DNSCheck Lib 0.91

Beskrivning av testflöde och stödmoduler

Calle Dybedahl <calle@init.se>

1. Zone

The zone module first does some logging and bookkeeping. It then runs the Delegation module, which returns a number of errors and a flag indicating if testing should proceed or not. If the indication is that it shouldn't continue, it skips to its end, does some closing bookkeeping and then exits. Otherwise, it proceeds.

After the delegation check, it fetches a list of all child nameservers. If that list is empty, it also skips to the end. That, however, should not be possible, since the Delegation module has already performed a similar test and indicated termination if it failed.

If there are child servers, the Nameserver module is run on each of them in turn. After that, the Consistency, SOA, Connectivity andDNSSEC modules are run on the zone name being tested.

Finally (and also if things terminate early) the total number of errors is returned.

2. Delegation

The Delegation module starts out by trying to fetch delegation history from the database, if such history was not provided when it was called and a database connection is configured. If there is a history, it is stored for later use.

The usual bookkeeping that starts every test comes next.

After that, there is a sanity check for undelegated tests. If the test has been marked as such but no usable fake glue has been provided, an error message is logged and the method returns as untestable.

A list nameservers for the tested zone is fetched form the parent zone. If the list is empty, the entire test is marked as untestable.

A list of nameservers for the tested zone is fetched from the child nameservers. If the list is empty, the entire test is marked as untestable.

It is checked that all entries in the list fetched from the parent side exists in the list fetched from the child side. Any extra entries are logged as problems.

It is checked that there are at least two nameservers that are included in both the list from the parent side and the list from the child side.

Notices are logged for all nameservers reported from the child side but not from the parent side.

If the test is marked as untestable, it proceeds to the final bookkeeping from here.

If it is testable, a list of nameservers if fetched by doing a normal recursive NS query for the zone. The IP addresses are looked up for all the names thus found, and it is checked that there are at least two IPv4 addresses and either zero or more than one IPv6 address.

Next, a list of glue records for the zone is fetched. This is done as follows:

- A list if nameservers is fetched from the parent side.

- The parent-side nameservers are queried for A and AAAA records for all nameservers thus found.
- A and AAAA records matching the names of the nameservers are put into a list, with all A records in alphabetical order by name first and then all AAAA records in alphabetical order by name.

For each glue record thus found, the following is done:

- The name is checked to see if the record is in the tested zone. If not, the test skips to the next record.
- The child servers are queried for the information in the glue record (same name, same class, same type).
- The response packet from the query is analyzed as follows:
  - If the packet is marked NOERROR and there is a record in the Answer section where all the parts match the query, this piece of glue is reported OK.
  - If the packet is marked NOERROR and there are Answer records but none of them match, this piece of glue is reported as inconsistent.
  - If the packet is reported as NOERROR, the Answer section is empty and the Authority section contains NS records, this glue record is reported as skipped.
  - If the packet is reported as NOERROR, the Answer section is empty and the Authority section contains a SOA record, this glue record is reported as missing at the child side.
  - If the packet is reported as REFUSED or SERVFAIL, this glue record is reported as skipped.
  - If none of the above apply, this glue record is reported as missing at the child side.
- After that two comments note that tests should be added for glue chain loops and glue chains longer than three hops.

If there is delegation history information, it is checked next. Any servers in the list that is also current servers (at the parent side) are filtered out of the history list, and the remaining ones are queried for SOA records for the zone being tested. If an authoritative reply is recieved, the server in question is logged as still being authoritative for the zone.

And finally some bookkeeping is done, and the total number of errors and the testable flag are returned to the caller.

3. Nameserver

The Nameserver test starts with the usual bookkeeping. Then the Host module is called to check that the name of the nameserver being tested is OK. If it isn't, this test jumps to the end. If it is, all IP addresses associated with the name are looked up and tested as follows:

- If it's an IPv4 address and IPv4 is disabled in the DNSCheck configuration, it is skipped.
- If it's an IPv6 address and IPv6 is disabled in the DNSCheck configuration, it is skipped.
- It's checked if the server is recursive, by calling the `address_is_recursive` method in the `DNSCheck::Lookup::DNS` module.

