DNSSEC-Tools Developers Guide

1 December 2006

SUBMITTED BY
SPARTA, Inc.
7110 Samuel Morse Dr.
Columbia, MD 21046-3401

This page intentionally left blank.

Table of Contents

This page intentionally left blank.

1 Introduction

This document is intended to assist developers wishing to enhance applications to perform DNSSEC validation when resolving domain names.

This document assumes a basic level of knowledge about DNSSEC.

2 Validator Library Developer's Guide

This section describes how one can use the interfaces provided by the validator library, ***libval(3)***, to build DNSSEC-aware programs.

The validator library allows applications to check the authenticity of data returned by the DNS by using the security extensions provided by DNSSEC. The library exports a number of interfaces that applications may use in place of legacy resolver interfaces provided by the ***libres(3)*** library. Additional interfaces that allow more visibility into the validation process are also available. This allows for the creation of applications that are either only interested in a basic "trusted" or "not-trusted" result, or more sophisticated applications that can look for specific errors as a sign of some network abnormality or attack.

The process of transitioning an application from using simple DNS towards being DNSSEC-aware consists of three steps –

1. Instrumentation – making applications capable of seeing DNSSEC results
2. Action – making applications act upon DNSSEC results
3. Policy – allowing applications control over how DNSSEC status values are determined.

Section 2.1 describes the interfaces an application would use in order access the results of DNSSEC evaluation. Section 2.2 describes the types of results the ***libval(3)*** interfaces can return after validation. Section 2.3 introduces validator policy and describes the syntax of the validator configuration file. Section 2.4 describes how applications can be made more flexible in their choosing of validator policy at run time. Section 2.5 describes other validator interfaces that allow the application to obtain greater detail about the validation process. Section 2.6 describes how one could write their own custom resolver interfaces using the interfaces described in Section 2.5. Section 2.7 describes how DNSSEC validation was added to an existing application.

2.1 Application Instrumentation for DNSSEC

Applications that only use **res_query()**, **gethostbyname()** and **getaddrinfo()** from the ***libres(3)*** for name resolution can be easily made DNSSEC-aware by using similarly named interfaces provided by the validator library - **val_res_query()**, **val_gethostbyname()** and **val_getaddrinfo()**. These interfaces perform DNSSEC validation of DNS responses in addition to name resolution. It is also possible for applications that use other interfaces for querying the DNS to be made DNSSEC-aware, but this requires a little more effort. This is described in Section 2.6.

The prototypes of **val_res_query()**, **val_gethostbyname()** and **val_getaddrinfo()** are given below:

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

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

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

val_res_query(), **val_gethostbyname()** and **val_getaddrinfo()** are semantically equivalent to **res_query()**, **gethostbyname()** and **getaddrinfo()** respectively, with slightly differing syntaxes. The essential difference between the interfaces provided by ***libval(3)*** and ***libres(3)*** is that the former includes additional parameters for controlling the validation process and returning the value of the DNSSEC status.

The validation policy is encapsulated in an opaque validator context object, which has a type of *val_context_t*. Setting the context parameter in interfaces exported by the ***libval*** library to NULL will result in the library using the default validator policy. Configuration of validator policy is described in section 2.3.

The **val_getaddrinfo()** function does have one important difference from its counterpart. The structure which is used to return the addresses is a *val_addrinfo* structure, which has an additional member, *ai_val_status*, to return the validation status for each address.

Return values from each of the libval APIs are similar to their ***libres(3)*** counterparts and similar tests for success and failure must be made following invocation of these calls. Also, the structures allocated by the **val_gethostbyname()** and **val_getaddrinfo()** interfaces must be released when they are no longer required, using the **val_freehostent()** and **val_freeaddrinfo()** interfaces, respectively.

2.1.1 Example code

The following code snippets gives examples of how one might change code using existing libres interfaces in place of the ones provided by ***libval(3)***.

2.1.1.1 val_res_query()

An application using **res_query()** can easily be updated to use **val_res_query()**.

```
int use_val_res_query(int argc, char *argv[])
{
    val_status_t val_status;
    unsigned char answer[8096];
    int ret_val, answer_len;

    if (argc < 2) {
        printf("Usage: %s <domain-name>\n", argv[0]);
        exit(1);
    }

    length = val_res_query(NULL, argv[1], ns_c_in, ns_t_a,
                           answer, sizeof(answer), &val_status);
    if ((lenght < 0) || (length > sizeof(answer)) {
       /* error handling */
    }
    else if (!val_istrusted(val_status))
        printf("*** WARNING: DNS resolution is not trusted (%s)",
               p_val_status(val_status));
    else
        printf("ValStatus: %s", p_val_status(val_status));

    return 0;
}
```

2.1.1.2 val_gethostbyname()

Similarly, **gethostbyname()** can be updated to **val_gethostbyname()**. Note that **val_freehostent()** must be used in place of any existing **freehostent()** calls.

```
int use_val_gethostbyname(int argc, char *argv[])
{
    val_status_t val_status;
    struct hostent *h = NULL;

    if (argc < 2) {
        printf("Usage: %s <hostname>\n", argv[0]);
        exit(1);
    }

    h = val_gethostbyname(NULL, argv[1], &val_status);
    if (h)
        printf("DNSSEC Status = %d [%s]\n", val_status, p_val_status(val_status));

    return 0;
}
```

2.1.1.3 val_getaddrinfo()

Finally, **getaddrinfo()** can be updated to **val_getaddrinfo()**. Note that **val_freeaddrinfo()** must be used in place of any existing **freeaddrinfo()** calls.

```
int use_val_getaddrinfo(int argc, char *argv[])
{
    int ret_val;
    struct val_addrinfo *ainfo = NULL;

    if (argc < 2) {
        printf("Usage: %s <hostname>\n", argv[0]);
        exit(1);
    }

    ret_val = val_getaddrinfo(NULL, argv[1], NULL, NULL, &ainfo);
    if (retval == VAL_NO_ERROR)
        printf("DNSSEC Status = %d [%s]\n", ainfo->ai_val_status,
               p_val_status(ainfo->ai_val_status));
    if (ainfo!= NULL)
        val_freeaddrinfo(ainfo);

    return 0;
}
```

2.2 DNSSEC Status Values Returned by libval

The result from DNSSEC processing is available to the application as an integer value. There are different ways to obtain the DNSSEC status for a response to a query based on the interface used.

