Originally published in the proceedings of the 5th Usenix Security Symposium

DNS and BIND Security Issues

Paul Vixie
<paul@vix.com>
Internet Software Consortium

2 May, 1995

Abstract

Efforts are underway to add security to the DNS protocol. We have observed that if BIND would just do what the DNS specifications say it should do, stop crashing, and start checking its inputs, then most of the existing security holes in DNS *as practiced* would go away. To be sure, attackers would still have a pretty easy time co-opting DNS in their break-in attempts. Our aim has been to get BIND to the point where its only vulnerabilities are due to the DNS protocol, and not to the implementation. This paper describes our progress to date.

1. Introduction

Many were the reasons for starting work on BIND again a few years back. The BIND server and resolver are critical to the daily activities of millions of Internet users, yet they have each been infested with bugs from their first day of use. We have made some good progress on plugging the memory leaks and core dumps that BIND is famous for, and along the way we have found a lot of ways to make BIND more secure.

Many of the classic security breaches in the history of computers and computer networking have had to do not with fundamental algorythm or protocol flaws, but with implementation errors. Sometimes those errors take the form of ignorant or "security unaware" programming, such as collecting potentially unbounded streams of data from the network using functions which do not know the length of their destination buffers, or the use of predictable magic cookies since the programmer's goal is to prevent accidental data errors rather than intentional ones. Other times, a code branch rarely or never taken in normal use is found to have "security fatal" bugs or even deliberate back doors or loopholes.

While we do not intend to demean the efforts of those involved in upgrading the Internet protocols to make security a more realistic goal, we have observed that if BIND would just do what the DNS specifications say it should do, stop crashing, and start checking its inputs, then most of the existing security holes in DNS *as practiced* would go away. To be sure, attackers would still have a pretty easy time co-opting DNS in their break-in attempts. Our aim has been to get BIND to the point where its only vulnerabilities are due to the DNS protocol, and not to the implementation.

2. Why Is DNS Security Important?

Let's say that a security conscious user always uses a DES challenge/response device when connecting to hosts outside the local network, but when connecting locally, she figures that it is safe to send her password in clear text since she knows[1] that outsiders cannot sniff on her private network. Further assume that hers is one of the many installations which does not restrict outbound TCP connections, on the assumption that firewalls are only necessary to keep people *out*[2]. If her name server is able to receive UDP packets on port 53 from outside her local network, then this security conscious user is in for a potentially rough ride.

Before we begin, we'd like to emphasize that the examples are not drawn from theoretical studies, but rather the **tcpdump** command running on real networks.

[1] We'll assume that she is correct.

[2] An assumption with which we do not agree.

Folks over on the Dark Side have tools to exploit these weaknesses, and they are real, right here, right now. We learned of these weaknesses by studying some successful attacks, not just by a careful examination of the protocol and the BIND source code.

2.1. Misdirected Destination

A user asks her telnet client to connect to **host1**. Her client asks the name server for the address of **host1**, receives a corrupt answer, and then initiates a TCP connection to the telnet server at that address. This address does not correspond to her intended host, but it displays the usual greeting, and she types her usual login and password. The connection drops, she tries it again, all is well, she chalks it up to a gremlin in the network and forgets all about it. But there *is* a gremlin in her network, and that gremlin just harvested her password.

2.2. Misdirected Source

If that same user depends on name based authentication when inside what she considers to be the safe confines of her internal network, she's in for another hellride. Anyone on any interior host can almost trivially bypass name based authentication, causing this user's hosts to believe that "they" are "her" and therefore allowing them to log in with her access rights and priviledges. Any host which is allowed to accept incoming connections from outside the local network could be fooled in this same way, but by an outside host.

3. How Did That Happen?

Clearly, the above activities were not design goals of the DNS protocol or of the BIND implementation of that protocol. Let's look at how they could occur.