- It's checked if the server is authoritative for the zone by calling the `address_is_authoritative` method in the `DNSCheck::Lookup::DNS` module. If it cannot be determined to be authoritative, the rest of the tests for this address are skipped.
- It is checked if the server answers queries over UDP, by asking for a SOA record for the zone being tested over that transport. Any response at all is considered OK.
- It is checked if the server answers queries over TCP, again by asking for a SOA record and seeing if there is a response at all.
- If the server did answer queries over TCP, it's checked if it allows zone transfers of the tested zone. This is done by calling the `check_axfr` method in the `DNSCheck::Lookup::DNS` module.
- If the server answers questions over at least one of TCP and UDP, it is queried for ID information. This is done by sending queries with the class `CH` and type `TXT` to it, asking for the names `hostname.bind`, `version.bind`, `id.server` and `version.server`. Any of those that get replies get logged as legacy IDs.

And finally there is the closing bookkeeping.

4. Consistency

There is initial bookkeeping, of course.

After that, all nameserver names are fetched from both the parent and child side. All IP addresses for all names are fetched (and thrown away again, if their respective IP version is disabled) and remembered for later use.

Each address in turn is queried for a SOA record for the zone being tested. If one is given, its serial number is stored for later. An SHA1 digest is calculated on the MNAME, RNAME, refresh time, retry time, expire time and minimum time joined together by colon characters. This digest is also stored for later.

Finally, the stored serial values and digests are checked to see how many different ones there are. If there are more than one of either, this is logged.

After that, there is closing bookkeeping.

5. SOA

After the initial bookkeeping, there is a test that a SOA record for the zone exists at the child side. This is done by sending a SOA query to the child servers and verifying that the first record in the Answer section in the response is of type SOA. A message is also logged if more than one SOA record is returned.

If no SOA record is returned, further tests are skipped.

If there is a SOA record, the first thing in it to be tested is the MNAME value:

- The MNAME must be a proper hostname, as determined by the Hostname module.
- If the MNAME is included in the list of servers returned by a recursive NS query for the zone being tested, it is listed as public. If not, it is listed as stealth. If the MNAME is a CNAME, that is also logged.
- All addresses for the MNAME are looked up. They are tested to check that they answer authoritatively for the zone being tested (by calling

`DNSCheck::Lookup::DNS::address_is_authoritative`), unless their IP version is disabled in the configuration. If the authoritativeness test does not return any errors, the MNAME is listed as authoritative. If the test does return a problem, the MNAME is listed as not authoritative. Note that this means that if the MNAME resolves to more than one IP address and those addresses point at some servers that are authoritative and some that are not, the MNAME will be reported as being both authoritative and not authoritative.

After that, it's the RNAME's turn. It's first checked to see that there is at least one unescaped dot in it. If there isn't, it's logged as having a syntax error and no further tests are made on it. Otherwise, the first unescaped dot is replaced by an at sign (@), any escaped dots are unescaped and unless SMTP is disabled in the DNSCheck configuration the resulting string is handed over to the Mail module for testing. If that module returns an error, the RNAME is logged as undeliverable. If not, it is logged as deliverable.

Then the various values in the SOA record are checked. TTL, REFRESH, EXPIRE and MINIMUM are compared to values from the DNSCheck ocnfiguration and problems logged in they are found to be smaller. It's also checked that RETRY is smaller than REFRESH, that EXPIRE is at least as many times as large as REFRESH as another configuration value, that MINIMUM is larger than yet another configuration value and finally that SERIAL is not zero.

That done, there is only the closing bookkeeping left.

6. Connectivity

First bookkeeping, as usual.

All nameserver IP addresses are fetched from the Lookup::DNS module. For each address in turn (unless its version is disabled in the config) it's looked up which AS it is announced from. Problems are logged for any number of ASs other than one. The various ASs are also remembered.

After all the IP addresses has been checked individually, a final check is done to see that the zone is announced via IPv4 from at least two separate ASs and via IPv6 from at least two separate ASs.

Then finishing bookkeeping, and return.

7. DNSSEC

After the initial bookkeeping, the parent-side servers are queried for DS records for the zone being tested (bypassing the DNS module's internal cache). If the query returns NXDOMAIN and the test is undelegated, reporting on the result of this query will be skipped. Otherwise, it's logged whether DS records were found or not.

