can affect when and how often the callback is called.

The specified user context will also be passed to the callback function. More information on the callback function and user_context can be found in Section 5.2.

The val_async_submit() function MUST return VAL_NO_ERROR on success and return a pointer to a newly allocated async_status object via the async_status parameter. More information on async_status objects can be found in Section 5.3.

On failure, the return code will be one of VAL_RESOURCE_UNAVAILABLE, VAL_BAD_ARGUMENT or VAL_INTERNAL_ERROR. The function MUST release any allocated data, and MUST NOT return a value via the async_status parameter. An implementation MAY set async_status to NULL.

The following flags may be set for the request. The numerical values for the flags are implementation-specific.

VAL_AS_IGNORE_CACHE: Don't use any internal cache for answers to this query. Answers MUST be from fresh responses to all queries. These new answers MAY be stored in the internal cache for use with future queries.

VAL_AS_NO_NEW_QUERIES: Don't send any new queries. Answers MUST come from the internal cache.

VAL_AS_NO_ANSWERS: Caller doesn't care about the answer results. This can be used for priming the cache.

VAL_AS_NO_CALLBACKS: Don't call any callbacks.

VAL_AS_NO_CANCEL_CALLBACKS: Call callbacks with results, but don't call any callbacks when the reques is canceled.

VAL_AS_INTERIM_CALLBACKS: Call the callback function with interim results. If this flag is not specified, the callback function will only be called when all validation results are ready.

5.1.2. val_async_select_info

```
#include <validator/validator.h>

int val_async_select_info(val_context_t *context,
                          fd_set *fds, int *max_fd,
                          struct timeval *timeout);
```

The val_async_select_info() function examines all outstanding asynchronous requests for the given context and sets the appropriate

file descriptors, timeout value and maximum file descriptor value in preparation for a call to select().

The file descriptor for each socket awating a response MUST be set in the fds parameter. The function MUST NOT initialize the fd_set, as the application may have already set its own file descriptors.

The integer value pointed to by max_fd MUST be set to the highest file descriptor number of any pending asynchronous request, unless that value is less than the current vaule of max_fd. In that case, the max_fd value MUST NOT be changed.

The timeout structure MUST be set to the lowest timeout value of any pending asynchronous query timeout which is less than the current value in timeout.

After the application call select, val_async_check_wait() (see Section 5.1.3) should be called with the fd_set and number of ready file descriptors returned by select. Example code is provided in Appendix C.

5.1.3. val_async_check_wait

```
#include <validator/validator.h>

int val_async_check_wait(val_context_t *ctx, fd_set *fds,
                         int *nfds, struct timeval *timeout,
                         unsigned int flags);
```

The val_async_check_wait() function handles timeouts or processes DNS responses to outstanding queries. It may also call callbacks for completed requests.

The function provides two modes of operation. The first is for use with an application that has its own select() loop. The applications sets its own file descriptors, calls val_async_select_info() to set file descriptors for pending queries and calls select(). The fds and nfds parameters from select are passed in to val_async_check_wait and the timeout value is ignored. If the implementation processes responses for a file descriptor, the implementation SHOULD clear the appropriate file descriptor in fds and decrement nfds.

In the second mode of operation, the fds and nfds parameters are set to NULL and a timeout value is specified. The function will call val_async_select_info() and select() internally, and process any responses received before the timeout value expires.

Example code is provided in Appendix C.

5.2. Asynchronous Callbacks

```
#include <validator/validator.h>

typedef struct val_cb_params_s {
    val_status_t             val_status;
    char                    *name;
    int                      class_h;
    int                      type_h;
    int                      retval;
    struct val_result_chain *results;
    struct val_answer_chain *answers;
} val_cb_params_t;

typedef int (*val_async_event_cb)(val_async_status *as, int event,
                                  val_context_t *ctx, void *user_ctx,
                                  val_cb_params_t *callback_params);
```

When an asynchronous request is submitted, a callback function and user context may be provided by the caller. This callback function is called when validation results are available. The user_ctx parameter MUST be the value given by the caller when the request was submitted.

