A Formal-Specification Based Approach for Protecting the Domain Name System

Steven Cheung
Department of Computer Science
University of California
Davis, CA 95616
cheung@cs.ucdavis.edu

Karl N. Levitt
Department of Computer Science
University of California
Davis, CA 95616
levitt@cs.ucdavis.edu

Abstract

Many network applications depend on the security of the domain name system (DNS). Attacks on DNS can cause denial of service and entity authentication to fail. In our approach, we use formal specifications to characterize DNS clients and DNS name servers, and to define a security goal: A name server should only use DNS data that is consistent with data from name servers that manage the corresponding domains (i.e., authoritative name servers). To enforce the security goal, we formally specify a DNS wrapper that examines the incoming and the outgoing DNS messages of a name server to detect messages that could cause violations of the security goal, cooperates with the corresponding authoritative name servers to diagnose those messages, and drops the messages that are identified as threats. Based on the wrapper specification, we implemented a wrapper prototype and evaluated its performance. Our experiments show that the wrapper incurs reasonable overhead and is effective against DNS attacks such as cache poisoning and certain spoofing attacks.

1. Introduction

This paper presents a detection-response approach for protecting the domain name system (DNS). DNS manages a distributed database to support a wide variety of network applications such as electronic mail, WWW, and remote login. For example, network applications rely on DNS to translate between host names and IP addresses. A compromise to DNS may cause denial of service (when a client cannot locate the network address of a server) and entity authentication to fail (when host names are used to specify trust relationships among hosts). For example, if DNS is compromised to cause a client to use incorrect DNS data, the client may be unable to obtain the IP address of a mail server and thus cannot communicate with it. As another example, if the DNS mapping for www.cnn.com is compromised, an attacker may be able to direct a web browser looking for the news web site to one that gives out counterfeit news. If the web browser does not authenticate the server, the user may use the counterfeit news as if they were genuine. Some applications (e.g., Unix *rlogin*) use name-based authentication. Attacking DNS could change the name-to-address mapping, and hence may allow an attacker's machine to masquerade as a trusted machine. Thus protecting DNS is security critical.

Our approach for protecting DNS is driven by formal specifications. The use of formal specifications enables reasoning, thus providing assurance for our solution. Formal methods have not been used in connection with an intrusion detection approach. Using Vienna Development Method (VDM), we developed formal specifications to characterize DNS clients and DNS servers, and to define a security goal as an invariant: A DNS server should only use DNS data that are consistent with those disseminated by the corresponding authoritative sources. We designed a DNS wrapper, also characterized by formal specifications, that enforces the security goal. Our DNS wrapper examines DNS messages entering and departing a protected name server to detect those messages that could lead to violations of our security goal. If the wrapper does not have enough information to determine whether a DNS message represents an attack, it collaborates with the name servers that manage the relevant part of the DNS name space. If the DNS wrapper cannot verify the data of the DNS message to be trustworthy, the wrapper logs the message and prevents it from reaching the protected name

0-7695-0707-7/00 $10.00 © 2000 IEEE

server.

Section 2 reviews the basics of the domain name system. (Readers are referred to [1, 13, 14] for more details about DNS.) Section 3 describes some known DNS vulnerabilities. Section 4 presents our system model. Section 5 presents a DNS wrapper that enforces our security goal for DNS. Based on the wrapper specification, we implemented a wrapper prototype. Section 6 describes our experiments for evaluating the performance of the wrapper implementation and their results. The results show that the DNS wrapper incurs reasonable overheads and is effective against some known DNS attacks. Section 7 concludes, compares our work with related work, and suggests future work. For the sake of brevity, we omit the formal specifications for DNS clients, DNS servers, and the DNS wrapper in this paper. See [7] for details of this work.

2. Overview of DNS

2.1. What is DNS?

DNS manages a distributed database indexed by *names*. The database has a hierarchical structure. A name (e.g., cs.ucdavis.edu.) has a structure that reflects the hierarchical name space, which is depicted in Figure 1.

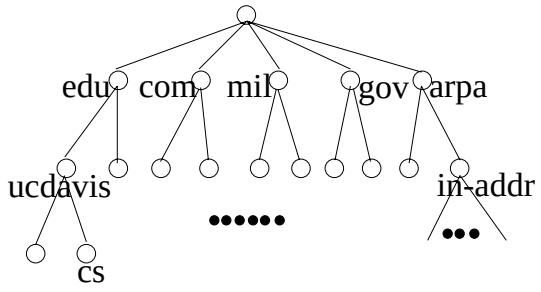

Figure 1. Hierarchical Structure of DNS Name Space

A *zone* is a contiguous part of the domain name space that is managed together by a set of machines, called *name servers*. The name of a zone is the concatenation of the node labels on the path from the topmost node of the zone to the root of the domain name space. The name servers that manage a zone are said to be *authoritative* for this zone. Every subtree of the domain name space is called a *domain*. The name of a domain is the same as the zone name of the topmost node of the corresponding subtree.

One of the main design goals for DNS is to have distributed administration. The distribution is achieved by delegation. For instance, instead of storing all the information about the entire edu domain, which is a very large domain, in a single name server, the responsibility of managing the ucdavis.edu domain is delegated to the authoritative name servers of UC Davis. The authoritative name servers of the edu zone are equipped with the names of the authoritative name servers of the ucdavis.edu zone. Thus if the edu servers need information about the ucdavis.edu domain, they know which servers to contact.