- **val_res_query()**: The *val_status* parameter.
- **val_gethostbyname()**: The *val_status* parameter.
- **val_getaddrinfo()**: The *ai_val_status* member of the *val_addrinfo* parameter.

A successful call will have a status of VAL_NO_ERROR. Several convenience functions are available to help interpret validator status values. Their prototypes are:

```
int val_istrusted(int valerrno);
int val_isvalidated(int valerrno);
char *p_val_status(int valerrno);
```

val_istrusted(), can be used to determine if a DNSSEC status values is considered trusted. This function returns 0 for untrusted status values, and a non-zero value for untrusted statuses. The status values that are considered trusted are: VAL_SUCCESS, VAL_NONEXISTENT_NAME, VAL_NONEXISTENT_TYPE, VAL_NONEXISTENT_NAME_NOCHAIN, VAL_NONEXISTENT_TYPE_NOCHAIN, VAL_NONEXISTENT_NAME_OPTOUT, VAL_PROVABLY_UNSECURE, VAL_IGNORE_VALIDATION, VAL_TRUSTED_ZONE and VAL_LOCAL_ANSWER.

val_isvalidated(), can be used to determine if a trusted DNSSEC status value is cryptographically validated. This function returns 0 for validated status values, and a non-zero

value for non-validated statuses (every thing except `VAL_SUCCESS`, `VVAL_NONEXISTENT_NAME` and `VAL_VAL_NONEXISTENT_TYPE`).

DNSSEC status values can be displayed as text using the **`p_val_status()`** interface.

2.3 Validator Configuration

The validator library reads configuration information from two separate files. By default, these files are `/etc/resolv.conf` and `/etc/dnsval.conf`.

2.3.1 Validator Resolver configuration

Resolver configuration is specified in `/etc/resolv.conf`. Only the `nameserver` option is supported in the `resolv.conf` file. All other options are silently ignored. The `nameserver` option is used to specify the IP address of the name server to which queries will be sent by default.

```
nameserver 10.0.0.1
```

An application that wishes to use a different file for the resolver configuration can specify the file by passing the full path to **`resolver_config_set()`**.

2.3.2 Validator Policy configuration

Local policy for the validator is stored in the local configration system (typically the configuration file, /etc/dnsval.conf). Policies are identified by simple text strings called labels, which must be unique within the configuration system. As an example, "browser" could be used as the label that defines the validator policy for all web-browsers in a system. A label value of ":" identifies the "default" policy, or the policy that is used when a NULL context is specified as the ctx parameter. The default policy is unique within the configuration system. Furthermore, the ':' character is only allowed in the default policy label. If the policy identified by the ':' label does not exist in the configuration system, the first policy in the configuration system is used as the default policy.

```
<label> <keyword> <additional-data>;
```

Currently two different values for *keyword* are defined: `trust-anchor` and `zone-security-expectation`.

An application that wishes to use a different file for the validator policy configuration can specify the file by passing the full path to **`dnsval_conf_set()`**.

2.3.2.1 Trust Anchor configuration

A trust-anchor policy fragment specifies the trust anchors for a sequence of zones. The additional data portion for this keyword is a sequence of the zone name and a quoted string

containing the RDATA portion for the trust anchor's DNSKEY. For example:

```
# <label> trust-anchor <zone> <zone-key> [<zone> <zone-key>];

mozilla trust-anchor
        fruits.netsec.tislabs.com.
            "257 3 5
            AQO8XS4y9r77X9SHBmrxMoJf1Pf9AT9Mr/L5BBGtO9/e9f/zl4FFgM2l
            B6M2XEm6mp6mit4tzpB/sAEQw1McYz6bJdKkTiqtuWTCfDmgQhI6/HaO
            EfGPNSqnY 99FmbSeWNIRaa4fgSCVFhvbrYq1nXkNVyQPeEVHkoDNCAlr
            qOA3lw=="
        netsec.tislabs.com.
           "257 3 5
            AQO8XS4y9r77X9SHBmrxMoJf1Pf9AT9Mr/L5BBGtO9/e9f/zl4FFgM2l
            B6M2XEm6mp6mit4tzpB/sAEQw1McYz6bJdKkTiqtuWTCfDmgQhI6/HaO
            EfGPNSqnY 99FmbSeWNIRaa4fgSCVFhvbrYq1nXkNVyQPeEVHkoDNCAlr
            qOA3lw=="
        ;
```

2.3.2.2 Zone Security Expectation configuration

A zone-security policy fragment specifies the local security expectation for a zone. The additional data portion for this keyword is a sequence of the zone name and its trust status – trusted, untrusted or ignore.

A zone security policy of ignore or trusted will result in no validation being done for the specified zone. A policy of validate indicates that the specified zone should be validated up to a configured trust anchor.

Here is an example configuration for zone security expectations:

```
# <label> zone-security-expectation <zone> <validate|ignore|untrusted>

validate_all zone-security-expectation . validate;

validate_none zone-security-expectation . ignore;

mozilla zone-security-expectation wesh.fruits.netsec.tislabs.com untrusted;
```

2.4 Controlling Validator Behavior Within Applications

The validator library allows multiple applications to operate with differing validator policies at the same instance. A handle to each of the different instantiations of the validator under different policies is available through the validator context. The prototypes for the policy manipulation functions are:

```
int val_create_context(const char *scope, val_context_t **newcontext);
void val_free_context(val_context_t *context);
```

Applications can create a new validator context using the **val_create_context()** method. This method parses the resolver and validator configuration files and creates a new context object, which is returned via the handle *newcontext*.

The val_create_context() creates a handle to a validator policy context. The scope parameter identifies the particular policy to be used as the active policy for the context during validation. Policy scopes have a hierarchical organization, with each member in the hierarchy identifying a policy label in the configuration system. The effective policy for the context is determined by the relative ordering of the first occurrence of the label in the policy scope. The validator forms an effective policy by cumulatively applying the policies for each label in the hierarchy. For example, for the policy scope "mozilla:browser", the effective policy is computed by applying policies for the "browser" and "mozilla" policies, in that order. A ':' character in the policy scope is used to indicate the default policy in the hierarchy. A NULL scope creates a context with only the default base policy .

