The following paper was originally published in the
Proceedings of the Sixth USENIX UNIX Security Symposium
San Jose, California, July 1996.

A DNS Filter and Switch for Packet-filtering Gateways

Bill Cheswick, Lucent Technologies
Steven M. Bellovin, AT&T Research

A DNS Filter and Switch for Packet-filtering Gateways

Bill Cheswick
Bell Laboratories
ches@bell-labs.com

Steven M. Bellovin
AT&T Research
smb@research.att.com

Abstract

IP-transparent firewalls require access to the external Domain Name System (DNS) from protected internal hosts. Misconfigurations and misuse of this system can create internal administrative and security problems.

Dnsproxy provides access to and protection from untrusted DNS services. It runs on a firewall, or on a trusted host just inside the firewall. The program receives (or intercepts) DNS queries and forwards them to an appropriate internal or external "realm" for processing. The responses can be checked, filtered, and modified before they are returned to the requester. The logging and consistency checks can provide information about possible DNS attacks and irregularities that are not available from most DNS implementations.

Introduction

For many years we have run application-level gateways on our interface to the Internet.[2][3, pp. 85–118] Behind these firewalls has grown an intimate community of perhaps 300,000 hosts, plus a (mostly) separate domain name system (DNS) arrangement. We've run our own DNS root and shielded ourselves from most of the security and administrative problems associated with the external DNS.

Recently we have installed a dynamic, or smart, packet filter. This newer technology allows us to have most of the security promised by an application-level approach, with much better performance.

A dynamic packet filter acts very much like traditional router, except that it keeps track of each TCP circuit, and permits bidirectional packet flow for each TCP circuit until the connection is terminated. Our users no longer need modified client software or proxies to access the net—they can get there themselves.

This gateway is transparent to permitted IP packets flows, which means we need something like standard DNS access to the Internet. This brings two problems:

1. Internally we rely on address-based authentication extensively for services like *rlogin*, *rsh*, *rcp*, and NFS. DNS is a vital part of these services, and it is easily subverted. Our crunchy outside now has a liquid center.

2. It's going to take a while to wean our companies away from our internal DNS root servers and our old gateways. During the transition period, internal exposure to external DNS information must be controlled and limited. We can't have external root NS entries knocking around inside, confusing innocent resolvers about where the answers lie.

The first problem is generic, and Internet-wide. Current, and especially older versions of *bind*, are much too trusting of the answers they receive.

The second problem is of our own making, but we expect that many sites are facing similar problems as their gateway technologies evolve.

What are we Afraid of?

There are many problems with the DNS protocol and its implementation. We don't intend to enumerate all the problems here—see references[1][5][6][7] for a detailed list.

Most of the known attacks are based on the fact that many popular Internet protocols rely on name-based authentication. If a client connects to a host using *rlogin*, the server does a reverse lookup of the client's IP address, and consults a table of trusted clients. If an attacker can subvert the DNS, this mechanism is broken. An attacker can force the target to make a normal query, and return additional irrelevant glue records to that query. These glue records are cached, and consulted on the subsequent query, giving the wrong answer.

We have seen DNS packet injectors and related tools in hackers' toolkits captured by law-enforcement folks.

It was clear to us that we could not allow an external name server to give us any information about an internal host or network. The DNS reply had to be unpacked, examined, censored and filtered, repacked, and forwarded to the recipient. This is an application-level gateway for external DNS services.

We also wanted to filter out all external NS records, to keep from polluting our internal DNS tree. In fact, we filter any record that is not explicitly necessary and common. There are a lot of miscellaneous and apparently harmless DNS records. We won't pass them until there is a need. As a performance optimization, we do pass back inside NS records; this allows inside resolvers to contact insider servers directly on subsequent queries.

We considered checking the address returned in the A record. Could an attacker harm us if he said his host was on one of our internal nets. We couldn't think of an attack, but one turned up in February involving Java (see CERT Advisory CA-96.05, March, 1996, and [4]). It was easy to install the check, because we had the right tool in place.

Dnsproxy

Dnsproxy is a DNS switch and filter. To clients it looks like a name server. We run it on two internal hosts; the resolvers inside point to these hosts. The next section discusses other deployment options. The program looks up the address in a configuration file and forwards the request to an appropriate "realm". Two realms are typically used: "inside" and "outside", though more may be configured. (We could set up separate `att.com` and `lucent.com` realms if we wished.) A realm is typically served by at least two name servers. *Dnsproxy* will forward the request to the least-busy server.