Clients of DNS are called *resolvers*, which are usually implemented as a set of library routines. Whenever an application on a machine needs to use the name service, it invokes the resolver on its local machine, and the resolver interacts with name servers to obtain the information needed. The most common implementations of resolvers are called *stub resolvers* (e.g., BIND[1] resolvers are stub resolvers). Stub resolvers only do the minimal job of assembling queries, sending them to servers, and re-sending them if the queries are not answered. Most of the work is carried out by name servers.

2.2. How does DNS Work?

The process of retrieving data from DNS is called *name resolution* or simply *resolution*. Suppose the host h1.cs.ucdavis.edu needs the IP address of h2.cs.foo.edu. The resolver will query a local name server in the cs.ucdavis.edu domain. There are two modes of resolution in DNS: *iterative* and *recursive*. In the iterative mode, when a name server receives a query for which it does not know the answer, the server will refer the querier to other servers that are more likely to know the answer. Each server is initialized with the addresses of some authoritative servers of the root zone. Moreover, the root servers know the authoritative servers of the second-level domains (e.g., edu domain). Second-level servers know the authoritative servers of third-level domains, and so on. Thus by following the tree structure, the querier can get "closer" to the answer after each referral. Figure 2 shows the iterative resolution scenario. For example, when a root server receives an iterative query for the domain name h2.cs.foo.edu, it refers the querier to the edu servers. Eventually, the querier will locate the authoritative servers of cs.foo.edu and obtain the IP address. In the recursive mode, a server either answers the query or finds out the answer by contacting other servers itself and then returns the answer to

[1] BIND stands for Berkeley Internet Name Domain, which is the most common implementation of DNS.

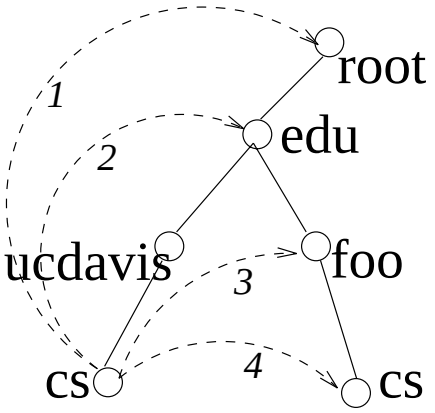

Figure 2. Iterative Name Resolution

the querier.

The above resolution process may be quite expensive in terms of resolution time and the number of messages sent. To speed up the process, servers store the results of the previous queries in their *caches*. Consider the above example. If h1.cs.ucdavis.edu asks its local server to resolve the same name twice, the server can reply immediately based on the information stored in its cache the second time. Also, if in a subsequent query h1.cs.ucdavis.edu asks its local server to find out the IP address of h3.cs.foo.edu, the local server can skip a few steps and contact a cs.foo.edu server directly. If the querier gets an answer from an authoritative server, the answer is called an *authoritative answer*. Otherwise, it is called a *non-authoritative answer*. Because there may be changes to the mapping, servers do not cache data forever. Authoritative servers attach time-to-live[2] (TTL) tags to data. Upon expiration, a name server should remove the data from its cache.

2.3. DNS Message Format

A DNS message consists of a header and four sections: *question*, *answer*, *authority*, and *additional*. A *resource record* (RR) is a unit of information in the last three sections. Here is a list of common resource record types [13]:

- An A record contains a 32-bit IP address for the specified domain name.
- A CNAME record lists the original (or *canonical*) name of the specified domain name. In other words, a CNAME resource record maps an alias to the canonical domain name.
- An HINFO record contains host information such as the operating system used.
- An MX record contains a host name acting as a mail exchange for the specified domain.
- An NS record contains a host name that is an authoritative name server for the specified domain.
- A PTR record contains a domain name corresponding to the specified IP address.
- An SOA record contains information for the entire specified domain such as the domain administrator's mail address.

The header has a query id field, which is used to facilitate requesters' matching up responses to outstanding queries. The question section carries a target domain name (QNAME), a query type (QTYPE), and a query class (QCLASS). For example, a query to find the IP address of the host h2.cs.foo.edu has QNAME=h2.cs.foo.edu, QTYPE=A, and QCLASS=IN (which stands for the Internet). The answer section carries RRs that directly answer the query. The authority section carries RRs that describe other authoritative servers. For instance, the authority section may contain NS RRs to refer the querier to other name servers during iterative resolution. The additional section carries RRs that may be helpful in using the RRs in the other sections. For instance, the additional section of a response may contain A RRs to provide the IP addresses for the NS RRs listed in the authority section.

3. DNS Vulnerabilities

Bellovin [3, 4], Gavron [10], Schuba and Spafford [15], Vixie [17], and CERT advisory CA-98.05 [5] discuss several security problems of DNS. In the following, we describe two well-known problems of DNS that are relevant to this paper—*cache poisoning* and failure to authenticate DNS responses.

In the cache poisoning attack, an attacker can trick a name server S_1 to query another name server S_2. If S_2 is a compromised name server, the attacker can have S_2 to return a DNS response that contains faked RRs.