Contexts that were created using the **val_create_context()** interface should be destroyed when they are no longer needed, using the **val_free_context()** method.

2.5 Observing the Finer Details

The **val_resolve_and_check()** interface forms the core of the validator functionality and can be used to query a set of name servers for a tuple, verify and validate the response. Verification is the step of checking the RRSIGs and validation includes performing verification up the chain of trust all the way to a trust anchor. All of the interfaces described above that are used to fetch and validate DNS answers make use of this interface.

```
int val_resolve_and_check( val_context_t *context, u_char *domain_name_n,
                           const u_int16_t type, const u_int16_t class,
                           const u_int8_t flags, struct val_result_chain
**results);
void free_result_chain(struct val_result **results);
```

The *domain_name_n* parameter is the queried name in DNS wire format. The conversion from host format to DNS wire format can be done using the **ns_name_pton()** ***libres(3)*** helper function that is also exported by ***libval(3)***.

The ***libval*** library internally allocates memory for this parameter, so that memory must be freed by the invoking application using the **val_free_result_chain()** interfaces.

val_resolve_and_check() returns `NO_ERROR` on success. Answers returned by **val_resolve_and_check()** are made available in the *results* chain. Each answer is a distinct RRset; multiple RRs within the RRset are treated as the same answer. Multiple answers are possible when *type* is `ns_t_any` or `ns_t_rrsig`.

The *val_rc_answer* member of *results* points to the authentication chain for the RRset. The authentication chain contains the actual RRset returned by the name server in response

to the query. These structures allow an application to investigate the DNSSEC validation status of each link in the validation chain. Most applications will only require the first *result* structure in the results chain, since this provides a single error code for representing the authenticity of returned data. Other more intrusive applications such as a DNSSEC troubleshooting utility may look at the individual authentication chain elements to identify which particular component in the chain-of-trust led to validation failure, if any.

Upon completion of the **val_resolve_and_check()** call, the *val_ac_status* element within the authentication chain element structure contains the validation status of a particular authentication chain element. This generally contains a different value than the final DNSSEC status returned by higher level applications since it only applies to the given authentication chain element. Successful DNSSEC Status values can only be generated if every component in the chain of trust was validated, and the chain culminates with a trusted record. The *val_ac_status* field may contain one of the values described in the following sections for an authentication chain element. authentication chain element status values may be displayed as text using the **p_val_status()** interface.

2.5.1 Authentication Chain Status Values

2.5.1.1 RRSet status values

- VAL_SUCCESS: Answer received and validated successfully.
- VAL_LOCAL_ANSWER: Answer was available from a local file.
- VAL_BARE_RRSIG: No DNSSEC validation possible, query was for an RRSIG.
- VAL_NONEXISTENT_NAME: No name was present and a valid proof of non-existence confirming the missing name (NSEC or NSEC3 span) was returned. The components of the proof were also individually validated.
- VAL_NONEXISTENT_TYPE: No type exists for the name and a valid proof of non-existence confirming the missing name (NSEC or NSEC3 span) was returned. The components of the proof were also individually validated.
- VAL_NONEXISTENT_NAME_NOCHAIN: No name was present and a valid proof of non-existence confirming the missing name (NSEC or NSEC3 span) was returned. The components of the proof were also identified to be trustworthy, but they were not individually validated.
- VAL_NONEXISTENT_TYPE_NOCHAIN: No type exists for the name and a valid proof of non-existence confirming the missing name (NSEC or NSEC3 span) was returned. The components of the proof were also identified to be trustworthy, but they were not individually validated.
- VAL_ERROR: Did not have sufficient or relevant data to complete validation, or encountered a DNS error.
- (VAL_DNS_ERROR_BASE + SR_error): This value contains a resolver error from libsres. The libsres error is added to VAL_DNS_ERROR_BASE, so this value will lie between VAL_DNS_ERROR_BASE and VAL_DNS_ERROR_LAST.
- VAL_INDETERMINATE: Lacking information to give a more conclusive answer.
- VAL_BOGUS: Validation failure condition.
- VAL_NOTRUST: All available components in the authentication chain verified properly, but there was no trust anchor available.
- VAL_IGNORE_VALIDATION: Local policy was configured to ignore validation for the zone from which this data was received.
- VAL_TRUSTED_ZONE: Local policy was configured to trust any data received from the given zone.
- VAL_UNTRUSTED_ZONE: Local policy was configured to reject any data received from the given zone.
- VAL_PROVABLY_UNSECURE: The record or some ancestor of the record in the authentication chain towards the trust anchor was known to be provably unsecure.

2.5.1.2 Validation Chain Status Values

- VAL_AC_UNSET: The status was not set.

- VAL_AC_DATA_MISSING: No data was returned for a query and the DNS did not indicate an error.
- VAL_AC_RRSIG_MISSING: RRSIG data could not be retrieved for a resource record.
- VAL_AC_DNSKEY_MISSING: The DNSKEY for an RRSIG covering a resource record could not be retrieved.
- VAL_AC_DS_MISSING: The DS record covering a DNSKEY record was not available.
- VAL_AC_UNTRUSTED_ZONE: Local policy defined a given zone as untrusted, with no further validation being deemed necessary.
- VAL_AC_UNKNOWN_DNSKEY_PROTOCOL: The DNSKEY protocol number was unrecognized.
- VAL_AC_NOT_VERIFIED: All RRSIGs covering the RRset could not be verified.
- VAL_AC_VERIFIED: At least one RRSIG covering a resource record had a status of VAL_AC_RRSIG_VERIFIED.
- VAL_AC_LOCAL_ANSWER: The answer was obtained locally (e.g., from /etc/hosts) and validation was not performed on the results.
- VAL_AC_TRUST_KEY: A given DNSKEY or a DS record was locally defined to be a trust anchor.
- VAL_AC_IGNORE_VALIDATION: Validation for the given resource record was ignored, either because of some local policy directive or because of some protocol-specific behavior.
- VAL_AC_TRUSTED_ZONE: Local policy defined a given zone as trusted, with no further validation being deemed necessary.
- VAL_AC_PROVABLY_UNSECURE: The authentication chain from a trust anchor to a given zone could not be constructed due to the provable absence of a DS record for this zone in the parent.
- VAL_AC_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_AC_NO_TRUST_ANCHOR: There was no trust anchor configured for a given authentication chain.
- (VAL_DNS_ERROR_BASE + SR_error): This value contains a resolver error from libsres. The libsres error is added to VAL_DNS_ERROR_BASE, so this value will lie between VAL_DNS_ERROR_BASE and VAL_DNS_ERROR_LAST. These values include the following:
    - SR_CONFLICTING_ANSWERS: Multiple conflicting answers received for a query.
    - SR_REFERRAL_ERROR: Some error encountered while following referrals.
    - SR_MISSING_GLUE: Glue was missing.