3.1. Misdirected Destination

It could be as simple as a forged response sent directly to her resolver. Even after 25 years of experience, the Internet still has no production routers which disallow packets with impossible source addresses. So if you can route packets to someone, you can make those packets look as though they came from a close and trusted host – even if they originated outside that host's network. If an attacker can predict the time that a query will be sent, he need only flood the resolver with bogus replies and hope that his bogons arrive earlier than the real answer. Predicting the UDP port used by the resolver for any given query might require that a novice attacker spend several minutes thinking about it, but many attackers will consider that time well spent.

This would not have worked in our example, since we're assuming a one-way firewall. Her resolver isn't reachable by packets from outside her net – but her name server is. If that name server can be corrupted, even for an instant, then an attacker can redirect telnet sessions (containing passwords), electronic mail (containing proprietary information), or even other DNS queries (thus using one name server to help corrupt others.) Every one of those things has been seen in action – we're not *just* being paranoid.

3.2. Misdirected Source

On late model BSD-derived systems, name based authentication usually takes the form of files containing lists of host names or addresses, possibly including a user name to be matched against the remote ("incoming") user name[1]. A convention is upheld whereby certain TCP port numbers[2] are able to be bound only by processes executing with so-called "super user" priviledges[3]. This rather brittle chain of causality permits the BSD **ruserok()** library call to assume that the remote user name given in the data stream is "authentic" from the point of view of the remote host and its administrators. Users are not allowed to claim, when they use the **rsh** or **rdist** or **rlogin** commands, that they are somebody they're not – at least on well run, trustworthy multiuser hosts.

BSD's security took a giant step forward back in 1989 or so, when the callers of **ruserok()** were encouraged to do more than blindly assume that the result of **gethostbyaddr(getpeername(**remote**))** was accurate. It used to be that whatever DNS gave as the name corresponding to the source address of a connection, was used directly as the search key when scanning **~/.rhosts** and its bretheren. After someone noticed that the name server being asked for this information was the one belonging to the connection's initiator, the convention changed: Now, after calling **gethostbyaddr()**, the result is passed back through **gethostbyname()** to see if the addresses and names all match. The name server for **gethostbyname()** will be, barring corruption, authoritative for any given host name in **~/.rhosts** (et al.) Someone who can make their address appear to map to one of your hosts will have to take some extra steps to also make your host appear to have one of his addresses.

(SunOS put this check into **gethostbyaddr()** – an error that will live in infamy, since not every caller of that function wants to get an "error" return status when the

[1] E.g., **hosts.equiv**, **hosts.lpd**, **~/.rhosts**

[2] Those from 512 to 1023.

[3] This convention is of course meaningless on single-user hosts.

forward and reverse lookups yield asymmetric results. The proper place for this mapping logic is in those applications and library calls who intend to use the data for some kind of authentication – it is not a naming issue per se, and does not belong in the resolver.)

As effective as that extra **gethostbyname()** call has been, its goal was to keep attackers from just editing their IN-ADDR.ARPA zones and zooming on in. No thought was given to whether the name servers could be corrupted. So while an attacker has a little more work to do now than in the Old Days, it is still trivially easy to pollute the caches of the set of servers who will be asked for the **gethostbyaddr()** and **gethostbyname()** answers, or to flood the resolvers with bogus responses at the time that they are predicted to be waiting for the answers.

If an attacker can reach the victim's host, they can probably make their host name seem to be almost any arbitrary string when viewed by the victim's **rlogind**. And, if they can also break "super user" on the source host (or if that host is their own office workstation), they can make the victim see any arbitrary remote user name. If this attacker knows any of the contents of your **~/.rhosts** files or your ~Bhosts.equiv file – and these are eminently guessable – then they are *in*.

4. Protocol View of Weaknesses

One way of looking at these weaknesses is from an operational point of view, which given the current state of the art, tells us: *name based authentication is inherently insecure*. Sessions (whether TELNET, NFS, or whatever) should require something stronger than trying to determine a host's name and and then looking for that name in some statically configured list. ([RFC1510] and [RFC1760] are each cause for optimism.)

From the bottom, though, these weaknesses all come with particular sets of details and can be described in terms of DNS protocol elements. As implementors we are more interested in this view than in the more political questions of Global Internet Authentication. So let's have a look at the packets, shall we? After that we'll take a look at the ways they can be perverted.