The callback params structure contains the orginal query parameters (name, class and type), the 'return value' for the operation, pointers to the result and answer chains, and the final validation status. This structure will be released when the callback is completed. The application can assume responsibility for any of the pointer values by copying them and setting the pointers in the callback param structure to NULL. The application then becomes responsible for releasing the memory with val_free_result_chain (see Section 4.1) and/or val_free_answer_chain (see Section 3.4), as appropriate.

The following event types are defined:

VAL_AS_EVENT_COMPLETED: The request completed.

VAL_AS_EVENT_INTERIM: The request is still being processed, but some interim results are available.

VAL_AS_EVENT_CANCELED: The request was canceled. The val_status, results and answers members of the callback parameter structure are undefined.

Possible codes for val_status are listed in Section 8.1. When evaluating the validity of a DNS response, applications SHOULD use the functions described in Section 8.2 instead of directly inspecting the DNSSEC validation status code returned in val_status.

After the callback function has completed, the implementation SHOULD release all resources allocated for the request.

5.3. Asynchronous Status

An application which submits asynchronous requests needs a way to refer to each request for future operations. This asynchronous status object is an implementation specific opaque object which an uniquely identifies a particular request.

When an asynchronous request is submitted, the implementation MUST create an asynchronous status object to return to the caller. The size of the object SHOULD be at least as large as the native pointer type.

5.3.1. Operations on asynchronous status objects

The API supports the following operations to manipulate asynchronous requests:

5.3.1.1. val_async_cancel

```
#include <validator/validator.h>

int val_async_cancel( val_context_t *context,
                      val_async_status *async_status,
                      unsigned int flags);
```

This function will cancel an outstanding asynchronous request. All resources used for the request SHOULD be released.

The following flags may be set for the request. The numerical values for the flags are implementation-specific.

VAL_AS_CANCEL_NO_CALLBACKS: Do not call completed or cancelled callbacks.

6. DNSSEC Validator Context API

DNSSEC validator policy can be used to influence the DNSSEC validation outcome. Examples of DNSSEC validator policy include DNSSEC trust anchors for different zones and acceptable clock-skew values for checking inception and expiration times on signatures from different zones.

DNSSEC validator policy is stored in the local system configuration (for example, the configuration file /etc/dnsval.conf) and could be configured differently for different applications and operating scenarios. Policies are identified by simple text strings called labels, which MUST be unique within the system configuration. As an example, "browser" could be used as the label that defines the DNSSEC validator policy for all web-browsers in a system. The manner of supplying the validation policy label to an application is implementation-specific, but the label MAY also be supplied during application-startup through the environment variable, VAL_CONTEXT_LABEL.

All DNSSEC validator policy definitions in the system configuration are implementation-specific.

6.1. val_create_context, val_free_context

```
#include <validator/validator.h>

int val_create_context( char *label,
                        val_context_t **newctx );

void val_free_context( val_context_t *ctx );
```

These function create and release, respectively, validator context objects.

An application maintains a run-time handle to its validator policy through the validator context. val_create_context() creates a new DNSSEC validator context. The label parameter identifies the DNSSEC validator policy to be used by the application for DNSSEC validation. The manner in which the label argument is used within the system configuration to identify specific validator policy settings is implementation-specific. However, all libraries that implement this API MUST internally create a DNSSEC validator context with a (system-defined) default DNSSEC validator policy if label is NULL.

The val_create_context() function MUST return 0 on success, and an error code from Section 7 on failure. Memory for the newly created DNSSEC validator context MUST be returned in the newctx field. This memory MUST be released when applications invoke the val_free_context() function. newctx MUST be set to NULL if an error is encountered.

6.2. val_context_setqflags

```
#include <validator/validator.h>

int val_context_setqflags(val_context_t *context,
                          unsigned char action,
                          unsigned int flags);
```

This function allows an application to set or reset default query flags for a given context. This enables the application to alter the DNSSEC validator processing, while still having most of the granular default configuration specified in its configuration file.

The application may specify one of the following action types, where their numeric values are implementation-specific.

VAL_CTX_FLAG_SET: Set the given flag as one of the default query flags for the context.

VAL_CTX_FLAG_RESET: Reset the given flag if it was set as one of the default query flags for the context.