[2] There is no single "best" TTL value for all resource records. The TTL value of a resource record is based on a tradeoff between consistency and performance. A small TTL will increase the average name resolution time because remote name servers will remove the resource record earlier and need to query the corresponding name servers more often. If a resource record is changed, a small TTL enables other name servers to purge the stale data and to use the new data earlier. One should reduce the TTL before the resource record is changed. A common TTL value is one day (e.g., the cs.ucdavis.edu zone), although some high-level zones (e.g., the root zone) use a multi-day TTL.

Otherwise, the attacker can masquerade as S_2 and send the DNS response to S_1 (see below). Recall that a name server caches the results of previous interactions with other servers to improve performance. When S_1 uses its contaminated cache to resolve a name, it may use the incorrect DNS data supplied by the attacker.

The message authentication mechanism used by most implementations of DNS is weak: A DNS server (or a DNS client) attaches an id to a query and uses it to match with the id of the corresponding response. Suppose a server S_1 sends a query to another server S_2. If an attacker can predict the query id used by S_1, the attacker can send a forged response that has a matching query id to S_1. When S_1 receives the response that claims to be from S_2, S_1 has no way to verify that the response actually comes from S_2. If S_2 is unavailable when the query is sent, the attacker can just masquerade as S_2 and send the forged response to S_1. If S_2 is operational, the attacker can mount a denial of service attack against S_2 to prevent S_2 from responding to S_1's query. Also, if a name server receives multiple responses for its query, it uses the first response. Thus even if S_2 can reply to S_1, the attacker can still succeed if the forged response reaches S_1 before S_2's response does.

4. System Model

In our model, there are two types of processes: DNS servers and DNS clients (or resolvers). These processes communicate with each other through message passing. Resolvers only communicate with servers; servers can communicate with other servers in addition to communicating with resolvers. These two types of processes are denoted by *Server* and *Resolver* respectively. Basically, we model DNS clients and DNS servers as an object that maintains a view on DNS data. The view may be changed only through communicating with other DNS components (i.e., sending DNS requests and receiving DNS responses) or by timeouts for DNS data.

We use the Vienna Development Method (VDM) to specify our system model, because VDM provides a formal language for specifying data and the associated operations, and includes a framework to perform refinements of data and operations. Another reason is that VDM provides a basis for performing formal verification, which makes it more convenient to extend our work in the future. Most of the symbols used in VDM are standard mathematical symbols. We will describe the non-standard or less commonly used ones as we need them. Readers are referred to [11, 2] for more details on VDM.

In the following, Section 4.1 presents our DNS data model. Section 4.2 defines our notion of a process' view on DNS data. Section 4.3 formalizes the DNS concept of authority. Section 4.4 discusses our assumptions about DNS. Section 4.5 presents our security goal for DNS.

4.1. DNS Data

DNS messages (of type *Msg*) are either a query (of type *Query*) or a response (of type *Resp*).

$$\begin{aligned} Query \cup Resp &= Msg \\ Query \cap Resp &= \emptyset \end{aligned}$$

A message m of type *Msg* consists of the following sections: header, question, answer, authority, and additional. We denote these sections of m by $Hdr(m)$, $Q(m)$, $Ans(m)$, $Auth(m)$, and $Add(m)$ respectively. The header section includes a query id, an opcode[3], a truncated message flag[4], and a response code[5]. We denote these fields of m by $id(m)$, $opcode(m)$, $tc(m)$, and $rcode(m)$ respectively. The question section consists of a domain name, a query type, and a query class. The answer, the authority, and the additional sections consists of resource records (RR). We denote the set of resource records of a message m by $RRof(m)$. A RR consists of a domain name, a type, a class, a 32-bit TTL (in seconds), and a resource data field. For a resource record r, we denote these fields by $dname(r)$, $type(r)$, $class(r)$, $ttl(r)$, and $rdata(r)$ respectively.

DNS manages a distributed database. The database is indexed by a tuple (dname, type, class) of type *Idx*. The range of the database is a set of resource records, abbreviated as RR. To denote this database type in VDM, we use a map type $DbMap : Idx \xrightarrow{m} RR\text{-set}$. A *map* type $T = D \xrightarrow{m} R$ has domain D and range R. The domain and the range of T are denoted by $\mathsf{dom}(T)$ and $\mathsf{rng}(T)$ respectively. A map of type T is a set that relates single items in D to single items in R.

$$\begin{aligned} RRType &= \{\mathsf{A}, \mathsf{PTR}, \mathsf{NS}, \mathsf{CNAME}, \mathsf{MX}, \mathsf{SOA}, \\ &= \mathsf{HINFO}, \ldots\} \\ RRClass &= \{\mathsf{IN}, \ldots\} \\ TTL &= \{0 \ldots 2^{32} - 1\} \end{aligned}$$

[3] The opcode of a DNS message distinguishes between different types of queries—standard queries and inverse queries. A standard query looks for the resource data given a domain name. An inverse query looks for the domain name given resource data.

[4] The truncated message flag indicates whether the DNS message is truncated. Message truncation occurs when the message length is greater than that allowed on the transmission medium.