We do not intend to present an exhaustive description of DNS – [RFC1034] and [RFC1035] already fill that need. Our goal in this section is to present enough information about DNS that someone unfamiliar with its details can still understand the security ramifications of some of DNS's design choices. If this report disagrees with [RFC1034] or [RFC1035] in any detail, it is most likely that the report is wrong.

4.1. DNS Datagram Formats

DNS queries and responses use a common format, though not all protocol elements are used all the time. The simplest case, described here, uses IP/UDP where each datagram contains one DNS query or response. DNS's use of IP/TCP is beyond the scope of this report other than as it affects zone transfers, which we will discuss shortly.

Header Section: Describes the other sections, has flags including RD (recursion desired) and AA (authoritative answer), and most important for our discussion, has a 16 bit "query ID."

Query Section: Contains the name, class, and type of the resource record set ("RRset") being queried for. DNS permits multiple queries in this section but this has never been tried and is not well specified.

Answer Section: Always empty in queries. Contains the RRset matching the query, or is empty if name doesn't exist, if no data matched the query, or if a nonrecursive query results in a referral.

Authority Section: Always empty in queries. Can be empty in responses. If nonempty, it contains the NS and SOA RRs for the enclosing zone. This is sometimes called "referral data."

Additional Data Section: Always empty in queries. Can be empty in responses. If the answer or authority section contains any RRs whose data fields contain RRnames, the RRsets for those RRnames appear here.

4.2. Servers and Resolvers

The client in DNS is called a "resolver." The server is called, appropriately enough, a "name server." Resolvers have some static configuration information, consisting of a domain "search list" and a list of name server addresses. Theoretically, a resolver can also be configured with a static map of domains to name server addresses, allowing queries to be forwarded directly to appropriate name servers for some set of locally known domains. BIND does not implement this last part yet. The resolver's list of name server addresses had better include at least one recursive name server, or the DNS name space is going to look pretty small.

4.3. Recursion

To "recurse" on a query means that when a query comes in for an RRset not known to the server receiving it, that server will forward it to some name server more likely to know the answer. In some cases, the forwarding server will know the name server list for the exact domain or parent domain of the query. More often, a grandparent domain's servers are known, or no servers are known and the query is sent all the way to the root name servers (which are co-operated by the InterNIC and a worldwide cadre of volunteers.) There is a flag in the query called RD which, if set, specifies that recursion is desired; if clear, a name server will answer queries for unknown RRsets with an appropriate error ("name unknown" or "no data," depending.)

Sending nonrecursive queries is a fine way to find out what a name server already knows, since, otherwise, you will get an answer even if the name server had to go searching for it at the time of your query.

4.4. Referrals

If a name server receives a query for a <*name*,*class*,*type*> tuple that it knows it has delegated, it answers with what's called a "referral." A referral response has an empty answer section but a nonempty authority section; the intent of this message is to tell another server "the name you asked for exists, but I don't have the answer, go try these other servers." Bogus referrals are a fine way to pollute a cache indirectly – if you can snoop on a forwarded query and then inject a referral response, you can make the forwarding server effectively believe that *you* are the delegated server for an entire subtree of the DNS name space. This is actually the easiest way to pollute a cache since there's no guessing involved: You know the source address, source UDP port, and query ID by inspection. You even know the query name. The only trick is in breaking into a host on a network backbone so that you can actually see the queries being forwarded to the root servers. This has been done[1], but not often.

4.5. Authority: Masters and Slaves

To be "authoritative" means that a name server has an entire "zone" loaded, either via a "master file" that was created by the name server administrator, or via a "zone transfer," which is a TCP session with another name server. The former kind of server is called the "master" and the latter is a "slave." Slaves generally do their zone transfers from the master, but sometimes firewalls are interposed and it becomes necessary to have slaves pull their data from other slaves, which are themselves stationed at the border, perhaps even on the firewall itself.