7. Function Return Codes and p_val_err()

The return values from functions defined in the low-level API, the DNSSEC validator-context API, and the val_get_rrset() function MUST be from the list below. Other high-level API functions mirror existing legacy DNS functions, so the return codes from these functions are identical to their predecessors. The numerical values for the return codes listed below are implementation-specific.

VAL_NO_ERROR: The function call was successful.

VAL_NOT_IMPLEMENTED: The implementation did not support a particular feature.

VAL_RESOURCE_UNAVAILABLE: Some resource necessary for an operation (such as memory) was unavailable.

VAL_BAD_ARGUMENT: An unexpected value was passed as an argument to a function.

VAL_INTERNAL_ERROR: An internal error was encountered by the DNSSEC validator.

VAL_CONF_PARSE_ERROR: The DNSSEC validator configuration was improperly specified in the system configuration.

VAL_CONF_NOT_FOUND: The DNSSEC validator configuration could not be located in the system configuration.

VAL_NO_POLICY: The DNSSEC validator policy identifier being referenced was invalid.

The p_val_err() function is used to convert an error code from the list above to a string representation.

```
#include <validator/validator.h>

const char *p_val_err(int err);
```

The returned value from p_val_err() MAY be the string conversion for the corresponding error code identifier. For example, the return value from p_val_err(VAL_NO_ERROR) MAY be "VAL_NO_ERROR".

8. Evaluating Response Validity

The result of DNSSEC validation for a resource record set, based on the individual status code of each element in an authentication chain, is returned in a variable of type val_status_t. val_status_t can contain one of the possible codes listed in Section 8.1. The functions provided in Section 8.2 simplify the task of evaluating validity of an answer by wrapping around the different status codes possible for each type of answer.

8.1. DNSSEC Validation Status Codes and p_val_status()

A variable of type val_status_t MUST contain one of the following codes (the numerical values for these codes are implementation-specific):

VAL_VALIDATED_ANSWER: Returned if the combined DNSSEC validation status for a set of resource record set responses represents a validated state.

VAL_TRUSTED_ANSWER: Returned if the combined DNSSEC validation status for a set of resource record set responses represents a trusted (but non-validated) state.

VAL_UNTRUSTED_ANSWER: Returned if the combined DNSSEC validation status for a set of resource record set responses represents an untrusted state.

VAL_SUCCESS: The response for the given resource record set was successfully validated through the DNSSEC validation process.

VAL_NONEXISTENT_NAME: The proof for denial of existence for a domain name validated successfully.

VAL_NONEXISTENT_TYPE: The proof for denial of existence for the resource record type for the given name was validated successfully.

VAL_NONEXISTENT_NAME_NOCHAIN: The proof for non-existence of a domain name was considered valid through local DNSSEC validator configuration; the authentication chain(s) for the different components of the proof were not validated.

VAL_NONEXISTENT_TYPE_NOCHAIN: The proof for non-existence of the resource record type for the name queried was considered valid through local DNSSEC validator configuration; the authentication chain(s) for the different components of the proof were not validated.

VAL_PINSECURE: The record or some ancestor of the record in the authentication chain towards a DNSSEC trust anchor was known to be provably insecure and DNSSEC validator policy is configured to trust provably insecure answers.

VAL_PINSECURE_UNTRUSTED: The record or some ancestor of the record in the authentication chain towards a DNSSEC trust anchor was known to be provably insecure, but DNSSEC validator policy is configured to not trust provably insecure answers.

VAL_BARE_RRSIG: The response was for a query of type RRSIG. RRSIGs contain the cryptographic signatures for other DNS data and cannot themselves be validated.

VAL_IGNORE_VALIDATION: DNSSEC validator policy was configured to ignore DNSSEC validation for the zone from where this data was received.

VAL_UNTRUSTED_ZONE: DNSSEC validator policy was configured to not trust any response from the zone that this data was received from.

VAL_OOB_ANSWER: The response was obtained using some out-of-band mechanism (for example, from a local configuration store such as /etc/hosts).

VAL_BOGUS: The response could not be validated due to signature verification failures or the inability to verify proofs of non-existence for one or more components in the authentication chain.

VAL_DNS_ERROR: Returned if a DNS error was encountered during the query resolution process.