Next, it tries to find DNSKEY records at the child side. For all nameserver IP addresses (obeying configuration disabling of v4 or v6), a DNSKEY query with the DNSSEC flag set to true is sent. A count is kept of responses with accompanying RRSIG records and those without. If there are any with RRSIG responses, the last one received is used for later processing. If none are received, the reply from the last server queried will be used for later processing.

After all the child servers are queried, the number of responses with and without RRSIG is analyzed. If both kinds have been received, a problem is logged about the child servers having inconsistent DNSSEC behavior. Otherwise, they are reported to be consistent. Finally, an entry is logged specifying if a DNSKEY was found or not.

It's then checked to see if the parent and child sides seem to agree on the zone's DNSSEC status. If there is a DS record at the parent side and a DNSKEY at the child side, the security is reported as consistent. If there is a DS but not a DNSKEY, it's reported as inconsistent. If there is no DS, nothing is reported here.

If there was no DNSKEY at the child side, the rest of the tests are skipped.

If there was a DNSKEY at the child but no DS at the parent, this is reported as a problem (unless the test is undelegated).

Closer checking of the DNSKEY response saved from earlier is then done as follows.

First, all separate DNSKEY records in the response are checked:

- A problem is logged if it has algorithm RSA/MD5.
- A counter is incremented if it has algorithm RSA/SHA1.
- If its protocol field is not 3, a SKIP_PROTOCOL message is logged.
- If the ninth bit in its flag field is not set a SKIP_TYPE message is logged.
- Its key tag is remembered.
- If it is a secure entry point (SEP), that is logged and remembered.

If the RSA/SHA1 counter has a value bigger than zero, it's logged that the mandatory algorithm is there. If not, it's logged that the mandatory algorithm is missing.

It is then checked if the DNSKEYs being processed has accompanying RRSIGs. If there aren't any, it's checked if RRSIGs can be explicitly fetched from a server at the child side. If that can be done, it is reported that DNSSEC Extra Processing is broken. If it can't be done, it is reported that RRSIGs are missing. In either case, further checking of the child data is skipped.

If here were RRSIGs included with the DNSKEYs, the RRSIGs are looped over to be checked:

- The inception and expiration times for the signature are extracted, compared to the current time and reported as not yet valid, inside the validity period or expired.
- The signature is then cryptographically validated against the records it's supposed to sign, and the result reported.
- If both of the above tests pass, the signature is reported as valid.

If at least one of the DNSKEY signatures is valid and if that key's keytag is the same as the keytag of exactly one of the DNSKEY records, it will be reported that the DNSKEY RRSet is validly signed. If not, it will be reported that it isn't.

After that, a SOA query is put to the child servers. The response is analyzed in the same way as the DNSKEY RRSet was, including checking if the RRSIG keytags match keytags in the DNSKEY set. At the end of the analysis, it is logged if the SOA RRSet is considered to be validly signed or not.

If there was a DS record up at the start of the DNSSEC tests, the parent level is then tested.

First, all records that resulted from the original DS query to the parent side are looped over and some checks made:

- If the record has algorithm RSA/SHA1 a counter is incremented.
- If the record has algorithm RSA/MD5, that is logged (but does not contribute to the total number of errors).

- The record's keytag must match the keytag of exactly one of the DNSKEY records earlier fetched from the child side.
- If the child side had one or more SEP records, it is checked if the record refers (via its keytag) to that record or not. The result is logged.

After the loop, it is checked that the counter increased for RSA/SHA1 records is larger than zero, and the result logged as appropriate.

And after all that, there is only the closing bookkeeping.

8. Host

This module tests a host specified by name. As usual, it starts with some bookkeeping. After that, the host name syntax is checked as follows:

- The complete name must be 255 characters or less.
- The name is split apart on period characters (.) to form a list of labels.
- Each label is checked to see that it starts and ends with alphanumeric characters, that the characters in between are alphanumeric or dashes (-) and that the total length of the label is less than 64 characters. As a consequence of the character rules, zero-length labels are also denied.
- It is checked that the rightmost label in the name does not consist entirely of numeric characters.

If any of those tests fail, the failure is logged and the entire test is immediately terminated with a failure result.

After the syntax check, recursive queries for A and AAAA records for the name are done. If no records are returned, it is logged that the name does not exist and the test ends.

If there are records returned, they are checked to see if any are of type CNAME. If so, it is logged as a problem and the test ends.

Finally, the addresses in all A and AAAA records found are fed to the Address test module.

The module closes with the usual bookkeeping.