[5] The response code field is used to indicate errors and exceptions.

$$
\begin{array}{lcll}
Idx & :: & \text{dname} & : DName \\
 & & \text{type} & : RRType \\
 & & \text{class} & : RRClass \\
RR & :: & \text{dname} & : DName \\
 & & \text{type} & : RRType \\
 & & \text{class} & : RRClass \\
 & & \text{ttl} & : TTL \\
 & & \text{rdata} & : RData
\end{array}
$$

$$DbMap = Idx \xrightarrow{m} RR\text{-set}$$

Db represents the data managed by DNS. *SubDomain* captures the domain-subdomain relationships. Given a domain d, the set of all the sub-domains of d is represented by $SubDomain(d)$. A *zone* contains the domain names and the associated data of a domain, except those that belong to a delegated domain. A zone is a contiguous part of the domain name space that is managed together by a set of name servers. A zone may have a set of delegated subzones, represented by the function *SubZone*. (In VDM, a *function* specification consists of two parts. The first part defines the argument types and the result type, which are separated by the symbol "→". The second part gives the function definition.) For a zone z, $ZoneData(z)$ contains all resource records whose domain names belong to zone z, the *zone cut data*, and the *glue data*. The zone cut data describe the cuts around the bottom of zone z: In particular the NS resource records of the name servers for the delegated zones of z. If there are name servers for the delegated zones residing below the zone cut, the glue data contain the addresses of these servers.

$$
\begin{array}{l}
Db : DbMap \\
SubDomain : Domain \xrightarrow{m} Domain\text{-set} \\
ZoneData : Zone \xrightarrow{m} DbMap \\
SubZone : Zone \to Zone\text{-set} \\
\forall z \in Zone \cdot SubZone(z) \triangleq \\
\quad \{cz \mid \exists rr \in \mathsf{rng}\, ZoneData(z) \cdot type(rr) = \mathsf{NS} \wedge \\
\quad dname(rr) \neq z \wedge cz = dname(rr)\}
\end{array}
$$

4.2. View

Every process maintains its view of the database. The view of a server s can be partitioned into the authority part (denoted by $View_{auth}(s)$) and the cache part (denoted by $View_{cache}(s)$), where the former takes precedence over the latter. The *map overwrite* operator † takes two map operands and returns a map that contains all the elements in the second operand and those in the first operand whose domain does not appear in the domain of the second operand. For a server that is not authoritative for any part of the database and for a resolver, the corresponding $View_{auth}$ is $\emptyset$.

$$
\begin{array}{l}
View_{auth} : Process \xrightarrow{m} DbMap \\
View_{cache} : Process \xrightarrow{m} DbMap \\
View : Process \to DbMap \\
\forall p \in Process \cdot View(p) \triangleq \\
\quad View_{cache}(p) \dagger View_{auth}(p)
\end{array}
$$

4.3. Authority

Some servers are said to be authoritative for a zone; their views on the zone data define them. *AuthServer* maps a zone to the list of authoritative servers. *AuthAnswer* defines the mapping from an index to the *authoritative answer*, defined by the view of the an authoritative server on the index. *Authoritative* returns true if and only if every resource record in the input resource record set is authoritative.

$$
\begin{array}{l}
AuthServer : Zone \xrightarrow{m} Server\text{-set} \\
AuthAnswer : Idx \to RR\text{-set} \\
\forall i \in dom(Db) \cdot AuthAnswer(i) = \\
\quad \mathsf{let}\ z \in Zone \wedge p \in Process \wedge \\
\quad i \in \mathsf{dom}\, ZoneData(z) \wedge p \in AuthServer(z)\ \mathsf{in} \\
\quad View_{auth}(p)(i) \\
Authoritative : RR\text{-set} \to Boolean \\
\forall rrs \in RR\text{-set} \cdot Authoritative(rrs) = \\
\quad \forall rr \in rrs \cdot rr \in \\
\quad AuthAnswer((dname(rr), type(rr), class(rr)))
\end{array}
$$

4.4. Assumptions

In this section, we explicitly list our assumptions for DNS. They concern with how name servers prioritize RR sets, the accuracy of authoritative DNS data, the effect of changes on DNS data, the accuracy of delegation data, and the power of attackers on eavesdropping DNS packets.

Assumption 1 *Protected servers do not add an RR to the $View_{cache}$ of a process if an RR that corresponds to the same index already exists in the $View_{cache}$. Moreover, protected servers prefer authoritative data over cache data.*

Both of them hold for "good" servers (i.e., servers that behave according to the DNS RFC [13, 14]). Some server implementations rank data from different sources at different credibility levels. Moreover, data from a higher credibility level can preempt data from a lower credibility level. We do not model data credibility levels in our work for the sake of simplicity. Because our DNS wrapper only allows authoritative data to reach a protected name server, this simplification does not affect the validity of our results.

Assumption 2 *Data from an authoritative server are correct.*

For example, if a server is authoritative for a machine h and the server says the IP address of h is i, then we believe that the IP address of h is i.

Assumption 3 *When a server attaches a TTL with t seconds to a resource record for which the server is authoritative, the resource record will be valid for the next t seconds.*

We state this assumption because there is no revocation mechanism in DNS. Without this assumption, one cannot determine the validity of DNS data as soon as they leave their authoritative servers. We argue that this assumption is reasonable. When a resource record needs to be changed, the TTL of this resource record is usually decreased before the changeover so that incorrect/stale records will timeout shortly after the changeover.