VAL_NOTRUST: The authentication chain does not lead up to a configured DNSSEC trust anchor.

The p_val_status() function is used to convert the DNSSEC validation status code stored in a variable of type val_status_t to a string representation.

```
#include <validator/validator.h>

const char *p_val_status(val_status_t status);
```

The value returned MAY be the string conversion for the corresponding val_status_t identifier. For example, the return value from p_val_status(VAL_SUCCESS) MAY be "VAL_SUCCESS".

8.2. High-Level Routines for Evaluating Validity

```
#include <validator/validator.h>

int val_istrusted(val_status_t status);

int val_isvalidated(val_status_t status);

int val_does_not_exist(val_status_t status);
```

These functions return a boolean value indicating whether or not the given val_status_t object is trusted, validated or does not exist (respectively).

Most applications will only be interested in a single value that represents the validity of DNS data. In some instances, an application may also need to distinguish between cases where the answer was cryptographically validated and cases where the answer was locally trusted. The val_istrusted() and val_isvalidated() functions allow an application to evaluate, at a high level, the validity of a response without having to inspect the exact status code returned.

The val_istrusted() function returns a single integer value representing the validity of information returned by the DNSSEC validator. The return value MUST be greater than 0 if status is one of VAL_SUCCESS, VAL_NONEXISTENT_NAME, VAL_NONEXISTENT_TYPE, VAL_NONEXISTENT_NAME_NOCHAIN, VAL_NONEXISTENT_TYPE_NOCHAIN, VAL_PINSECURE, VAL_IGNORE_VALIDATION, VAL_TRUSTED_ANSWER, or VAL_VALIDATED_ANSWER and MUST be equal to 0 for other status codes.

The val_isvalidated() function returns a single integer value that indicates if the answer cryptographically chains down from a configured DNSSEC trust anchor. The return value MUST be greater than 0 if status is one of VAL_SUCCESS, VAL_NONEXISTENT_NAME, VAL_NONEXISTENT_TYPE, or VAL_VALIDATED_ANSWER and MUST be equal to 0 for other status codes.

The val_does_not_exist() function allows an application to determine from the DNSSEC validation status value if the answer was provably non-existent. In combination with the val_istrusted() and val_isvalidated() functions, it can give an indication about the manner in which validity was determined (cryptographically verified or trusted through local DNSSEC validator policy). The return value from val_does_not_exist() MUST be greater than 0 if status is one of VAL_NONEXISTENT_TYPE, VAL_NONEXISTENT_NAME, VAL_NONEXISTENT_NAME_NOCHAIN, or VAL_NONEXISTENT_TYPE_NOCHAIN and MUST be equal to 0 for other status codes.

9. Notes On DNS Data Caching By Appplications

Certain applications are known to cache DNS data for an application-specific length of time, independent of the TTL limits placed on the relevant DNS resource records. Since DNS data is ephemeral by design, any caching performed independently by applications may conflict with zone publishers' needs to change such DNS records frequently. An extension to this problem is the scenario where an application caches DNS data for an application-specific length of time during which period a zone operator may revoke a DNSSEC key, thus rendering that particular cached data as untrustworthy.

It is recommended that applications MUST NOT cache DNS data in a

manner that would violate the TTL limits placed on DNS records. Applications must, instead delegate the function of caching DNS data to a stub resolver or a local recursive resolver library, and to only use DNS API functions to request answers whenever necessary. The stub or recursive resolver libraries should, in turn, determine from the resource record TTLs if a cached answer is available or if a fresh DNS query needs to be issued.

10. IANA Considerations

This document has no actions for IANA.

11. Security Considerations

In certain cases DNS responses may be returned from the local system configuration (for example, from the /etc/hosts file on some systems). The application cannot assume that these answers are valid, unless the application is certain that the local configuration store contains valid data. If this information is modified during a DHCP lookup, for example, the client system should ensure that the DHCP server is a trusted source, and that the communication path between the DHCP server and the client system is secured. If these conditions are not satisfied and if the application chooses to trust a locally available answer, an attacker may be able to poison the system configuration and cause an application to use invalid answers. If applications are to treat out-of-band answers as trusted, this choice SHOULD be made explicit through a validator policy configuration knob.