Masters and slaves will set the AA flag on any response whose answer section contains only RRsets from authoritive zones. The AA flag will be clear if any RRset in the answer section came from the the "cache," which is what we call the portion of the DNS name space that is outside all of a server's zones of authority. If a server has no zones of authority, then all of its answers will be nonauthoritative since all it has is a cache. This kind of server is sometimes called a "caching only" or "forwarding" server.

4.6. Forwarding -vs- Recursion

When a name server receives a query for data it doesn't have, it can either send back an error response (if it is authoritative for the name's zone, it knows that either the name or data doesn't exist), send back a referral (if running in "nonrecursive mode" as the root servers all do, or if the RD flag is clear in the query), or it can forward the query. This last possibility is of interest to us in our security study, because of what will happen when some response finally comes back. Forwarding is not a three-party transaction – a forwarded query results in a response to the forwarder who must then complete the original transaction by forwarding the response back to the originator.

BIND takes its forwarding duties one step further, as an optimization attempt: It caches all the RRsets in the forwarded response. This promiscuity is the source of most of BIND's bad reputation in both the operations and the security fields. Other servers are free to put almost anything into the response, even if it has nothing to do with the query. As shown in [Bel95a], this has disasterous effects on security.

It is worth noting that the first query handled by a forwarding or recursive name server for a given RRset is likely to result, ultimately, in it forwarding back an answer obtained from an authoritative name server – thus the AA flag will be set in the response, even though the forwarder is not itself authoritative for the name. Subsequent queries to the same name server for the same RRset will probably be satisfied from the cache, and in that case the AA flag will not be set in the response. You can see this in action using the ISI **dig** tool from the BIND kit.

4.7. Forwarding -vs- Timeouts

When BIND's resolver needs to forward a query, it chooses the next name server address from its statically configured list, sends the query, waits a short time for an answer, chooses the next name server address, sends and waits, and so on. BIND's timeouts are fairly short; It will often send a query to name server #1, then to name server #2, then the response will come in from name server #1, and the resolver will close its socket such that when name server #2's response comes in a second or so later the kernel sends back an ICMP Port Unreachable

[1] No, we're not going to name names.

message. We wish there were a way to ask the kernel not to send these, other than keeping the socket open longer (which would lead to resource starvation among kernel protocol control blocks.) Lengthening the timeout would lead to longer application-visible delays when a statically configured name server goes off the air, but life is full of hard choices.

4.8. Query IDs and UDP Ports

Each query sent out by a resolver will come from some UDP port on some address of the resolver's host, and its header will contain a unique (in the context of the source address and port number) query ID. UDP port numbers and DNS query IDs are both unsigned 16 bit quantities, giving a range from 0 to 65535 for each. Port numbers could be conserved and reused by the resolver, but BIND currently opens a new socket for each query, and kernels tend to use an LRU mechanism when assigning port numbers to new sockets. The tuple <*address*,*port*,*queryID*> forms a unique identifier that servers can use to keep track of queries in progress. Resolvers should verify that the query ID of the response matches that of their query.

4.9. Delegations, Zones, Domains, and Subdomains

Strictly speaking, every DNS name is a domain. All domains except the root are also "subdomains." Any time a subdomain is delegated to some other master name server, a "zone cut" is said to exist. A zone consists of all names from a zone cut downward to either terminal names (sometimes called "leaf domains") or other, deeper zone cuts.

The most common case of a zone begins at a subdomain and has no zone cuts beneath it. The most famous zone is the root ("**.**") which has no terminal names, just delegations.

There are two views of a delegation: The parent zone, which has some NS RRs at the cut, and the child zone, which has a superset of those NS RRs and also an SOA RR. When we say "superset" we mean that a child will have at least the NS RRs known by its parent, and perhaps some additional NS RRs that the parent does not know about.

4.10. Lame Delegations

If a delegation NS RR names a host which is not authoritative for the zone, then that host when queried nonrecursively for names in that zone will answer with a delegation to a higher (that is, closer to the root) authority. This is an error condition as perceived by the server that forwarded a nonrecursive query – if a name server is listed in an NS RR, it is supposed to have the zone. It is reasonable to declare failure at this point, though perhaps a bit severe.