9. Address

The Address module starts with some bookkeeping, as usual.

It then uses the `Net::IP` module to check that the string that was passed to the test was a syntactically valid IPv4 or IPv6 address. If not, that is logged and the test ends.

If the address is of version 4, it is then checked that the address is not in any of the private ranges as defined by RFC 1918. If so, that is logged and the test ends.

After that, any version 4 addresses are checked not to be in the reserved ranges defined in RFC 3330. If they are, that is logged and the test ends.

Then any version 6 addresses are checked not to be in any of the reserved ranges defined by the IETF RFC draft extending RFC 3330 to IPv6. If they are, that is logged and the test ends.

Following that, a recursive PTR query is done for the reverse of the address (as given by `Net::IP::reverse_ip`). If that query does not return a response with at least one record in the Answer section, the address is logged as having no reverse lookup and the test ends. All the names given in the PTR records are then looped over and recursive A and

AAAA queries made. Any names that gets neither A nor AAAA responses are logged as not resolvable.

And then there is only the closing bookkeeping.

10. Mail

Following the opening bookkeeping, the address given to the test is broken up into a local part and a remote part by simply splitting it at the first at-sign character (@). This may fail for addresses with such characters in a comment in the local part of the address, but is probably not a concern in practice since such addresses should be exceedingly uncommon in SOA records.

After that, `Mail::RFC822::Address` checks if the whole address is valid according to the RFC. If not, that is logged and the test ends.

If the address is valid, the DNS module is asked to find an MX record for the remote part. If any were found, that is logged. It's checked if all servers found are in the zone being tested, and if so a warning is logged. After that, it's checked if the list of mail servers is empty, and if so that is logged and the test terminated.

Next, the mailserver names found are looped over:

- The name is fed to the Hostname module. If that module indicates a problem, that is logged and the loop skips to the next name.
- Recursive queries for A and AAAA records are made. If either query fails, the loop skips to the next name.
- If the A query gets no records in the Answer section but the AAAA query does, it is logged that the server is only reachable via IPv6.
- If IPv4 is enabled in the DNSCheck configuration and the A query got a result, those results are looped over. If the flag indicating that a successful SMTP test for v4 has been done is not set, the address from the query result is handed over to the SMTP module for testing. If that test indicates success, the flag is set. If not, the error count is incremented by the value returned by the SMTP module.
- If IPv6 is enabled in the DNSCheck configuration and the AAAA query got a result, those results are looped over. If the flag indicating that a successful SMTP test for v6 has been done is not set, the address from the query result is handed over to the SMTP module for testing. If that test indicates success, the flag is set. If not, the error count is incremented by the value returned by the SMTP module.

When the loop has finished, it is logged if the v4 and v6 delivery flags are set or not. Then there is the usual closing bookkeeping.

11. SMTP

Notice: As of this writing, the DNSCheck::Test::SMTP module will at compile time internally replace the IO::Socket::INET module with the IO::Socket::INET6 module in order to enable testing SMTP deliveries over IPv6. If (or, hopefully, when) the Net::SMTP standard module is upgraded to handle IPv6 natively, the code doing the replacement should be taken out ~~and shot~~.

At first the test checks if SMTP has been disabled in the DNSCheck config. If so, that is logged and the test simply returns. If not, the usual initial bookkeeping is run.

The test starts by trying to establish an SMTP connection to the host given. If that fails, a problem is logged and the rest of the tests are skipped.

The banner text from the remote server is stored.

The status value of the Net::SMTP object is checked, and if it is not equal to 2, a message is logged that the `HELO` command failed and the rest of the tests are skipped.

Next, the remote server is asked to accept a message from the sender "<>" (the special bounce address). Status is again checked, and if found to not be 2 a problem is logged and further tests skipped.

It's then asked to deliver the message to the mail address provided to the test. Again, the result is checked. In this case, both the result codes 2 and 4 are accepted, to allow for servers using greylisting.

Finally, the connection is reset before a message is actually delivered, and then final bookkeeping is done.

12. Lookup::DNS

Unlike the test modules, `DNSCheck::Lookup::DNS` does not have any sort of global execution flow. It is simply a number of methods meant to provide information for the test modules to look at. Because of that, it will be described one method at a time. Not all methods will be described, only the interesting ones.

