ISC Passive DNS Architecture

Robert Edmonds
Internet Systems Consortium, Inc.

June 2012

Contents

1 Introduction

This document describes the architecture of *ISC Passive DNS*, an implementation of Florian Weimer's "Passive DNS Replication" technique [1]. This document will describe at a high level the internal implementation details of the various stages that compose *ISC Passive DNS* as well as the message passing interfaces between those stages.

ISC Passive DNS itself is one component in a larger suite of technologies employed by the ISC Resiliency and Security Forum (RSF). Other components include the ISC Security Information Exchange (SIE), which is used to receive the collected input of ISC Passive DNS sensors distributed throughout the Internet and to send the coalesced output to downstream consumers, such as ISC DNSDB, which is a historical database of DNS records. These three systems are part of a loosely coupled architecture and since this document focuses on ISC Passive DNS, the details of the SIE and DNSDB systems will not be covered.

1.1 Domain Name System

Understanding passive DNS replication requires a robust understanding of the Domain Name System, which is, roughly speakly, a distributed hierarchical key-value store. Rather than paraphrase existing descriptions, here is Paul Vixie's "simplified view" [2] of the DNS:

> The DNS namespace has a tree structure, where every node has a parent except the root node, which is its own parent. Nodes have labels that are from 1 to 63 characters long, except the root node whose label is empty. A domain is a node in context, and a fully qualified domain name has a presentation form that is just the node names, bottom up, with each followed by a period (`.`). For example, `www.google.com` is the fully qualified name of a node whose name is `www`, whose parent is `google`, whose grandparent is `com`, and whose great-grandparent is the DNS root.
>
> Nodes are grouped together into zones, the apex of each being called a start of authority and the bottom edges being called delegation points if other zones exist below them, or leaf nodes if not. Zones are served by authority servers that are either primary (if the zone data comes to them from outside the DNS) or secondary (if their zone data comes to

> them from primary servers via a zone transfer procedure). For example, root, `org`, `acm.org`, and `hq.acm.org`[1] are separate zones of administrative authority.
>
> Every node can have RRs (resource records) that contain the actual content of DNS. Depending on its name, type, and data, an RR can map a host name to an IP address or vice versa, or describe the mail servers for a domain, or serve a growing variety of other purposes. Every RR has a name, class, type, TTL (time to live), and data. TTL is measured in seconds and begins to decrement whenever an RR is transmitted from an authority server. This TTL eventually ticks down to zero inside intermediate caching servers; thus, the authoritative server's stated TTL puts an upper bound on the reuse lifetime of an RR.
>
> DNS clients are most often found inside the runtime libraries of TCP/IP initiators. These runtime libraries are called resolvers and most often will not have caches of their own (thus, they are stub resolvers). Stub resolvers request recursive service from their designated upstream full resolvers. A full resolver is capable of caching data for reuse, and of surfing the zone hierarchy to locate a DNS RR no matter where in the namespace it is located or on which authority servers it may be stored.
>
> – Paul Vixie, "DNS Complexity"

To this it is necessary to add the concept of a Resource Record Set, or RRset, which is a group of Resource Records having identical name, class, and type.

1.2 Passive DNS Replication

Weimer's *Passive DNS Replication* is a technique "to obtain domain name system data from production networks, and store it in a database for later reference [...] once the data has been stored in a local database, more elaborate queries are possible, which leads to further applications." [1] These more elaborate queries are usually inverse queries, as opposed to the more typical "reverse" queries encountered in the DNS.[2] (The original DNS specification supported an `IQUERY` opcode,

[1] Illustrative of the dynamic nature of the DNS, `hq.acm.org` is no longer a separate zone as of the time of the present writing.

[2] An "inverse" query referred to the location of keys given a particular value, while a "reverse" query refers to how IP address keys are formatted in order to look up values in the `in-addr.arpa` and `ip6.arpa` DNS trees.

which, due to the hierarchical, distributed nature of the DNS, never worked very well, and it was formally deprecated in RFC 3425 [7].) Depending on the layout of the database, other types of queries not directly supported by the DNS protocol are possible as well, such as enumerating all known nodes below a certain point in the DNS tree.