Applications can similarly choose to trust data from provably insecure zones. Not performing DNSSEC validation for a zone that has DNSSEC intentionally turned off is no worse than the current situation of DNSSEC-unaware applications not being able to detect the integrity of DNS data for such zones.

The DNS search path may affect the result of DNSSEC validation, especially in the current Internet environment where not all DNS name servers are expected to be DNSSEC-aware. If the name server pointed to by the system configuration is not DNSSEC-aware (i.e. it does not return DNSSEC records), DNSSEC validation will not work as expected, unless the validator has certain fallback mechanims in place to try and route around such broken behavior.

The DNSSEC validator configuration information needs to be protected so that it cannot be overwritten by unauthorized users or processes. The system administrator must ensure that the list of DNSSEC trust

anchors is kept accurate and up-to-date. If the DNSSEC trust anchors are outdated (in the event of key-rollovers), the DNSSEC validator may either falsely mark zones as bogus or may operate with the false belief of having validated a response when the response should really have been flagged as bogus. Any subversion of the DNSSEC policy configuration (including definition of new trust anchors) can similarly completely undermine the value provided by DNSSEC.

12. Acknowledgements

A number of individuals have provided valuable feedback and suggestions for improving this document including the following: Lindy Foster, Wayne Morrison, Russ Mundy, Bill Sommerfeld, Wes Hardaker, Giovanni Marzot and Alfred Hoenes. The list of authentication status codes in Section 4.2 was generated through multiple brainstorming sessions at various IETF meetings; this draft draws on the results from that effort.

13. References

13.1. Normative References

[refs.DLV]
Weiler, S., "DNSSEC Lookaside Validation (DLV)", RFC 5074, November 2007.

[refs.IEEE.1003.1-2004]
IEEE and The Open Group, http://www.opengroup.org, "IEEE Std 1003.1-2004 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open Group Technical Standard: Base Specifications, Issue 6", ISO/IEC 9945:2003, February 2004.

[refs.RFC3493]
Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, February 2003.

[refs.RFC4034]
Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[refs.RFC4035]
Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Protocol Modifications for the DNS Security

Extensions", RFC 4035, March 2005.

13.2. Informative References

[refs.RFC1034]
Mockapetris, P., "Domain Names - Concepts and Facilities", RFC 1034, November 1987.

[refs.RFC2672]
Crawford, M., "Non-Terminal DNS Name Redirection", RFC 2672, August 1999.

[refs.RFC4033]
Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

Appendix A. Zone-Specific Validator Policy Settings

Zone-specific validator policy settings may have the following structure.

<label> <attribute> <additional-data>;

Sample values for <attribute> are "trust-anchor", "zone-security-expectation", "provably-insecure-status", "clock-skew". The value for <additional-data> would depend on the type of attribute specified.

o For the "trust-anchor" attribute additional-data could be a sequence of ordered pairs, each consisting of the zone name and a string containing the resource record data for the trust anchor's DNSKEY or DS record. An example is given below.

```
browser trust-anchor
  .   DS  19036  8  2  \
      49AAC11D7B6F6446702E54A1607371607A1A41855200FD2CE\
      1CDDE32F24E8FB5
  example.org DNSKEY 257 3 5 AQO8XS4y9r77X 9SHBmrx MoJf\
      1Pf9AT9Mr/L5BBGtO9/e9f/zl4FFgM2l B6M2 XEm6mp6 mit\
      4tzpB/sAEQw1McYz6bJdKkTiqtuWTCfDmgQhI6 /Ha0 Ef GP\
      NSqnY 99FmbSeWNIRaa4fgSCVFhvbrYq1nXkNVy QPeEVHk o\
      DNCA lr qOA3lw==
;
```

- For the "zone-security-expectation" attribute additional-data could be a sequence of <domain name,value> tuples representing the security expectation for names in that domain, where value could be one of the following:

  ignore: Ignore DNSSEC validation for names under this domain.
  validate: Perform DNSSEC validation of answers received for names under this domain.
  untrusted: Reject all answers received for names under this domain.