BINDs from version 4.9 have **syslog**'ed the condition and gone on to try the other delegated servers. The **syslog** volume generated by this condition is the cause of more than half the questions we see about BIND from new name server administrators. The only way to fix the condition is to get someone to edit the delegation to remove the nonauthoritative name server, or to get someone to make the name server authoritative. Either way it's not something the detecting server's administrator can do anything about directly; we hope that the continued **syslog** volume will lead to more hate mail being sent to the administrators of broken zones, thus ultimately leading to a decline in the number of broken zones. We have been accused of optimism in this matter.

4.11. Glue

When transmitting a zone via a TCP "zone transfer," the general rule is to send only the RRsets whose names lie within the zone being transferred, which is to say starting from the initial zone cut, and proceeding downward (away from the root) to include all names which are not further delegated. There is an exception to this, called "glue." Any address records (A RRs) which are referred to by an NS RR inside the zone (at the initial cut or any downward cuts) must be included, even if they lie beneath one of the downward zone cuts.

If this information is not included in the zone transfer, then referral responses won't be able to include those addresses in their additional data sections. In the absence of that additional data, the name servers will not be reachable except by servers who have the zone – and that's not very useful. It is important that a server only send (or accept) relevant glue during zone transfers, since otherwise this becomes an easy way for your cache to become polluted.

5. What We Have Fixed

BINDs from version 4.9 have plugged a lot of holes with respect to earlier versions. An incomplete list follows:

5.1. Cache Tagging

BIND now maintains for each cached RR a "credibility" level showing whether the data came from a zone, an authoritative answer, an authority section, or additional data section. When a more credible RRset comes in, the old one is completely wiped out. Older BINDs blindly aggregated data from all sources, paying no attention to the maxim that some sources are better than others.

Each RR also has the address of the name server who sent it to us. This can be seen in cache dump when you're looking at some bad data and wondering how it got to you.

5.2. Additional Data Promiscuity

We accelerate the TTL decline for data which arrived as additional data. We are considering not caching it at all other than as necessary for forwarding the response – see below.

5.3. Irrelevant Answers

We check the response to ensure that all RRsets in each section have names and types that make sense in the context of the query and answer sections. Including spurious additional data won't automatically pollute a cache any more; As of BIND 4.9.3 it is necessary that the answer section contain a CNAME RR to introduce an arbitrary name, after which it's business as usual for cache polluters. This is the best we can do without a protocol change.

5.4. Nonmatching Answers

Believe it or not, older BINDs did not check that the answer name matched the query name. Now, within the limits of CNAMEs and wildcard answers, BIND will insist that a response answers the right question. This error was particularly pernicious with respect to some of the name ↔ address symmetry checking, since the answer's RRname sets the name in the resolver's response structure, which meant that callers of **gethostbyname()** could end up comparing a foreign name to another foreign name.

5.5. Logging

Many of the detectable conditions indicating a probable break-in attempt were in the past either not detected, or treated as protocol errors (which is to say, silently worked around). BIND now fairly shrieks whenever it has even the slightest cause for alarm, which is a mixed blessing since the volume of its complaints is so high that most name server administrators pay no attention.

The **syslog** data is of greatest interest during the post mortem analysis of a break-in attempt. The log of unsolicited responses, for example, can show attempts at cache pollution during the early stages – before the attackers switched to whatever technology actually got them in, or set off your alarms, or whatever. Be aware while examining these logs that some systems (most notably SunOS) cannot cause packets to come from a particular address if they have more than one interface – so if you're on the wrong side of a multihomed SunOS name server, *all* of its responses will appear to be "unsolicited."

5.6. Glue

BINDs from version 4.9 restrict glue to just the A RRs under the delegation point, whereas previous versions included all the A RRs referred to by a zone's NS RRs – even those above the zone. By "restrict" we mean that BIND will be conservative both in what it generates *and what it accepts*. This may fly in the face of the Robustness Principle[1] of [RFC1123], but the old behaviour was just simply *wrong*.