Note that a database built from passive DNS information does not have to adhere to the same data model as a full recursive DNS resolver. Since the focus of such a database is diagnostic and introspective in nature, it can for instance accommodate disparate, conflicting records existing simultaneously at the same location in the DNS tree. And since passive DNS systems are not nameservers, records do not necessarily have to be expired based on time-to-live values. A passive DNS database can thus functions as a historical log of a partial subset of the records that have appeared in the DNS.

Paul Mockapetris, the inventor of the domain name system, noted in one of the foundational documents [3] of the DNS that:

> The sheer size of the database and frequency of updates suggest that it must be maintained in a distributed manner, with local caching to improve performance. Approaches that attempt to collect a consistent copy of the entire database will become more and more expensive and difficult, and hence should be avoided. The same principle holds for the structure of the name space, and in particular mechanisms for creating and deleting names; these should also be distributed.
>
> – Paul Mockapetris, RFC 1034

Mockapetris was referring to the *active*, most current version of the database — a historical database that included previous versions of changed records would necessarily be even larger. While we cannot "collect a consistent copy of the entire database", we can however collect a subset of the database by replicating the DNS records received by a sample of the DNS resolvers on the Internet, namely the ones participating in the ISC Passive DNS data collection effort.

We are aided by the historical, exponential increase in computer CPU power and storage capacity. We assert that this decades-long period of exponential growth has perhaps outstripped the growth of the DNS and that relatively modest investments in computer hardware can process extremely large quantities of DNS data.

Some of the issues encountered in collecting and processing a contemporaneously large amount of DNS data was described by Mark Lottor [4]:

> ZONE currently runs on a DECsystem-20 and is written in assembler. The amount of data is quickly reaching the limits of the DEC-20 section address space, and the hardware's ability to survive gets slimmer each day. ZONE assembles all its data in core before dumping it to disk. [...] A new version of ZONE needs to be written to run on a modern computer system. A completely new architecture should be designed to handle the enormous amount of data collected and expected in the future.
>
> – Mark Lottor, RFC 1296

1.3 ISC Passive DNS

The ISC Passive DNS system consists of the following conceptual stages:

(1) The initial collection stage, where the packets between DNS resolvers and authoritative DNS servers are collected together at a central processing point.

(2) The decomposition of individual DNS response messages into a stream of individual RRsets. Each RRset is annotated with the IP address of the server that sent the RRset.

(3) The de-duplication of this stream of RRsets.

(4) A reduction stage where the name of the RRset and the response IP address which originated the RRset are used to infer based on available information the closest enclosing zone or "bailiwick" of the RRset. (This stage locates an RRset within the DNS hierarchy using what we call the "passive DNS bailiwick reconstruction algorithm".) The response IP address metadata is stripped from the RRset and the bailiwick is added.

(5) A second de-duplication of the stream of RRsets, which are now annotated with zone information.

(6) A filtering stage where "undesirable" records are eliminated, based on static blacklists maintained by hand.

Note that while stages 2 and 3 and stages 4 and 5 are conceptually distinct, they are actually performed back-to-back and combined internally in our implementation. Thus, the message passing interface consists of the following simplified set of

stages, which are made available via the ISC SIE *channel* system, and correspond respectively to the SIE channels numbered 202, 207, 208, and 204:

(1) The initial message collection stage.

(2) Decomposition and de-duplication of messages into individual RRsets.

(3) Reduction of RRsets via bailiwick reconstruction and further de-duplication.

(4) Filtering.

2 Processing stages

This section describes each conceptual stage of the ISC Passive DNS system in detail.