12.1. new

Creates a new lookup object. A global one is created at the start of a full test run, and several more specialized ones during the course of the testing.

12.2. query_resolver

This method first looks if the information requested (defined by the name, class and type triple) is already available in the internal cache. If so, it is returned. If not, a global recursive query is done by calling out to the `DNSCheck::Lookup::Resolver` module, the result put in the cache and then returned. If the result was a timeout, that is put into a log entry.

12.3. query_parent

The actual method `query_parent` only looks in the cache, and if the requested information isn't there calls out to `query_parent_nocache`.

The `query_parent_nocache` method takes five arguments: the zone, the name being asked for, the class, the type and a hashref with flags. It starts out by checking if the zone being asked about is a faked zone. If it is and the query type is NS, a fatal error results, since that is not supposed to ever happen. If the query type is A or AAAA, the global resolver is asked to fake up a response packet which is then returned to the caller. For any other type, the return will be empty.

If the zone is not faked, the method proceeds to call `find_parent` in order to find the zone from which the zone being queried about is delegated. If a parent zone can't be found, the method returns an empty value. If it did find one, `init_nameservers` is called for the found zone.

the, `get_nameservers_ipv4` and `get_nameservers_ipv6` are called. The resulting list of IP addresses is shuffled to randomize its order, and a query is finally sent by calling the `_query_multiple` method. Whatever us returned from that is returned as-is from this method.

12.4. query_child

As for the previous method, this is just a cache-checking wrapper for `query_child_nocache`, which also takes five arguments (zone, name, class, type and flags). It proceeds in largely the same way as `query_parent_nocache`, except that there is no lookup for a parent zone or handling of faked data. `init_nameservers` is called on the zone specified, adresses are asked for with `get_nameserver_ipv4` and `get_nameserver_ipv6`, and finally a query is sent with `_query_multiple`, if any servers were found (if not, that is logged). The `aaonly` flag is also unconditionally set before the query is made.

12.5. query_explicit

This method takes five arguments: name, class, type, address and flags. The first three is the triple to be queried for, the address specifies a nameserver to send the query to and the flags specify how the query is to be made.

The method starts out by calling `_querible` to see if the address provided makes sense. If not, the method returns empty.

If the address is ok, a new resolver object is created by calling `_setup_resolver` with the flags that were provided as an argument. The returned object is stored for later use.

The internal address blacklist is checked, and if the combination of name, type, class and address is found to to be blocked the method returns empty.

If the address is querible and not blacklisted, the previously made resolver object is used to call `DNSCheck::Lookup::Resolver::get`. Whatever it returns is stored for later use.

The returned value is checked to see if it is a timed out DNS response. If so, the name, class, type and address is added to the blacklist and the fact of the timeout and blacklist addition logged. An empty value is also returned.

After that, it's checked if the returned value is empty. If it is, the `errorstring` value from the resolver object is logged, and an empty value returned.

If the returned packet's rcode is `FORMERR` and one or both of the flags `bufsize` and `dnssec` were set, it is logged that the queried server does not support EDNS0 and an empty value is returned.

If the response is `FORMERR` but neither of the flag mentioned just above were set, a general lookup failure is logged together with the resolver object's `errorstring`. And, of course, an empty value is returned.

Next, it's checked if the rcode was `SERVFAIL`, the type was SOA and the flag `noservfail` was not set. If so, that is logged, the query and address is blacklisted and the packet is returned.

After that, it is checked if the rcode was anything other than `NOERROR`. If it was, it is logged that no answer was received, and an empty value is returned.

Then it is checked if the `aaonly` flag was set while the response had the aa bit unset. If so, that is logged and an empty value returned.

And finally the entire response is put into the log and the response packet returned.

12.6. _query_multiple

This method takes five or more arguments. Name, class, type, flags and one or more IP addresses. It fetches a fresh resolver object from `_setup_resolver`, and then starts looping over the provided IP addresses. Any address for which `_querible` does not return true is skipped, and for the rest a query is fired off with the resolver's `get`.

If the response does not have the aa bit set but the flags given include `aaonly`, the current response is discarded and the next server is tried instead.

If the rcode is anything other than `SERVFAIL`, the loop is exited.

If the rcode was `SERVFAIL`, the response is checked for timeout, and if found to have timed out a counter is incremented.