6. What We Cannot Fix

We are counting on the IETF DNSSEC effort to bring us a DNS protocol revision that authoritatively signs responses. With that in place we will all stop worrying about attackers who spoof their source addresses, predict our UDP port numbers and query ID numbers, and so on. Response data will be objectively verifiable, independent of whether it is even a response to some query we have sent. Until DNSSEC is finished and in wide use, there are some things we're just going to have to live with.

6.1. Query ID Prediction

With only 16 bits worth of query ID and 16 bits worth of UDP port number, it's hard not to be predictable. A determined attacker can try all the numbers in a very short time and can use patterns derived from examination of the freely available BIND source code. Even if we had a white noise generator to help randomize our numbers, it's just too easy to try them all.

6.1. CNAME Indirection

As mentioned previously, a CNAME response allows a remote name server to introduce a new name for an RRset of arbitrary type. Forwarders receiving such a response should not cache those RRsets (as BIND currently does), but even with that precaution it will be possible to use a CNAME response to bypass the name/address symmetry checking.

[1] "Be liberal in what you accept, and conservative in what you send."

7. What We Would Like To Fix

Every change to BIND has the potential to push the Internet into the final abyss. We are therefore quite conservative about anything that looks like it could have far reaching consequences, which is to say, just about anything[1].

7.1. Query Restarts

Some of the information needed to properly validate a DNS response is expensive (in terms of bandwidth and delay) to obtain, and for that reason it is inappropriate for every resolver to exhaustively validate every response it receives. Recursive or forwarding name servers, on the other hand, have (or should be able to obtain) all the information the DNS has to offer, and it would be a good thing if the name server validated responses before forwarding them to the client. BIND does not currently do this, since it is not possible to edit responses *in situ* and we are uncomfortable with the idea of BIND autonomously deciding that certain responses should not be forwarded at all.

Our current plan for circumventing this problem is to restart all queries. To "restart" means that upon receiving an answer from a forwarded query, a name server will validate the response and insert "known good" data into its cache, and then pretend that the original query had "just now" been received. All the original RRsets would be looked up again, and if any are still missing (either because no response has yet included them, or because the responses that included them were invalid in some way), new queries would be generated to bring in the missing data. Query restarts are the *only* way to solve certain other problems currently being encountered by BIND[2] – the security benefits will be a happy side effect.

One interesting question we're pondering about query restarts is whether to preserve the AA flag, which as discussed earlier will tend to be set on forwarded responses if those responses come from an authoritative server, but will tend to be clear on responses satisfied from the forwarder's cache. We could maintain the current semantics with the hierarchical cache described below, but it's not clear that the AA flag on forwarded responses really matters that much. DNSv2 will probably have a AD flag – authority desired – to force forwarding in spite of any cache. The proposed AD flag will probably have to bypass the query restart logic described here.

[1]A Usenet article once opined, "BIND is like a train wreck inside."

[2]Out of zone CNAMEs, for example.

7.2. Hierarchical Cache

We would like to segment the cache such that additional data can be cached for the duration of a query's restarts, but not used to satisfy other queries (either as answer data, authority data, or additional data). Ideally, the only things we would ever cache would be the answer and authority sections, and only those from authoritative answers (AA flag set). BIND's current cache design is not ready for this kind of overloading – we've pushed it about as far as it will go just by adding the credibility tags described earlier. What's needed is a multilevel translucent cache such that each lookup can specify a stack of caches to be searched, and each cache can be managed by an appropriate purge policy.

7.3. Empty Nonterminal Names

One of the gaping holes in BIND's new nonpromiscuous policy towards cache data is that the credibility and zone tags are held in the RR, not in the name. It is possible to determine, knowing only a name, whether that name lies within any of a server's zones of authority. BIND doesn't do that right now, it currently checks the RRs looking for any that have a zone tag, and if none are found it assumes that it is in the cache. This is bad news in the case of empty nonterminal names – those names which have no RRs and are only present to keep two dots from smashing into each other.

The **ARPA** domain was once empty other than for its **IN-ADDR.ARPA** subdomain, and eventually someone accidentally fed a root server some NS RRs at that name. That root server told the other root servers, and those root servers told every name server on the Internet, and pretty soon nobody anywhere could do address → name translations. We quickly added some NS RRs at the **ARPA** domain and cold started the universe.