2.5.1.3 Signature Status Values

- VAL_AC_RRSIG_VERIFIED: The RRSIG verified successfully.
- VAL_AC_WCARD_VERIFIED: A given RRSIG covering a resource record shows that the record was wildcard expanded.
- VAL_AC_RRSIG_VERIFY_FAILED: A given RRSIG covering an RRset was bogus.
- VAL_AC_DNSKEY_NOMATCH: An RRSIG was created by a DNSKEY that did not exist in the apex keyset.

- VAL_AC_RRSIG_ALGORITHM_MISMATCH: The keytag referenced in the RRSIG matched a DNSKEY but the algorithms were different.
- VAL_AC_WRONG_LABEL_COUNT: The number of labels on the signature was greater than the count given in the RRSIG RDATA.
- VAL_AC_BAD_DELEGATION: An RRSIG was created with a key that did not exist in the parent DS record set.
- VAL_AC_RRSIG_NOTYETACTIVE: The RRSIG's inception time is in the future.
- VAL_AC_RRSIG_EXPIRED: The RRSIG had expired.
- VAL_AC_INVALID_RRSIG: The RRSIG could not be parsed.
- VAL_AC_ALGORITHM_NOT_SUPPORTED: The RRSIG algorithm was not supported.
- VAL_AC_UNKNOWN_ALGORITHM: The RRSIG algorithm was unknown.
- VAL_AC_ALGORITHM_REFUSED: The RRSIG algorithm was not allowed as per local policy.

2.5.1.4 DNSKEY Status Values

- VAL_AC_SIGNING_KEY: This DNSKEY was used to create an RRSIG for the resource record set.
- VAL_AC_VERIFIED_LINK: This DNSKEY provided the link in the authentication chain from the trust anchor to the signed record.
- VAL_AC_UNKNOWN_ALGORITHM_LINK: This DNSKEY provided the link in the authentication chain from the trust anchor to the signed record, but the DNSKEY algorithm was unknown.
- VAL_AC_INVALID_KEY: The key used to verify the RRSIG was not a valid DNSKEY.
- VAL_AC_KEY_TOO_LARGE: Local policy defined the DNSKEY size as being too large.
- VAL_AC_KEY_TOO_SMALL: Local policy defined the DNSKEY size as being too small.
- VAL_AC_KEY_NOT_AUTHORIZED: Local policy defined the DNSKEY to be unauthorized for validation.
- VAL_AC_ALGORITHM_NOT_SUPPORTED: The DNSKEY or DS algorithm was not supported.
- VAL_AC_UNKNOWN_ALGORITHM: The DNSKEY or DS algorithm was unknown.
- VAL_AC_ALGORITHM_REFUSED: The DNSKEY or DS algorithm was not allowed as per local policy.

2.5.1.5 Insufficient Data status values

In cases where data is insufficient to generate a validation result, *val_ac_status* may also contain the following status values. The final validation result in such circumstances is VAL_ERROR.

- VAL_AC_DATA_MISSING: No data was returned in the response.
- VAL_AC_IRRELEVANT_PROOF: An NSEC received does not contribute towards proving non-existence.

2.5.1.6 DNS Error status values

There are also many DNS error status values.

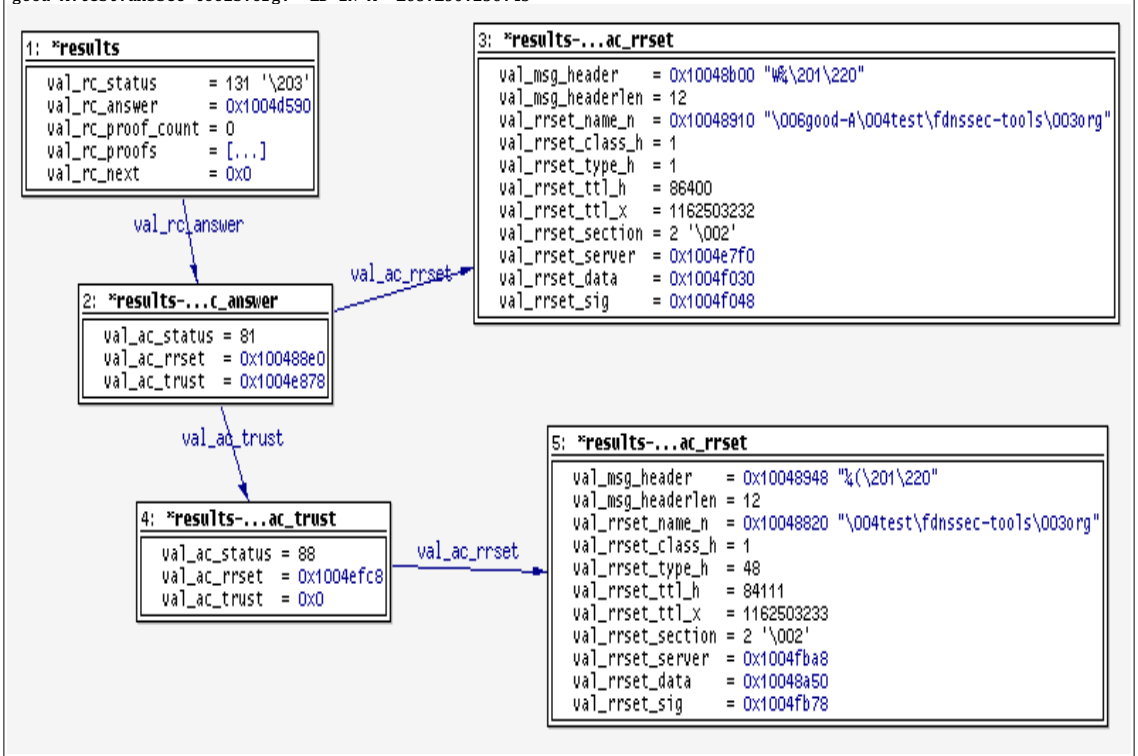
- SR_REFERRAL_ERROR: Some error encountered while following referrals.
- SR_MISSING_GLUE: Glue was missing.
- SR_INTERNAL_ERROR: The resolver encountered an internal error.
- SR_TSIG_ERROR: An error occurred during TSIG processing.
- SR_NO_ANSWER: No response was received.
- SR_WRONG_ANSWER: The response was not a valid response for the query.
- SR_HEADER_BADSIZE: The message size was not consistent with header values.
- SR_NXDOMAIN: The rcode of the response was NXDOMAIN.
- SR_FORMERR: The rcode of the response was FORMERR.
- SR_SERVFAIL: The rcode of the response was SERVFAIL.
- SR_NOTIMPL: The rcode of the response was NOTIMPL.
- SR_REFUSED: The rcode of the response was REFUSED.
- SR_DNS_GENERIC_ERROR: The response was received with the rcode set to one of the well-known error values (NXDOMAIN, FORMERR, SERVFAIL, NOTIMPL or REFUSED).
- SR_EDNS0_VERSION_ERROR: The EDNS0 version was not recognized.
- SR_UNSUPP_EDNS0_LABEL: The EDNS0 label is not supported.
- SR_NAME_EXPANSION_FAILURE: DNS name uncompression failed.
- SR_REFERRAL_ERROR: Referral could not be successfully followed.
- SR_MISSING_GLUE: Glue records were not available for a referral.
- SR_CONFLICTING_ANSWERS: Multiple conflicting answers received for a query.