If the final response fetched by the loop had rcode `SERVFAIL`, something is logged. If the timeout counter is non-zero, it is a timeout. If it is zero, a general lookup error and the `errorstring` from the resolver object is logged.

And finally the last response processed by the loop is returned to the caller.

12.7. _setup_resolver

This method takes only one argument, a hashref with flags.

It starts by creating a `DNSCheck::Lookup::Resolver` object. In this object it then sets the `cdflag` ("Checking Disabled") flag to true, `usevc` (use TCP transport) to false and `defnames` (append system-configured suffix to names) to false. These defaults are then modified according the provided flag hashref. The state of the `usevc` flag is logged, and then the resolver object is returned.

12.8. get_nameservers_ipv4

Given a name and a query class, calls `init_nameservers` on the name and then returns any IPv4 addresses cached for that name and class.

12.9. get_nameservers_ipv6

Given a name and a query class, calls `init_nameservers` on the name and then returns any IPv6 addresses cached for the name and class.

12.10.get_nameservers_at_parent

Takes two arguments, a name and a class.

First, it checks if the name given is that of a faked zone. If so, it returns the fake data to the caller. If not, it makes a call to `query_parent` for NS records. If no response is returned, it returns an empty value to the caller.

If a response is returned, a list is created of the name server name parts of all NS records in the Authority section if there is one and the Answer section otherwise. The resulting list of names is then sorted in default order and returned to the caller.

12.11.get_nameservers_at_child

Takes a name and a class as arguments. Uses `query_child` to ask for NS records, and then returns either an empty value (if the query did not return anything) or a default-sorted list of the nameserver names given in any NS records in the Answer section of the response.

12.12.init_nameservers

This method takes two arguments, name and class. It looks in the local cache for any NS data for that name and class, does nothing if it finds it an otherwise calls `_init_nameservers_helper`.

The latter method takes the same arguments. It starts out by clearing the local cache of any data already filed under that name and class, and then issues a global recursive query for NS records for the given name. If there is no response at all, it terminates and returns an empty value. Otherwise it gets the names of all nameservers listed in the Answer section of the response and puts them in the local cache.

If the list of nameservers for the given name and class is empty after all records in the Answer section of the response has been processed, the method simply returns and empty value.

If there are NS names saved, they are looped over. For each name, a global recursive A query is made and the IP addresses in any A records in the Answer section of the response where the address field is not empty [1] is remembered. The process is repeated for AAAA records. Finally, the list of IPv4 addresses found is unconditionally put into the local cache, and the list of IPv6 addresses found put into the local cache only if it is not empty.

The method returns the value returned by the `DNSCheck::Logger::auto` method logging that nameserver data has been initialized for the given name.

12.13.prep_fake_glue

Takes a zone name as an argument. Asks the parent object (`DNSCheck`) for lists of nameserver names and IP addresses that are supposed to be faked, and inserts them into the local cache.

This method is no longer called form anywhere and should probably be removed.

12.14.find_parent

This method takes a name and a class as arguments. Like many other methods, it's just a caching wrapper for another method that does the real work. In this case, `_find_parent_helper`.

The real work begins by calling `_find_soa` to get a starting point.

`_find_soa` issues a global recursive SOA query for the given name. If there is no response at all (that is, an error of some kind) it returns the name itself. If there is a response, it goes through the Answer section. If there is a CNAME record in that section, the method returns the given name. If not, it proceeds to go through the Authority section. If that contains a SOA record, the domain name of that record is returned. If there is not, the name being asked for is again returned.

There used to be a bit in `_find_soa` that would return an empty value if the query it makes gets an NXDOMAIN response, and a corresponding check in `_find_parent_helper` that would terminate the search and return an empty value to the caller. This has been removed, since a bug in BIND versions 9.0.x to 9.3.x made it give an NXDOMAIN response for empty nodes with children. Which in practice meant that many names were erroneously considered not to have a parent domain.

---

[1] Is it actually possible for an A or AAAA record to have an empty address field?

Anyway, whatever was returned from `_find_soa` gets split into separate labels. The leftmost label is thrown away. The remaining labels are joined together and `_find_soa` is called on the result. If the value returned from there is the same as the value that was passed in, it is returned as the value of the entire method. If not, another label is chopped off and the attempt is done again with the new, shorter name. If nothing else, this looping will terminate when it reaches the root zone (a name with no labels).