Assumption 4 *For every zone, the delegation data and the glue data of its child zones correspond to the* NS *RRs and the* A *RRs of the name servers of the child zones.*

An example violation of this assumption is called *lame delegation*. Lame delegation is caused by operational errors: A system administrator changes the name servers for a zone without changing the corresponding delegation information in the parent zone or notifying the system administrator of the parent zone about the change.

Assumption 5 *Attackers cannot eavesdrop on the DNS packets sent between our protected servers and the legitimate name servers.*

This is a limit we place on the attackers; if attackers can monitor the communication, our scheme may fail to cope with spoofing attacks. In the future, when the use of the DNS security extensions [8] (DNSSEC)—which employs digital signatures to authenticate DNS data—is widespread, we may drop this assumption. An implication of this assumption is that by randomizing the query id used, the probability that an attacker can forge a response whose id matches the randomized query id is small. Thus attempts for sending forged responses by guessing the query id used can be detected by the wrapper.

4.5. Our Goal

Our goal is to ensure that the view of a protected name server agrees with those of the corresponding authoritative name servers. This goal is specified using a VDM data invariant. A *data invariant* of a data type specifies the predicates that must hold true during the execution of a system. Our name server specification, which reflects the minimal functionalities of DNS servers among existing implementations, does not satisfy this data invariant because it allows non-authoritative DNS data to be used by a name server. Thus for a name server s, $Authoritative(\mathsf{rng}\ View(s))$ may not hold. In the next section, we will present our solution—a security wrapper for protecting name servers. Our DNS wrapper filters out DNS messages containing resource records that cannot be verified as authoritative. Therefore, a protected name server that satisfies the data invariant can be constructed by composing a name server and our DNS wrapper.

```
state DNS of
    protectedNS: Server-set
...
inv mk-DNS(protectedNS) △
    ∀s ∈ protectedNS Authoritative(rng View(s))
end
```

5. Our DNS Wrapper

We use *security wrapper* (or simply wrapper) to refer to a piece of software that encapsulates a component, such as a nameserver, to improve its security. Using wrappers to enhance the security of existing software is not a new idea. Related work includes TCP wrapper [16] and TIS' generic software wrappers [9]. However, our work is different in that it addresses problems that are DNS specific and it involves the use of formal specifications.

Consider a wrapper w. Wrapper w checks DNS response packets going to a name server and ensures that they are authenticated[6] and they agree with authoritative answers. If a resource record in the response does not come from an authoritative server, wrapper w locates an authoritative server and queries that server for the authoritative answer. To locate an authoritative server for a zone, say z, the wrapper starts with a server, say s, that is known to be an authoritative

[6] Data authentication checks can be performed by matching the query id's of queries to those of responses, or by using DNSSEC. However, the query id generation process used in some implementations of name servers is quite predictable. Before DNSSEC is widely deployed, we need a means to protect these name servers from spoofing attacks.

server for an ancestor zone of z, and queries server s for authoritative servers of the child zone that is either an ancestor zone of z or z itself. The search is performed by traversing the domain name tree, one zone at a time, until an authoritative server for the DNS data being verified is located. Recall that the zone data maintained by a server include the name server data of the delegated zones. Moreover, the zone data, including the zone cut data and the glue data, take precedence over RRs obtained from outside sources. Thus the delegation data is immune from cache poisoning attacks. Our scheme exploits this fact to securely locate the authoritative servers.

Let *ns* denote the name server protected by wrapper w. Our wrapper consists of two main parts: $Wrapper_sq$ for processing queries, and $Wrapper_sr$ for processing responses. (The subscript s stands for "server".) Wrapper w processes queries generated by *ns* before they are sent out, and processes queries destined for *ns*. Wrapper w also processes responses destined for *ns*; those that are accepted by w will be forwarded to *ns*.

When *ns* sends a query, wrapper w generates a random query id and uses it to replace the original query id (used by *ns*). We use a translation table to track the mapping between the random query id's used by w and the original query id's used by *ns*.

$Wrapper_sq$ processes queries that involve *ns*. These queries can be partitioned into two types. The first type corresponds to the queries that are sent to *ns*. The second type corresponds to the queries that are generated by *ns*. These two types of queries are treated differently. For the first type, wrapper w checks the queries to determine whether they are well-formed (e.g., the answer, the authority, and the additional sections for a standard query should be empty). For the second type, the wrapper generates a random query id, replaces the query id used in the original query by this randomly generated query id, and updates the local query id translation table.

$Wrapper_sr$ processes responses that are received by the wrapper. $Wrapper_sr$ has two components: $Wrapper_sr1$ and $Wrapper_sr2$. $Wrapper_sr1$ screens out forged response messages. In other words, response authentication is hardened. $Wrapper_sr2$ verifies the response messages to ensure that they agree with authoritative answers, and copes with cache poisoning attacks. There are two types of responses received by a wrapper: responses for queries generated by the protected name server *ns*, and responses for queries generated by the wrapper itself (for message diagnosis purposes). When a response for a query generated by *ns* is received, the wrapper uses the query id translation table to restore the query id (to the one used by *ns*) before passing the response to $Wrapper_sr2$.

6. Experiments

6.1. Overview

We conducted experiments to evaluate the response time (i.e., the elapsed time between sending a query to a name server and receiving a response from it) of a wrapped name server, and to evaluate the false positive rate, the false negative rate, and the computational overhead (i.e., CPU time used) of our wrapper.