2.5.2 Example Results for validate Application

Here are a few examples of some results returned by the validator for the dnssec-tools.org test zone. To help in understanding the returned data, debugging options are specified, which output the individual status values for each record. Additionally, screen captures from a debugging session are included, show the relationships between the data structures.

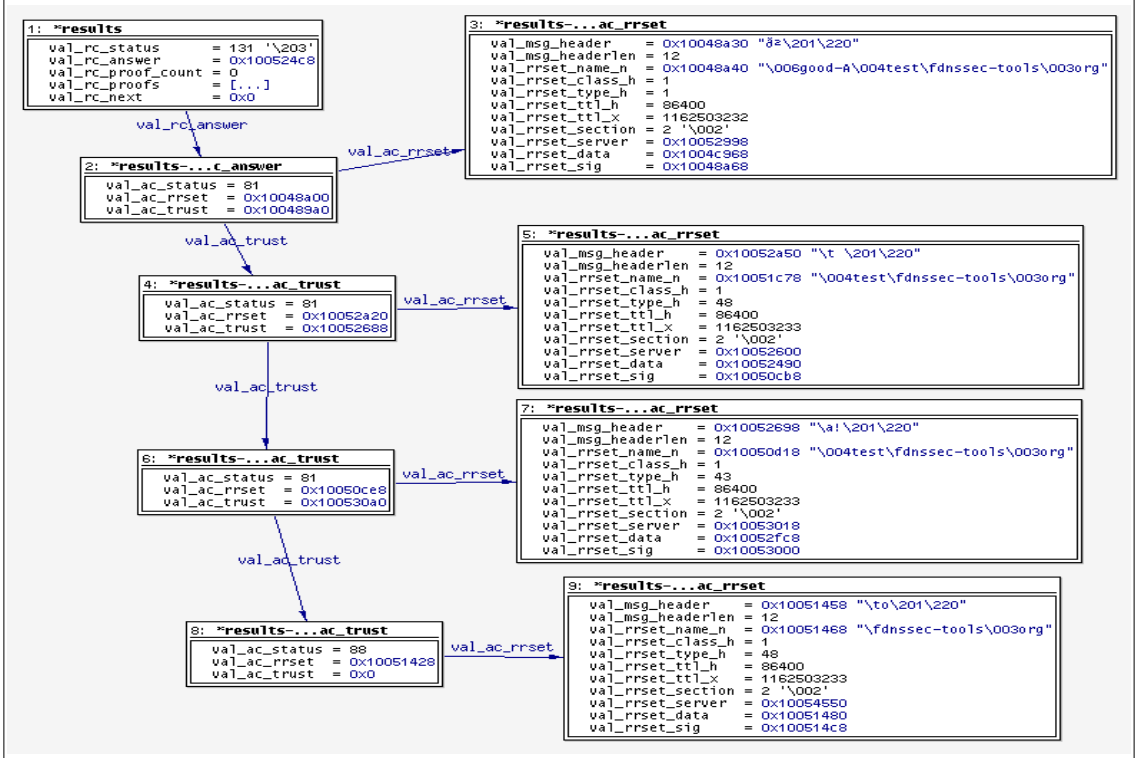
2.5.2.1 Validated Answer from a Trusted Zone

```
$validate -p -T 1
Original query: name=good-A.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3, Query-status=Q_ANSWERED:4
  Result: VAL_SUCCESS:131
    name=good-A.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3 status=VAL_AC_VERIFIED:81
    name=test.dnssec-tools.org class=IN type=DNSKEY[tag=28827] from-server=192.168.1.3 status=VAL_AC_TRUST_KEY:88
DNSSEC status: VAL_SUCCESS [131]
Validated response:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 0
;; flags: ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;;        good-A.test.dnssec-tools.org, type = A, class = IN
good-A.test.dnssec-tools.org.  1D IN A  168.150.236.43
```