The realm's name server sends its reply to *dnsproxy*, which examines the response. The following checks are made:

- Is the record malformed in some way? If so, drop it.
- Does the query in the response match the query we sent out? If not, drop the whole thing.
- Did the response come from the expected IP address? If so, drop the response. This can obviously be spoofed, but the check is cheap. On the other hand, it can cause problems when talking to some multi-homed DNS servers, as the answers may appear to come from a different address for that machine.
- In each resource record, do all domain names refer to the relevant realm? If a query for `inside.com` would normally be directed to realm `inside`, an answer containing it will not be tolerated from another realm. This rule has some administrative implications, discussed below.
- Is there a filter rule for the responding realm to drop or modify a particular resource record? If so, apply it.

Deployment Choices

There are several different ways in which a DNS proxy can be deployed. First, it can simply be a server within a comparatively small lab. Each machine has its `resolv.conf` file pointing to the server; it in turn queries the outside or inside as needed. A variant points the `forwarders` option of Bind at `dnsproxy`, to allow for local caching of responses. We use both of these variations now. A second method would be to have the proxy act as the root server within a large organization. As per normal practice, all unresolved queries would be redirected to the root; it would take appropriate action. The third way is to install the proxy as a filter in some sort of dynamic packet filter. The packet filter could intercept outgoing DNS queries and kick them up to *dnsproxy* for processing.

Each of these three schemes has its advantages and disadvantages. The third scheme is likely the best. Insiders see the same image of the world as do outsiders. NS records need not be deleted, save for those that refer to the inside domain. And it doesn't matter where a host learned of a DNS server's address; the address will just work.

The problem is that this deployment mechanism is very dependent on the details of your firewall. Not all firewalls permit this sort of dynamic processing—simple packet filters do not—and those that do differ widely in their design.

The second scheme works well for large organizations; it has the disadvantage that it produces an inconsistent view of the world. There are then two sets of root servers, the outside's and the inside's. A laptop that lives in both worlds would need two different configuration files.

It would be least intrusive to use local `resolv.conf` files or `forwarders` entries in `named.boot` files is the least intrusive mechanism. Only the local machine is affected; it does not require organization-wide deployment. Furthermore, it works even with ordinary packet filters. The problem is that this approach doesn't scale; it may not be feasible to change all of the machines in a large organization. As such, it

```
realm
        inside  135.104.2.10,135.104.26.141      error
        outside 192.20.225.4,192.20.225.9        default

switch
        outside any     www-db.research.att.com
        outside any     www.research.att.com
        outside any     ampl.com
        outside any     dnstest.research.att.com
        inside  any     att.com
        inside  any     ncr.com
        inside  any     lucent.com
        inside  any     attgis.com
        inside  any     135.in-addr.arpa
        inside  ptr     127.in-addr.arpa
        inside  any     11.192.in-addr.arpa
        inside  any     19.192.in-addr.arpa
        inside  any     94.128.in-addr.arpa
        inside  any     127.192.in-addr.arpa
        inside  any     222.131.in-addr.arpa
        inside  any     243.132.in-addr.arpa
        inside  any     206.141.in-addr.arpa
        inside  any     25.149.in-addr.arpa
        inside  any     52.153.in-addr.arpa
...
        inside  any     87.153.in-addr.arpa
        outside any     *

filter  outside block   * NS *
        outside block   * A 135.104/16
        outside block   * A 135.180/16
        outside block   * A 127/8
```

Figure 1: A sample configuration file.

is best for experimental use or within relatively small groups. We have found that the benefits of the dynamic packet filter have given our users strong incentive to point their resolvers at the *dnsproxy* service.

We are currently using this latter choice, precisely to avoid impact on the rest of the company. At this stage, the code is experimental, and we are not prepared to deploy it widely. Ultimately, we will likely modify it so that it can be integrated with our dynamic packet filter.

It's not surprising that we have found it vital to duplicate the *dnsproxy* service. Since *all* name server queries come through it, its host provides an annoying single point of failure. Each realm has at least two servers, as well.

Performance

We've made no attempt to optimize the code in *dnsproxy* at this point: we are pleased enough that it works well. At present, the program is a CPU hog. We are processing 40,000 requests an hour on an NCR 3430 server with two 60MHz Pentiums. It uses about 10 CPU seconds per minute.

But the users see snappy response times. Typical round-trip times are about 10ms.