2.1 Initial message collection

ISC Passive DNS makes use of a software package called *nmsg*[12], which contains a module called *dnsqr*, which reconstructs UDP DNS query-response transactions based on the capture of network packets. This *passive DNS sensor* is deployed directly on recursive, caching DNS servers or a nearby network tap and collects only the query-response transactions that occur between the recursive DNS resolver and authoritative DNS servers. It does not collect any of the query-response traffic that occurs when the client sets the `RD` or "Recursion Desired" bit to 1, that is, the traffic that occurs between DNS "stub" clients and the caching server itself, since only the traffic generated in response to a cache miss (`RD` bit set to 0) is strictly needed in order to build a passive DNS database.

The passive DNS sensor is stateful and collects both the outgoing query packets from the caching server's DNS resolver as well as the incoming response packets from authoritative DNS servers. The incoming responses are coordinated with the outgoing queries based on a tuple of the following values (the "DNS 9-tuple"):

Element	Description
Query IP	The address of the **initiator**.[a]
Response IP	The address of the **target**.[b]
IP protocol	Usually 17, `UDP`.
Query port	The port number of the initiator.
Response port	The port number of the target, 53.
ID	Randomized 16 bit nonce.
QNAME	The domain name being looked up.
QTYPE	Its type.
QCLASS	Its class.

[a] This is the source in queries and the destination in responses.
[b] This is the destination in queries and the source in responses.

Responses can arrive in multiple packets due to `EDNS0` fragmentation[6]. The sensor automatically captures and reassembles these packets, so that downstream consumers process these large responses transparently.

When the sensor receives a query packet, it temporarily buffers the packet into an in-memory structure, the *state table*. When a response message is received that corresponds to an outstanding query in the state table, the query and response packets along with metadata such as timestamps are encapsulated into a single *dnsqr message* and written to the capture output stream. A configurable upper limit on the total number of entries in the state table (128K) as well as a time limit on the oldest query in the state table (60 seconds) are enforced.

For the purpose of replicating DNS protocol transactions, the state table is actually indexed by the first six values in the 9-tuple (the "DNS 6-tuple"), in order to handle responses which lack a question section (`QDCOUNT` is 0), but only if the response code is `FORMERR`, `SERVFAIL`, `NOTIMP`, or `REFUSED`. If the response code is any other value, such as `NOERROR` or `NXDOMAIN`, or if the response contains a question section, then the full 9-tuple must match for the query and response to be considered part of the same transaction.

Individual *dnsqr* messages are classified into three separate types based on the result of query-response state reconstruction:

- `UDP_QUERY_RESPONSE`. Matching query and response messages.
- `UDP_UNANSWERED_QUERY`. A query message was sent, but expired from the state table without a matching response message being received.
- `UDP_UNSOLICITED_RESPONSE`. A response message was received, but a corresponding query message was not present in the state table.

This laborious reconstruction of DNS query-response state is necessary in order to defeat a trivial blind spoofing vulnerability. If an attacker suspected that a specific recursive DNS server were particating in passive DNS replication, he could send DNS response packets containing arbitrary data to the IP address of the recursive server. Without correlation of query and response packets, the passive DNS system would have no way to tell these unsolicited response packets apart from legitimate responses. This reconstruction process adheres to the resiliency principles described in RFC 5452[9].

The aggregate stream of *dnsqr* messages from multiple passive DNS sensors forms the input into the remaining stages of the passive DNS system.

2.2 Decomposition into RRsets

Once collected, the raw, individual response messages are decomposed into a finer stream of RRsets. These RRsets[5] consist of one or more Resource Records from the `Answer`, `Authority`, and `Additional` sections (the "response sections") of the DNS response message having the same name, type, and class.

The collected *dnsqr* messages are subject to several checks before further processing:

- Only messages of type `UDP_QUERY_RESPONSE` are processed; the other *dnsqr* message types are discarded.
- The time of capture is checked to make sure that the data is not too old. Messages older than 12 hours[3] are discarded, to prevent accidentally uploaded old data or data from a system with a wildly inaccurate system clock from entering the system.[4]
- The UDP checksum of the response message is verified to be either present and correct, or absent. (The UDP checksum is absent from only a very tiny minority of response packets.) The UDP checksum is either verified on the fly at this stage if the data came from an older release of the *dnsqr* sensor software, or is a simple check of a field in the *dnsqr* metadata if the data came from a newer release. (Newer releases of the *dnsqr* software allow the passive DNS sensor operator to zero the IP address of his resolvers in captured packets while still communicating the status of the UDP checksum to downstream components of the passive DNS system.)
- Response messages that fail to be decoded correctly by the DNS message parser are discarded.
- Response messages where the `TC` bit is set are discarded.
- Response messages where the `QDCOUNT` is not 1 are discarded.