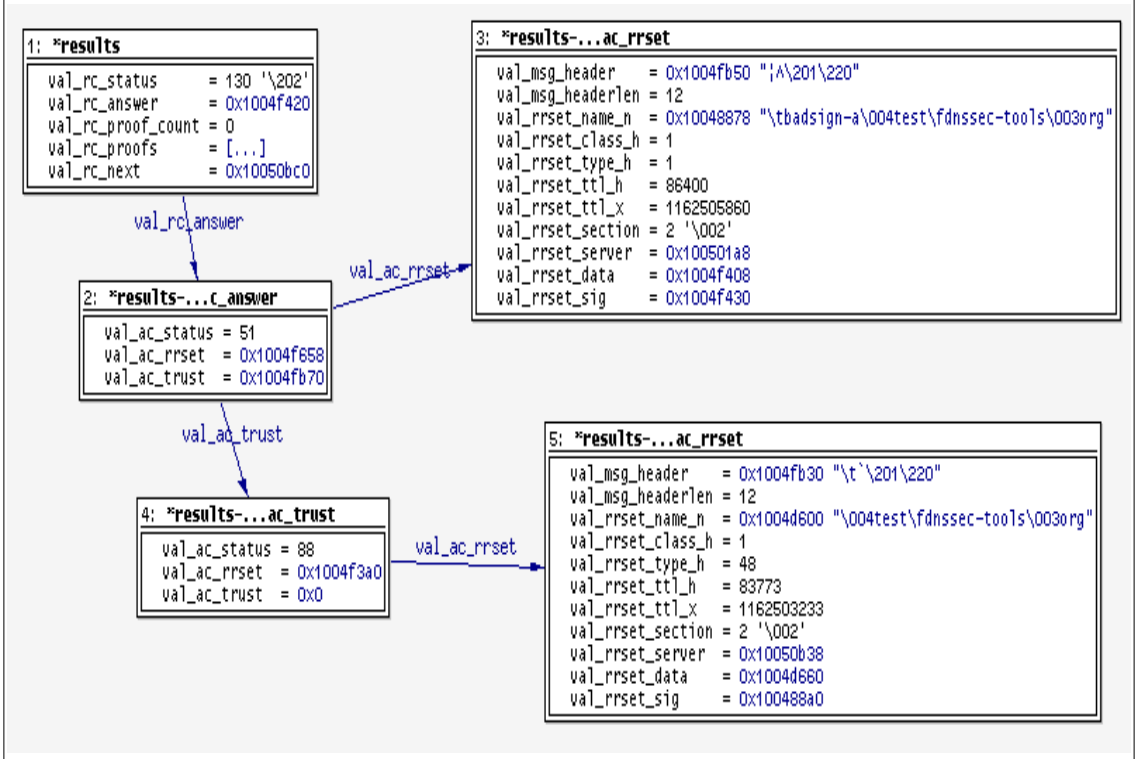
2.5.2.2 Validated Answer from a Sub-domain of a Trusted Zone

```
$ validate  -p -T 1
Original query: name=good-A.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3, Query-status=Q_ANSWERED:4
  Result: VAL_SUCCESS:131
    name=good-A.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3 status=VAL_AC_VERIFIED:81
    name=test.dnssec-tools.org class=IN type=DNSKEY from-server=192.168.1.3 status=VAL_AC_VERIFIED:81
    name=test.dnssec-tools.org class=IN type=DS from-server=192.168.1.3 status=VAL_AC_VERIFIED:81
    name=dnssec-tools.org class=IN type=DNSKEY[tag=42375] from-server=192.168.1.3 status=VAL_AC_TRUST_KEY:88
DNSSEC status: VAL_SUCCESS [131]
Validated response:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 0
;; flags: ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;;        good-A.test.dnssec-tools.org, type = A, class = IN
good-A.test.dnssec-tools.org.  1D IN A  168.150.236.43
```



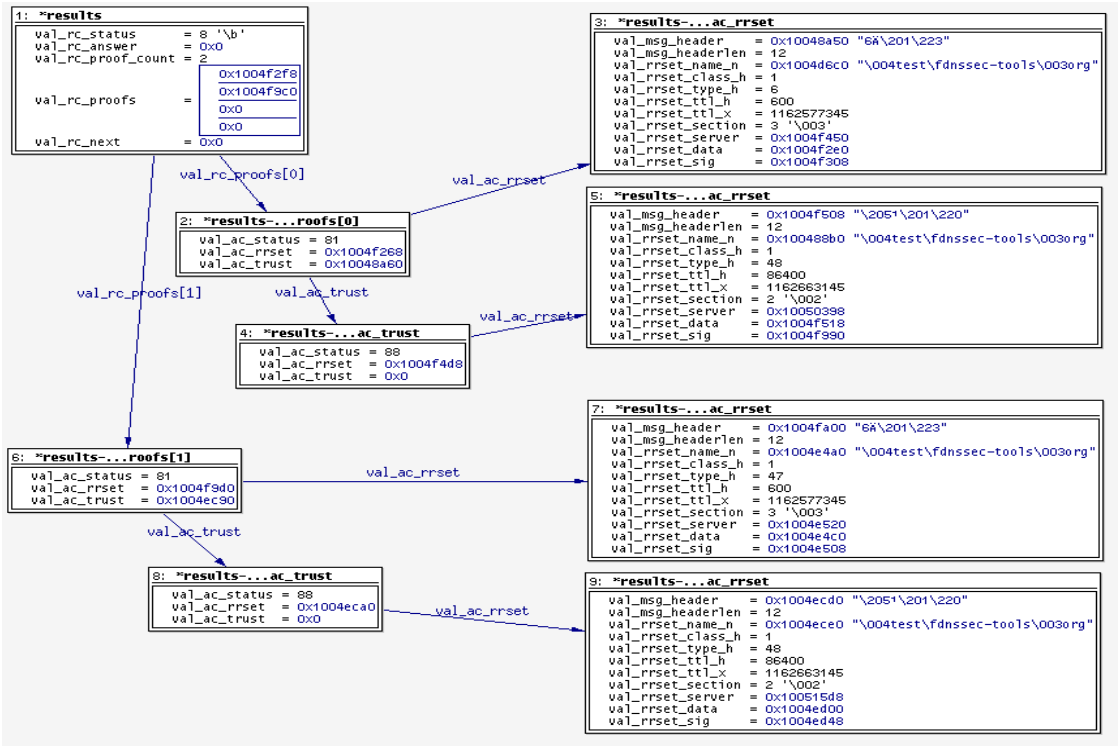
2.5.2.3 Non-validated Answer from a Trusted Zone (Bogus)

```
$ validate -p -T 8
Original query: name=good-cname-to-badsign-A.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3, Query-
status=Q_ANSWERED:4
  Result: VAL_BOGUS:130
    name=badsign-a.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3 status=VAL_AC_NOT_VERIFIED:51
    name=test.dnssec-tools.org class=IN type=DNSKEY[tag=28827] from-server=192.168.1.3 status=VAL_AC_TRUST_KEY:88
  Result: VAL_SUCCESS:131
    name=good-cname-to-badsign-A.test.dnssec-tools.org class=IN type=CNAME from-server=192.168.1.3
status=VAL_AC_VERIFIED:81
    name=test.dnssec-tools.org class=IN type=DNSKEY[tag=28827] from-server=192.168.1.3 status=VAL_AC_TRUST_KEY:88
DNSSEC status: VAL_SUCCESS [131]
Validated response:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 0
;; flags: ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;;        good-cname-to-badsign-A.test.dnssec-tools.org, type = A, class = IN
good-cname-to-badsign-A.test.dnssec-tools.org.  1D IN CNAME  badsign-a.test.dnssec-tools.org.

DNSSEC status: VAL_BOGUS [130]
Non-validated response:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 0
;; flags:; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;;        good-cname-to-badsign-A.test.dnssec-tools.org, type = A, class = IN
badsign-a.test.dnssec-tools.org.  1D IN A  168.150.236.43
```