12.15.find_mx

This method takes a name as an argument. It issues a global recursive MX query for the name, collects all MX records from the Answer section of the response and returns those to the caller sorted in ascending order of preference. If no MX records can be found, it issues a global recursive A query for the name, and if at least one such record is found it returns the given name to the caller. If not, it does the same again but with AAAA records. And if none such are found either, it returns an empty value to the caller.

12.16.find_addresses

Takes a name and a class as arguments. It issues global recursive queries for the name, class and respectively A and AAAA types. If either query gets an error, the method returns an empty value. Unless at least one of the queries returns something in its Answer section, an empty value is also returned.

The Answer sections that have something in them are gone through, and the addresses of all A and AAAA records are collected. The collection is then returned to the caller.

12.17.address_is_authoritative

This method takes an address, a name and a class as arguments. It uses `query_explicit` to send a query to the given address, asking for a SOA record for the given name and class. If the query returns an error, the method returns 0. If there is a response, the method returns 0 if is has the aa flag set and 1 if not.

12.18.address_is_recursive

This takes an address and a class as arguments. The address is passed to `_querible`, and if that returns false the method returns an empty value.

If the address is considered OK, a new resolver object is created, and its Recursion and Checking Disabled flags are set to true.

That done, a random name is generated. In order to do that, `Crypt::OpenSSL::Random` is used to generate 64 bytes of random data, on which a SHA1 digest is then calculated, and which is then converted to readable text via BubbleBabble. The resulting string is concatenated with ‘nxdomain.example.com’.

A SOA query for the generated name is sent to the address given at the start. If the query times out or encounters an error, an empty value is returned.

A zero value is returned if the response packet’s ra flag is false, if the response packet’s rcode is REFUSED or SERVFAIL, or if the rcode is NOERROR, the Answer section is empty and the Authority section is not.

In any case not covered above, the value 1 is returned.

12.19.check_axfr

This method takes address, name and class arguments. The address is passed to `_querible`, and if that returns a false value the method returns the value zero.

If the address is OK, a new resolver object is created. The objects recursion, dnssec, usevc and defnames flags are all set to false. The nameserver to query is set to the given address, and an AXFR query is then sent for the given name. If the object can return at least one record from the transfer, the value 1 is returned from the method. Otherwise, the value 0 is returned.

12.20._querible

This method takes an address as its only argument.

If the address is of an IP version that is disabled in the DNSCheck configuration, the value zero (false) is returned.

If the adress class is not one of PUBLIC and GLOBAL-UNICAST, it is logged that the address is not querible and the value zero is returned.

If neither of those is the case, the value 1 is returned.

12.21.preflight_check

This method takes a name as its only argument. A global recursive NS query is made for that name. If there as an error or if there was a response packet and it contained an NS record in the Answer section, the value 1 is returned.

Otherwise, a global recursive SOA query is made for the name. If there as an error or if there was a response packet and it contained a SOA record in the Answer section, the value 1 is returned.

If not, it checks if the last packet returned had rcode SERVFAIL, in which case the value 1 is again returned.

If none of the above turns out to be the case, an empty value is returned.

13. Lookup::Resolver

This module implements a recursive DNS resolver. The reason we do this ourselves rather than simply use an existing library (like version 0.81 and earlier of DNSCheck does) is twofold: we want to be able to lie to ourselves during the recursion process in order to do testing of undelegated domains, and we do not want our results affected by various caches.

The Resolver has a number of external entrypoints. Some of them are mere passthroughs to the underlying `Net::DNS::Resolver` object that is used to send queries to specific nameservers. Some of them have to do with undelegated testing, one is a class function to get root zone data in a particular format and two methods are for actual lookup use.

13.1. get_preload_data

This is a class method, and as such should be called as `DNSCheck::Lookup::Resolver->get_preload_data()`. It is the only method in the module that pays attention to the DNS settings of the surrounding system, and it only uses those to do an NS lookup for the root zone and then address (A and AAAA) lookups for the various nameservers returned. It will store the information in the same format as the Resolver module's internal cache, and return a reference to that structure. In the normal course of things, the `DNSCheck` module will call this method and stick the resulting structure in its `DNSCheck::Config` object, from which `DNSCheck::Lookup::Resolver::new` will later retrieve it. This fairly convoluted way of doing it is a speed optimization. The lookup process here can take a fairly long time, and since several resolver objects will be created in the course of a normal zone test, caching the startup data externally can save many seconds of runtime.