If these checks succeed, each Internet-class (`IN`) RRset from each of the three response sections are taken one at a time, canonicalized, and annotated with the timestamp of the response and the IP address of the server which originated the response, and passed to the next stage.

[3] 12 hours is selected as a value which is a bit longer than a typical temporary network partition.
[4] Ideally, all capture systems should have NTP-synced system clocks.

The canonicalization process downcases the RRset's name, sorts the record data values if the RRset consists of more than one RR, and downcases the domain names that appear in record data values according to RFC 4034 [8] section 6.2 and `draft-ietf-dnsext-dnssec-bis-updates` [10] section 5.1.

2.3 Initial de-duplication

De-duplication of the RRset stream is performed by keeping a window of RRsets in memory. This *suppression window* is a `FIFO`-expired store of key-value entries with a hard limit on the amount of total memory consumed by the entries. In this initial or "front" de-duplication stage, these keys are the tuples generated by the RRset decomposition stage.

Each key is a tuple of the following fields:

Element	Description
rrname	RRset owner name.
rrclass	RRset class.
rrtype	RRset type.
rdata	An array of one or more record data values, in sorted order.
response_ip	Response IP address.

The value for each entry in the suppression cache is a structure containing the following fields:

Element	Description
time_first	Earliest timestamp that the key was seen.
time_last	Latest timestamp that the key was seen.
count	Number of times the key was seen between `time_first` and `time_last`.

The de-duplication cache works as follows:

(1) Each incoming tuple is converted into a key. If the key is not present in the de-duplication cache, a new key-value entry will be created, and an `INSERTION` message is written to the output stream. This `INSERTION` message will contain

a tuple of some of the above fields from the key and value: `rrname`, `rrclass`, `rrtype`, `rdata`, `response_ip`, and `time_first`.

The `time_last` and `count` fields are not present in the `INSERTION` messages in the output stream because this is the first time the key has been seen since the beginning of the suppression window. Internally, the `count` field is initialized to the value `1`, and the `time_last` field is initialized to the same value as the `time_first` field.

(2) If the key of the incoming RRset is already present in the suppression cache, the entry's `count` field is incremented by `1`, the `time_first` field is updated if the incoming timestamp is earlier, and the `time_last` field is updated if the incoming timestamp is later. No message is sent to the output stream.

(3) If the total memory size of the suppression cache exceeds a set limit, one or more of the oldest entries are expired until the cache is under the limit again.[5] These expired entries are sent to the output stream and contain the full set of combined key-value fields: `rrname`, `rrclass`, `rrtype`, `rdata`, `response_ip`, `time_first`, `time_last`, and `count`.

That is, `EXPIRATION` messages look similar to `INSERTION` messages except with the addition of the `time_last` and `count` fields.

Note that over the course of the suppression window from the time a unique entry enters the cache to the time it leaves, the entry will only be seen in the output stream twice; the `INSERTION` message and the corresponding `EXPIRATION` message, of course.

While the use of `INSERTION` messages in the output stream doubles the number of output messages for entries that are only seen once in the output stream, and indeed many RRsets are only ever seen once by our passive DNS system, a large number of RRsets are suppressed by the de-duplication cache. Additionally, *no matter the size of the suppression window*, generating an `INSERTION` message when an entry enters the cache results in downstream consumers receiving timely, nearly instant notification of new RRsets. Eventually, at the reception of the corresponding `EXPIRATION` message the downstream consumer is made aware of how many times the RRset occurred during its time in the suppression cache. That is, due to the repetitiveness in the raw stream of DNS input data, downstream consumers of

[5] As of the time of this writing, our deployment uses a limit of 16 gigabytes of memory for the suppression cache, which is sufficient to create a window of about 3-4 hours of data.

the output of this stage trade the knowledge of the exact arrival times of individual occurrences of an RRset for a substantial reduction in the total volume of data.