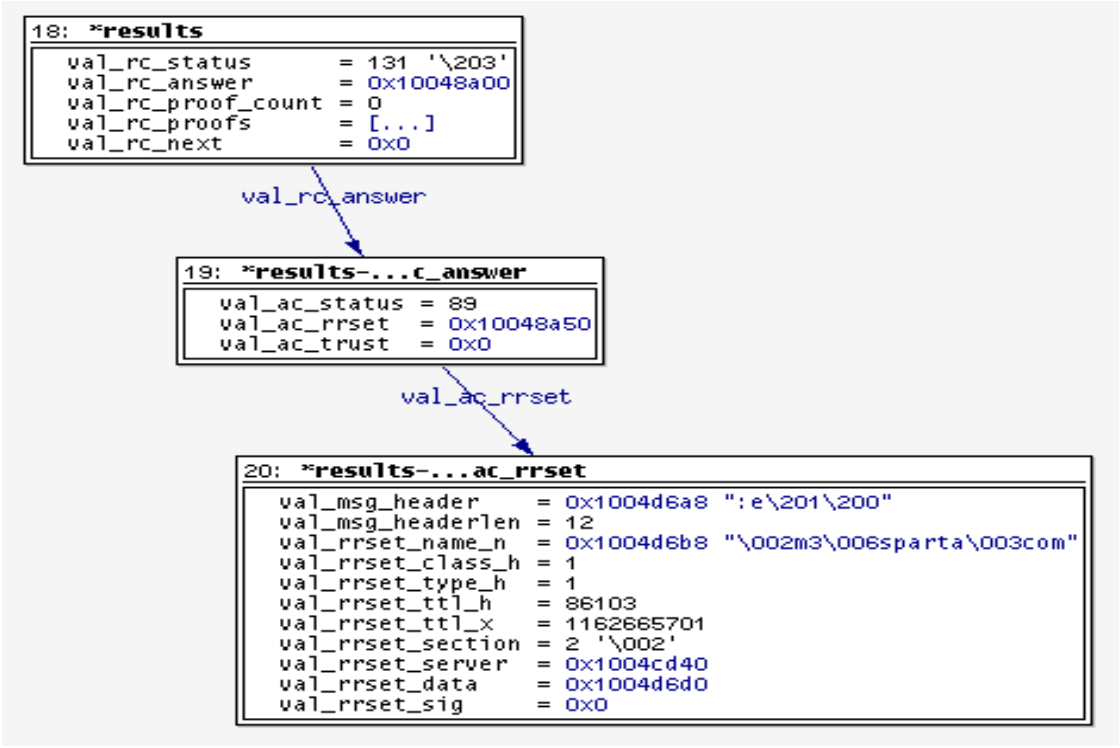
2.5.2.4 NXDOMAIN from a Trusted Zone (Proof of non-existence)

```
Original query: name=addedlater-A.test.dnssec-tools.org class=IN type=A from-server=192.168.1.3, Query-
status=Q_ANSWERED:4
  Result: VAL_NONEXISTENT_NAME:8
    Associated Proofs Follow:
      name=test.dnssec-tools.org class=IN type=SOA from-server=192.168.1.3 status=VAL_AC_VERIFIED:81
      name=test.dnssec-tools.org class=IN type=DNSKEY[tag=28827] from-server=192.168.1.3 status=VAL_AC_TRUST_KEY:88
      name=test.dnssec-tools.org class=IN type=NSEC from-server=192.168.1.3 status=VAL_AC_VERIFIED:81
      name=test.dnssec-tools.org class=IN type=DNSKEY[tag=28827] from-server=192.168.1.3 status=VAL_AC_TRUST_KEY:88

DNSSEC status: VAL_NONEXISTENT_NAME [8]
Non-validated response:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 0
;; flags: ad; QUERY: 1, ANSWER: 0, AUTHORITY: 2, ADDITIONAL: 0
;;        addedlater-A.test.dnssec-tools.org, type = A, class = IN
test.dnssec-tools.org.        10M IN SOA         dns.test.dnssec-tools.org. hardaker.test.dnssec-tools.org. (
                                                 1161967857          ; serial
                                                 2H                  ; refresh
                                                 1H                  ; retry
                                                 1W                  ; expiry
                                                 10M )               ; minimum

test.dnssec-tools.org.        10M IN NSEC        baddata-a.test.dnssec-tools.org. NS SOA TXT RRSIG NSEC DNSKEY
```