Based on the DNS wrapper specification, we implemented a prototype of the DNS wrapper for BIND release 4.9.5, which was the latest release for BIND when we started our implementation. The DNS wrapper was written in C. We modified the BIND name server source code to invoke the DNS wrapper upon receiving queries and responses and upon sending queries to other name servers.

In this section, we describe two sets of experiments and their results. In Experiment A, we examined the response time, the false positive rate, and the computational overhead of our wrapper using a trace of DNS queries received by a name server in an operational setting. In Experiment B, we examined the false negative rate of our wrapper with respect to four attacks: three cache poisoning attacks and one spoofing attack.

6.2. General Experimental Setup

In these experiments, our name servers (BIND 4.9.5) listened to port 4000 instead of port 53 (the *de facto* standard port number for name servers) for DNS queries to prevent queries outside our experiments from affecting our results.

In every run of our experiments, we started a fresh copy of our name server because name servers maintain a cache for DNS information obtained through interacting with other name servers. The behavior of a name server can be quite different depending on whether the DNS information queried can be found in the cache. Restarting name servers can avoid interference between consecutive runs of the experiment.

We used a modified version of *nslookup* as the DNS client in our experiments. (See [1] for a good tutorial on *nslookup*.) We chose *nslookup* because it is a convenient tool for generating DNS queries and displaying DNS responses. Moreover, *nslookup* can be easily configured to use a specified name server port number and to query a specified name server. Our modified *nslookup* uses Unix *gethrtime*() system calls to record

the time when a query is sent and when the corresponding response is received. Unless otherwise specified, we will use *nslookup* to refer to this modified version of *nslookup*.

Our experiments were performed on a lightly loaded Sun SPARC-5 running Solaris 2.5.1. We ran our name servers and *nslookup* on the same machine to eliminate the network latency for the communication between them, thus reducing the influence of the local area network load on the experimental results.

Because we did not have control over external name servers, and the inter-network links between our name server and external name servers, we performed Experiment A multiple times and calculated the average response time.

6.3. Experiment A

6.3.1 Data Set

The data set for Experiment A consisted of a trace of 1340 DNS queries received by a name server in a "real world" setting. To gather the trace of DNS queries, we modified a name server to log all DNS queries it received and ran it for two days. We also modified the local BIND resolver configuration file to direct all DNS queries to this name server. In the resolver configuration file, the *search list* was consisted of cs.ucdavis.edu., ucdavis.edu., and ucop.edu. When a BIND resolver is invoked to resolve a *relative* domain name—a domain name that does not have a trailing dot—it appends the domain names in the order specified in the search list and attempt to resolve them until a positive response is received. If none of them results in a successful resolution, the resolver then generates a query for the relative domain name itself. For example, when the BIND resolver is invoked for domain name dn, it attempts to resolve for dn.cs.ucdavis.edu., dn.ucdavis.edu., dn.ucop.edu., and dn in that order until a successful resolution is obtained.

6.3.2 Experimental Procedure

1. Start a wrapped name server.
2. Run *nslookup* to query the wrapped name server for resolving the 1340 DNS queries sequentially.
3. Record the total system CPU time and the total user CPU time used.
4. Terminate the wrapped name server.
5. Repeat the above procedure using an unmodified name server instead of a wrapped name server.

6.3.3 Experimental Results

Table 1 shows the statistics related to response times recorded by *nslookup* based on 33 runs of this experiment. The mean response time for the wrapped server was 0.12 second per query, and that for the unmodified server was 0.08 second per query. We examined the trace segments that correspond to "steep" increases in the response times (e.g., 400^{th}-600^{th} query), we found that they could be explained by DNS queries generated by web surfing sessions, which involved mostly remote and distinct domain names. Specifically, the trace segment for the 400^{th}-600^{th} query included 43 remote and distinct domain names. The average total response times for those 43 queries for the unmodified server and the wrapped server were 28.29 seconds and 47.54 seconds respectively, which accounted for 83% and 88% of the total response times for that interval respectively.

Table 2 shows the CPU times used by the unmodified server and the wrapped server. The figures show that the average CPU times used are a small fraction (8% for the unmodified server and 7% for the wrapped server) of the total response time. Thus the response time overhead of the wrapper reported in Table 1 was largely due to waiting for the response messages in the message diagnosis process. The average total CPU time increased from 9.33 seconds to 11.29 seconds (i.e., a 21% increase).

The number of false positives ranged from 2-10 per run, with the mean being 5.85 and the standard deviation being 1.89. Among the false positives, 80% of them were caused by name server behaviors that violate our name server specification or to a violation of our assumptions. For example, false positives caused by misconfigurations of name servers are in this category. The remaining 20% of the false positives were generated when the wrapper gave up on diagnosing a DNS message after the amount of resources spent (e.g., the number of DNS queries issued) had reached a threshold. The threshold is used to ensure that the amount of resources used for verifying a message is bounded, thus protecting the wrapper from problems like denial of service attacks.

6.4. Experiment B

The main goal of Experiment B is to examine the detection rate of malicious attacks of a wrapped name server (i.e., false negative rate). We investigated the following four types of attacks:

- *Sending incorrect resource records for a remote domain name to the target*: This is accomplished by

Table 1. Cumulative Response Time (in Sec.) for the "Two-day trace" Data Set.