Additionally, this strategy for data volume reduction is easy to scale horizontally by partitioning the input over multiple, independent de-duplication cache instances and combining the output.

2.4 Bailiwick reconstruction

Bailiwick reconstruction is a passive technique that approximates the location of an RRset within the DNS hierarchy. Modern recursive DNS servers have been hardened to perform rigorous checks on the data returned by authoritative nameservers in order to prevent cache poisoning, and unfortunately the results of these checks are not directly inferrable by an observer of DNS packet data without keeping a significant amount of state. (It would instead be preferable to instrument the recursive DNS server such that packet capture is performed by a routine running internal to the DNS server process, so that the relevant internal state could be replicated into the output stream.)

ISC Passive DNS implements a *passive bailiwick reconstruction algorithm* that serves two purposes: 1) the location of a given DNS record within the DNS hierarchy is an important piece of metadata that is necessary to place that DNS record in context; and 2) it prevents "untrustworthy" records that are a result of intentional or unintentional cache poisoning attempts from being replicated by downstream consumers.

As an example of the first purpose, consider the normal process of delegation in the DNS. A domain name is delegated by a parent zone to a child zone by the presence of `NS` records (and potentially in-zone "glue" address records) — the "delegation point" at the bottom of the parent zone. The child zone receives the delegation by placing its own, authoritative versions of these records at the "zone apex" at the top of the child zone. The authoritative version of these records in the child zone are considered to be more trustworthy — see RFC 2181 [5] §5.4.1 — and hence will replace the version of the records from the parent zone if they differ. (Note that the DNS delegation uses identical values for the Resource Record name and type between the child and parent zones.) When the bailiwick reconstruction process annotates an RRset with the closest known enclosing zone it is made possible to distinguish between the versions of a record from a parent zone versus those from a child zone, and we can disambiguate between various types of misconfiguration, such as inconsistency between child and parent zone nameservers, or inconsistency among sibling nameservers for a particular zone.

As an example of the second purpose — that of reducing, hopefully to zero, the number of "untrustworthy" records in the output RRset stream — nameservers which are authoritative for a large number of zones are sometimes misconfigured so that the nameserver believes it is serving its records from a common ancestor of these zones. This zone contains synthetic `NS` records for the common ancestor zone which will be discarded by modern DNS servers because of the internal state that they keep about the DNS hierarchy, and it is important to be able to distinguish these records (which frequently squat on, e.g., the `com` or root names) from the genuine records served by the real nameservers for those zones.

It is important to note that bailiwick reconstruction has a heavy dependency on the query-response verification described earlier. Without reasonable assurance that a response came from a genuine nameserver and not an attacker with the ability to spoof his source address, the bailiwick reconstruction algorithm can be easily fooled. This isn't a perfect solution, of course, because of the possibility of Kaminsky-style retry-until-success blind response spoofing, but we have endeavored to raise the level of assurance of DNS records obtained from packet capture based passive DNS replication to the level of those obtained directly from a modern, hardened recursive DNS server[9].

2.4.1 Algorithm

The passive DNS bailiwick reconstruction algorithm takes the name of a given Resource Record and the IP address of the nameserver which originated the Resource Record, and answers the question, "Is the IP address a nameserver for a zone that contains or can contain this name, and if so, what is the closest such containing zone known?"

It does this by keeping a cache of `NS`, `A`, and `AAAA` records which have themselves been verified by this algorithm. This cache needs to be initially bootstrapped somehow, analogously to how a real recursive DNS server needs to have a set of "root hints". Root hints could be used to bootstrap the bailiwick cache, but in practice it's quicker to use a full copy of the root zone, as the delegations from the root to top-level domains are long-lived and are infrequently received by recursive DNS servers in production.