2.5.2.5 Answer from an Unsigned Zone (Unsecure)

```
Original query: name=m3.sparta.com class=IN type=A from-server=192.168.1.3, Query-
status=Q_ANSWERED:4
  Result: VAL_SUCCESS:131
    name=m3.sparta.com class=IN type=A from-server=192.168.1.3 status=VAL_AC_IGNORE_VALIDATION:89

DNSSEC status: VAL_SUCCESS [131]
Validated response:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 0
;; flags: ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;;       m3.sparta.com, type = A, class = IN
m3.sparta.com.           23h45m35s IN A   157.185.61.3
```



2.6 DNSSEC Instrumentation For Unsupported Resolver Calls

Applications that make use of resolver calls that are not directly exported by the validator library would need to implement such wrapper functions themselves. The **val_resolve_and_check()** interface is generic enough to allow almost any name resolution functionality to be made DNSSEC-aware. The following code snippet illustrates how **val_query()** is internally implemented using the **val_resolve_and_check()** interface.

```
int
val_query(val_context_t * ctx,
          const char *domain_name,
          const u_int16_t class_h,
          const u_int16_t type,
          const u_int8_t flags, struct val_response **resp)
{
    struct val_result_chain *results = NULL;
    int             retval;
    val_context_t  *context;
    u_char          name_n[NS_MAXCDNAME];

    if ((resp == NULL) || (domain_name == NULL))
        return VAL_BAD_ARGUMENT;
    *resp = NULL;

    if (ctx == NULL) {
        if (VAL_NO_ERROR != (retval = val_create_context(NULL, &context)))
            return retval;
    } else
        context = (val_context_t *) ctx;

    val_log(context, LOG_DEBUG,
            "val_query called with dname=%s, class=%s, type=%s",
            domain_name, p_class(class_h), p_type(type));

    if (ns_name_pton(domain_name, name_n, sizeof(name_n)) == -1) {
        if ((ctx == NULL) && context)
            val_free_context(context);
        return (VAL_BAD_ARGUMENT);
    }

    /*
     * Query the validator
     */
    if (VAL_NO_ERROR ==
        (retval =
         val_resolve_and_check(context, name_n, class_h, type, flags,
                               &results))) {
        /*
         * Construct the answer response in resp
         */
        retval =
            compose_answer(name_n, type, class_h, results, resp, flags);
    }

    val_log_authentication_chain(context, LOG_DEBUG, name_n, class_h, type,
                                 context->q_list, results);

    val_free_result_chain(results);

    if ((ctx == NULL) && context)
        val_free_context(context);

    return retval;
}
```

2.7 Case Study: OpenSSH

In this section, we present some case studies for adding local validation to an application.

The application we selected is the 'ssh' program from OpenSSH. ***Note Well: the code changes below are meant as quick examples of adding DNSSEC validation to an existing example. They are not the complete set of changes that would be needed for the whole set of applications OpenSSH offers, and unconditionally replace code (i.e. not configurable options).***

2.7.1 Simple Client Validation On Connection

OpenSSH uses the **getaddrinfo()** function to resolve host names when a client connects to a ssh server. It is fairly trivial to change this to use **val_getaddrinfo()** instead. Code changes are highlighted in bold below.

```
/* ... */

#include "validator.h"

/* ... */

int
ssh_connect(const char *host, struct sockaddr_storage * hostaddr,
    u_short port, int family, int connection_attempts,
    int needpriv, const char *proxy_command)
{
        int gaierr;
        int on = 1;
        int sock = -1, attempt;
        char ntop[NI_MAXHOST], strport[NI_MAXSERV];
        struct addrinfo hints, *ai, *aitop;
        struct val_addrinfo *val_ainfo = NULL;

        /* ... */

        memset(&hints, 0, sizeof(hints));
        hints.ai_family = family;
        hints.ai_socktype = SOCK_STREAM;
        snprintf(strport, sizeof strport, "%u", port);
        /*if ((gaierr = getaddrinfo(host, strport, &hints, &aitop)) != 0)*/
        if ((gaierr = val_getaddrinfo(NULL, host, strport, &hints, &val_ainfo)) != 0)
                fatal("%s: %.100s: %s", __progname, host,
                    gai_strerror(gaierr));
        if (!val_istrusted(aitop->ai_val_status))
                error("*** WARNING: DNS resolution is not trusted (%s)",
                      p_val_status(aitop->ai_val_status));
        else
                debug("ValStatus: %s", p_val_status(aitop->ai_val_status));

        /* ... */

        /*freeaddrinfo(aitop);*/
        val_freeaddrinfo(aitop);

        /* ... */
```

2.7.2 Validation of SSHFP Record Retrieval

OpenSSH has support for validating a server's SSH key fingerprint against a SSHFP resource record stored in the DNS. Obviously, this is a piece of data that should be validated if possible. While OpenSSH does have a configuration option to use DNSSEC, it does so by trusting the validation status returned from a remote DNS server. In this example, we modify the code to use ***libval*** to do the validation locally. Code changes are highlighted in bold below.

```
/* ... */

#include "validator.h"

/* ... */

int
getrrsetbyname(const char *hostname, unsigned int rdclass,
    unsigned int rdtype, unsigned int flags,
    struct rrsetinfo **res)
{
        struct __res_state *_resp = _THREAD_PRIVATE(_res, _res, &_res);
        int result;
        struct rrsetinfo *rrset = NULL;
        struct dns_response *response = NULL;
        val_status_t val_status;
        struct dns_rr *rr;
        struct rdatainfo *rdata;
        int length;
        unsigned int index_ans, index_sig;
        u_char answer[ANSWER_BUFFER_SIZE];

/* ... */

        /* make query */
        /*length = res_query(hostname, (signed int) rdclass, (signed int) rdtype,
            answer, sizeof(answer));*/
        length = val_res_query(NULL, hostname, rdclass, rdtype,
                               answer, sizeof(answer), &val_status);
        if (!val_istrusted(val_status))
                error("*** WARNING: DNS resolution is not trusted (%s)",
                      p_val_status(aitop->ai_val_status));
        else
                debug("ValStatus: %s", p_val_status(aitop->ai_val_status));
        if (length < 0) {
                switch(h_errno) {
                case HOST_NOT_FOUND:
                        result = ERRSET_NONAME;
                        goto fail;
                case NO_DATA:
                        result = ERRSET_NODATA;
                        goto fail;
                default:
                        result = ERRSET_FAIL;
                        goto fail;
                }
        }

/* ... */
```