It would be better if BIND did not need data to be present at a name in order to know that that name was inside a local zone of authority. Astute readers will note that it's really quite easy to add new names to someone else's authority zones – just keep in mind during your experiments that these new names won't appear in zone transfers, so you will have to infect each authoritative name server manually.

7.4. Unified Zone Cut View

Right now the answer you'll get for an NS query for a domain will depend on who you ask. If you ask a server of the parent zone, you will get the delegation information from "above" the zone cut. If you ask the a server of the zone itself, you will get the actual authority data (an NS RRset and an SOA.) We believe it would be better in most cases to have the server for the parent zone use its delegation data only as hints, and that it should go out and ask the servers named therein for their view of the real delegation data. This would prevent most of the current instances of lame delegation, since the lameness would be detected by the server for the parent zone where it can most likely be fixed by the local name server administrator. The lame data can be elided from delegation responses, thus preventing other servers from following it and having each other server **syslog** the lameness information to their local, helpless, name server administrator. Naturally we would extend the logic so that the zone servers validate their own delegation information and likewise elide lame information from their responses.

This unification would put a stop to the unpleasant question, "how can both the parent and child zones answer authoritatively if they are allowed to answer differently?" We may implement a stopgap whereby parents stop setting the AA flag on referral responses – since the child is really the authority. Unfortunately, last time we changed the way we handed out referrals, some major clients could not handle it and we had to back out to older, broken behaviour. Keeping track of client sensitivities has become a first order task for us.

What we're wrestling with on the unification theory is whether the root servers should try to verify their delegation data. With millions of zones delegated, it could take quite a while for each root server to get this done at startup time, so if we do it, it'll have to come after we make the cache persistent.

8. DNSSEC – The IETF DNS Security WG

As we've mentioned several times in this paper, there is presently work underway to add security to DNS. The current model is something like a "web of trust," using public key technology. A new KEY RR holds the public key and is added to the delegation data. This key is sufficient to validate signed answers but not to actually sign them. Signing is done by the authoritative servers, and the SIG RR is used to carry the signature of any given RRset.

Once DNSSEC is widely implemented, it will be possible to determine from examination of a DNS response whether its contents are authentic. This sounds simple but it has deep reaching consequences in both the protocol and the implementation – which is why it's taken more than a year to choose a security model and design a solution. We expect it to be another year before DNSSEC is in wide use on the leading edge, and at least a year after that before its use is commonplace on the Internet.

9. Which BIND Version Plugs Which Hole?

Always assume that you need the latest BIND you can lay your hands on. Our RCS libraries have the whole sordid story, and from them we could derive a table of Versions -vs- Vulnerabilities. You can bet that the upper class of attackers can do this as well. Deriving that table would be a lot of work and publishing it might do more harm (giving folks the false idea that they don't need to upgrade their BIND) than good (letting folks see how bad things really are.) When we took over BIND, the latest version was UCB 4.8.3. Our first release was DECWRL 4.9, which contained quite a few security related changes. Our current release as of this writing is ISC 4.9.3[1], and it also contains quite a few security related changes.

References

[Bel95a] Steven M. Bellovin. Using the Domain Name System for Syetem Break-ins. In *Proceedings of the Fifth Usenix UNIX Security Syposium, Salt Lake City, UT.* AT&T Bell Laboratories, 1995.

[RFC1034] Paul V. Mockapetris (ISI). RFC 1034 – Domain Concepts and Facilities, IETF, 1987.

[RFC1035] Paul V. Mockapetris (ISI). RFC 1035 – Domain Implementation and Specification, IETF, 1987.

[RFC1123] R. Braden, Editor. RFC 1123 – Requirements for Internet Hosts – Application and Support, IETF, 1989.

[RFC1510] John T. Kohl, et al. RFC 1510 – The Kerberos Network Authentication Service (V5), IETF, 1993.

[RFC1760] N. Haller. RFC 1760 – The S/KEY One-Time Password System, IETF, 1995.

[1] see **http://www.isc.org/isc/**.