Given a Resource Record name and nameserver address, the algorithm works in the following manner:

(1) Walk up the DNS tree by stripping leading labels from the Resource Record name under consideration, until the root label is reached. This generates a

list of potential zones, one of which may be the closest known enclosing zone for the given Resource Record. (For instance, if the Resource Record name is `www.isc.org`, the potential zones are `www.isc.org`, `isc.org`, `org`, and the root.)

(2) Consider each potential zone, starting with the deepest one first. Check if there are any `NS` records in the bailiwick cache for the potential zone. If so, convert the nameserver names thus returned into a list of IP addresses by looking up the needed `A` or `AAAA` records in the bailiwick cache. If the nameserver address which originated the Resource Record under consideration is on this list then the potential zone is the closest known enclosing zone, and the algorithm terminates.

(3) If the list of potential zones is exhausted, then there is no closest known enclosing zone. Either the bailiwick cache does not contain the needed records[6], or the record is "poison".

2.5 Re-de-duplication

The re-de-duplication stage is a second or "back" de-duplication process that is very similar to the initial stage described earlier, and in fact a large amount of code is shared in the implementation of these stages. This stage consumes the stream of RRsets after it has been transformed by the bailiwick reconstruction stage, so there are the following key differences with the initial de-duplication stage:

- Because this is a *re*-de-duplication stage, the stream of RRsets that are being consumed are not a simple stream of RRset observations but rather are a more complex stream of `INSERTION` / `EXPIRATION` pairs, as transformed by the bailiwick reconstruction stage. In operation, the second stage cache has a larger capacity than the first stage cache, even at the same absolute memory limit[7], due to the reason that the response server IP address metadata is stripped from the RRset and replaced by the result from the bailiwick stage,

[6] Note that the bailiwick cache should be able to comfortably accommodate these records. If it is too small, false negatives can be generated due to records that are missing from the bailiwick cache but have not yet expired from the DNS caches being replicated. Our deployment uses 16 gigabytes of memory for the bailiwick cache, and records are expired in `LRU`-order.

[7] As of the time of this writing, our deployment uses a limit of 16 gigabytes of memory for the second suppression cache (the same limit as the first cache), which is sufficient to create a window of about 12 hours of data, about 4 times the size of the window of the first cache.

which is a domain name, combined with the fact that the average DNS zone has more than one nameserver. (As of January 2012, the real DNS data processed by our system is roughly cut in half by absolute volume by the second stage of de-duplication.)

This also means that the sequence of `INSERTION` / `EXPIRATION` messages from the "front" cache need to be coherently combined into a single `INSERTION` / `EXPIRATION` pair generated by the "back" cache. This is done by taking the sum of the `count` field that appears in the front cache's `EXPIRATION` messages, setting the `time_first` field in the back cache to the earliest `time_first` value observed in the front cache's output stream, and similarly setting the `time_last` field to the latest `time_last` value.

- The aggregation of `INSERTION` / `EXPIRATION` messages from the front cache into the back cache makes it possible for an `EXPIRATION` message for an entry in the front cache to be generated that, after transformation, has no corresponding entry in the back cache (most likely, because that entry has expired out of the back cache). Because `EXPIRATION` messages from the front cache will have the `count` and `time_last` fields set, these fields will be set in `INSERTION` messages from the back cache in contrast to the `INSERTION` messages from the front cache, which do not have these fields set.

2.6 Static filtering

This stage performs no actual modification to the contents of the messages in its input stream but rather completely discards messages if they match any entries found on a blacklist. This blacklist has to be manually updated and is maintained with the goal of a substantial reduction in volume by discarding records that are unlikely to be of interest in the assembling of a historical database of DNS records.

The blacklist includes a number of domain names which, after the aggressive de-duplication performed by previous stages, generate a large number of unique records. These domains are typically DNSBLs or other types of tunneled database lookups which are especially prevalent in certain kinds of security software. CDNs and load-balancing services are also capable of generating a large number of unique records. Another voluminous source are PTR records; in the data actually processed our deployment, the overwhelming majority of PTR records appear to be generically named.