Scheduling

Dnsproxy supports multiple servers per realm. How should we use these? It would be nice to share the load, and if one server goes down, to rely on the others. How do we tell if a server is down? We could ping the server, but that would require additional code, and possibly a new hole in the firewall. Why not use our DNS traffic flow?

The problem is that a server may fail to respond because it hasn't obtained an answer, or because it is down. With a hefty load such as ours, we get plenty of indications that a server is working, but it is harder to tell if it isn't.

At first we tried a moving average of server response times. They all started with a high average, and quick responses brought the average down. If we gave up on a query, we included the timeout in our average. This worked well enough, and wasn't hard to implement. But we found a simpler way.

Resolvers are accustomed to timing out, and reissuing requests: typical retries arrive within three seconds. We can use this flexibility to waste an occasional request in the name of scheduling. To do this, we keep track of the number of outstanding requests to each server. We send the request to the server with the shortest queue. Even under heavy load, we usually get our answer before a new query comes in. Typically, one server handles all of the load until a single query gets stalled. Then it switches to the other. If one server goes down, the second starts handling the entire load within a couple of requests.

Other approaches

Paul Vixie[7] is working to add these and similar features to Bind. We hope he succeeds. For example, he is modifying the forwarding code to implement checks like ours, and others.

On the other hand, Bind is already a very large program; adding more security-critical functionality to it may not be a good idea. We're likely to continue running *dnsproxy* no matter how the base code changes.

Administration details

Dnsproxy normally runs as a detached daemon, and listens to queries on UDP port 53. (To access this port, it currently runs as *root*, which it shouldn't.) It forwards its queries from a predictable UDP port based on the realm, so it is easy to install appropriate filtering rules in a gateway. For example, in the following realms:

```
inside  135.104.70.9  error
outside 192.20.225.4  default
```

queries to 135.104.70.9 port 53 would be forwarded from port 54 in *dnsproxy*, and queries to 192.20.225.4 would come from UDP port 55. If a debugging version of *dnsproxy* is listening to port 9953, these ports would be 9954 and 9955 respectively.

Figure 1 shows a sample configuration file. The `realm` section describes the realm name and servers. It also can process erroneous requests or be the default (keywords `error` and `default`) if a query does not match any entry in the `switch` section.

The `switch` section is an ordered list of query types and the realms that process them. When a query arrives, *dnsproxy* runs through the list to find the first match, and dispatches the query to the given realm. This crude data structure eats much of our processing time, and should probably be improved.

The `filter` section is for responses, and is specified per realm. In this example we explicitly filter out all NS records, and any A records that refer to our internal addresses. Other filtering occurs as well, as described above. Our filter rules are fairly primitive. One should probably be able to match any specific type of resource record. We haven't needed this generality yet, so the code is currently fairly crude.

Dnsproxy generally logs to *syslog*. The usual *syslog* level controls the detail and severity of logging information. At the debugging level it produces a full dump of every query and response, producing a torrent of output. At a typical *syslog* logging level of *notice*, only records with unusual reason codes are displayed.

The routine logging can show a host of ills and configuration mistakes that might be normally missed. After we sifted through these, we raised the logging level. Error logs should be normally silent, so really unusual events won't get buried in a sea of mundane trivia. It has been a bit difficult to get the logging level right.

Dnsproxy does not handle tcp requests at present. This hasn't been a problem in our environment, but there are name servers that rely on this ability.

Availability

At present this code is not available outside of Lucent and AT&T.

References

[1] Bellovin, S. *Using the Domain Name System of System Break-ins.* Proceedings of the Fifth Usenix Unix Security Symposium, June 1995, pps. 199–208.

[2] Cheswick, W. R. *The Design of a Secure Internet Gateway.* Proceedings of the Usenix Summer '90 Conference.

[3] *Firewalls and Internet Security; Repelling the Wily Hacker.* Cheswick and Bellovin. Addison Wesley, 1994.

[4] Dean, D. and Wallach, D. *Security Flaws in the HotJava Web Browser.* Proceedings of the IEEE Symposium on Security and Privacy, May 1996.

[5] Mockapetris, P. *Domain names–concepts and facilities.* RFC 1034, Nov. 1987. Updated by RFC1101.

[6] Mockapetris, P. *Domain names–concepts and facilities.* RFC 1035, Nov. 1987. Updated by RFC1348.

[7] Vixie, Paul. *DNS and Bind Security Issues.* Proceedings of the Fifth Usenix Unix Security Symposium, June 1995, pps. 209–216.