  This zone-security-expectation DNSSEC validator policy construct makes it possible to define various islands of trust for DNSSEC-enabled zones and to ignore or trust data from selected zones. The default zone security expectation for a domain should be "validate". In the following example, for DNSSEC validator contexts created with a DNSSEC validator policy label of "browser", DNSSEC validation would only be performed for names under the example.com domain; names under the somebogusdomain.org domain would always considered to be untrusted and DNSSEC validation for all other domain names would be ignored.

```
 browser zone-security-expectation
     example.com  validate
     somebogusdomain.org untrusted
     . ignore
     ;
```

- For the "provably-insecure-status" attribute additional-data could be a sequence of <domain name,value> tuples representing the validity of the provably insecure condition, where value could be one of the following:

  trusted: Treat the provably insecure condition as valid.
  untrusted: Treat the provably insecure condition as invalid.

  The default value for the provably insecure status for a domain should be "trusted". In the following example, for DNSSEC validator contexts created with the default label, the provably insecure condition would be treated as trustworthy for all domains except the net domain, where this condition would be treated as invalid.

```
 : provably-insecure-status
     . trusted
     net untrusted
     ;
```

- For the "clock-skew" attribute additional-data could be a sequence of the domain name and the number of seconds of clock-skew acceptable for signatures on names in that domain. A clock skew value of -1 could have the effect of turning off inception and expiration time checks on signatures from that domain. The default clock skew should be 0. In the following example, for DNSSEC validator contexts created with the "mta" label, signature inception and expiration checks would be disabled for all names under the example.com domain.

```
 mta clock-skew
     example.com -1
     ;
```

Appendix B. Global Validator Policy

Global policy options guide validator behavior across multiple zones. Global policy options for the DNSSEC validator could be defined under a separate section within the validator system configuration. Some of the possible configuration knobs for global validator policy include the following.

- trust-oob-answers <yes/no>: policy on whether or not the validator should trust answers received out-of-band.
- edns0-size <default-edns0-size>: the default EDNS0 size to be advertized in queries sent out by the validator.

Appendix C. Asynchronous API Example Code

The general flow for asynchronous request processing can be described with the following pseudo-code:

```
#include <validator/validator.h>

int done = 0;

int my_callback(val_async_status *async_status, int event,
                val_context_t *ctx, void *user_ctx,
                val_cb_params_t *cbp) {
   if (event == VAL_AS_EVENT_CANCELED) {
      fprint("canceled: %s", (char*)user_ctx);
      return;
   }

   fprintf("final status for %s: %d\n", (char*)user_ctx,
           val_async_status(async_status));
```

```
      done = 1;
      return 0;
   }

   main() {
      val_async_status *async_status;
      struct timeval    tv;
      fd_set            fds;
      int               nfds, ready;
      val_context      *ctx = NULL;
      char             *domain = "www.example.com";

      /* submit request */
      rc = val_async_submit(ctx, domain, ns_c_in, ns_t_a, 0,
                            my_callback, (void*)domain, &async_status);
      while (!done) {
         tv.usec = 0;
         tv.sec = 10;         /* maximum timeout 10 sec */

   #ifdef NO_APPLICATION_FDS

         val_async_check_wait(ctx, NULL, NULL, &tv, 0);

   #else /* HAVE_OUR_OWN_FDS_TO_WATCH */

         FD_ZERO(&fds);       /* clear fd_set */
         nfds = 0;            /* no FDs yet */

         /* set FDs for pending requests. application should also set
          * its own FDs, if any, before calling select */
         val_async_select_info(&ctx, &fds, &numfds, &tv);
         ready = select(numfds+1, &fds, NULL, NULL, &tv);
         if ( ready < 0 ) {
            break; /* or continue... */
         } else if ( ready == 0 ) {
            /* application timeout processing */
         } else {
            /* application FD processing */
         }
         /* handle async FDs/timeouts */
         val_async_check(&ctx, &fds, &numfds, flags);

   #endif
      }
   }
```

Authors' Addresses

Suresh Krishnaswamy
SPARTA, Inc.
7110 Samuel Morse Dr.
Columbia, MD 21046
US

Email: suresh AT sparta.com

Robert Story
SPARTA, Inc.

Email: rstory AT sparta.com

Abhijit Hayatnagarkar