3 Message passing interface

This section describes the message passing interface between each concrete stage of the ISC Passive DNS system. These concrete stages make up a loosely coupled data processing pipeline.

An encapsulation protocol called `NMSG`, also developed by ISC, is used as the actual communications channel. The messages passed between the stages of the ISC Passive DNS system are serialized using Google's Protocol Buffers[11] encoding and delivered over a network socket. Multiple message types are supported by `NMSG` by identifying different message types with unique codes. Multiple individual message payloads are automatically coalesced into a single `NMSG` *container* to reduce the total number of network packets sent.

These `NMSG` *containers* are carried over *channels* provided by the ISC Security Information Exchange (SIE). These channels provide a way of grouping and constraining different message streams and referring to the network identifiers (VLAN, IP, port) that make up the stream with a convenient short-hand.

3.1 Raw response messages

SIE output channel:	202
Output message type:	*ISC/dnsqr*

The *ISC/dnsqr* message format is the native output format of the *dnsqr* capture tool described earlier. As mentioned in that section, the individual message units of this format usually contain the multiple packets that make up the query and response of a given DNS transaction. While these messages encapsulate the raw IP packets verbatim, the message API provides hooks for directly extracting the DNS response message, transparently performing IP datagram reassembly if necessary.

3.2 "Front" de-duplication

SIE input channel:	202
SIE output channel:	207
Input message type:	*ISC/dnsqr*
Output message type:	*SIE/dnsdedupe*

This is the combination of message decomposition and initial de-duplication.

3.3 "Back" de-duplication

SIE input channel:	207
SIE output channel:	208
Input message type:	*SIE/dnsdedupe*
Output message type:	*SIE/dnsdedupe*

This is the combination of bailiwick reconstruction and re-de-duplication.

3.4 Filtering

SIE input channel:	208
SIE output channel:	204
Input message type:	*SIE/dnsdedupe*
Output message type:	*SIE/dnsdedupe*

This is the static blacklist-based filtering.

3.5 Discards

SIE channel:	206
Message type:	*ISC/dnsqr*, *SIE/dnsdedupe*

This channel provides a sink where data discarded from further processing by stages described in 2.2, 2.4, and 2.6 are sent. These can be messages of type *ISC/dnsqr* (if the message was rejected from stage 2.2; for instance, a UDP checksum failure or rejection as malformed by the DNS message parser) or of type *SIE/dnsdedupe* (if the message was rejected from stage 2.4 or 2.6; for instance, rejection by the bailiwick algorithm or the static filter). The rejection reason is annotated onto the message payload.

4 References

References

[1] F. Weimer. 2005. *Passive DNS Replication*, FIRST.

[2] P. Vixie. 2007. *DNS Complexity*, ACM Queue 5, no. 3: 24–29.

[3] P. Mockapetris. 1987. *Domain names - concepts and facilities*, IETF RFC 1034.

[4] M. Lottor. 1992. *Internet Growth (1981-1991)*, IETF RFC 1296.

[5] R. Elz, R. Bush. 1997. *Clarifications to the DNS Specification*, IETF RFC 2181.

[6] P. Vixie. 1999. *Extension Mechanisms for DNS (EDNS0)*, IETF RFC 2671.

[7] D. Lawrence. 2002. *Obsoleting IQUERY*, IETF RFC 3425.

[8] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. *Resource Records for the DNS Security Extensions,* IETF RFC 4034.

[9] A. Hubert and R. van Mook. 2009. *Measures for Making DNS More Resilient against Forged Answers*, IETF RFC 5452.

[10] S. Weiler and D. Blacka. 2012. *Clarifications and Implementation Notes for DNSSECbis*, IETF RFC `draft-ietf-dnsext-dnssec-bis-updates-16`.

[11] Google. *Protocol Buffers*. `http://code.google.com/p/protobuf/`.

[12] Internet Systems Consortium, Inc. *nmsg*. `ftp://ftp.isc.org/isc/nmsg/`.