	Unmodified Name Server				Wrapped Name Server			
# queries	Mean	Min	Max	Std Dev	Mean	Min	Max	Std Dev
200	6.24	3.75	19.78	3.62	8.83	4.27	24.52	4.66
400	11.63	7.44	23.99	4.06	19.56	10.48	34.53	6.37
600	45.59	22.29	147.99	28.31	73.42	40.66	270.41	45.65
800	59.71	35.60	171.83	28.99	94.66	58.53	312.98	49.46
1000	74.15	40.28	263.69	50.93	111.69	70.72	332.60	62.77
1200	96.71	55.36	370.10	75.22	145.10	85.13	396.51	87.50
1340	111.05	70.52	392.48	78.75	165.96	102.38	439.51	91.96

Table 2. System and User Times Used (in Sec.) for the "Two-day Trace" Data Set.

	Unmodified Name Server				Wrapped Name Server			
Type	Mean	Min	Max	Std Dev	Mean	Min	Max	Std Dev
System	4.01	3.43	4.90	0.36	5.32	4.59	5.82	0.33
User	4.35	3.73	4.90	0.26	6.94	6.31	7.90	0.43

using a CNAME resource record in the answer section of a response message to introduce (in the resource data field) an arbitrary domain name for which the target server is not authoritative, and then including incorrect resource records for this remote domain name in the additional section of the response message.

- *Sending incorrect resource records that conflict with the zone data for which the target is authoritative*: In particular, the attacker uses a CNAME resource record to link to an A resource record for which the target is authoritative.
- *Sending resource records that correspond to a non-existing domain name that lives in the target server's zone.*
- *Sending a response with a guessed query id*: In this attack, one queries the target server to trigger it to send a query to the attacker, whom records the query id used. A second query is then issued to trigger the target to query the attacker again. Instead of using the query id of the second query, the attacker adds one to the query id used in the first query and uses the result as the query id in its second response.

The first three types of attacks correspond to sending incorrect DNS data to a name server (i.e., cache poisoning attacks). The fourth type of attacks corresponds to masquerading attacks. Our wrapper used randomized query id's for outgoing queries. Thus attackers who do not have access to those queries will have to guess the query id's used for their forged response messages. As a result, their forged messages will be detected with high probability.

6.4.1 Data Set

In Experiment B, we modified the data set used in Experiment A by inserting two queries that correspond to each of the four types of attacks at random locations in the two-day trace. Moreover, we also inserted four queries at random locations in the trace as controls. These queries correspond to different domain names in the domain for which a malicious name server is authoritative but do not trigger an attack.

6.4.2 Experimental Procedure

1. Start a malicious name server for a new subdomain dns.cs.ucdavis.edu. When that malicious name server is asked to resolve for certain domain names that reside in the dns.cs.ucdavis.edu. domain, depending on the domain names queried, it will either return incorrect DNS resource records or send out response messages with an incorrect query id or a predicted query id.
2. Start a wrapped name server.
3. Run *nslookup* with the modified trace of DNS queries as input and send the queries sequentially

to the wrapped name server.

4. Terminate the wrapped name server.
5. Terminate the malicious name server.
6. Repeat the above procedure using an unmodified name server instead of a wrapped name server.

6.4.3 Experimental Results

We ran the experiment five times. In all five runs, all eight attacks (i.e., two from each of the four attack types) were reported correctly by the wrapped name server, and none of the response messages corresponding to the control queries were misclassified as attacks.

When we applied these four types of attacks to an unmodified name server, the first type of attacks succeeded in planting incorrect DNS data into the cache of the target server. For the second and the third type, the unmodified name server did not cache the incorrect DNS data for domain names that belong to its authoritative domain. However, the name server did forward the entire response message received, including those incorrect resource records for which the name server was authoritative, to its client. That did not make much difference for our experiments because the client used was *nslookup* which did not perform caching. However, if the client was another name server that was not authoritative for those incorrect DNS data, the cache of the client would be corrupted. This situation may occur when the client is a *caching-only server*[7] that uses another name server as a *forwarder*[8]. The fourth type of attacks succeeded for an unmodified name server. It was because the query id used by the unmodified name server was predictable: the query id used in successive queries always differed by one.

7. Conclusions and Future Work

This paper presents a detection-response approach for protecting DNS. Our approach consists of the following steps. First, we define a security goal—name servers only use DNS data that are consistent with the corresponding authoritative data. Second, we declare the threats, namely cache poisoning and spoofing attacks. Third, we develop a DNS model, which includes formal characterizations of DNS clients and DNS servers. Fourth, we design a DNS wrapper with the objective that the composition of the specification for a protected name server and that for the wrapper satisfies our security goal for DNS. If the DNS wrapper receives a DNS message that may cause violations of the security goal, the wrapper drops the message instead of forwarding it to the protected name server. Fifth, we use the formal specification for the wrapper to guide our implementation of a wrapper prototype.