13.2. get

This is a very straightforward method. It takes four or more arguments: a name, a type, a class and one or more IP addresses. It will then send a query with the specified name, type and class to the specified servers and return the first result that comes back. The information in the result will also be inserted in the object cache.

One slight quirk here is that if no IP addresses are provided the system defaults (as specified in `/etc/resolv.conf` will be used). This can occur not only if no address argument is given, but also if none of the given addresses are valid IP numbers.

13.3. add_fake_glue

This method takes three arguments: a zone name, an NS name, and an NS IP address. If the IP address is a valid address (defined as that a `Net::IP` object can be created from it), the information given is added to the object cache and the list of faked data.

13.4. faked_zones

Takes no arguments. Returns a list of all zone names for which `add_fake_glue` has been run.

13.5. faked_zone

Takes a zone name as an argument. Returns the NS names for the zone if there are any, and an empty value otherwise.

13.6. fake_packet

Takes three arguments, a zone name, an NS name and a record type. If the type is either A or AAAA, a `Net::DNS::Packet` object is created and filled with the information previously given to `add_fake_glue` for the given zone and NS names.

13.7. recurse

This is the big scary method that does actual recursive lookups. It takes three useful arguments: a name, a type and a class. If the type and class are not provided, they default to NS and IN respectively. There is also a fourth argument, but that is only used internally to break CNAME loops.

The first thing the method does is to check if this is information that should be faked. If so, it calls `fake_packet` to do so, returns the result and that is that.

If not, the recursing starts. The first thing that is done is to look up (in the object cache) the closest ancestral zone to the one asked for that there is already nameserver information for. This is done by looking in the cache and repeatedly lopping off labels until something is found. Initially, this will be the root zone. The nameservers thus found are put on a stack. One server is popped off the stack, and the query sent to it via `get`. The result is then analyzed as follows:

- ★ If there was an error, it skips to the next server.
- ★ If the response packet has the AA flag set, but the rcode is neither NOERROR nor NXDOMAIN, the packet is saved as a potential answer and it skips to the next server.
- ★ If the packet has the AA flag set and there is an Answer record of type CNAME, the CNAME is looked up, the CNAME record is added to the packet returned from the CNAME lookup and the whole shebang is returned to the caller.
- ★ If the AA flag is set but neither of the two previous are true, the packet is simply returned to the caller as is.

- ★ If the rcode is anything other than NOERROR, it is saved as a potential answer and the process proceeds to the next server on the stack.
- ★ If the AA flag is not set but there is a CNAME record in the answer section, that is resolved the same as if the AA flag had been set.
- ★ If the AA flag is not set and the Authority section contains records, we have a referral. It checks that the name being referred to has at least as many labels in common with the name being looked up as the set of servers currently being processed. If not, it is a bad referral and the process proceeds to the next server on the stack. If the referral is OK, the IP addresses for the NS records are looked up, the current server stack is discarded and the new set of addresses is put on the stack in their place. The process then starts over with getting a new server address off the top of the stack and sending a query to it.
- ★ Eventually, a good answer will be found and returned, or the stack address cache will run out. If the stack runs out and there is a potential answer stored, it will be returned to the caller. Otherwise, an empty value will be returned.

1. Logger

The Logger module is in theory simple: it retains an ordered list of messages produced in the course of a DNSCheck run. In practice it is somewhat more complex, since it also maintains some information for the use of the web GUI and controls the localization of the various messages (with help from the next module, `Locale`).

1.1. auto

The actual of data is done with `DNSCheck::Logger::auto`. This method takes one or more arguments. The first argument is the message tag, a string value that must exist in the `policy.yaml` file and all locale specification files. The tag is stored together with its severity level (from the policy file), a timestamp (with as good resolution as `Time::HiRes` can provide on the system) and any other arguments provided. Internally, a module id and a parent module id are also stored. These are for use by the web GUI.

Note that if DNSCheck is being run with the debug flag set, the information here will be immediately printed out rather than stored.

1.2. export

This method returns a list with all the data stored in the object.

1.3. dump

This method prints all data stored to standard error.

1.4. print

This method takes a locale specification as an argument, and prints stored data formatted according to the data in that locale file. It will not print entires with severity level DEBUG unless DNSCheck is being run with the debug flag set.