To counter cache poisoning, Vixie [17] presents enhancements to BIND. Briefly, BIND version 4.9.3 checks the input resource records more carefully before caching them. Moreover, it implements a credibility level scheme in which resource records from a more credible source take precedence over those from a less credible one. Cheswick and Bellovin [6] present a design for a DNS proxy (*dnsproxy*). In their design, the domain name space is partitioned into regions called *realms*. A realm is served by a set of servers. Depending on the query name of a DNS request, *dnsproxy* forwards the request to the servers responsible for the corresponding realm. Certain resource records in response messages—those that do not refer to realm to which the query name belongs, and those that satisfy a set of filtering rules—are removed to protect the queriers. Eastlake and Kaufman [8] present security extensions to DNS (DNSSEC) that uses digital signatures to support data authentication for DNS data. In DNSSEC, new resource record types are introduced for public keys and digital signatures. Security-aware servers and security-aware resolvers can use zone keys, which are either statically configured or learned by chaining through zones, to verify the origins of resource records. Compared to the prior work for protecting DNS, our DNS wrapper has the following advantages:

- Provides assurance by employing formal specifications (written in VDM) to characterize DNS components, to state the security goal, and to characterize our solution.
- Effective against cache poisoning attacks and certain spoofing attacks (i.e., query id guessing) when the assumptions in Section 4.4 are met.
- Compatible with existing DNS implementations.
- Does not require changes for the DNS protocol.
- Incurs reasonable performance overhead.
- Can be deployed locally; does not depend on changes to other remote DNS components.

In November 1998, a company called Men & Mice surveyed the status of name servers on the Internet

[7] A caching-only server is a name server that is not authoritative for any domain.

[8] A forwarder is a name server to which other name servers forward their recursive queries. A forwarder is useful for building a large cache for remote DNS data, especially when communication between local machines and remote machines is slow or restricted.

[12]. Among 4184 randomly picked com zones, 1344 of them (i.e., 32.1%) were found to be vulnerable to cache poisoning attacks. In other words, the name servers of those zones could be compromised and gave out incorrect information about other domains, including its delegated domains. We note that the effectiveness of our DNS wrapper is not affected by attacks against external name servers as long as our assumptions are met.

There are several directions for future research.

- To further raise the assurance level of our wrapper, one may perform a complete formal verification from specification to implementation. The VDM specifications developed can be used as the basis for conducting the formal verification.
- Results from Experiment A showed a 0.437% false positive rate for the DNS wrapper. Because the majority of these false positives were caused by misconfigurations of external name servers, a non-trivial modification for the DNS wrapper may be needed to significantly reduce the false positive rate.
- We have not discussed protecting DNS resolvers. If the communication path between a resolver and its trusted local name server is secure, and the name server is protected by the DNS wrapper, the DNS data received by the resolver is "safe" because a wrapped name server only uses DNS data that are consistent with the corresponding authoritative answers. Future research may be conducted to protect DNS resolvers when the resolver-server communication path is insecure. A possibility is to adapt the DNS wrapper to protect resolvers.
- One may apply our approach to protect other network services and privileged processes.

8. Acknowledgments

This work was supported by DARPA under grant ARMY/DAAH 04-96-1-0207.

References

[1] P. Albitz, and C. Liu, "DNS and BIND." O'Reilly and Associates, Inc., 1992.

[2] D. Andrews, and D. Ince, "Practical Formal Methods with VDM." McGraw-Hill, 1991.

[3] S.M. Bellovin, "Security Problems in the TCP/IP Protocol Suite." *Computer Communications Review*, Vol.19, No.2, April 1989, pp.32-48.

[4] S. Bellovin, "Using the Domain Name System for System Break-ins." *Proc. of the 5th UNIX Security Symposium,* June 5-7, 1995, pp.199-208.

[5] CERT Coordination Center, "Multiple Vulnerabilities in BIND." CERT Advisory CA-98:05, April 8, 1998.

[6] B. Cheswick, and S. Bellovin, "A DNS Filter and Switch for Packet-filtering Gateways." *Proc. of the 6th UNIX Security Symposium,* July 22-25, 1996, pp.15-19.

[7] S. Cheung, "An Intrusion Tolerance Approach for Protecting Network Infrastructures." Ph.D. Dissertation, University of California, Davis, September 1999.

[8] D. Eastlake, 3rd, and C. Kaufman, "Domain Name System Security Extensions." RFC 2065, January 1997.

[9] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers." *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 5-7, 1999, pp.2-16.

[10] E. Gavron, "A Security Problem and Proposed Correction with Widely Deployed DNS Software." RFC 1535, October 1993.

[11] C.B. Jones, "Systematic Software Development using VDM." Prentice-Hall, 1990.

[12] Men and Mice, "Domain Health Survey." http://www.menandmice.com, November 1998.

[13] P. Mockapetris, "Domain Names – Concepts and Facilities." RFC 1034, November 1987.

[14] P. Mockapetris, "Domain Names – Implementation and Specification." RFC 1035, November 1987.

[15] C.L. Schuba, and E.H. Spafford, "Addressing Weaknesses in the Domain Name System Protocol." Technical Report, Department of Computer Sciences, Purdue University, 1994.

[16] W. Venema, "TCP Wrapper: Network Monitoring, Access Control, and Booby Traps." *Proc. of the 3rd UNIX Security Symposium,* September 1992, pp.85-92.

[17] P. Vixie, "DNS and BIND Security Issues." *Proc. of the 5th UNIX Security Symposium,* June 5-7, 1995, pp.209-216.